

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

## Introduction, Overview, and Syllabus

Christian Kästner   Vincent Hellendoorn



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

## Introduction, Overview, and Syllabus

Christian Kästner

Vincent Hellendoorn



# A Few Questions

How many lines of code behind twitter.com?

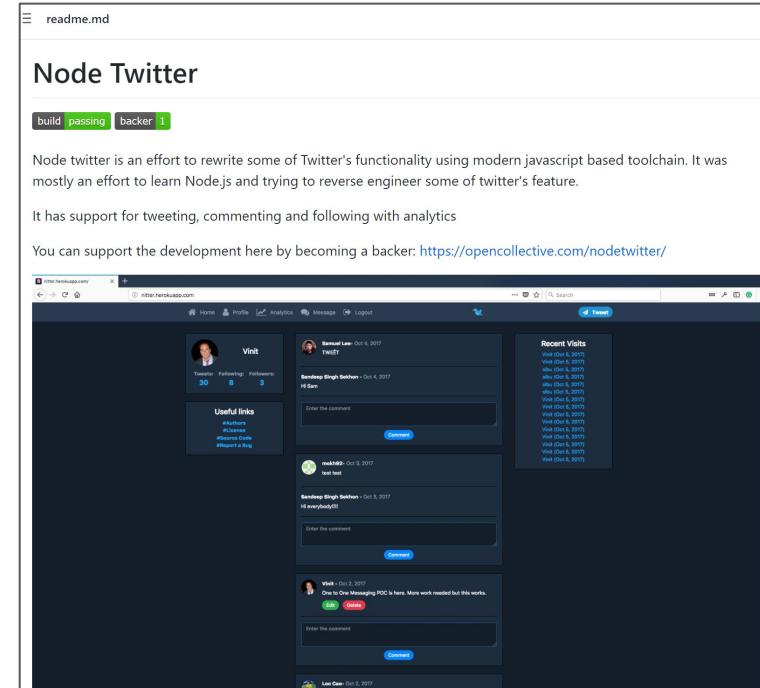
> A few million, maybe many

How many LOC to build an okay Twitter replica?

> A few 10K

How many LOC to run a Twitter replica?

> A few



<https://github.com/vinitkumar/node-twitter>

# Modern Software Engineering

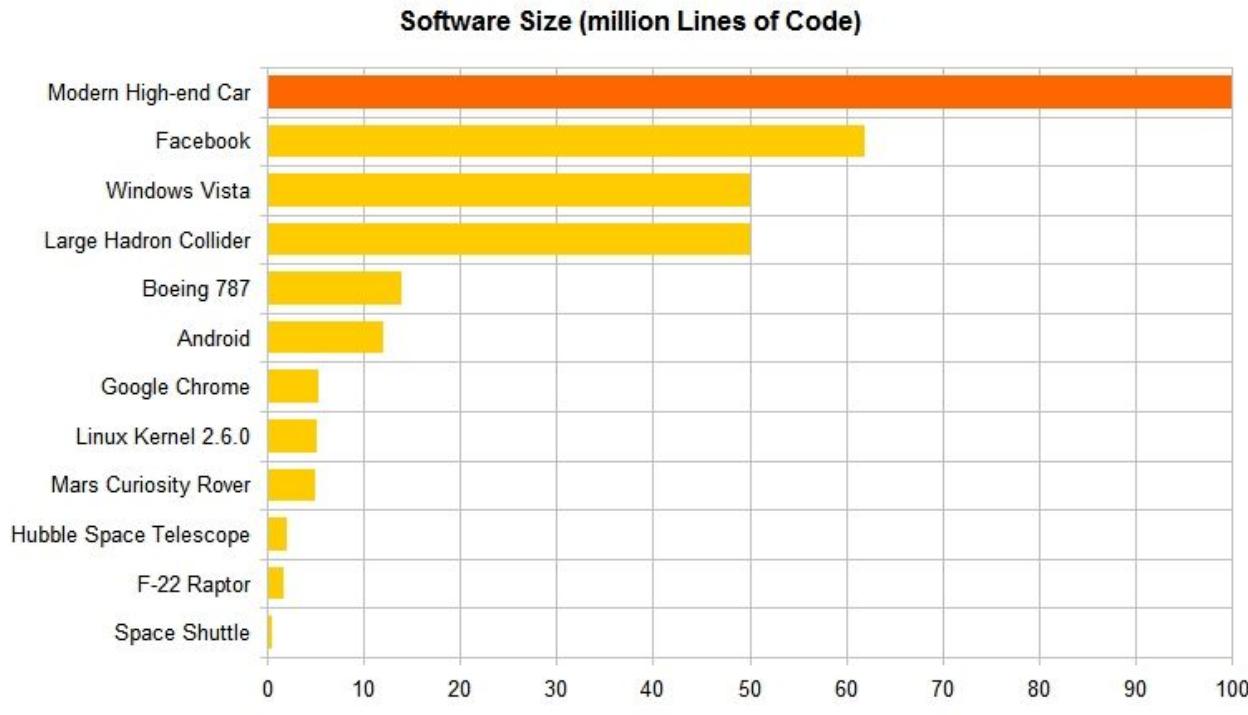
So, why learn “principles of software construction”?

- You don't want to build Twitter
  - But you can reuse the components
- “A few lines of code” does not mean easy
  - You'll need to become fluent in using new systems

There are many ways to compose applications

- An engineer knows the pieces **and** how to put them together.

# Welcome to the era of “big code”

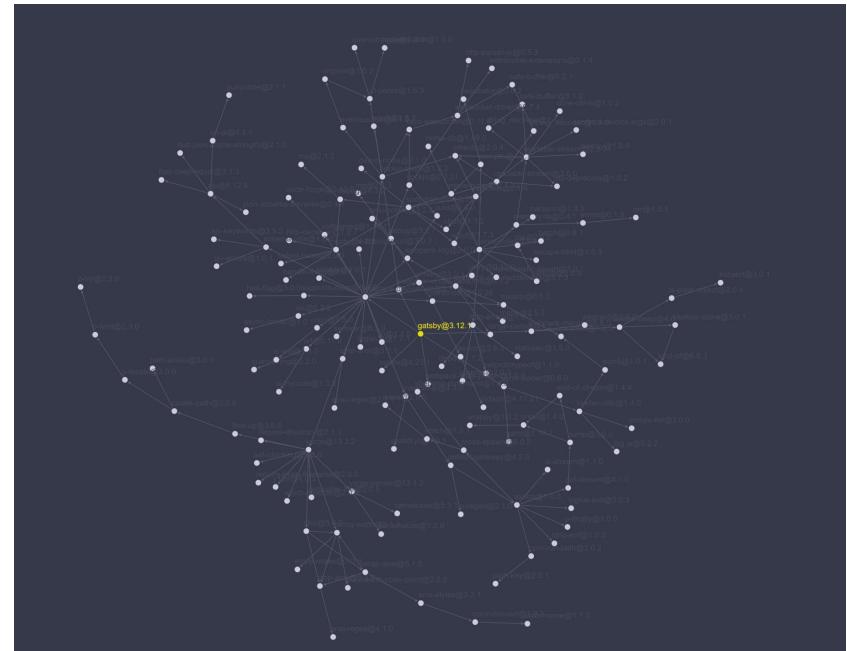


(informal reports)

# Modern Software Engineering

Nobody wants to write a million lines of code.

- Instead, you use libraries
    - E.g., import Android => +12M LOC
    - You don't write most of the code you use
      - And why would you want to?
  - And your libraries use libraries
    - Et cetera
    - <https://npm.anvaka.com/#/view/2d/gatsby>

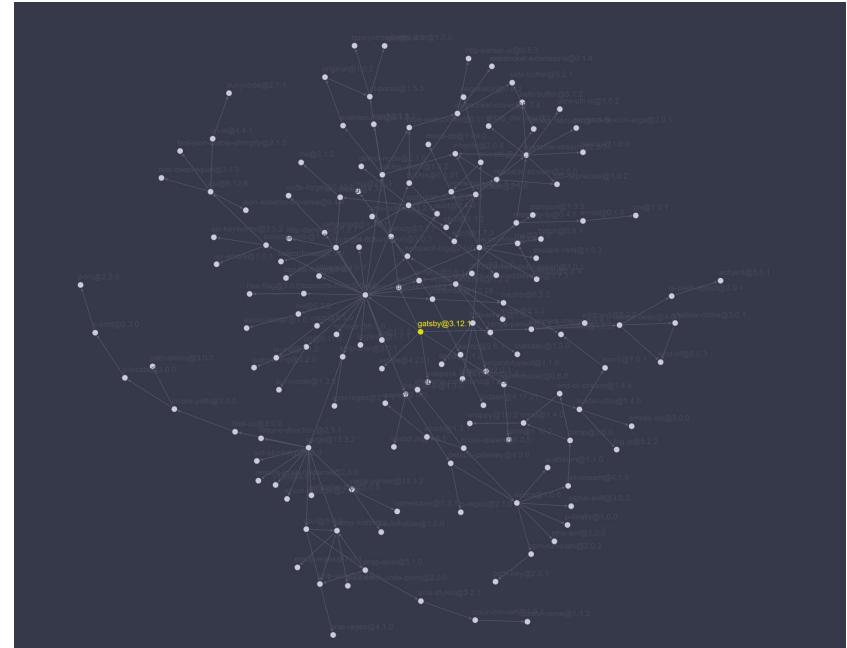


# Modern Software Engineering

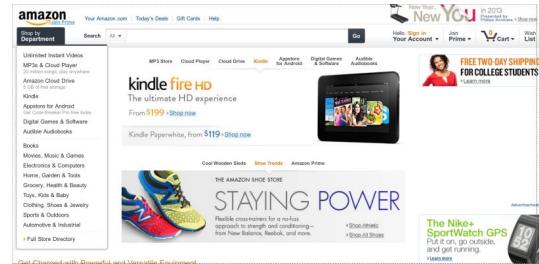
Nobody wants to write a million lines of code.

- Instead, you use libraries
    - E.g., import Android => +12M LOC
    - You don't write most of the code you use
      - And why would you want to?
  - And your libraries use libraries
    - Et cetera
    - <https://npm.anvaka.com/#/view/2d/gatsby>

## What are the implications?



primes  
binary  
tree  
sorting  
**BDDs**



# Equipment of a Modern Programmer

**Less emphasis on:** (*though not unimportant!*)

- Clever algorithmics
- Low-level code (kernels, drivers)
- Writing common components (command-line parsers, HTML)

# Equipment of a Modern Programmer

More emphasis on:

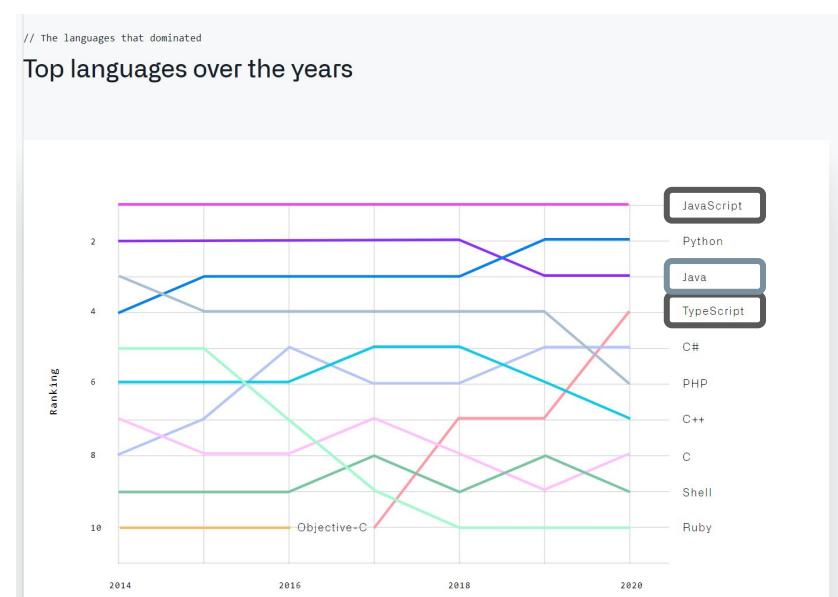
- Using APIs, libraries (hw1)
- Quality assurance (hw2)
- Design for reuse, extension (hw3+)
- Flexibility in ecosystems (all)

# Flexibility & Ecosystems

*Flexibility is perhaps the key skill, besides good design.*

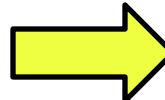
In this course:

- learn to choose & use libraries
- Adopting new tools, troubleshooting
- Also, Java vs. JavaScript/TypeScript



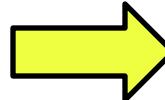
# From Programs to Applications and Systems

Writing algorithms, data structures from scratch



Reuse of libraries, frameworks

Functions with inputs and outputs



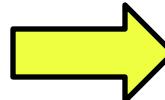
Asynchronous and reactive designs

Sequential and local computation



Parallel and distributed computation

Full functional specifications



Partial, composable, targeted models

Our goal: understanding both the **building blocks** and also the **design principles** for construction of software systems at scale

# Outcomes, hopefully

You'll learn to be:

- An architect, approaching programming as design
  - This is the only way to scale up to larger systems
  - You'll learn a rich vocabulary, of both components and their combinations
- An omniglot, able to pick up new languages and libraries
  - Because you know the underlying concepts
  - And you've had plenty of practice reading documentation, debugging setups
- An engineer, safeguarding the quality of your programs
  - You'll get dexterous at testing, be explicit about specification
  - You'll know the tools that improve your work

# Applications in this Course (Homeworks)

Flashcard learning app (command line)

Static website generator / CMS (command line application)

Board game with web interface (could also be a mobile app)

Data analysis and visualization tool (desktop/web application)

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

## Introduction, Overview, and Syllabus

Christian Kästner   Vincent Hellendoorn



# Objects in the real world



# Object-oriented programming

Programming based on structures that contain both data and methods

```
public class Bicycle {  
    private int speed;  
    private final Wheel frontWheel, rearWheel;  
    private final Seat seat;  
    ...  
    public Bicycle(...) { ... }  
  
    public void accelerate() {  
        speed++;  
    }  
  
    public int speed() { return speed; }  
}
```



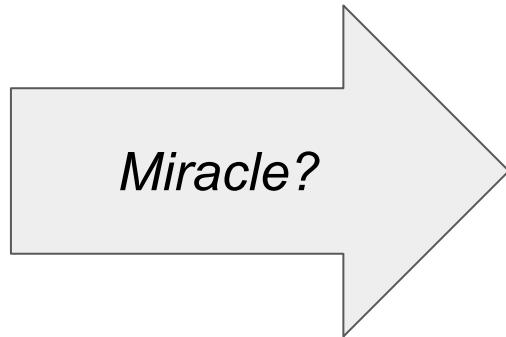
# Principles of Software Construction: Objects, **Design**, and Concurrency

## Introduction, Overview, and Syllabus

Christian Kästner    Vincent Hellendoorn

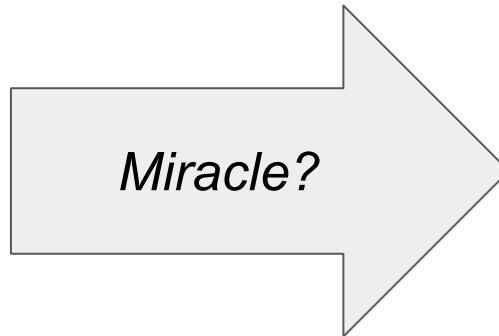


User needs  
(Requirements)



Code

User needs  
(Requirements)



Code

Maintainable?  
Testable?  
Extensible?  
Scalable?  
Robust? ...

# Semester overview

- Introduction to Object-Oriented Programming
- Introduction to **design**
  - **Design** goals, principles, patterns
- **Designing** objects/classes
  - **Design** for change
  - **Design** for reuse
- **Designing** (sub)systems
  - **Design** for robustness
  - **Design** for change (cont.)
- **Design** for large-scale reuse

## Crosscutting topics:

- Building on libraries and frameworks
- Building libraries and frameworks
- Modern development tools: IDEs, version control, refactoring, build and test automation, static analysis
- Testing, testing, testing
- Concurrency basics

# Sorting with a configurable order, version A

```
static void sort(int[] list, boolean ascending) {  
    ...  
    boolean mustSwap;  
    if (ascending) {  
        mustSwap = list[i] > list[j];  
    } else {  
        mustSwap = list[i] < list[j];  
    }  
    ...  
}
```

# Sorting with a configurable order, version B

```
interface Order {  
    boolean lessThan(int i, int j);  
}  
  
class AscendingOrder implements Order {  
    public boolean lessThan(int i, int j) { return i < j; }  
}  
class DescendingOrder implements Order {  
    public boolean lessThan(int i, int j) { return i > j; }  
}  
  
static void sort(int[] list, Order order) {  
    ...  
    boolean mustSwap =  
        order.lessThan(list[j], list[i]);  
}
```

# Sorting with a configurable order, version B'

```
const ASC = function(i: number, j: number): boolean {
    return i < j;
}

const DESC = function(i: number, j: number): boolean {
    return i > j;
}

function sort(
    list: number[],
    order: (number, number) => boolean) {

    ...
    boolean mustSwap = order(list[j], list[i]);
    ...
}

> sort(list, ASC);
```

# Which version is better?

Version A:

```
static void sort(int[] list, boolean ascending) {  
    ...  
    boolean mustSwap;  
    if (ascending) {  
        mustSwap = list[i] > list[j];  
    } else {  
        mustSwap = list[i] < list[j];  
    }  
    ...  
}
```

```
interface Order {  
    boolean lessThan(int i, int j);  
}  
class AscendingOrder implements Order {  
    public boolean lessThan(int i, int j) { return i < j; }  
}  
class DescendingOrder implements Order {  
    public boolean lessThan(int i, int j) { return i > j; }  
}
```

```
static void sort(int[] list, Order order) {  
    ...  
    boolean mustSwap =  
        order.lessThan(list[j], list[i]);  
    ...
```

Version B':

**it depends**

# **it depends**

**Depends on what?  
What are scenarios?  
What are tradeoffs?**

# **it depends**

**Depends on what?  
What are scenarios?  
What are tradeoffs?**

**In this specific case, what  
would you recommend?  
(Engineering judgement)**

Software engineering is the branch of computer science that creates **practical, cost-effective solutions** to computing and information processing problems, preferably by applying scientific knowledge, developing software systems in the service of mankind.

**"Software engineering** is the branch of computer science that creates practical, cost-effective solutions to computing and information processing problems, preferentially by applying scientific knowledge, developing software systems in the service of mankind.

Software engineering entails making **decisions** under constraints of limited time, knowledge, and resources. [...]

Engineering quality resides in engineering judgment. [...]

Quality of the software product depends on the engineer's faithfulness to the engineered artifact. [...]

Engineering requires reconciling conflicting constraints. [...]

Engineering skills improve as a result of careful systematic reflection on experience. [...]

# Goal of software design

- Think before coding
- For each desired program behavior there are infinitely many programs
  - What are the differences between the variants?
  - Which variant should we choose?
  - How can we synthesize a variant with desired properties?
- Consider qualities: Maintainability, extensibility, performance, ...
- Make explicit design decisions

# Tradeoffs?

```
static void sort(int[] list, boolean ascending) {  
    ...  
    boolean mustSwap;  
    if (ascending) {  
        mustSwap = list[i] > list[j];  
    } else {  
        mustSwap = list[i] < list[j];  
    }  
    ...  
}
```

```
interface Order {  
    boolean lessThan(int i, int j);  
}  
class AscendingOrder implements Order {  
    public boolean lessThan(int i, int j) { return i < j; }  
}  
class DescendingOrder implements Order {  
    public boolean lessThan(int i, int j) { return i > j; }  
}
```

```
static void sort(int[] list, Order order) {  
    ...  
    boolean mustSwap =  
        order.lessThan(list[j], list[i]);  
    ...
```

# Some qualities of interest, i.e., *design goals*

Functional correctness	Adherence of implementation to the specifications
Robustness	Ability to handle anomalous events
Flexibility	Ability to accommodate changes in specifications
Reusability	Ability to be reused in another application
Efficiency	Satisfaction of speed and storage requirements
Scalability	Ability to serve as the basis of a larger version of the application
Security	Level of consideration of application security

Source: Braude, Bernstein,  
Software Engineering. Wiley  
2011

# A typical Intro CS design process

1. Discuss software that needs to be written
2. Write some code
3. Test the code to identify the defects
4. Debug to find causes of defects
5. Fix the defects
6. If not done, return to step 1

# Better software design

- Think before coding: broadly consider quality attributes
  - Maintainability, extensibility, performance, ...
- Propose, consider design alternatives
  - Make explicit design decisions

# Using a design process

- A design process organizes your work
- A design process structures your understanding
- A design process facilitates communication

# Preview: Design goals, principles, and patterns

- **Design goals** enable evaluation of designs
  - e.g. maintainability, reusability, scalability
- **Design principles** are heuristics that describe best practices
  - e.g. high correspondence to real-world concepts
- **Design patterns** codify repeated experiences, common solutions
  - e.g. template method pattern

# Software Engineering at CMU

- 17-214: “Code-level” design
  - extensibility, reuse, concurrency, functional correctness, medium-size to large programs
- 17-313: “Human aspects” of software development
  - requirements, team work, balancing qualities, scheduling, costs, risks, business models
- 17-413 Practicum, Seminar, Internship
- SE electives: SE4Startups, Program Analysis, SE4ML
- Various master-level courses on requirements, architecture, software analysis, etc
- SE Minor/Concentration: <http://isri.cmu.edu/education/undergrad/>

**This is not a  
Java course**

**This is not a  
Java course**

**but you will write a  
lot of  
Java/JavaScript code**

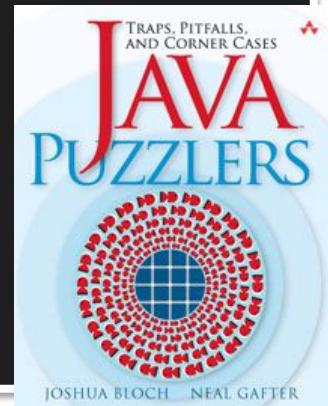
```
int a = 010 + 3;  
System.out.println("A" + a);
```

```
int a = 010 + 3;  
System.out.println("A" + a);
```

```
const a = 010 + 3;  
console.log("A" + a);
```

```
int a = 010 + 3;  
System.out.println("A" + a);
```

```
const a = 010 + 3;  
console.log("A" + a);
```



# Java + JavaScript / TypeScript

Focus on design concepts, not programming language

Language proficiency through practice and homeworks

Lectures show examples in pseudo code, Java, JavaScript, TypeScript, and other languages

Pick Java or TypeScript for homeworks

```
int a = 010 + 3;
```

```
System.out.println("A"
```

```
const a = 010 + 3;  
console.log("A" + a);
```

# Java **OR** TypeScript/JavaScript

Your choice!

Pick in each homework assignment

Most students will use one language for all assignments

Each recitation focuses on tools/examples in one language

tentatively: A, C, E for Java - B, D, F for TypeScript

# COURSE ORGANIZATION

A photograph of a wildfire in a dry, brushy area. In the foreground, a wooden signpost stands upright. The top horizontal bar of the signpost has the words "SENIOR CENTER" in white capital letters. Below this, the main vertical sign has four lines of yellow capital letters: "WEAR A MASK", "WASH YOUR HANDS", "SOCIAL DISTANCE", and "STAY SAFE". At the bottom of the signpost, another horizontal bar reads "COME JOIN US" in white capital letters. The background is filled with intense orange and yellow flames, smoke, and charred vegetation.

**SENIOR CENTER**

**WEAR A MASK  
WASH YOUR HANDS  
SOCIAL DISTANCE  
STAY SAFE**

**COME JOIN US**

A photograph of a wooden signpost set against a backdrop of intense orange and yellow flames. The sign has four horizontal panels. The top panel reads "SENIOR CENTER". The middle-left panel contains the text "WEAR A MASK", "WASH YOUR HANDS", "SOCIAL DISTANCE", and "STAY SAFE" stacked vertically. The bottom-left panel reads "COME JOIN US". The sign is partially obscured by smoke and fire.

Trying to get back to normal with ...  
*\*gestures widely\** everything

Talk to us about concerns and accommodations



## **Disclaimer:**

This semester, we are changing a lot in this course.  
Some things will go wrong.  
Have patience with us.  
Give us feedback.

# Course materials

Course website (syllabus, slides, calendar): <https://cmu-17-214.github.io/f2021/>

Discussions, questions, announcements: Piazza

Assignments, readings, and grades: Canvas and Gradescope

Homework submission: GitHub (signup instructions in assignment) and other tools

# Course preconditions

- 15-122 or equivalent: Basic programming skills in any language, algorithms and data structures (lists, graphs, sorting, binary search)
- 21-127 or equivalent: Basic discrete math concepts, logic

# Course staff

- Christian Kästner  
[kaestner@cs.cmu.edu](mailto:kaestner@cs.cmu.edu), TCS 345
- Vincent Hellendoorn  
[vhellendoorn@cmu.edu](mailto:vhellendoorn@cmu.edu), TCS 320
- Teaching assistants:  
Esther, Katrina, Kevin, Jeff, Olivia, Sean, Sophie, Ye, and Zhifeng



# Course meetings

- Lectures: Tuesday and Thursday 11:50 – 1:10pm here :)
- Recitations: Wednesdays 9:30 - ... - 3:20pm
  - Preparing for homeworks, hands-on practice, supplementary material
  - Starting tomorrow! (version control and git -- relevant for HW1)
- Office hours: see course web page

*Recitation  
attendance  
is required*

# Homework & Exams

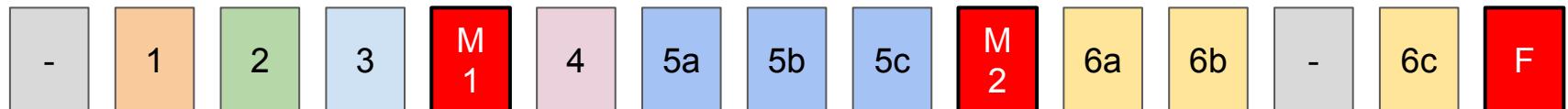
6 homeworks, 4 small + 2 large (with milestones), 1000 points total

- (1) intro, (2) testing, (3) first design, (4) fixing design,
- (5) extensibility + GUI, (6) framework and API design

Homeworks and milestones usually due Sunday night, see course calendar

Homework 1 will be released Thursday, due Sep 12

Two midterms + final

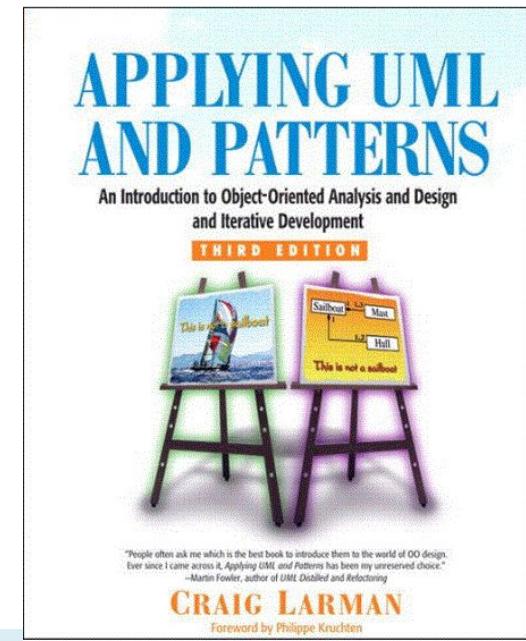


# Late day policy

- See syllabus on course web page for details
- 2 possible late days per deadline (some exceptions will be announced)
  - 5 total free late days for semester (+ separate 2 late days for assignments done in pairs)
  - 10% penalty per day after free late days are used
  - but we won't accept work 3 days late
- Extreme circumstances – talk to us

# Textbooks

- Craig Larman. **Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.** 3rd Edition. Prentice Hall. 2004. ISBN 0-13-148906-2
- Joshua Bloch. Effective Java, Third Edition. Addison-Wesley, ISBN 978-0-13-468599-1.
- Selective other readings throughout the semester
- Occasional in-class reading quizzes after reading assignment due
- Electronic version available through CMU library



# Approximate grading policy

- 50% assignments
- 20% midterms (2 x 10% each)
- 20% final exam
- 10% quizzes and participation

This course does not have a fixed letter grade policy; i.e., the final letter grades will not be A=90-100%, B=80-90%, etc.

# Collaboration policy

- See course web page for details!
- We expect your work to be your own
- Do not release your solutions (not even after end of semester)
- Ask if you have any questions
- If you are feeling desperate, please reach out to us
  - Always turn in any work you've completed before the deadline
- We run cheating detection tools. Trust us, academic integrity meetings are painful for everybody

# 10% quizzes and participation

- Recitation participation counts toward your participation grade
- Lecture has in-class quizzes

*The key to your success in this course is your regular, engagement with course activities, staff, and other students*

# Summary

- Software engineering requires decisions, judgment
- Good design follows a process
- You will get lots of practice in 17-214!

# Principles of Software Construction (Design for change, class level)

## Starting with Objects (dynamic dispatch, encapsulation)

Christian Kästner      Vincent Hellendoorn





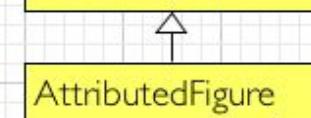
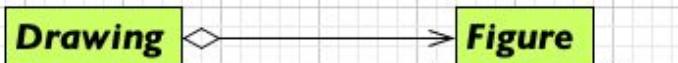
# Tradeoffs?

```
void sort(int[] list, String order) {  
    ...  
    boolean mustswap;  
    if (order.equals("up")) {  
        mustswap = list[i] < list[j];  
    } else if (order.equals("down")) {  
        mustswap = list[i] > list[j];  
    }  
    ...  
}
```

```
void sort(int[] list, Comparator cmp) {  
    ...  
    boolean mustswap;  
    mustswap = cmp.compare(list[i], list[j]);  
    ...  
}  
interface Comparator {  
    boolean compare(int i, int j);  
}  
class UpComparator implements Comparator {  
    boolean compare(int i, int j) { return i < j; }  
}  
class DownComparator implements Comparator {  
    ...  
}
```

unnamed  
jhotdraw7.xml

JHotDraw7  
A Java Framework for Structured Drawing Editors



TextAreaFigure

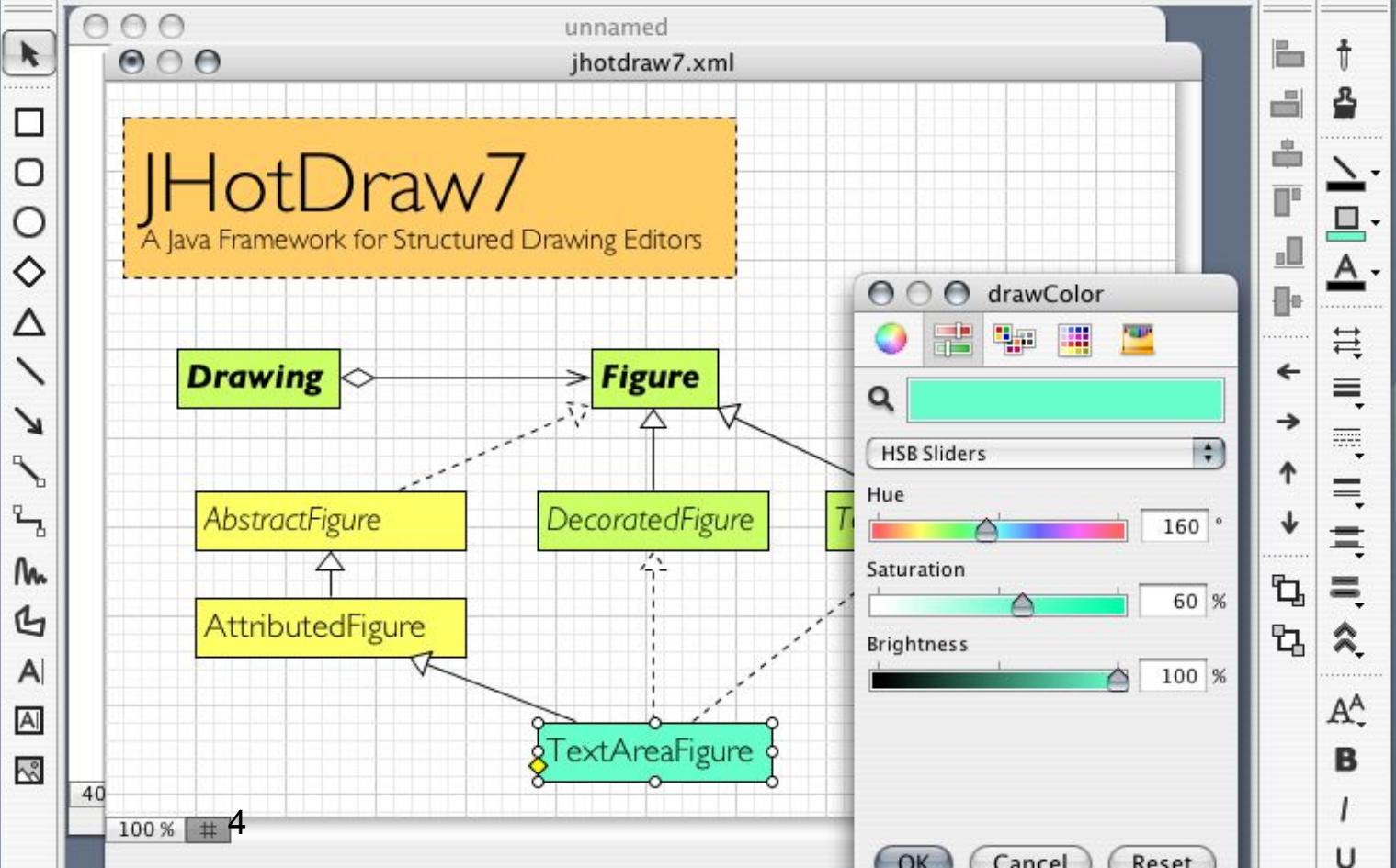
drawColor



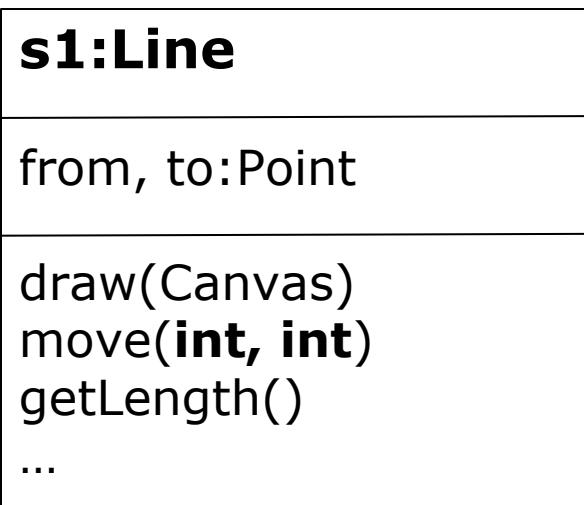
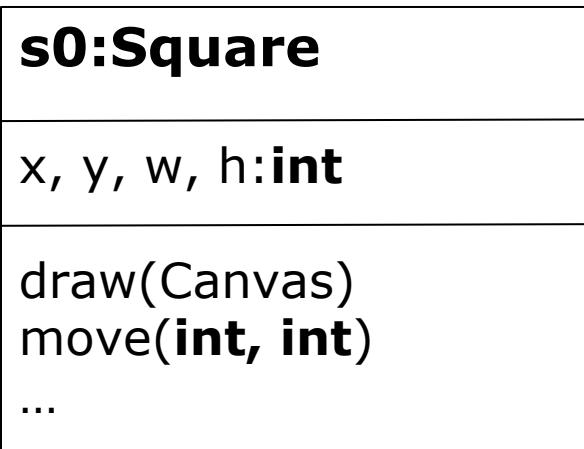
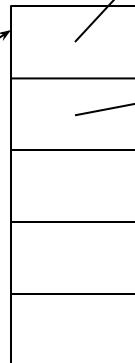
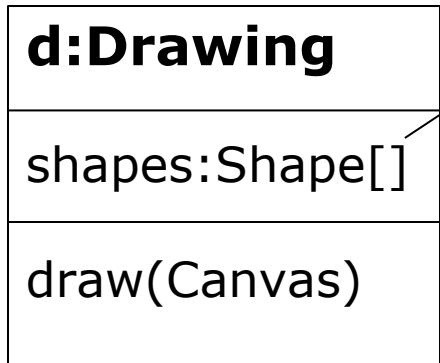
HSB Sliders



OK Cancel Reset



# Today: How Objects Respond to Messages



# Learning Goals

- Explain the need to design for change and design for division of labor
- Understand subtype polymorphism and dynamic dispatch
- Use encapsulation mechanisms
- Distinguish object methods from global procedures
- Start a program with entry code

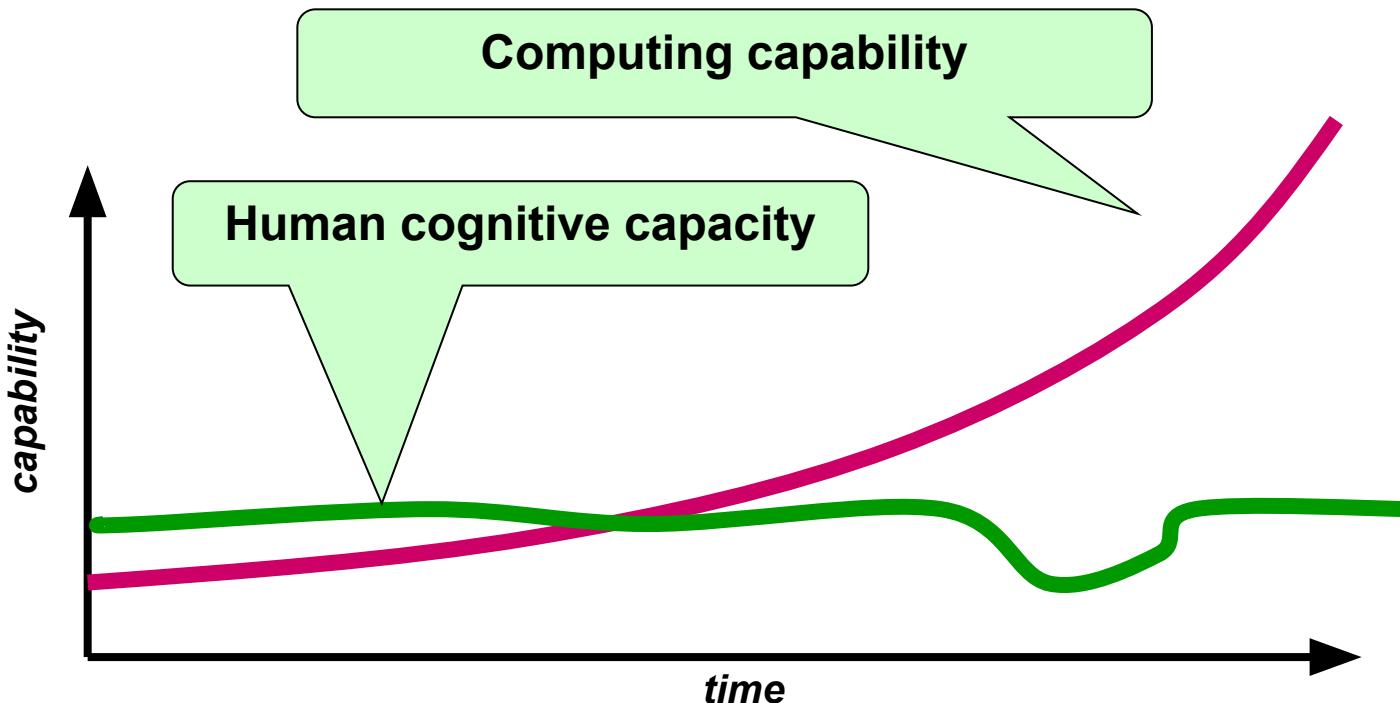
# Design Goals, Principles, and Patterns

- Design Goals
  - Design for Change
  - Design for Division of Labor
- Design Principles
  - Explicit Interfaces (clear boundaries)
  - Information Hiding (hide likely changes)
- Design Patterns
  - Strategy Design Pattern
  - Composite Design Pattern
- Supporting Language Features
  - Subtype Polymorphism
  - Encapsulation

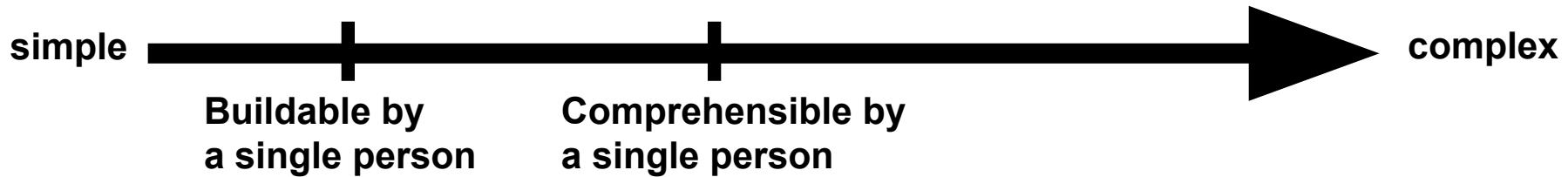
# Software Change

- ...accept the fact of change as a way of life, rather than an untoward and annoying exception.  
—Brooks, 1974
- Software that does not change becomes useless over time.  
—Belady and Lehman
- For successful software projects, most of the cost is spent evolving the system, not in initial development
  - Therefore, reducing the cost of change is one of the most important principles of software design

# The limits of exponentials



# Building Complex Systems



- Division of Labor
- Division of Knowledge and Design Effort
- Reuse of Existing Implementations

# Today: Key OOP Features that Support:

- **Design for Change** (flexibility, extensibility, modifiability)
- Design for Division of Labor
- Design for Understandability

# Programming without Objects

# Data structures and procedures

```
struct point {  
    int x;  
    int y;  
};  
  
void movePoint(struct point p, int deltax, int deltay) { p.x = ...; }  
  
int main() {  
    struct point p = { 1, 3 };  
    int deltaX = 5;  
    movePoint(p, 0, deltaX);  
    ...  
}
```

# Data structures and procedures

Data is stored in memory in a certain format

All data the same memory layout, procedures expect that layout

Each procedure is compiled to an address

Procedure invocations jump to that address

Single address for procedure

(Function pointers provide more flexibility)

# Objects

# Object (JavaScript)

A program abstraction with internal state (data) and behavior (actions, methods)

Interact through messages (invoking methods)

- perform an action, update state (e.g., move)
- request some information (e.g., getSize)

```
const obj = {  
    print: function() { console.log("foo"); }  
}  
  
obj.print()  
// foo
```

Functions in an object  
are typically called  
*methods*

This is a  
*method invocation*  
(conceptually by sending  
a message to the object)

# Objects can contain state

```
const obj = {  
    v: 1,  
    print: function() { console.log(this.v); },  
    inc: function() { this.v++; }  
}  
obj.print()  
// 1  
obj.print()  
// 1  
obj.inc()  
obj.print()  
// 2
```

The object contains a variable *v*, called a *field*, to store state

Multiple methods in the object

# Objects respond to messages, methods define interface

```
const obj = {  
    v: 1,  
    inc: function() { this.v++; },  
    get: function() { return this.v; },  
    add: function(y) { return this.v + y; }  
}  
  
obj.get() + 2  
// 3  
obj.add(obj.get()+2)  
// 4  
obj.send()  
// Uncaught TypeError: obj.send is not a function
```

Calling a method that does not exist results in an error

# Interface declared explicitly with TypeScript

```
interface Counter {  
    v: number;  
    inc(): void;  
    get(): number;  
    add(y: number): number;  
}  
  
const obj: Counter = {  
    v: 1,  
    inc: function() { this.v++; },  
    get: function() { return this.v; },  
    add: function(y) { return this.v + y; }  
}  
  
obj.foo();  
// Compile-time error: Property 'foo' does not exist
```

v must be part of the interface in TypeScript.  
Ways to avoid this later.

The object assigned to *obj* must have all the same methods as the interface.

# Interfaces and Objects in Java

```
interface Counter {  
    int get();  
    int add(int y);  
    void inc();  
}  
  
Counter obj = new Counter() {  
    int v = 1;  
    public int get() { return this.v; }  
    public int add(int y) { return this.v + y; }  
    public void inc() { this.v++; }  
};  
  
System.out.println(obj.add(obj.get()));  
// 2
```

This uses anonymous classes to create an object without a class.  
More later.

Object-oriented language feature enabling flexibility

# **SUBTYPE POLYMORPHISM ,**

## **DYNAMIC DISPATCH**

# Subtype Polymorphism / Dynamic Dispatch

- There may be multiple implementations of an interface
- Multiple implementations coexist in the same program
- May not even be distinguishable
- Every object has its own data and behavior, internals can be very different

# Programming against interfaces, not internals

```
interface Point {  
    int getX();  
    int getY();  
    void moveUp(int y);  
    Point copy();  
}
```

```
Point p = ...  
int x = p.getX();
```

```
interface IntSet {  
    boolean contains(  
        int element);  
    boolean isSubsetOf(  
        IntSet otherSet);  
}
```

```
IntSet a = ...; IntSet b = ...  
boolean s = a.isSubsetOf(b);
```

# Creating Objects

```
interface IntSet {  
    boolean contains(int element);  
    boolean isSubsetOf(IntSet otherSet);  
}  
  
IntSet emptySet = new IntSet() {  
    boolean contains(int element) { return false; }  
    boolean isSubsetOf(IntSet otherSet) { return true; }  
}
```

# Creating Objects

```
interface IntSet {  
    boolean contains(int element);  
    boolean isSubsetOf(IntSet otherSet);  
}  
  
IntSet threeSet = new IntSet() {  
    boolean contains(int element) {  
        return element == 3;  
    }  
    boolean isSubsetOf(IntSet otherSet) {  
        return otherSet.contains(3);  
    }  
}
```

# Classes as Object Templates

```
interface Point {  
    int getX();  
    int getY();  
}  
  
class CartesianPoint implements Point {  
    int x,y;  
    CartesianPoint(int x, int y) {this.x=x; this.y=y;}  
    int getX() { return this.x; }  
    int getY() { return this.y; }  
}  
  
Point p = new CartesianPoint(3, -10);
```

class as template for objects with Point interface

*Constructor* initializes the object

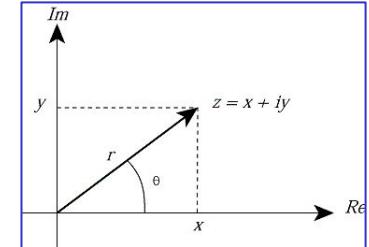
Calling *constructor* of class to create object

# Multiple Implementations of Interface

```
interface Point {  
    int getX();  
    int getY();  
}  
class SkewedPoint implements Point {  
    int x, y;  
    SkewedPoint(int x, int y) {  
        this.x = x + 10; this.y = y * 2; }  
    int getX() { return this.x - 10; }  
    int getY() { return this.y / 2; }  
}  
Point p = new SkewedPoint(3, -10);  
p.getX()
```

# Multiple Implementations of Interface

```
interface Point {  
    int getX();  
    int getY();  
}  
  
class PolarPoint implements Point {  
    double len, angle;  
    PolarPoint(double len, double angle)  
        {this.len=len; this.angle=angle;}  
    int getX() { return this.len * cos(this.angle);}  
    int getY() { return this.len * sin(this.angle); }  
    double getAngle() {...}  
}  
Point p = new PolarPoint(5, .245);
```



# Multiple Implementations of Interface

```
interface Point {  
    int getX();  
    int getY();  
}  
  
class MiddlePoint implements Point {  
    Point a, b;  
    MiddlePoint(Point a, Point b) { this.a = a; this.b = b; }  
    int getX() { return (this.a.getX() + this.b.getX()) / 2; }  
    int getY() { return (this.a.getY() + this.b.getY()) / 2; }  
}  
  
Point p = new MiddlePoint(new PolarPoint(5, .245),  
                         new CartesianPoint(3, 3));
```

Works with multiple implementations of Point

# Clients work with all implementations of Interface

```
interface Point {  
    int getX();  
    int getY();  
}  
r = new Rectangle() {  
    Point origin;  
    int width, height;  
    void draw() {  
        this.drawLine(this.origin.getX(), this.origin.getY(),  
                    this.origin.getX()+this.width, this.origin.getY());  
        ... // more lines here  
    }  
};
```

Works with all implementations of Point

# Subtype Polymorphism / Dynamic Dispatch

- There may be multiple implementations of an interface
- Multiple implementations coexist in the same program
- May not even be distinguishable
- Every object has its own data and behavior, internals can be very different

# Points and Rectangles: Interface

```
interface Point {  
    int getX();  
    int getY();  
}  
  
interface Rectangle {  
    Point getOrigin();  
    int getWidth();  
    int getHeight();  
    void draw();  
}
```

**What are possible implementations of the Rectangle interface?**

# Sets: Interface

```
interface IntSet {  
    boolean contains(int element);  
    boolean isSubsetOf(  
        IntSet otherSet);  
}
```

**What are possible implementations of the IntSet interface?**

# Java Twist: Classes implicitly have Interfaces

Classes can be used as types,  
like interfaces

All (public) methods can be  
called

No alternative implementations  
of class type

*Prefer interfaces over class  
types!*

```
class PolarPoint implements Point {  
    double len, angle;  
    ...  
    int getX() {...}  
    int getY() {...}  
    double getAngle() {...}  
}  
PolarPoint pp = new PolarPoint(5, .245);  
Point p = new PolarPoint(5, .245);  
pp.getAngle(); // okay  
p.getAngle(); // compilation error
```

# Programming against interfaces, not internals

```
interface Point {  
    int getX();  
    int getY();  
    void moveUp(int y);  
    Point copy();  
}
```

```
Point p = ...  
int x = p.getX();
```

```
interface IntSet {  
    boolean contains(  
        int element);  
    boolean isSubsetOf(  
        IntSet otherSet);  
}
```

```
IntSet a = ...; IntSet b = ...  
boolean s = a.isSubsetOf(b);
```

# JavaScript Twist: No Interfaces!

All methods of objects can be called

Objects with the same method can be called

No static checking by compiler; runtime error if method not exist

*TypeScript adds type system with interfaces*

```
const pp = {  
    len: 1, angle: 0,  
    getX: function() {...}  
    getAngle: function() {...}  
}  
  
const p = {  
    x: 1, y: 0;  
    getX: function() {...}  
}  
  
pp.getX(); p.getX(); // okay  
pp.getAngle(); // okay  
p.getAngle() // runtime error
```

# JavaScript and Classes

JavaScript traditionally had no classes; easy to create objects directly

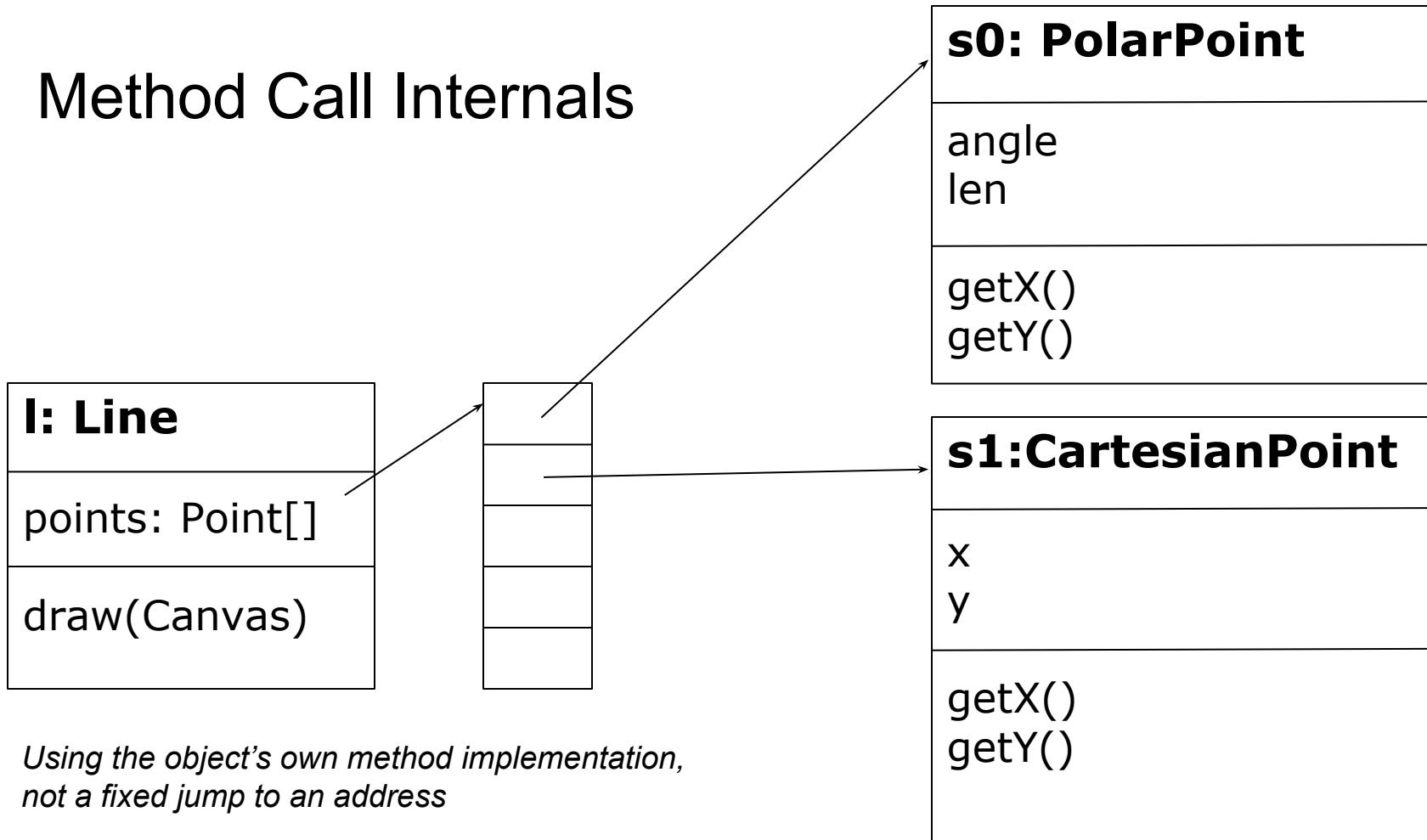
Classes introduced later in ECMAScript 2015 (ES6)

TypeScript supports classes and interfaces

Use somewhat controversial

```
class Point {  
    constructor(x, y) {  
        this.x = x; this.y = y;  
    }  
    getX() { return this.x; }  
    getY() { return this.y; }  
}  
  
const p = new Point(4, 5);  
p.getX();
```

# Method Call Internals



# Check your Understanding

```
interface Animal {  
    void makeSound();  
}  
  
class Dog implements Animal {  
    public void makeSound() { System.out.println("bark!"); } }  
  
class Cow implements Animal {  
    public void makeSound() { mew(); }  
    public void mew() {System.out.println("Mew!"); } }  
  
Animal x = new Animal() {  
    public void makeSound() { System.out.println("chirp!"); } }  
  
x.makeSound();  
Animal a = new Animal();  
a.makeSound();  
Animal d = new Dog();  
d.makeSound();  
Animal b = new Cow();  
b.makeSound();  
b.mew();
```

chirp

bark

mew

Dynamic Dispatch

# **Object Methods vs Global Functions/Procedures**

# Flexibility of dynamic dispatch (JavaScript)

Each object decides implementation,  
client does not care

Method is decided at runtime

Only single implementation of global function (and module)

```
// top-level function
function movePoint(p, x, y) { ... }

// create object, implementation unknown
const p = createPoint(...)

// call object's method
// object determines implementation
p.move(3, 5);

// single global implementation
// less flexibility
movePoint(p, 3, 5)
```

# Flexibility of dynamic dispatch (Java)

Each class decides implementation,  
client does not care

**Static** methods are *global functions*, only single copy exists;  
class provides only namespace

Java does not allow global functions outside of classes

```
interface Point {  
    void move(int x, int y) { ... }  
}  
class Helper {  
    static void movePoint(Point p,  
                          int x, int y) {...}  
}  
  
Point p = createPoint(...);  
// dynamic dispatch, object's method  
p.move(4, 5);  
  
// single global method, less flexible  
Helper.movePoint(p, 4, 5);
```

Dynamic Dispatch

# Benefits of Dynamic Dispatch

# Discussion Dynamic Dispatch

- A user of an object does not need to know the object's implementation, only its interface
- All objects implementing the interface can be used interchangeably
- Allows flexible change (modifications, extensions, reuse) later without changing the client implementation, even in unanticipated contexts

dynamic dispatch的目的是为了支持在编译期无法确定最终最合适的实现的操作.这种情况一般是因为在运行期才能通过一个或多个参数确定对象的类型.例如 B继承自A, 声明var obj : A = B(), 编译期会认为是A类型,但是真正的类型B,只能在运行期确定.

Design for Change!

◦

# Why multiple implementations?

Different performance

- Choose implementation that works best for your use

Different behavior

- Choose implementation that does what you want
- Behavior must comply with interface spec (“contract”)

Often performance and behavior both vary

- Provides a functionality – performance tradeoff
- Example: HashSet, TreeSet

```
interface Order {  
    boolean lessThan(int i, int j);  
}  
  
class AscendingOrder implements Order {  
    public boolean lessThan(int i, int j) { return i < j; }  
}  
class DescendingOrder implements Order {  
    public boolean lessThan(int i, int j) { return i > j; }  
}  
  
static void sort(int[] list, Order order) {  
    ...  
    boolean mustSwap =  
        order.lessThan(list[j], list[i]);  
    ...  
}
```

# Other Examples of Multiple Implementations

Change the sorting criteria in a list

Change the aggregation method for computations over a list (e.g., fold)

Compute the tax on a sale

Compute a discount on a sale

Change the layout of a form

# Historical note: simulation and the origins of OO programming

Simula 67 was the first object-oriented language

Developed by Kristin Nygaard and Ole-Johan Dahl  
at the Norwegian Computing Center

Developed to support discrete-event simulation

- Application: operations research, e.g. traffic analysis
- Extensibility was a key quality attribute for them
- Code reuse was another



Dahl and Nygaard at the time of Simula's development

Information Hiding

# Encapsulation

# Encapsulation / Information hiding

- Well designed objects project internals from others
  - both internal state and implementation details
- Well-designed code hides all implementation details
  - Cleanly separates interface from implementation
  - Modules communicate only through interfaces
  - They are oblivious to each others' inner workings
- Hidden details can be changed without changing client!
- Fundamental tenet of software design

# How to hide information?

```
class CartesianPoint {  
    int x,y;  
    Point(int x, int y) {  
        this.x=x;  
        this.y=y;  
    }  
    int getX() { return this.x; }  
    int getY() { return this.y; }  
    int helper_getAngle();  
}
```

```
const point = {  
    x: 1, y: 0,  
    getX: function() {...}  
    helper_getAngle:  
        function() {...}  
}
```

# get & set

## Java: Access modifier to hide private details

```
public class PolarPoint implements Point {  
    private double len, angle;  
    private int xcache = -1;  
    public PolarPoint(double len, double angle)  
        {this.len=len; this.angle=angle; computeX(); }  
    public int getX() { return xcache; }  
    public int getY() {...}  
    private int computeX() {  
        xcache = this.len * cos(this.angle);  
    }  
}  
PolarPoint p = new PolarPoint(5, .245);  
p.xcache // type error, trying to access private member  
p.computeX(); // type error, private method
```

# Benefits of information hiding

**Decouples** the objects that comprise a system: Allows them to be developed, tested, optimized, used, understood, and modified in isolation

**Speeds up** system development: Objects can be developed in parallel

**Eases maintenance burden**: Objects can be understood more quickly and debugged with little fear of harming other modules

**Enables effective performance tuning**: “Hot” classes can be optimized in isolation

**Increases software reuse**: Loosely-coupled classes often prove useful in other contexts

# Java: Information hiding with interfaces

```
public interface Point { ... }
private class PolarPoint implements Point {
    private double len, angle;
    public void computeX() { ... }
    public int getX() { return xcache; }
}
public class Factory {
    public Point createPoint(int x, int y) {
        return new PolarPoint(x, y);
    }
}
Point p = new Factory().createPoint((5, .245));
p.computeX(); // type error, method not in interface Point
```

# Information hiding with interfaces (Java)

Declare variables using interface types, not class types

- Client can use only interface methods
- Fields and implementation-specific methods not accessible from client code

Use **private** for fields and internal methods to restrict access also in class types; accessible only from within same class

Interface methods must be **public**.

Other modifiers **protected** (for inheritance, more later) and package

# JavaScript: Closures for Hiding

All methods and fields are public, no language constructs for access control (only recent TypeScript)

Encoding with closures

```
function createPolarPoint(len, angle) {  
    let xcache = -1;  
    let internalLen=len;  
    function computeX() {...}  
    return {  
        getX: function() {  
            computeX(); return xcache; },  
        getY: function() {  
            return len * sin(angle); }  
    };  
}  
const pp = createPolarPoint(1, 0);  
pp.getX(); // works  
pp.computeX(); // runtime error  
pp.xcache // undefined  
pp.len // undefined
```

# Closures

In nested functions/classes, inner functions/classes can access variables and arguments of outer functions

Frequently used in JavaScript

In Java: Closures for nested classes and lambda functions, but outer variables need to be final

```
function a(x) {  
    const z = 3;  
    function b(y) {  
        x++;  
        console.log(x+y+z);  
    }  
    b(5);  
    console.log(x);  
}  
a(3);  
// 12  
// 4
```

# JavaScript: Modules

Information hiding at the file level!

Decide what functions, variables, classes to keep private in a file

Traditionally, all code in one file; later multiple competing module systems

Standardized since ECMAScript 2015 (ES6)

import interfaces / functions from other modules

```
import { f, b }  
      from 'dir/file'  
import fs from 'fs'  
  
interface Point { ... }  
  
function createP(a, b) {...}  
  
function helper() { ... }  
  
export { Point, createP }
```

decide what functions / interfaces can be accessed from other modules

# Java: Packages and classes

Each class in file with same name; classes grouped in packages (directories)

Fully qualified name = Package + Class name (e.g. `java.lang.String`)

All public classes from all packages can be used

Imports simplify names

```
import me.util.PolarPoint; PolarPoint p = new PolarPoint(...);
```

instead of

```
me.util.PolarPoint p = new me.util.PolarPoint(...);
```

# Java 9: Modules

Advanced feature, discussed in later lecture

# Best practices for information hiding

- Carefully design your API
- Provide only functionality required by clients
  - All other members should be private / hidden through interfaces or closure
- You can always make a private member public later (or export an additional method) without breaking clients but not vice-versa!

# Starting a Program

# Starting a Program

Objects do not do anything on their own, wait for method calls

Every program needs a starting point or waits for events

```
// start with: node file.js
function createPrinter() {
    return {
        print: function() { console.log("hi"); }
    }
}
const printer = createPrinter();
printer.print()
// hi
```

Defining interfaces,  
functions, classes

Starting:  
Creating objects and  
calling methods

# Starting Java Code

All Java code is in classes, so how to create an object and call a method?

Special syntax for *main* method in class (`java X` calls *main* in *X*)

```
// start with: java Printer
class Printer {
    void print() {
        System.out.println("hi");
    }
    public static void main(String[] args) {
        Printer obj = new Printer();
        obj.print();
    }
}
```

Main method to be executed, here used to create object and invoke method

Static methods belong to class not the object, generally avoid them

# Summary

Need to divide work, divide and conquer

Objects encapsulate state and behavior

Static/global functions: Only a single function provided, less flexibility

Dynamic dispatch: Each object's own method is executed, multiple implementations possible

Encapsulation: Hide object internals behind interface

## Quick Survey: The Muddiest Point

Link also on Piazza under “Resources”

<https://forms.gle/W2YoKCvHE9ayqZb27>



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

## IDEs, Build system, Continuous Integration, Libraries

Christian Kästner

Vincent Hellendoorn



# Outline

- On Homework 1
- Abstraction, Reuse, and Programming Tools
- For each in {IDE, Build systems, libraries, CI}:
  - What is it today?
  - What is under the hood?
  - What is next?

# Homework 1

Welcome to the deep end!

- Java/TS + IDE + Maven/Npm + GitHub + Travis + linter!?

# Homework 1

Welcome to the deep end!

- Java/TS + IDE + Maven/Npm + GitHub + Travis + linter!?
- We're here to help:
  - Recitation tomorrow, walks through all this setup
  - Find some clarifications on Piazza
    - E.g., only implement what is asked for; all other functionality (repeating, flipping question/answer) is already there.
  - Use office hours (see course calendar)

# Homework 1

Welcome to the deep end!

- Java/TS + IDE + Maven/Npm + GitHub + Travis + linter!?
- We're here to help:
  - Recitation tomorrow, walks through all this setup
  - Find some clarifications on Piazza
    - E.g., only implement what is asked for; all other functionality (repeating, flipping question/answer) is already there.
  - Use office hours (see course calendar)
- Actual coding effort is small -- reading & setting up is the point
- Small typo detected on Piazza in `mostmistakes.ts`; fixed now. Not essential to your HW.

# Mini-quiz

<https://forms.gle/9tnB5BszVz9KTY7r5>



# Outline

- On Homework 1
- **Abstraction, Reuse, and Programming Tools**
- For each in {IDE, Build systems, libraries, CI}:
  - What is it today?
  - What is under the hood?
  - What is next?

# Automation Requires Abstraction



# Automation Requires Abstraction

The image shows a software interface with two main panes. The left pane is titled "C++ source #1" and contains the following C++ code:

```
1 // Type your code here, or load an example.
2 int square(int num) {
3     return num * num;
4 }
```

The right pane is titled "#1 with MSP430 gcc 4.5.3" and displays the generated assembly code:

```
11010 .LX0: .text // Intel A A A +
1
2 ****
3 * Function `square(int)'
4 ****
5 square(int):
6     push    r10
7     push    r4
8     mov     r1, r4
9     add    #4, r4
10    sub    #2, r1
11    mov     r15, -6(r4)
12    mov     -6(r4), r10
13    mov     -6(r4), r12
14    call   #__mulhi3
15    mov     r14, r15
16    add    #2, r1
17    pop    r4
18    pop    r10
19    ret
20;; End of function
```

The assembly code is color-coded by line number, showing different sections and labels.

# Automation Requires Abstraction

We all treat familiar levels of abstraction as normal/natural

- That's fine if you only drive your car
  - Not so much if you are a mechanic
  - How to debug a broken transmission?
- Also slow to evolve
  - *Conf.* people adamantly refusing to use an automatic
- Engineers seek out abstractions that simplify their work, help focus on the hard parts
  - They also know what is beneath the abstractions

# Automation Requires Abstraction

Today's "normal":

- Integrated-development environments (IDEs) galore
  - Web-based too! Press “.” on a GitHub (file) page 😱
- Frequent build, test, release
  - In some companies, every commit is a “release”
- Never write code for which there is a useful library
  - Define “useful” (we will)
- All of the above, entangled

# Outline

- On Homework 1
- Abstraction, Reuse, and Programming Tools
- For each in {IDE, Build systems, libraries, CI}:
  - **What is it today?**
  - What is under the hood?
  - What is next?

# Today's toolchain: a quick overview

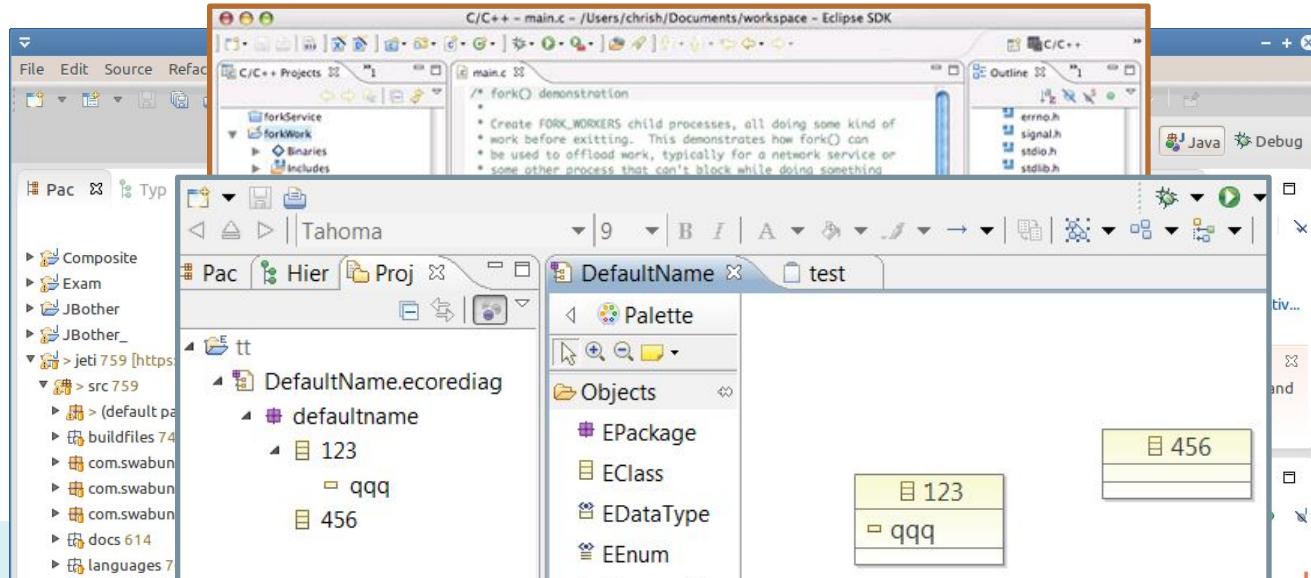
IDEs:

- Integrated Development Environments, bundle development workflows in a single UI
  - Editing, refactoring, running & debugging, adding dependencies, compiling, deploying, plugins, you name it
  - They often try to be everything, with mixed results
  - Leverage them to the fullest extent, to automate and check your work

# Today's toolchain: a quick overview

IDEs:

- Eclipse was the dominant player in Java for 20-odd years, owing to its powerful backbone and plugin architecture



# Today's toolchain: a quick overview

IDEs:

- Recently, IntelliJ has been more dominant
  - Packs a lot of “recipes” to create certain types of projects (e.g., web-app with Spring & Maven)

# Today's toolchain: a quick overview

IDEs:

- Recently, IntelliJ has been more dominant
  - Packs a lot of “recipes” to create certain types of projects (e.g., web-app with Spring & Maven)
- VSCode is surging in popularity
  - Local & web, lightweight but with a massive plugin ecosystem
    - Quick tangent: if you can build either a large product or a platform, build a platform

# Today's toolchain: a quick overview

IDEs:

- Recently, IntelliJ has been more dominant
  - Packs a lot of “recipes” to create certain types of projects (e.g., web-app with Spring & Maven)
- VSCode is surging in popularity
  - Local & web, lightweight but with a massive plugin ecosystem
    - Quick tangent: if you can build either a large product or a platform, build a platform
- But choose based on need!
  - You can relearn key-bindings; “killer features” are rare and temporary
  - E.g., Android: might want Android Studio (itself built on IntelliJ) since Google supports it
  - For this homework, choose what you’d like. We suggest IntelliJ for Java, VSCode for TS

# Today's toolchain: a quick overview

## Build Systems:

- How does this happen?

The image shows a software interface for developing and compiling C++ code for an MSP430 microcontroller. On the left, a window titled "C++ source #1" displays the following code:

```
1 // Type your code here, or load an example.
2 int square(int num) {
3     return num * num;
4 }
```

On the right, a window titled "#1 with MSP430 gcc 4.5.3" shows the generated assembly code:

```
1
2 ****
3 * Function 'square(int)'
4 ****
5 square(int):
6     push    r10
7     push    r4
8     mov     r1, r4
9     add     #4, r4
10    sub    #2, r1
11    mov     r15, -6(r4)
12    mov     -6(r4), r10
13    mov     -6(r4), r12
14    call    #__mulhi3
```

The assembly code is color-coded to highlight different instructions and labels.

# Today's toolchain: a quick overview

## Build Systems:

- Compiling is “easy” when all your source code is here
  - (Please don't tell a compiler expert I said that)
- Nowadays, your code is not “here”
  - Even libraries that you use in the IDE!
  - Interfaces make that possible

# Today's toolchain: a quick overview

## Build Systems:

- Compiling is “easy” when all your source code is here
  - (Please don’t tell a compiler expert I said that)
- Nowadays, your code is not “here”
  - Even libraries that you use in the IDE!
  - Interfaces make that possible
- Study the Travis log:
  - What is it doing?
  - Downloading, compiling, running checks
  - Most of this is “building”, using Maven
  - More on Travis later

```
[INFO] -----< org.example:FlashCards >-----  
210 [INFO] Building FlashCards 1.0-SNAPSHOT  
211 [INFO] -----[ jar ]-----  
220 [INFO] Downloading from google-maven-central: https://maven-central.storage-download.googleapis.com/maven2/org/apache/maven/plugins/maven-resources-plugin/2.6/maven-resources-plugin-2.6.pom  
221 [INFO] Downloaded from google-maven-central: https://maven-central.storage-download.googleapis.com/maven2/org/apache/maven/plugins/maven-resources-plugin/2.6/maven-resources-plugin-2.6.pom (8.1 kB at 30 kB/s)  
222 [INFO] Downloading from google-maven-central: https://maven-central.storage-download.googleapis.com/maven2/org/apache/maven/plugins/maven-plugins/23/maven-plugins-23.pom  
223 [INFO] Downloaded from google-maven-central: https://maven-central.storage-download.googleapis.com/maven2/org/apache/maven/plugins/maven-plugins/23/maven-plugins-23.pom (9.2 kB at 708 kB/s)  
224 [INFO] Downloading from google-maven-central: https://maven-central.storage-download.googleapis.com/maven2/org/apache/maven/maven-parent/22/maven-parent-22.pom  
225 [INFO] Downloaded from google-maven-central: https://maven-central.storage-download.googleapis.com/maven2/org/apache/maven/maven-parent/22/maven-parent-22.pom (30 kB at 1.5 MB/s)  
226 [INFO] Downloading from google-maven-central: https://maven-central.storage-download.googleapis.com/maven2/org/apache/apache/11/apache-11.pom  
227 [INFO] Downloaded from google-maven-central: https://maven-central.storage-download.googleapis.com/maven2/org/apache/apache/11/apache-11.pom (15 kB at 1.5 kB/s)  
228 [INFO] Downloading from google-maven-central: https://maven-central.storage-download.googleapis.com/maven2/org/apache/maven/plugins/maven-resources-plugin/2.6/maven-resources-plugin-2.6.jar  
229 [INFO] Downloaded from google-maven-central: https://maven-central.storage-download.googleapis.com/maven2/org/apache/maven/plugins/maven-resources-plugin/2.6/maven-resources-plugin-2.6.jar (30 kB at 1.6 MB/s)  
230 [INFO] Downloading from google-maven-central: https://maven-central.storage-download.googleapis.com/maven2/org/apache/maven/plugins/maven-compiler-plugin/3.1/maven-compiler-plugin-3.1.pom  
231 [INFO] Downloaded from google-maven-central: https://maven-central.storage-download.googleapis.com/maven2/org/apache/maven/plugins/maven-compiler-plugin/3.1/maven-compiler-plugin-3.1.pom (10 kB at 928 kB/s)  
232 [INFO] Downloading from google-maven-central: https://maven-central.storage-download.googleapis.com/maven2/org/apache/maven/plugins/maven-plugins/24/maven-plugins-24.pom  
233 [INFO] Downloaded from google-maven-central: https://maven-central.storage-download.googleapis.com/maven2/org/apache/maven/plugins/maven-plugins/24/maven-plugins-24.pom (11 kB at 982 kB/s)  
234 [INFO] Downloading from google-maven-central: https://maven-central.storage-download.googleapis.com/maven2/org/apache/maven/maven-parent/23/maven-parent-23.pom  
235 [INFO] Downloaded from google-maven-central: https://maven-central.storage-download.googleapis.com/maven2/org/apache/maven/maven-parent/23/maven-parent-23.pom (35 kB at 1.4 MB/s)  
236 [INFO] Downloading from google-maven-central: https://maven-central.storage-download.googleapis.com/maven2/org/apache/apache/13/apache-13.pom  
237 [INFO] Downloaded from google-maven-central: https://maven-central.storage-download.googleapis.com/maven2/org/apache/apache/13/apache-13.pom (14 kB at 1 kB/s)  
238 [INFO] Downloading from google-maven-central: https://maven-central.storage-download.googleapis.com/maven2/org/apache/maven/plugins/maven-compiler-plugin/3.8/maven-compiler-plugin-3.8.pom
```

# Today's toolchain: a quick overview

## Build Systems:

- Has a few basic tasks:
  - Compiling & linking, to produce an executable
  - Creating secondary *artifacts*, e.g. documentation-pages, linter reports, test suite reports
  - Different levels of “depth” may be appropriate, for large code bases (e.g. Google)

# Today's toolchain: a quick overview

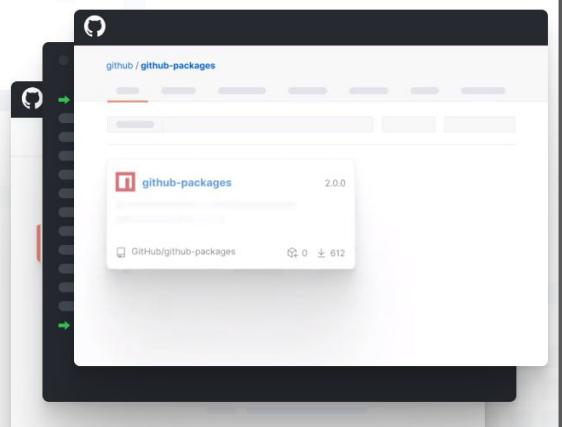
## Build Systems:

- Has a few basic tasks:
  - Compiling & linking, to produce an executable
  - Creating secondary *artifacts*, e.g. documentation-pages, linter reports, test suite reports
  - Different levels of “depth” may be appropriate, for large code bases (e.g. Google)
- Popular options:
  - For Java: Maven and Gradle -- historically Ant.
    - You could do any homework in either; we’re not attached to one
  - For JS/TS: Node(JS)
    - Generally coupled with the Node Package Manager (NPM)
  - Often built into IDEs, as plugins

# Today's toolchain: a quick overview

## Libraries:

- Myriad. Publicly hosted on various *package managers*
  - Often tied, but not inextricably linked, to build tools, and languages
  - Maven/Gradle for Java, NPM for JS/TS, Nuget for C#, ...
  - Registries of managers, e.g., GitHub Packages



The image shows a screenshot of the GitHub Packages interface. On the left, there is a sidebar with a dark theme. In the center, a modal window is open, also with a dark theme. The title of the modal is "github / github-packages". Inside the modal, there is a single item listed: "github-packages" by "GitHub/github-packages" with a version of "2.0.0". Below the modal, the main GitHub interface is visible, showing a repository named "github-packages". At the bottom of the screen, there is a navigation bar with icons for "Home", "Search", "Explore", "Notifications", and "Profile".

**Easy publishing**

Use industry and community-standard package managers with native tooling commands. Then authenticate and publish directly to GitHub.

npm docker nuget  
Maven Gradle

# Today's toolchain: a quick overview

## Libraries:

- Myriad. Publicly hosted on various *managers*
  - Often tied, but not inextricably linked, to build tools, and languages
  - Maven, Gradle, NPM, Nuget, Docker, ...
  - Registries of managers, e.g., GitHub Packages
- Releases are generally fast-paced or frigid
  - Almost all volunteer-based, so support waivers, as does documentation quality
  - Often open-source, so you can check out the status & details on GitHub
  - Beware of vulnerabilities and bugs, esp. with minor-releases and nightly's, old packages

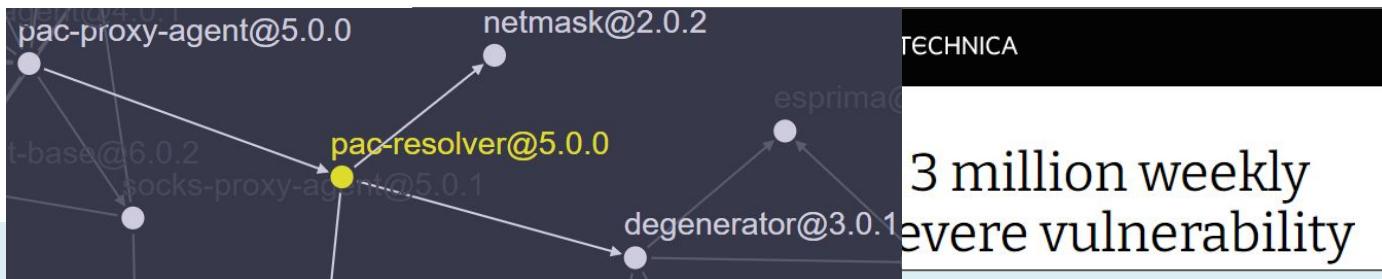


NPM package with 3 million weekly  
downloads had a severe vulnerability

# Today's toolchain: a quick overview

## Libraries:

- A Case-Study:
  - ‘pac-resolver’ (3M weekly downloads) has a major security vulnerability
    - Uses ‘degenerator’ (same author), which misuses a Node module
    - “The vm module is not a security mechanism. Do not use it to run untrusted code.”
    - (a mistake that’s been made before: people rarely read disclaimers)
  - ‘pac-proxy-agent’ (2M weekly downloads, same author) uses the above
    - Is widely popular, the main reason people use ‘degenerator’
    - Most people using this package have never heard of the latter -- many never will



# Today's toolchain: a quick overview

## Continuous Integration:

- Automates standard build, test, deploy pipelines
  - Technically, the latter is “CD”
  - Typically builds from scratch in a clean *container*
  - Often tied to code-review; triggers on new commits, pull requests
    - Ideally, official releases pass the build
  - Produces (long) logs with debugging outputs

# Today's toolchain: a quick overview

Not mentioned:

- Docker: containerize applications for coarse-grained reuse
- Cloud: deploy and scale rapidly, release seamlessly
- Bug/Issue trackers, often integrated with reviews

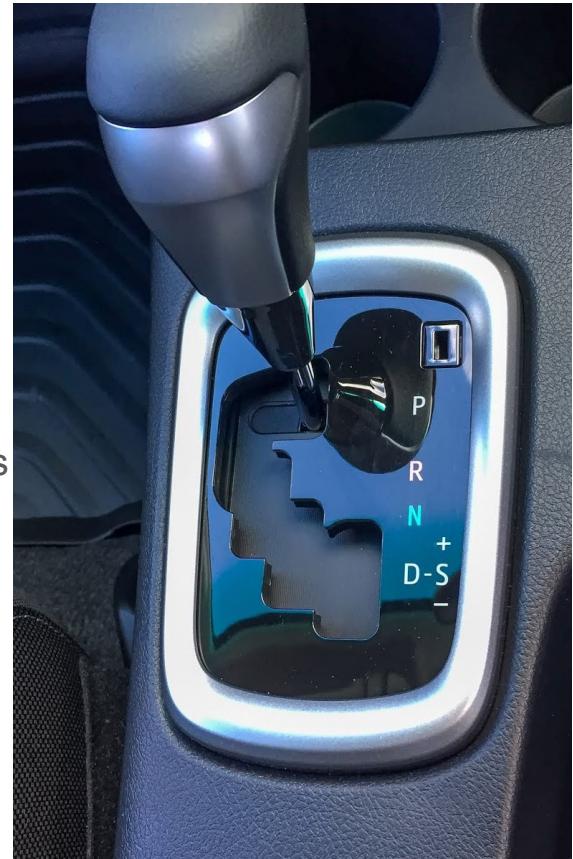
# Outline

- On Homework 1
- **Abstraction, Reuse, and Programming Tools**
- For each in {IDE, Build systems, libraries, CI}:
  - What is it today?
  - What is under the hood?
  - What is next?

# Behind the Abstraction

First, a bit of nuance:

- Automation vs. Reuse
  - We tend to automate common chains of actions
    - Gear-up := {Press clutch, switch gear, release clutch while accelerating}
  - To facilitate reusing such “subroutines”, we introduce abstractions
    - Accelerate in ‘D’ => Gear-up when needed



# Behind the Abstraction

First, a bit of nuance:

- Automation vs. Reuse
  - We tend to automate common chains of actions
    - Gear-up := {Press clutch, switch gear, release clutch while accelerating}
  - To facilitate reusing such “subroutines”, we introduce abstractions
    - Accelerate in ‘D’ => Gear-up when needed
- Reuse vs. Interfaces
  - Interfaces facilitate reuse through abstraction
    - Allow upgrading implementation without breaking things
    - Provide explicit & transparent contract



# Behind the Abstraction

First, a bit of nuance:

- Most tools are abstractions of common commands
  - Typically operated via GUI and/or a DSL
  - Obvious for Travis: just read the Yaml
    - Script-like languages are common
    - Involving a vocabulary of “targets”
    - E.g., `mvn site`



The screenshot shows a GitHub Classroom project titled "hw1-flash-cards-VHellendoorn / .travis.yml". The project is set to Java. It has one contributor, "github-classroom", who has initialized the Java environment. The repository contains 4 lines (4 sloc) of 72 Bytes of code:

```
language: java
jdk: oraclejdk16
script:
  - timeout 5m mvn site
```

# Behind the Abstraction

First, a bit of nuance:

- Most tools are abstractions of common commands
  - Typically operated via GUI and/or a DSL
  - Obvious for Travis: just read the Yaml
    - Script-like languages are common
    - Involving a vocabulary of “targets”
    - E.g., `mvn site`
- Abstraction can also “trap” us
  - When/how do we leave the abstraction?
  - Command-line comes built into IDEs for a reason
  - Non-trivial in general! May require switching/“patching” libraries
    - E.g., Maven → Gradle for more unusual build routines

# Outline

- On Homework 1
- Abstraction, Reuse, and Programming Tools
- For each in {IDE, Build systems, libraries, CI}:
  - What is it today?
  - **What is under the hood?**
  - What is next?

# IDEs

Automate common programming actions:

- Handy refactorings, suggestions
  - E.g., just press `alt+enter` in IntelliJ while highlighting nearly any code
    - Keyboard shortcuts are super useful: explore your IDE!
  - These can make you a better programmer: encode a lot of best-practices
    - Though, don't read into them too much

The screenshot shows a Java code editor in IntelliJ IDEA. A tooltip is displayed over the line of code: `throw new UnsupportedOperationException();`. The tooltip has a blue header with the text "Add runtime exception(s) to method signature" and a grey body with the text "Press Ctrl+Shift+I to open preview". The code editor shows a class named `Main` with a constructor that throws an `UnsupportedOperationException`.

```
public final class Main {  
  
    private Main() {  
        // Disable instantiating this class.  
        throw new UnsupportedOperationException();  
    }  
  
    public static void main(String[] args) throws IOException {  
        // TODO: set up options, extract command line arguments, fill in the relevant objects  
        CardStore cards = new CardLoader().loadCardsFromFile(new File("cards/designs"));  
    }  
}
```

# IDEs

- The engine: continuous parsing, building
  - Key feature: most partial programs don't parse, but IDEs make sense of them
  - That allows quickly relaying compile warnings/errors and useful suggestions
  - Same with API resolution
- Powered by rapid incremental compilation
  - Only build what has been updated
    - Virtually every edit you make triggers a compilation, re-linking
    - Of just the changed code and its dependencies
  - Works because *very little* of the code changes most of the time
    - But no free lunch: tends to drop optimizations (mostly fine), may struggle with big projects
  - Just try it: call an API with the wrong parameters & see how fast it triggers an alert; contrast with running a full Maven build (e.g., with `mvn install`)

# IDEs

Automate common programming actions:

- Debugging
  - Often the default mode when you run in the IDE

The screenshot shows a debugger interface with the following details:

- File:** translate.py > ...
- Line 4:** en, de = l.split(' - ')
- Exception:** Exception has occurred: ValueError
- Description:** not enough values to unpack (expected 2, got 1)
- Call Stack:** File translate.py", line 5, in <module>
- Line 5:** index[en] = de

# IDEs

Automate common programming actions:

- Debugging
  - Often the default mode when you run in the IDE
  - Allows setting breakpoints
    - Which give you rich insight into execution

The screenshot shows a Python IDE interface. The top bar includes buttons for 'RUN ...', 'No Configuration', settings, and tabs for 'en\_de.txt' and 'translate.py'. The main area displays the code for 'translate.py':

```
1 index = {}
2 with open('en_de.txt') as f:
3     for l in f:
4         en, de = l.split(' - ')
5         index[en] = de
6
```

To the left, a 'VARIABLES' panel shows the state of variables:

- Locals**:
  - > special variables
  - > f: <\_io.TextIOWrapper name='en\_de.txt' encoding='utf-8'>
  - > index: {}
  - l: 'I -- ich\n'
- Globals**: (empty)

# IDEs

Automate common programming actions:

- Debugging
  - Often the default mode when you run in the IDE
  - Allows setting breakpoints
    - Which give you rich insight into execution

The screenshot shows a Python debugger interface. The top bar includes buttons for RUN, configuration, settings, and tabs for 'en\_de.txt' and 'translate.py'. The main area has a toolbar with icons for play, step, and refresh. On the left, a 'VARIABLES' sidebar shows 'Locals' with variables like 'de' ('ich\n'), 'en' ('I'), and 'index' ({'I': 'ich\n'}). The right side is a code editor for 'translate.py' with the following code:

```
1 index = {}
2 with open('en_de.txt') as f:
3     for l in f:
4         en, de = l.split(' -- ')
5         index[en] = de
6 
```

A red dot at line 4 indicates a breakpoint. The line 'for l in f:' is highlighted with a yellow background.

# IDEs

- IDE designers spend a lot of time automating common development tasks
  - Sometimes they get a little too helpful (modifying pom's)
  - Many plugins provide customized experience
  - Mostly evolve with new tools, prioritizing emerging routines
- Useful to know how these actions work
  - Often not much more than invoking commands for you
    - VSCode, IntelliJ are very explicit about this in the terminal -- great for customization

```
"C:\Program Files\Java\jdk-16.0.1\bin\java.exe" -ea -Didea.test.cyclic.buffer.size=1048576 "-javaagent:C:\Program Files\JetBrains\IntelliJ ID  
Process finished with exit code 0
```

# Build Systems

- These days: intricately tied with IDEs, package managers
- Projects often come with a build config file or two
  - ‘pom.xml’ for Maven
  - ‘tsconfig.json’ + ‘package.json’ for TypeScript+NPM -- the second deals with packages
  - These can be nested, one per (sub-)directory, to compose larger systems
    - On GitHub, you can create links across repositories

# Build Systems

- These days: intricately tied with IDEs, package managers
- Projects often come with a build config file or two
  - ‘pom.xml’ for Maven
  - ‘tsconfig.json’ + ‘package.json’ for TypeScript+NPM -- the second deals with packages
  - These can be nested, one per (sub-)directory, to compose larger systems
    - On GitHub, you can create links across repositories
  - Specifies:
    - Compilation source and target version
    - High-level configuration options
    - Targets for various phases in development
      - “lifecycle” in Maven; e.g. ‘compile’, ‘test’, ‘deploy’
    - Often involving plugins
    - Dependencies with versions
      - Not shown: in package.json

```
1  {  
2      "compilerOptions": {  
3          "target": "es2016",  
4          "module": "commonjs",  
5          "sourceMap": true,  
6          "strict": true,  
7          "esModuleInterop": true,  
8          "moduleResolution": "node",  
9          "outDir": "dist"  
10     }  
11 }
```

# Libraries & Frameworks

Packages can be either:

- Libraries:
  - A set of classes and methods that provide reusable functionality
  - Typically: programmer calls, library returns data, that's it.

# Libraries & Frameworks

Packages can be either:

- Libraries:
  - A set of classes and methods that provide reusable functionality
  - Typically: programmer calls, library returns data, that's it.
- Frameworks:
  - Reusable skeleton code that can be customized into an application
  - Framework calls back into client code
    - The Hollywood principle: “Don’t call us. We’ll call you.”
  - E.g., Android development: you declare your UI elements, activities to be composed
  - Principle: inversion of control

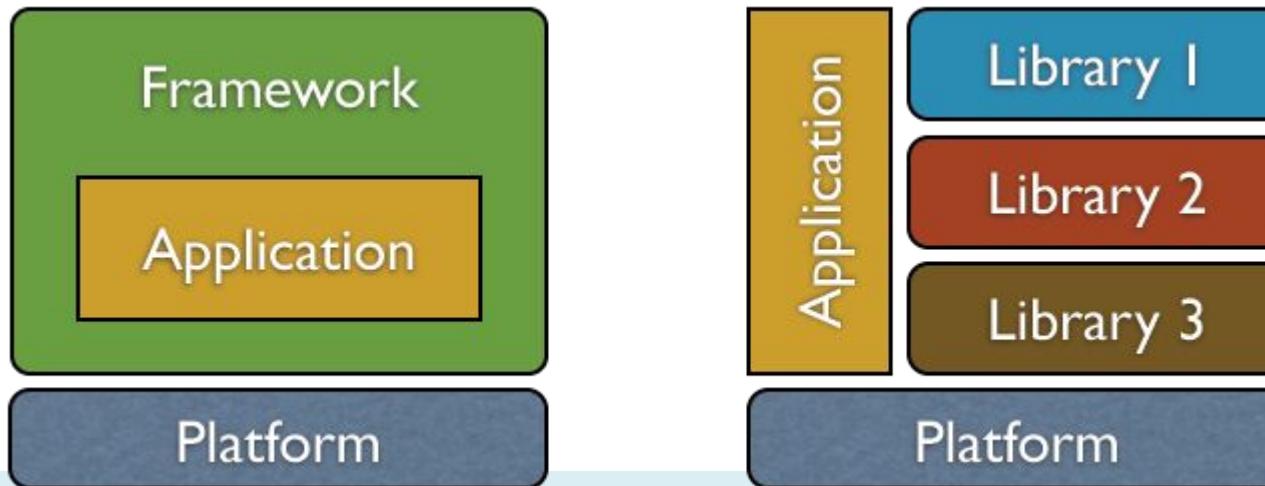
# Libraries & Frameworks

Packages can be either:

- Libraries:
  - A set of classes and methods that provide reusable functionality
  - Typically: programmer calls, library returns data, that's it.
- Frameworks:
  - Reusable skeleton code that can be customized into an application
  - Framework calls back into client code
    - The Hollywood principle: “Don’t call us. We’ll call you.”
  - E.g., Android development: you declare your UI elements, activities to be composed
  - Principle: inversion of control
- You typically use zero/one framework and many libraries
  - Frameworks might be especially constraining, but for good reason.
  - Some tools are a bit of both, and not all frameworks quite invert control

# Libraries & Frameworks

Which kind is a command-line parsing package?

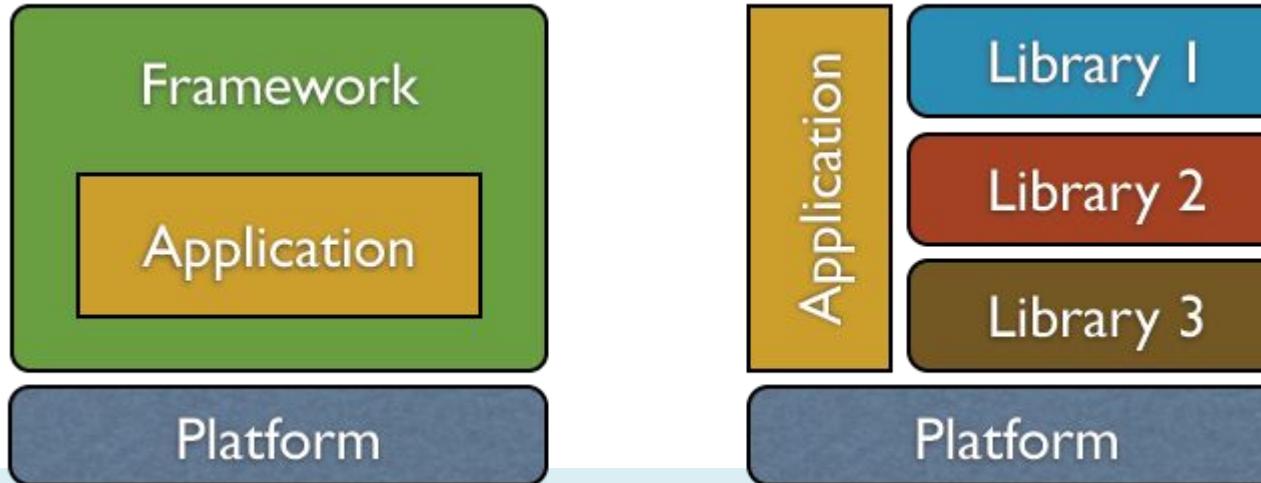


# Libraries & Frameworks

Which kind is a command-line parsing package?

How about a tool that runs tests based on annotations you add in your code?

- More on Thursday



# Libraries

Look into:

- Stated Goal:
  - A simple interface (“get started in one line!”) also means lots of abstraction
  - That’s neither good nor bad; know what you need
  - Docs with “advanced use cases” are always neat
- Maintenance:
  - Active release cycle, recent updates to documentation
  - GitHub build status, issue tracker (filled with unmerged ‘dependabot’ PRs?)
  - Lots of companies deliberately lag by one minor (or even major) version
- Recursive dependencies
  - Myriad, beyond inspection. Using OSS in corporate environments is a headache

# Frameworks

Whitebox:

- Extension via subclassing and overriding methods
- Common design pattern(s):
  - Template method
- Subclass has main method but gives control to framework

Blackbox:

- Extension via implementing a plugin interface
- Common design pattern(s):
  - Command
  - Observer
- Plugin-loading mechanism loads plugins and gives control to the framework

# Continuous Integration

Defines a series of actions to be run in a clean build:

- Actions start from the very top:
  - Clone repository, checkout branch
  - Download & install Java/Node
  - Invoke commands with timeouts
- Travis allocates a new (Docker) container for each build
  - Think of this like a fresh, temporary computer
  - Usually with a few default libraries present (i.e., based on an *image*)
- That means: **fully replicable builds**

```
▶ 163 Installing SSH key from: default repository key
164 Using /home/travis/.netrc to clone repository.
165
166
▼ 167 $ git clone --depth=50 --branch=TypeScript https://github.com/CMU-17-214/template-21f-hw1.git CMU-17-214/template-21f-hw1
168 Cloning into 'CMU-17-214/template-21f-hw1'...
169 remote: Enumerating objects: 117, done.
170 remote: Counting objects: 100% (117/117), done.
171 remote: Compressing objects: 100% (73/73), done.
172 remote: Total 117 (delta 50), reused 104 (delta 37), pack-reused 0
173 Receiving objects: 100% (117/117), 69.89 KiB | 2.25 MiB/s, done.
174 Resolving deltas: 100% (50/50), done.
175 $ cd CMU-17-214/template-21f-hw1
176 $ git checkout -qf 0d657225c8cbdd52751c2f88527f93f4099b041e
177
178
▼ 179 $ nvm install 16
180 Downloading and installing node v16.8.0...
181 Downloading https://nodejs.org/dist/v16.8.0/node-v16.8.0-linux-x64.tar.xz...
182 Computing checksum with sha256sum
183 Checksums matched!
184 Now using node v16.8.0 (npm v7.21.0)
185
▶ 186 Setting up build cache
187
188
▶ 189 $ npm ci
190
191
192
▶ 193 $ node --version
193 v16.8.0
194 $ npm --version
195 7.21.0
196 $ nvm --version
197 0.38.0
198
199
200
201
▶ 202 $ npm ci
202
203
204
205
206
207
208
209
210
211 $ timeout 5m npm run compile
212
213 > hw1-flashcards@1.0.0 compile
214 > tsc
215
```

# Continuous integration – Travis CI

Automatically builds, tests,  
and displays the result

The screenshot shows the Travis CI web interface for the repository `wyvernlang/wyvern`. The build number is `#17`, which is marked as `passing`. The build duration was 16 seconds, and it finished 3 days ago. The commit author is `potanin`. A note at the bottom indicates that the job ran on legacy infrastructure and suggests upgrading. The build log shows standard setup steps like cloning the repository and switching Java versions.

```
1 Using worker: worker-linux-027f0490-1.bb.travis-ci.org:travis-linux-2
2
3 Build system information
67
68 $ git clone --depth=50 --branch=SimpleWyvern-devel
69 $ jdk_switcher use oraclejdk8
70 Switching to Oracle JDK8 (java-8-oracle), JAVA_HOME will be set to /usr/lib/jvm/java-8-oracle
71 $ java -Xmx32m -version
72 java version "1.8.0_31"
73 Java(TM) SE Runtime Environment (build 1.8.0_31-b13)
74 Java HotSpot(TM) 64-Bit Server VM (build 25.31-b07, mixed mode)
75 $ javac -J-Xmx32m -version
76 javac 1.8.0_31
77 $ cd tools
78
79 The command "cd tools" exited with 0.
80 $ ant test
81
82 BUILD SUCCESSFUL
83 Total time: 0.815 seconds
84
```

# Continuous integration – Travis CI

You can see the results of builds over time

The screenshot shows a web browser window displaying the Travis CI build history for the repository `wyvernlang/wyvern`. The URL in the address bar is `https://travis-ci.org/wyvernlang/wyvern/builds`. The page header includes the Travis CI logo, navigation links for Blog, Status, and Help, and a user profile for Jonathan Aldrich.

The main content area is titled "wyvernlang / wyvern" and shows the "Build History" tab selected. A search bar at the top allows searching for all repositories. Below it, a section titled "My Repositories" lists the repository `wyvernlang/wyvern`, which has 17 builds. The details for this repository show a green checkmark icon, the repository name, the build number (# 17), the duration (16 sec), and the completion time (3 days ago).

The "Build History" table lists ten builds, each with a status indicator (green checkmark for passed, red X for failed), the commit message, the number of passes, the duration, and the timestamp. The builds are ordered by their creation date, from most recent at the top to oldest at the bottom. The first nine builds are green, indicating they passed, while the last one is red, indicating it failed.

Build	Commit Message	Passes	Duration	Time Ago
# 17 passed	asserting false (works on L)	17	16 sec	3 days ago
# 16 passed	Debugging mac bug.	16	22 sec	3 days ago
# 14 passed	Zooming in on Mac's IRBuilder	14	15 sec	4 days ago
# 13 passed	Zooming in on Mac LLVM build	13	16 sec	4 days ago
# 7 passed	Removed outdated tests	7	15 sec	11 days ago
# 6 passed	Merge branch 'master' of https://github.com/wyvernlang/wyvern	6	14 sec	11 days ago
# 5 passed	Build with JDK 8	5	13 sec	11 days ago
# 4 failed	fixed Travis build script syntax error	4	5 sec	11 days ago
# 3 passed	Initial commit	3	16 sec	11 days ago

# Outline

- On Homework 1
- Abstraction, Reuse, and Programming Tools
- For each in {IDE, Build systems, libraries, CI}:
  - What is it today?
  - What is under the hood?
  - **What is next?**

# What's Next

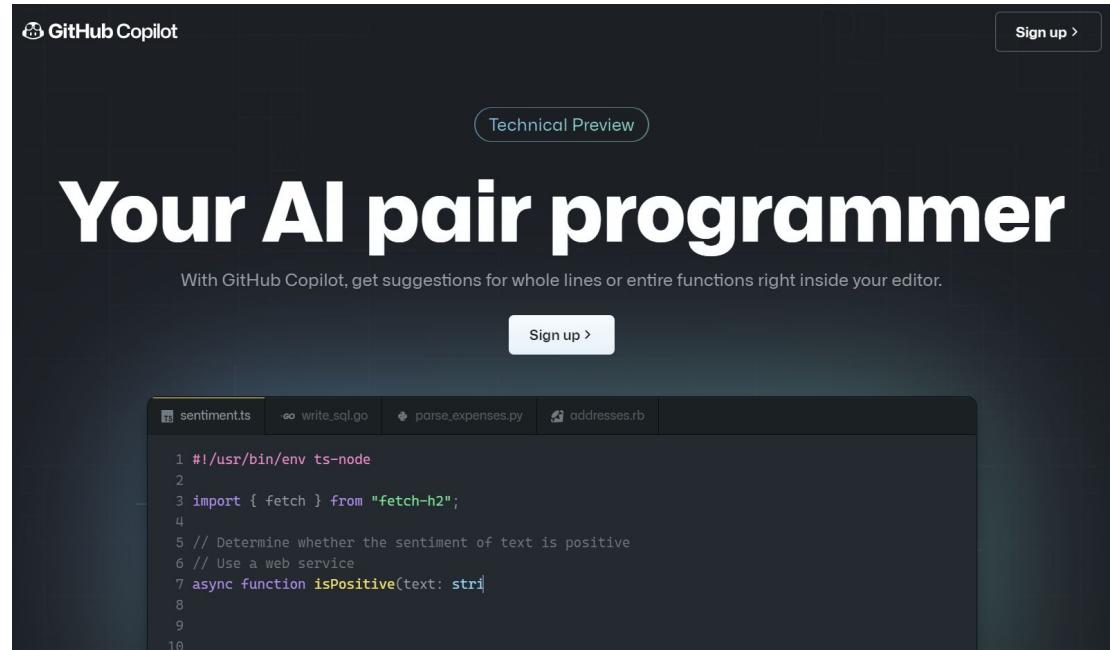
Anyone care to guess?

- Can be based on something you've seen, but think will boom

# What's Next

## AI Powered Programming

- Easier in Web IDEs
  - Which are themselves “next”



# What's Next

## Collaborative online coding

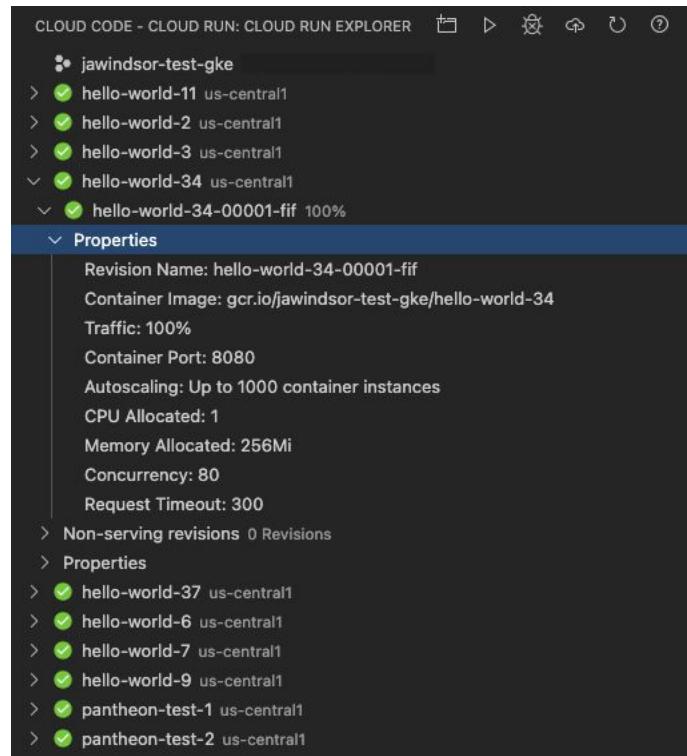
- Think: Google Docs for code
- E.g. VS Life Share
- How will this change “commits”?



# What's Next

## Tighter IDE-to-cloud integration

- Google Cloud is pushing on this with VSCode
- We will (lightly) touch on Containers & Clouds in this course



# Summary

- Programming Tools are abundant, and rapidly evolving
  - Learn multiple; you will have to inevitably
- They rely on abstractions through interfaces to facilitate reuse
  - Which come in many shapes: GUI, API, DSL
  - And can be a limitation -- choose wisely
- Your HW1 toolchain sets you up for all homeworks
  - With modest variations (frameworks, new build targets)
  - Self-discovery is a big asset
  - Tomorrow's recitation offers help

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

## Specifications and unit testing, exceptions

Christian Kästner

Vincent Hellendoorn



# Explicit over Implicit

Can anything go wrong with this?

```
int add(int a, int b) {  
    return a + b;  
}
```

# Explicit over Implicit

Can anything go wrong with this?

```
int add(int a, int b) {  
    return a + b;  
}
```

How about this:

```
int divide(int a, int b) {  
    return a / b;  
}
```

# Explicit over Implicit

Can anything go wrong with this?

```
int add(int a, int b) {  
    return a + b;  
}
```

How about this:

```
int divide(int a, int b) {  
    return a / b;  
}  
divide(4, 3); // 1
```

# Explicit over Implicit

Can anything go wrong with this?

```
int add(int a, int b) {  
    return a + b;  
}
```

How about this:

```
int divide(int a, int b) {  
    return a / b;  
}  
divide(4, 3); // 1  
divide(2, 0); // Exception  
java.lang.ArithmetricException: / by zero
```

# Explicit over Implicit

BTW, harder to force in TS\*:

```
function divide(a: bigint, b: bigint): bigint {  
    return a / b;  
}  
divide(4n, 3n); // 1  
divide(2n, 0n); // RangeError: Division by zero
```

\*Compile with: --target es2020

# Explicit over Implicit

Most real-world code has a **contract**.

- It might not be obvious to you!
- This is why we:
  - Encode specifications
  - Test
  - Use exceptions
- Imperative to build systems that scale

# Today

1. Exception Handling
2. Unit Testing
3. Specifications

# Exceptions

- Inform caller of problem by transfer of control
  - They split control-flow into a “normal” and an “erroneous” branch
  - Compare “if/else”
- Semantics
  - Propagates up the call stack until exception is caught, or main method is reached
    - So, it can terminate the program!
- Where do exceptions come from?

# Exceptions

Just try:

```
String read(String path) {  
    return Files.lines(Path.of(path))  
        .collect(Collectors.joining("\n"));  
}
```

# Handling Exceptions

```
String read(String path) {  
    try {  
        return Files.lines(Path.of(path))  
            .collect(Collectors.joining("\n"));  
    }  
    catch (IOException e) {  
        // implement fall-back behavior.  
    }  
}
```

# Handling Exceptions

```
String read(String path) throws IOException {  
    return Files.lines(Path.of(path))  
        .collect(Collectors.joining("\n"));  
}
```

# Benefits of exceptions

- You can't forget to handle common failure modes
  - Explicit > implicit
  - Compare: using a flag or special return value
- Provide high-level summary of error
  - Compare: core dump in C/C++
- Improve code structure
  - Separate normal code path from exceptional
  - Error handling code is segregated in catch blocks
- Ease task of writing robust, maintainable code

# Exception Handling

Undeclared

vs.

Declared

```
int divide(int a, int b) {  
    return a / b;  
}
```

```
String read(String path) throws  
        IOException {  
    return Files.lines(Path.of(path))  
        .collect(Collectors.joining("\n"));  
}
```

# Exception Handling

Undeclared

vs.

Declared

```
int divide(int a, int b) {  
    return a / b;  
}
```

```
String read(String path) throws  
        IOException {  
    return Files.lines(Path.of(path))  
        .collect(Collectors.joining("\n"));  
}
```

Unchecked

vs.

Checked

```
divide(4, 3); // Compiles  
              fine
```

```
read("test.txt"); // Unhandled  
                  exception: java.io.IOException
```

# Exception Handling

Handling unchecked exceptions is not enforced by the compiler

These are quite common

- E.g., all exceptions in C++
- In Java: any exception that extends Error or RuntimeException

# Exception Handling

Handling unchecked exceptions is not enforced by the compiler

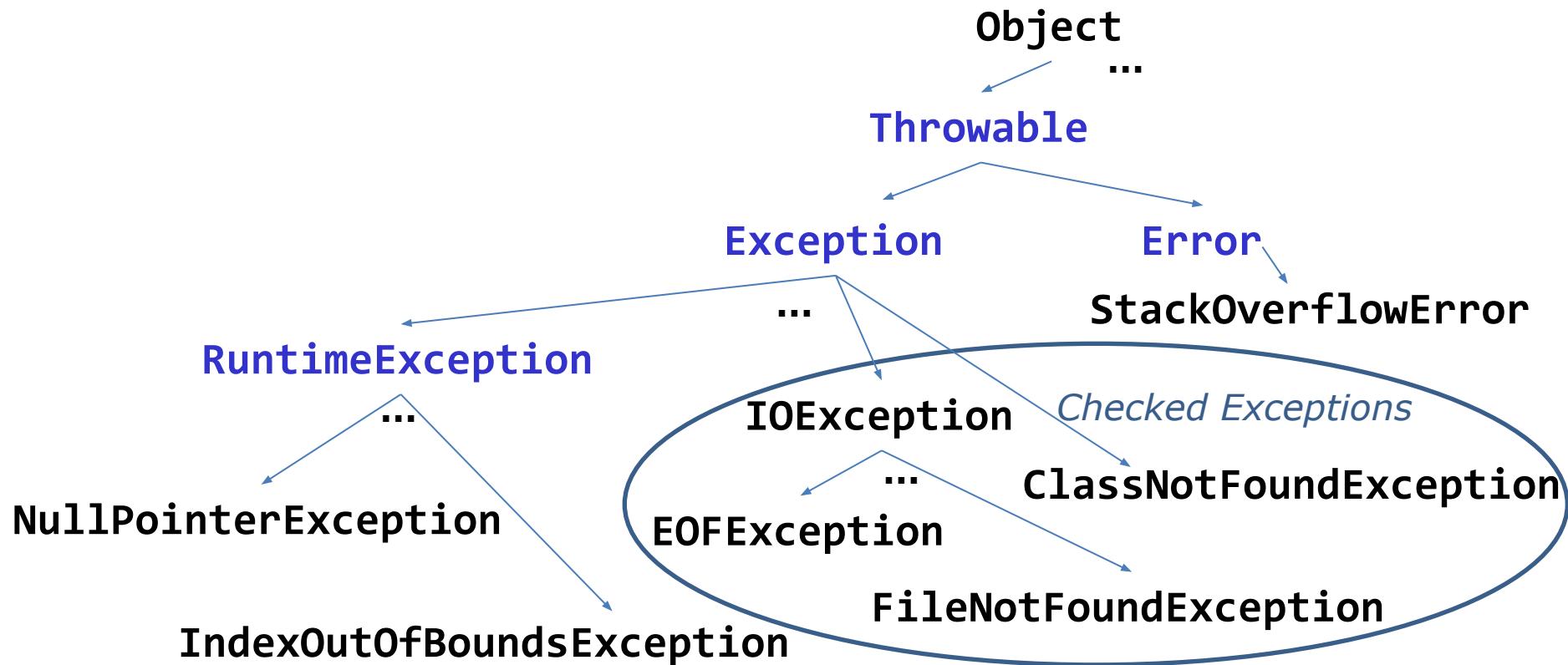
These are quite common

- E.g., all exceptions in C++
- In Java: any exception that extends Error or RuntimeException
  - E.g.:

```
int divide(int a, int b) throws ArithmeticException {  
    return a / b;  
}  
divide(4, 3); // Compiles fine
```

- Note: we don't typically declare unchecked exceptions.

# Java's exception hierarchy (messy)



# Design choice: checked vs. unchecked

- Unchecked exception
  - Programming error, other unrecoverable failure
- Checked exception
  - An error that every caller should be aware of and handle
- Special return value (e.g., null from Map.get)
  - Common but atypical result (not erroneous!)
- Do not use error codes – too easy to ignore
- Avoid null return values
  - Never return null instead of zero-length list or array

# Defining & using Exception Types

```
class BufferBoundsException extends Throwable {  
    public BufferBoundsException(String message) {  
        ...  
    }  
}  
  
void atIndex(int[] buff, int i) throws CustomException {  
    if (buff.length <= i)  
        throw new BufferBoundsException("...");  
    return buff[i];  
}
```

# Exception Handling

- It's still wise to guard for "obvious" unchecked exceptions

```
if (arr.length > 10)  
    return arr[10];
```

- Or explicitly signal the problem, recall:

```
if (buff.length <= i)  
    throw new BufferBoundsException("...");  
return buff[i];
```

- Why is this better than letting the index fail?

# Exception Handling

- It's still wise to guard for "obvious" unchecked exceptions

```
if (arr.length > 10)  
    return arr[10];
```

- Or explicitly signal the problem, recall:

```
if (buff.length <= i)  
    throw new BufferBoundsException("...");  
return buff[i];
```

- Why is this better than letting the index fail?
  - BufferBoundsException can be a checked exception!
  - Which forces someone to handle it
  - Here, we declared: `atIndex(int[] buff, int i) throws BufferBoundsException`
  - So every calling method must handle it, or throw it on

# Guidelines for using exceptions (1)

- Avoid unnecessary checked exceptions (EJ Item 71)
- Favor standard exceptions (EJ Item 72)
  - `IllegalArgumentException` – invalid parameter value
  - `IllegalStateException` – invalid object state
  - `NullPointerException` – null param where prohibited
  - `IndexOutOfBoundsException` – invalid index param
  - `IOException` -- and its subclasses, mostly for File-related actions
- Throw exceptions appropriate to abstraction (EJ Item 73)

# Guidelines for using exceptions

- Document all exceptions thrown by each method
  - Unchecked as well as checked (EJ Item 74)
  - But don't *declare* unchecked exceptions!
- Include failure-capture info in detail message (Item 75)

```
throw new IllegalArgumentException(  
    "Quantity must be positive: " + quantity);
```

# Guidelines for using exceptions (2)

- Document all exceptions thrown by each method
  - Unchecked as well as checked (EJ Item 74)
  - But don't *declare* unchecked exceptions!
- Include failure-capture info in detail message (Item 75)

```
throw new IllegalArgumentException(  
    "Quantity must be positive: " + quantity);
```

- Don't ignore exceptions (EJ Item 77)

```
try {  
    processPayment(payment);  
}  
catch (Exception e) { // BAD!  
}
```

# Cleanup

Exception handling often also supports cleaning up

```
openMyFile();
try {
    writeMyFile(theData); // This may throw an error
} catch(e) {
    handleError(e); // If an error occurred, handle it
} finally {
    closeMyFile(); // Always close the resource
}
```

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control\\_flow\\_and\\_error\\_handling](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling)

# Manual Resource Termination

Is ugly and error-prone, especially for multiple resources

- Even good programmers usually get it wrong
  - Sun's Guide to Persistent Connections got it wrong in code that claimed to be exemplary
  - Solution on page 88 of Bloch and Gafter's Java Puzzlers is badly broken; no one noticed for years
- 70% of the uses of `close` **in the JDK itself** were wrong in 2008!
- Even the “correct” idioms for manual resource management are deficient

# The solution: try-with-resources

Automatically closes resources!

```
try (DataInputStream dataInput =
      new DataInputStream(new FileInputStream(fileName))) {
    return dataInput.readInt();
} catch (IOException e) {
    ...
}
```

# Exceptions Across Languages

Alas, try-with-resources does not exist in JS/TS

- Neither does ‘throws’

Exception structures differ radically across languages

- Most languages have ‘try/catch’ and ‘throw’
  - Some have ‘finally’
- Python has ‘with’ for resource management (since 2006)
  - C# has ‘using’
  - Java’s try-with-resources was added in 2011
- Go returns an error-typed value, to be checked for nullity

# Exceptions Across Languages

Use what you have

- When possible, be explicit
  - Use the compiler to enforce, where possible
  - Pro-actively pre-empt corner-cases, where not
    - Unchecked exceptions, JS/TS
- Make exceptions part of your contract

# Outline

1. Exception Handling
2. **Unit Testing**
3. Specifications

# Testing

How do we know  
this works?

```
int isPos(int x) {  
    return x >= 1;  
}
```

# Testing

How do we know  
this works?

Testing

Are we done?

```
int isPos(int x) {  
    return x >= 1;  
}  
  
@Test  
void testIsPos() {  
    assertTrue(isPos(1));  
}
```

# Testing

How do we know  
this works?

Testing

Are we done?

```
int isPos(int x) {  
    return x >= 1;  
}  
  
@Test  
void testIsPos() {  
    assertTrue(isPos(1));  
}  
  
@Test  
void testNotPos() {  
    assertFalse(isPos(-1));  
}
```

# Testing

How do we know  
this works?

Testing

Are we done?

```
int isPos(int x) {  
    return x >= 0; // What if?  
}  
  
@Test  
void testIsPos() {  
    assertTrue(isPos(1));  
}  
  
@Test  
void testNotPos() {  
    assertFalse(isPos(-1));  
}
```

# Testing

How do we know  
this works?

Testing

Are we done?

```
int isPos(int x) {  
    return x >= 0; // What if?  
}  
  
@Test  
void test1IsPos() {  
    assertTrue(isPos(1));  
}  
  
@Test  
void test0IsNotPos() {  
    assertFalse(isPos(0)); // Fails  
}
```

# Testing

How do we know a program is correct?

- In a perfect world (maybe): formal verification
  - Easy enough for proving that `isPos(x)` -- the implementation is the definition
  - Tedious, cannot be done automatically
- Hence, testing

# Testing

- Execute the program with selected inputs in a controlled environment
  - Why is this related to contracts?

# Testing

- Execute the program with selected inputs in a controlled environment
  - Why is this related to contracts?
  - Because we need to know what to test!

# Testing

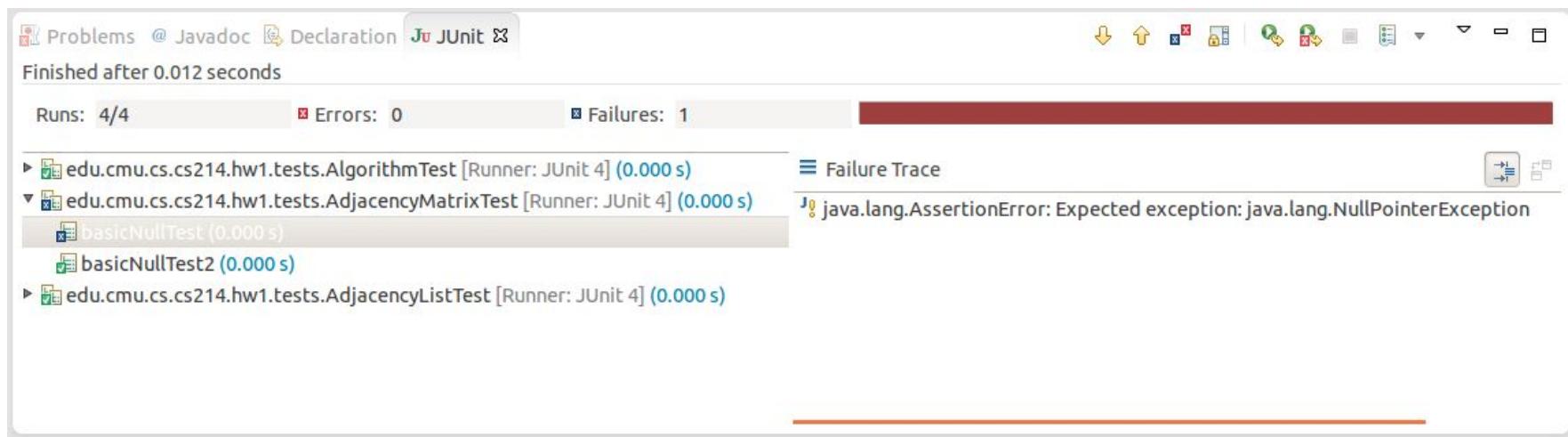
- Execute the program with selected inputs in a controlled environment
  - Why is this related to contracts?
  - Because we need to know what to test!
- Goals
  - Reveal bugs, so they can be fixed (primary goal)
  - Clarify the specification, documentation

# Unit Tests

- For “small” units: methods, classes, subsystems
  - Unit is smallest testable part of system
  - Test the parts before assembling them
  - Intended to catch local bugs
- Typically (but not always) written by developers
- Many small, fast-running, independent tests
- Few dependencies on other system parts or environment
- Insufficient, but a good starting point

# For Java: JUnit

- Popular unit-testing framework for Java
- Easy to use
- Tool support available, e.g., IntelliJ integration



# For Java: JUnit

Syntax:

```
import static org.junit.Assert.*;  
  
class PosTests {  
  
    @Before  
    void setUp() {  
        // Anything you want to run  
        // before each test  
    }  
  
    @Test  
    void test1IsPos() {  
        assertTrue(isPos(1));  
    }  
}
```

# For TS: Jest

- In particular, ts-jest
  - Many other options; your choice
- Requires a few files:
  - jest.config.js, to specify testing mode
  - package.json with (ts-)jest dependencies
- Provides useful features:
  - ‘test’, ‘expect’ (= ‘assert’)
  - ‘toBe’, ‘toEqual’
  - ‘fn’, for Mocking (later)

```
test > TS isPos.test.ts > ...
1 import { isPos } from "../src/isPos"
2
3 test('1 is positive', () => {
4   expect(isPos(1)).toBe(true);
5 });
6
7 test('-1 is not positive', () => {
8   expect(isPos(-1)).toBe(false);
9 });
10
11 test('0 is not positive', () => {
12   expect(isPos(0)).toBe(false);
13 });
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

at Object.<anonymous> ([test/isPos.test.ts:12:19](#))

Test Suites: 1 failed, 1 total  
Tests: 1 failed, 2 passed, 3 total  
Snapshots: 0 total

# Writing Testable Code

- Think about testing when writing code
  - Unit testing encourages you to write testable code
- Modularity and testability go hand in hand
  - Same test can be used on multiple implementations of an interface!
- Test-Driven Development
  - A design and development method in which you write tests before you write the code
  - Writing tests can expose API weaknesses!

# Run Tests Often

- You should only commit code that passes all tests...
- So run tests before every commit
- If test suite becomes too large & slow for rapid feedback
  - Run local package-level tests (“smoke tests”) frequently
  - Run all tests nightly
  - Medium sized projects often have thousands of test cases
- Continuous integration (CI) servers help to scale testing

# Reflections on Testing

“Testing shows the presence, not the absence of bugs.”

Edsger W. Dijkstra, 1969

“Functionality that can't be demonstrated by automated test simply don't exist.”

Kent Beck

# Boundary Value Testing

We cannot test for every integer.

Choose *representative* values:  
1 for positives, -1 for negatives

And *boundary cases*: 0 is a likely candidate for mistakes

- Think like an attacker

```
int isPos(int x) {  
    return x >= 0; // What if?  
}  
  
@Test  
void test1IsPos() {  
    assertTrue(isPos(1));  
}  
  
@Test  
void test0IsNotPos() {  
    assertFalse(isPos(0)); // Fails  
}
```

# Outline

1. Exception Handling
2. Unit Testing
3. **Specifications**

# Specifications

So what exactly do you test?

- What it claims to do: specification testing
- What it does: structural testing

# What is a contract?

- Agreement between an object and its user
  - What object provides, and user can count on
- Includes:
  - Method signature (type specifications)
  - Functionality and correctness expectations
  - Sometimes: performance expectations
- **What the method does, not how it does it**
  - Interface (API), not implementation
- “Focus on concepts rather than operations”

# Method contract details

- Defines method's and caller's responsibilities
- Analogy: legal contract
  - If you pay me this amount on this schedule...
  - I will build a room with the following detailed spec
  - Some contracts have remedies for nonperformance
- Method contract structure
  - Preconditions: what method requires for correct operation
  - Postconditions: what method establishes on completion
  - Exceptional behavior: what it does if precondition violated
- Defines correctness of implementation

# How to Encode Specifications?

Formal frameworks exist, to capture pre- and post-conditions

- E.g., ‘requires arr != null’
- Useful for formal verification
- But rarely used
  - Takes a lot of effort, and doesn’t scale well

# How to Encode Specifications?

More common: prose specification. Document:

- Every parameter
- Return value
- Every exception (checked and unchecked)
- What the method does, including
  - Primary purpose
  - Any side effects
  - Any thread safety issues
  - Any performance issues

# How to Encode Specifications?

More common: prose specification. Document

- Every parameter
- Return value
- Every exception (checked and unchecked)
- What the method does, including
  - Primary purpose
  - Any side effects
  - Any thread safety issues
  - Any performance issues
- **Do not** document implementation details
  - Known as overspecification

# Docstring Specification

```
class RepeatingCardOrganizer {  
    ...  
  
    public boolean isComplete(CardStatus card) {  
        return card.getResults().stream()  
            .filter(isSuccess -> isSuccess)  
            .count() >= this.repetitions;  
    }  
}
```

# Docstring Specification

```
class RepeatingCardOrganizer {  
    ...  
    /**  
     * Checks if the provided card has been answered correctly the required  
     * number of times.  
     * @param card The {@link CardStatus} object to check.  
     * @return {@code true} if this card has been answered correctly at least  
     * {@code this.repetitions} times.  
     */  
    public boolean isComplete(CardStatus card) {  
        return card.getResults().stream()  
            .filter(isSuccess -> isSuccess)  
            .count() >= this.repetitions;  
    }  
}
```

# Docstring Specification

```
class RepeatingCardOrganizer {  
    ...  
    /**  
     * Checks if the provided card has been answered correctly the required  
     * number of times.  
     * @param card The {@link CardStatus} object to check.  
     * @return {@code true} if this card has been answered correctly at least  
     * {@code this.repetitions} times.  
     */  
    public boolean isComplete(CardStatus card) {  
        // IGNORE THIS WHEN SPECIFICATION TESTING!  
    }  
}
```

# Docstring Specification

```
/**  
 * Checks if the provided card has been answered correctly the required  
number of times.  
 * @param card The {@link CardStatus} object to check.  
 * @return {@code true} if this card has been answered correctly at least  
{@code this.repetitions} times.  
 */  
public boolean isComplete(CardStatus card);  
  
// What is specified?
```

# Docstring Specification

```
/**  
 * Checks if the provided card has been answered correctly the required  
number of times.  
 * @param card The {@link CardStatus} object to check.  
 * @return {@code true} if this card has been answered correctly at least  
{@code this.repetitions} times.  
 */  
public boolean isComplete(CardStatus card);  
  
// What is specified?  
// - Parameter type (no constraints)
```

# Docstring Specification

```
/**  
 * Checks if the provided card has been answered correctly the required  
number of times.  
 * @param card The {@link CardStatus} object to check.  
 * @return {@code true} if this card has been answered correctly at least  
{@code this.repetitions} times.  
 */  
public boolean isComplete(CardStatus card);  
  
// What is specified?  
// - Parameter type (no constraints)  
// - Return constraints: "at least" this.repetitions correct answers
```

# Docstring Specification

```
/**  
 * Checks if the provided card has been answered correctly the required  
number of times.  
 * @param card The {@link CardStatus} object to check.  
 * @return {@code true} if this card has been answered correctly at least  
{@code this.repetitions} times.  
 */  
public boolean isComplete(CardStatus card);  
  
// What is specified?  
// - Parameter type (no constraints)  
// - Return constraints: "at least" this.repetitions correct answers  
// So what do we test?
```

# Docstring Specification

```
/**
 * Checks if the provided card has been answered correctly the required
number of times.
 * @param card The {@link CardStatus} object to check.
 * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
 */
public boolean isComplete(CardStatus card);

@Test
public void testIsCompleteSingleSuccess() {
    CardRepeater repeater = new RepeatingCardOrganizer(1); // Single repetition
    CardStatus cs = new CardStatus(new FlashCard("", ""));
    cs.recordResult(true); // Single Success
    assert???(repeater.isComplete(cs));
}
```

# Docstring Specification

```
/**  
 * Checks if the provided card has been answered correctly the required  
 * number of times.  
 * @param card The {@link CardStatus} object to check.  
 * @return {@code true} if this card has been answered correctly at least  
 * {@code this.repetitions} times.  
 */  
public boolean isComplete(CardStatus card);  
  
@Test  
public void testIsCompleteSingleSuccess() {  
    CardRepeater repeater = new RepeatingCardOrganizer(1); // Single repetition  
    CardStatus cs = new CardStatus(new FlashCard("", ""));  
    cs.recordResult(true); // Single Success  
    assertTrue(repeater.isComplete(cs));  
}
```

# Docstring Specification

```
/**  
 * Checks if the provided card has been answered correctly the required  
 * number of times.  
 * @param card The {@link CardStatus} object to check.  
 * @return {@code true} if this card has been answered correctly at least  
 * {@code this.repetitions} times.  
 */  
public boolean isComplete(CardStatus card);  
  
@Test  
public void testIsNotCompleteSingleFailure() {  
    CardRepeater repeater = new RepeatingCardOrganizer(1); // Single repetition  
    CardStatus cs = new CardStatus(new FlashCard("", ""));  
    cs.recordResult(false); // Single failure  
    assertFalse(repeater.isComplete(cs));  
}
```

# Docstring Specification

```
class RepeatingCardOrganizer {  
    ...  
    /**  
     * Checks if the provided card has been answered correctly the required  
     * number of times.  
     * @param card The {@link CardStatus} object to check.  
     * @return {@code true} if this card has been answered correctly at least  
     * {@code this.repetitions} times.  
     */  
    public boolean isComplete(CardStatus card) {  
        return card.getResults().stream()  
            .filter(isSuccess -> isSuccess)  
            .count() >= this.repetitions;  
    }  
}
```

We've now run this twice.  
Are we done testing?

# Specification vs. Structural Testing

You can test for different objectives

- Specification-based testing: test solely the specification
  - Ignores implementation, use inputs/outputs only
  - Cover all specified behavior
- Structural Testing: consider implementation
  - Optimize for various kinds of code coverage
    - Line, Statement, Data-flow, etc. -- More next week

# Specification vs. Structural Testing

You can test for different objectives

- Structural Testing:
  - By some definitions, we are done. Full line coverage, branch coverage.
  - Rarely enough, but often adequate
- Specification Testing:
  - Do not rely on code; need to consider corner-cases
  - Think like an attacker

# Specification vs. Structural Testing

```
/**  
 * Checks if the provided card has been answered correctly the required  
 * number of times.  
 * @param card The {@link CardStatus} object to check.  
 * @return {@code true} if this card has been answered correctly at least  
 * {@code this.repetitions} times.  
 */  
public boolean isComplete(CardStatus card) {  
    return card.getSuccesses.get(0); // <-- Bad, but passes both tests  
}
```

# Outlook

Homework 2 is all about testing

- Specification-testing the FlashCard system
- Some structural testing as well
  - More next Tuesday, also on coverage, test-case design
- To be released fairly soon

# Summary

- Being explicit about program behavior is ideal
  - Helps you detect bugs
  - Forces handling of special cases -- a key source of bugs
  - Increases transparency of your program's interface
- Specification comes in multiple forms
  - Explicit contracts, formal or informal
  - Compile-time signals, e.g. through exceptions
  - Testing helps clarify, often improve specifications
    - TDD takes this to the extreme
    - You rarely know your code until you test it

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

## Test case design

Christian Kästner

Vincent Hellendoorn



# Last Week

- Unit testing: small, simple, per-method tests
- Specification vs. Structural testing

# Little Quiz

<https://forms.gle/Am48bu6avqLh7ytb9>



# Note on Precondition Testing

question @175 ▾

24 views

Actions ▾

## HW2 - Testing constructor for RepeatingCardOrganizer

How should I test the constructor for RepeatingCardOrganizer?

The javadoc mentions that `repetitions` must be positive, but it doesn't explicitly say that an exception / error will be thrown (like `AssertionError`) if that is violated.

```
/**  
 * Creates a RepeatingCardSorter instance.  
 *  
 * @param repetitions The number of repetitions to require of each card. Must be positive.  
 */  
public RepeatingCardOrganizer(int repetitions) {  
    assert repetitions >= 1;  
    this.repetitions = repetitions;  
}
```

run code snippet

Visit 'Manage Class' to disable runnable code snippets

I understand that we shouldn't assume anything not stated (an exception / error will be thrown). But if we don't do that, the behavior of the `RepeatingCardOrganizer` will be undefined if we pass an invalid value.

How should we deal with that?

# Today

- Structural Testing Strategies
  - Statement, branch, path coverage; limitations
- Writing testable code & good tests
- Specification Testing Strategies
  - Boundary value analysis, combinatorial testing, decision tables
- Bit of both

# Structural Testing: a closer look

Takes into account the internal mechanism of a system (IEEE, 1990).

- Approaches include tracing data and control flow through a program

# Case Study

Assume various Wallets

```
public interface Wallet {  
    boolean pay(int cost);  
    int getValue();  
}
```

# DebitWallet.pay()

What should we test in this code?

```
public boolean pay(int cost) {  
    if (cost <= this.money) {  
        this.money -= cost;  
        return true;  
    }  
    return false;  
}
```

# DebitWallet.pay()

```
public boolean pay(int cost) {  
    if (cost <= this.money) {  
        this.money -= cost;  
        return true;  
    }  
    return false;  
}  
  
new DebitWallet(100).pay(10);
```

# DebitWallet.pay()

```
public boolean pay(int cost) {  
    if (cost <= this.money) {  
        this.money -= cost;  
        return true;  
    }  
    return false;  
}  
  
new DebitWallet(0).pay(10);
```

# CreditWallet.pay()

How about now?

```
public boolean pay(int cost, boolean useCredit) {  
    if (useCredit) {  
        if (this.credit + cost <= this.maxCredit) {  
            this.credit += cost;  
            return true;  
        }  
    }  
    if (cost <= this.cash) {  
        this.cash -= cost;  
        return true;  
    }  
    return false;  
}
```

# CreditWallet.pay()

```
public boolean pay(int cost, boolean useCredit) {  
    if (useCredit) {  
        if (enoughCredit) {  
            ...  
        }  
    }  
    if (enoughCash) {  
        ...  
    }  
    return false;  
}
```

Exercise: think about as many test scenarios as you can

# CreditWallet.pay()

```
public boolean pay(int cost, boolean useCredit) {  
    if (useCredit) {  
        if (enoughCredit) {  
            ...  
        }  
        if (enoughCash) {  
            ...  
        }  
        return false;  
    }  
}
```

Test case	useCredit	Enough Credit	Enough Cash	Result	Coverage
1	T	T	-	Pass	??

# CreditWallet.pay()

```
public boolean pay(int cost, boolean useCredit) {  
    if (useCredit) {  
        if (enoughCredit) {  
            ...  
        }  
        if (enoughCash) {  
            ...  
        }  
        return false;  
    }  
}
```

Test case	useCredit	Enough Credit	Enough Cash	Result	Coverage
1	T	T	-	Pass	--
2	F	-	T	Pass	--
3	F	-	F	Fails	Statement

# Coverage

We have tested every statement; are we done?

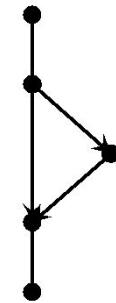
Depends on desired **coverage**:

- Provide at least one test for distinct types of behavior
- Typically on control flow paths through the program
- Statement, branch, basis paths, MC/DC

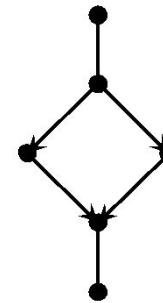
# Structures in Code



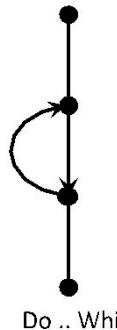
sequence



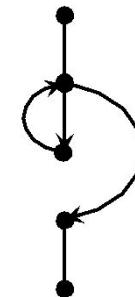
If .. then



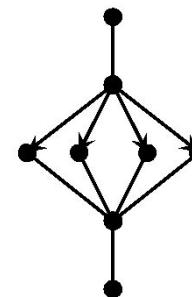
If .. then .. else



Do .. While

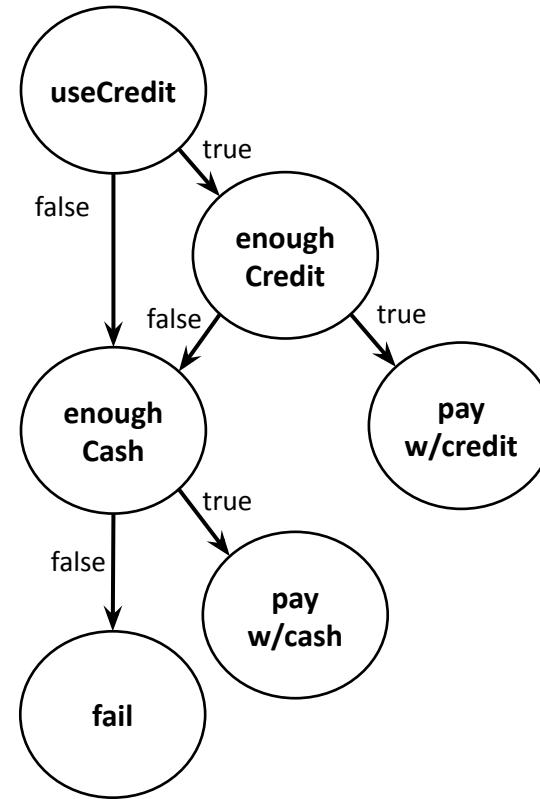


While .. Do



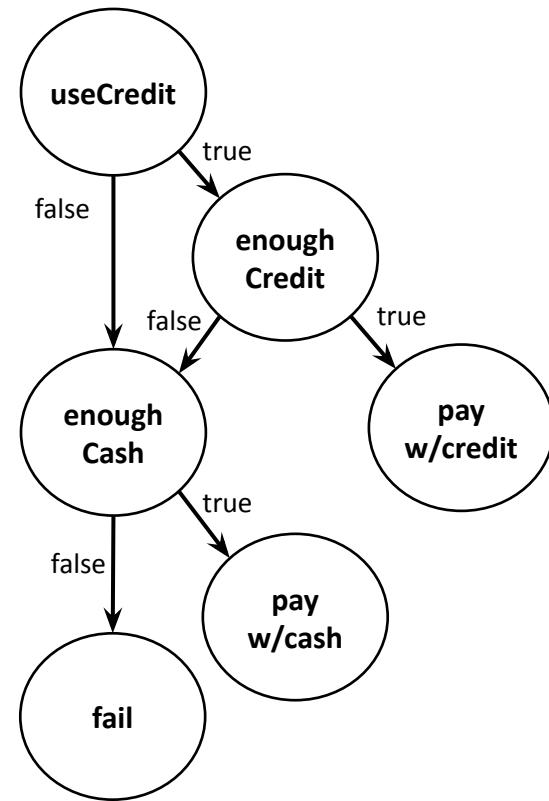
Switch

# Control-Flow of CreditCard.pay()



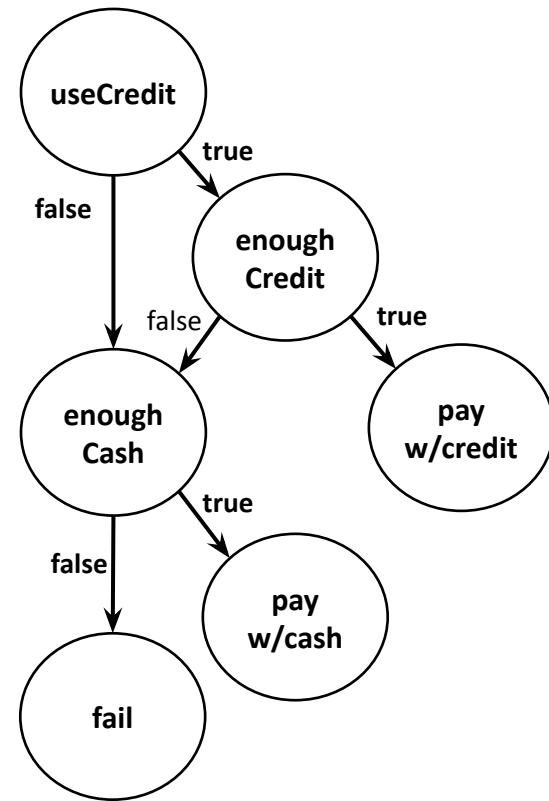
# Control-Flow of CreditCard.pay()

Test case	useCredit	Enough Credit	Enough Cash	Result	Coverage
1	T	T	-	Pass	--
2	F	-	T	Pass	--
3	F	-	F	Fails	Statement



# Control-Flow of CreditCard.pay()

Test case	useCredit	Enough Credit	Enough Cash	Result	Coverage
1	T	T	-	Pass	--
2	F	-	T	Pass	--
3	F	-	F	Fails	Statement



# CreditWallet.pay()

```
public boolean pay(int cost, boolean useCredit) {  
    if (useCredit) {  
        if (enoughCredit) {  
            ...  
        }  
    } else if (enoughCash) {  
        ...  
    }  
    return false;  
}
```

Test case	useCredit	Enough Credit	Enough Cash	Result	Coverage
1	T	T	-	Pass	--
2	F	-	T	Pass	--
3	F	-	F	Fails	Statement
4	T	F	T	Pass	Branch

# Path Coverage

We have seen every condition ... what else is missing?

# Path Coverage

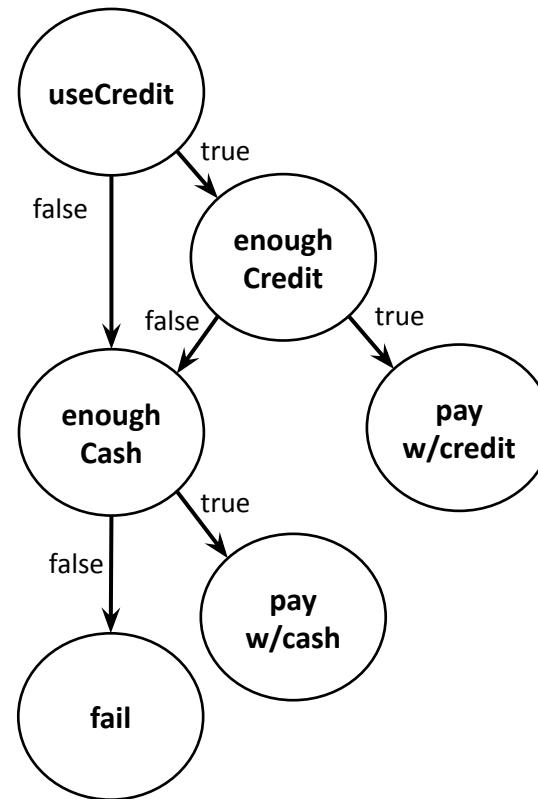
We have seen every condition ... but not every path.

- 3 conditions, each with two values = 8 permutations
- Some permutations are impossible
- Still one *path* left

# Control-Flow of CreditCard.pay()

Paths:

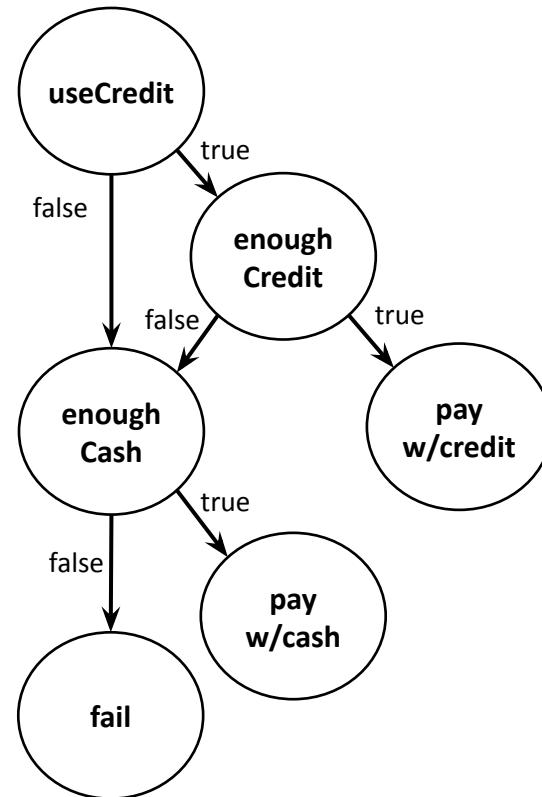
- {true, true}: pay w/credit
- {false, true}: pay w/cash
- {false, false}: fail



# Control-Flow of CreditCard.pay()

Paths:

- {true, true}: pay w/credit
- {false, true}: pay w/cash
- {false, false}: fail
- {true, false, true}: pay w/cash after failing credit
- {true, false, false}: try credit, but fail, **and** no cash



# CreditWallet.pay()

```
public boolean pay(int cost, boolean useCredit) {  
    if (useCredit) {  
        if (enoughCredit) {  
            ...  
        }  
    } else if (enoughCash) {  
        ...  
    }  
    return false;  
}
```

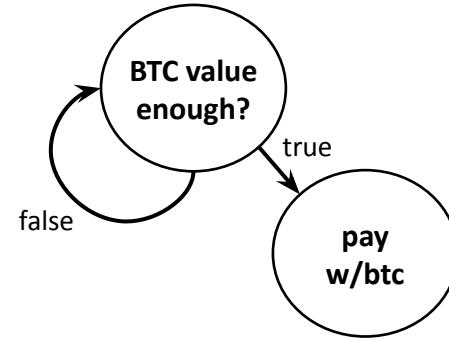
Test case	useCredit	Enough Credit	Enough Cash	Result	Coverage		
1	T	T	-	Pass	--		
2	F	-	T	Pass	--		
3	F	-	F	Fails	Statement		
4	T	F	T	Pass	Branch		
5	T	F	F	Fails	(Basis) paths		

# BitCoinWallet.pay()

```
public boolean pay(int cost) {  
    int currValue;  
    while ((currValue = getValue()) < cost) {  
        // Just wait.  
    }  
    this.btc -= cost / currValue;  
    return true;  
}  
  
public int getValue() {  
    return (int)  
        (this.btc * Math.pow(2, 20*Math.random()));  
}
```

# Control-flow of BitCoinWallet.pay()

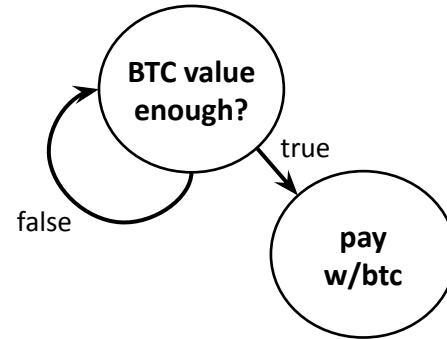
What are all the paths?



# Control-flow of BitCoinWallet.pay()

What are all the paths?

- {true}
- {false, true}
- {false, false, true}
- {false, false, false, true}
- ...

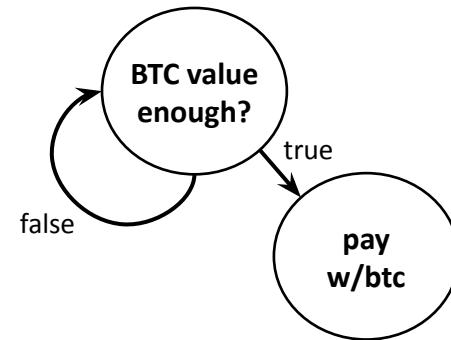


# Control-flow of BitCoinWallet.pay()

Perfect “general” path coverage is elusive

But “adequate” coverage criteria exist:

- Basis paths: each path must cover one new edge
  - {true} and {false, true} are sufficient
  - As is just {false, true}
- Loop adequacy: iterate each loop zero, one, and 2+ times

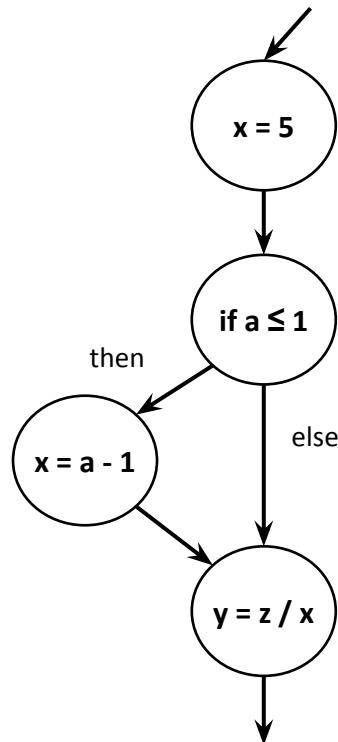


# More Coverage

Many more criteria exist:

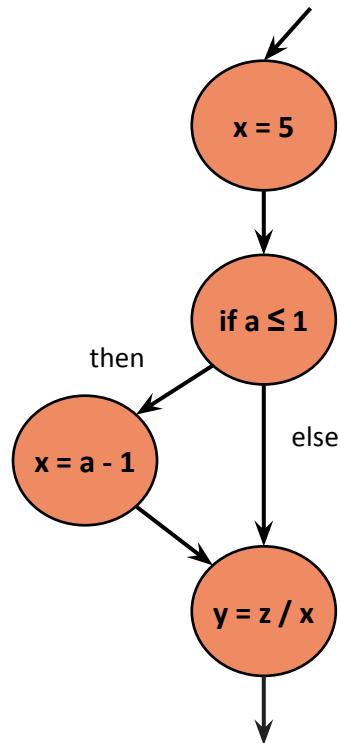
- For branches with multiple conditions
  - Modified Condition/Decision Coverage is quite popular
- For loops
  - Boundary Interior Testing
- Branch coverage is by far the most common

# Coverage and Quality



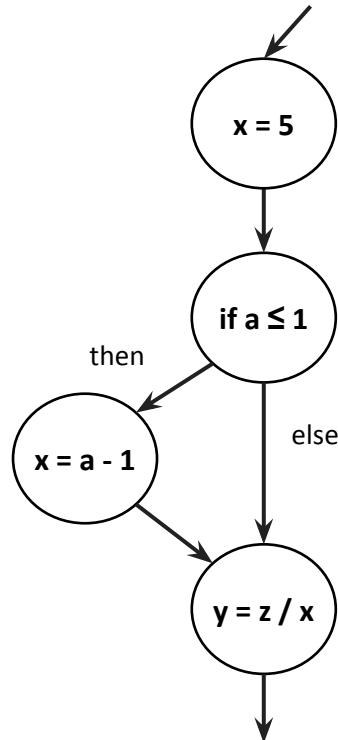
Question 1: Is there a defect?

# Coverage and Quality



Question 2: Can we achieve 100% **statement** coverage and miss the defect?

# Coverage and Quality



Question 3: Can we achieve 100% branch coverage and miss the defect?

# Outline

- Structural Testing Strategies
- **Writing testable code & good tests**
- Specification Testing Strategies

# Writing Testable Code

What is the problem with this?

```
public boolean hasHeader(String path) throws IOException {  
    List<String> lines = Files.readAllLines(Path.of(path));  
    return !lines.get(0).isEmpty()  
}  
  
// complete control-flow coverage!  
hasHeader("cards.csv") // true
```

# Writing Testable Code

What is the problem with this?

```
public boolean hasHeader(String path) throws IOException {
    List<String> lines = Files.readAllLines(Path.of(path));
    return !lines.get(0).isEmpty()
}

// to achieve a 'false' output:
try {
    Path tempFile = Files.createTempFile(null, null);
    Files.write(tempFile, "\n".getBytes(StandardCharsets.UTF_8));
    hasHeader(tempFile.toFile().getAbsolutePath()); // false
} catch (IOException e) {
    e.printStackTrace();
}
```

# Writing Testable Code

Exercise: rewrite to make this easier

- And: what would you test?

```
public boolean hasHeader(String path) throws IOException {  
    List<String> lines = Files.readAllLines(Path.of(path));  
    return !lines.get(0).isEmpty()  
}
```

# Writing Testable Code

Aim to write easily testable code

- Which is almost by definition more modular

```
public List<String> getLines(String path) throws IOException {
    return Files.readAllLines(Path.of(path));
}

public boolean hasHeader(List<String> lines) {
    return !lines.get(0).isEmpty()
}

// Test:
// - hasHeader with empty, non-empty first line
// - getLines (if you must) with null, real path
```

# Writing Testable Code

What is the problem with this?

```
public String[] getHeaderParts(List<String> lines) {  
    if (!lines.isEmpty()) {  
        String header = lines.get(0);  
        if (header.contains(", ")) {  
            return header.split(", ");  
        } else {  
            return new String[0];  
        }  
    } else {  
        return null;  
    }  
}
```

# Writing Testable Code

Split functionality into easily testable units

```
public String[] getHeaderParts(List<String> lines) {
    if (!lines.isEmpty()) {
        return getHeaderParts(lines.get(0));
    } else {
        return null;
    }
}

public String[] getHeaderParts(String header) {
    if (header.contains(",")) {
        return header.split(",");
    } else {
        return new String[0];
    }
}
```

# Clean Testing

What is the problem with this?

```
public String[] getHeaderParts(String header) {  
    if (header.contains(", ")) {  
        return header.split(", ");  
    } else {  
        return null;  
    }  
}  
  
@Test  
public void testGetHeaderParts() {  
    for (String header : List.of("line", "", "one, two")) {  
        String[] parts = getHeaderParts(header);  
        if (header.contains(", ")) assertNull(parts);  
        else assertEquals(header.split(", "), parts.length);  
    }  
}
```

# Clean Testing

Keep tests simple, small

```
public String[] getHeaderParts(String header) {  
    if (header.contains(", ")) {  
        return header.split(", ");  
    } else {  
        return null;  
    }  
}  
  
@Test  
public void testGetHeaderPartsNoComma() {  
    String[] parts = getHeaderParts("line");  
    assertNull(parts);  
}  
  
@Test  
...
```

# Testing Best Practices

Coverage is useful, but no substitute for your insight

- Cannot capture all paths
  - Especially beyond “unit”
  - Write testable code
- You may be testing buggy code
  - (add regression tests)
- Aim for at least branch coverage
  - And think through scenarios that demand more

# Outline

- Structural Testing Strategies
- Writing testable code & good tests
- **Specification Testing Strategies**

# Back to Specification Testing

What would you test differently in this situation?

- Previously identified five paths through the code. Are there still?
- Should we test anything new?

```
/** Pays with credit if useCredit is set and enough  
 * credit is available; otherwise, pays with cash if  
 * enough cash is available; otherwise, returns false.  
 */  
public boolean pay(int cost, boolean useCredit);
```

# Back to Specification Testing

What would you test differently in this situation?

- “if `useCredit` is set and enough credit is available”:
  - Test both true, either/both false
- “pays with cash if enough cash is available; otherwise”:
  - Test true, false
- Could do this with three test cases

```
/** Pays with credit if useCredit is set and enough
 * credit is available; otherwise, pays with cash if
 * enough cash is available; otherwise, returns false.
 */
public boolean pay(int cost, boolean useCredit);
```

# Specification Testing

We need a *strategy* to identify plausible mistakes

# Specification Testing

We need a *strategy* to identify plausible mistakes

- Random: avoids bias, but inefficient
  - Yet potentially *very* valuable, because automatable
  - Not for today

# Boundary Value Testing

We need a *strategy* to identify plausible mistakes

- Boundary Value Testing: errors often occur at boundary conditions
  - E.g.:

```
/** Returns true and subtracts cost if enough
 * money is available, false otherwise.
 */
public boolean pay(int cost) {
    if (cost < this.money) {
        this.money -= cost;
        return true;
    }
    return false;
}
```

# Boundary Value Testing

We need a *strategy* to identify plausible mistakes

- Boundary Value Testing: errors often occur at boundary conditions
  - Identify equivalence partitions: regions where behavior should be the same
    - `cost <= money: true, cost > money: false`
    - Boundary value: `cost == money`

```
/** Returns true and subtracts cost if enough
 * money is available, false otherwise.
 */
public boolean pay(int cost) {
    if (cost < this.money) {
        this.money -= cost;
        return true;
    }
    return false;
}
```

# Boundary Value Testing

We need a *strategy* to identify plausible mistakes

- Boundary Value Testing: errors often occur at boundary conditions
  - Select: a nominal/normal case, a boundary value, and an abnormal case
  - Useful for few *categories* of behavior (e.g., null/not-null) per value
- Test: `cost < credit`, `cost == credit`, `cost > credit`,  
`cost < cash`, `cost == cash`, `cost > cash`

```
/** Pays with credit if useCredit is set and enough
 * credit is available; otherwise, pays with cash if
 * enough cash is available; otherwise, returns false.
 */
public boolean pay(int cost, boolean useCredit);
```

# Combinatorial Testing

We need a *strategy* to identify plausible mistakes

- Combinatorial Testing: focus on tuples of boundary values
  - Captures bugs in **interactions** between risky inputs
  - Rarely need to test pairs of “invalid” values (cost too high for credit & cash)

```
/** Pays with credit if useCredit is set and enough
 * credit is available; otherwise, pays with cash if
 * enough cash is available; otherwise, returns false.
 */
public boolean pay(int cost, boolean useCredit);
```

# Combinatorial Testing

We need a *strategy* to identify plausible mistakes

- Combinatorial Testing: focus on tuples of boundary values
  - Captures bugs in **interactions** between risky inputs
  - Rarely need to test pairs of “invalid” values (cost too high for credit & cash)
- Include: {cost > credit && cost == cash}
- Maybe: {cost < credit && cost == cash}

```
/** Pays with credit if useCredit is set and enough
 * credit is available; otherwise, pays with cash if
 * enough cash is available; otherwise, returns false.
 */
public boolean pay(int cost, boolean useCredit);
```

# Decision Tables

We need a *strategy* to identify plausible mistakes

- Decision Tables
  - You've seen one already
  - Enumerate condition options
    - Leave out impossibles
    - Identify “don't-matter” values
  - Useful for redundant input domains

Test case	useCredit	Enough Credit	Enough Cash	Result
1	T	T	-	Pass
2	F	-	T	Pass
3	F	-	F	Fails
4	T	F	T	Pass
5	T	F	F	Fails

# Specification Tests

So what is the right granularity?

- It depends
- We are still aiming for coverage
  - Just of specifications, and their innumerable implementations
  - BVA (& its cousins), decision tables tend to provide good coverage

# Structural Testing vs. Specification Testing

You will *typically have both* code & (prose) specification

- Test specification, but know that it can be underspecified
- Test implementation, but not to the point that it cannot change
- Use testing strategies that leverage both
  - There is a fair bit of overlap; e.g., BVA yields useful branch coverage

# Further Testing Strategies

Many more aspects, some later in this course:

- Stubbing/Mocking, to avoid testing dependencies
- Integration testing: scenarios that span units
- Beyond correctness: performance, security

# Summary

Testing comprehensively is hard

- Tailor to your task: specification vs. structural testing
  - Do not assume unstated specifications for part 2; spend your energy wisely in part 3
- Pick a strategy, or a few
  - Be systematic; defend your decisions
- Tomorrow's recitation covers:
  - Unit test best practices
  - Test organization
  - Running tests, coverage; Travis setup

# Bonus: Coding like the tour the france

```
public boolean foo() {  
    try {  
        synchronized () {  
            if () {  
            } else {  
            }  
            for () {  
                if () {  
                    if () {  
                        if () {  
                            if () {  
                                if () {  
                                    for () {  
                                }  
                            }  
                        }  
                    }  
                } else {  
                    if () {  
                        for () {  
                            if () {  
                                } else {  
                                }  
                            if () {  
                                } else {  
                                if () {  
                                    if () {  
                                        if () {  
                                            for () {  
                                        }  
                                    }  
                                }  
                            }  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

<https://thedailywtf.com/articles/coding-like-the-tour-de-france>

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

## Object-oriented analysis

Christian Kästner   Vincent Hellendoorn



# 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

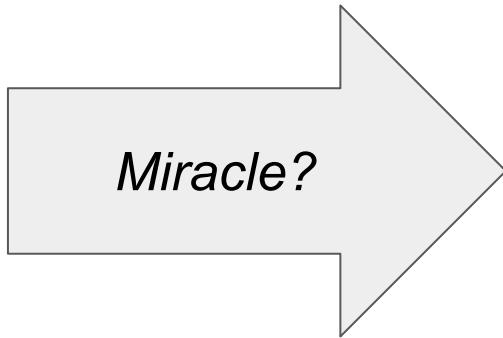
```
@Test ...  
Comparator x =  
    myComplexImpl();  
  
List l =  
    loadFromFile();  
  
l.sort(x);
```

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

# Learning Goals

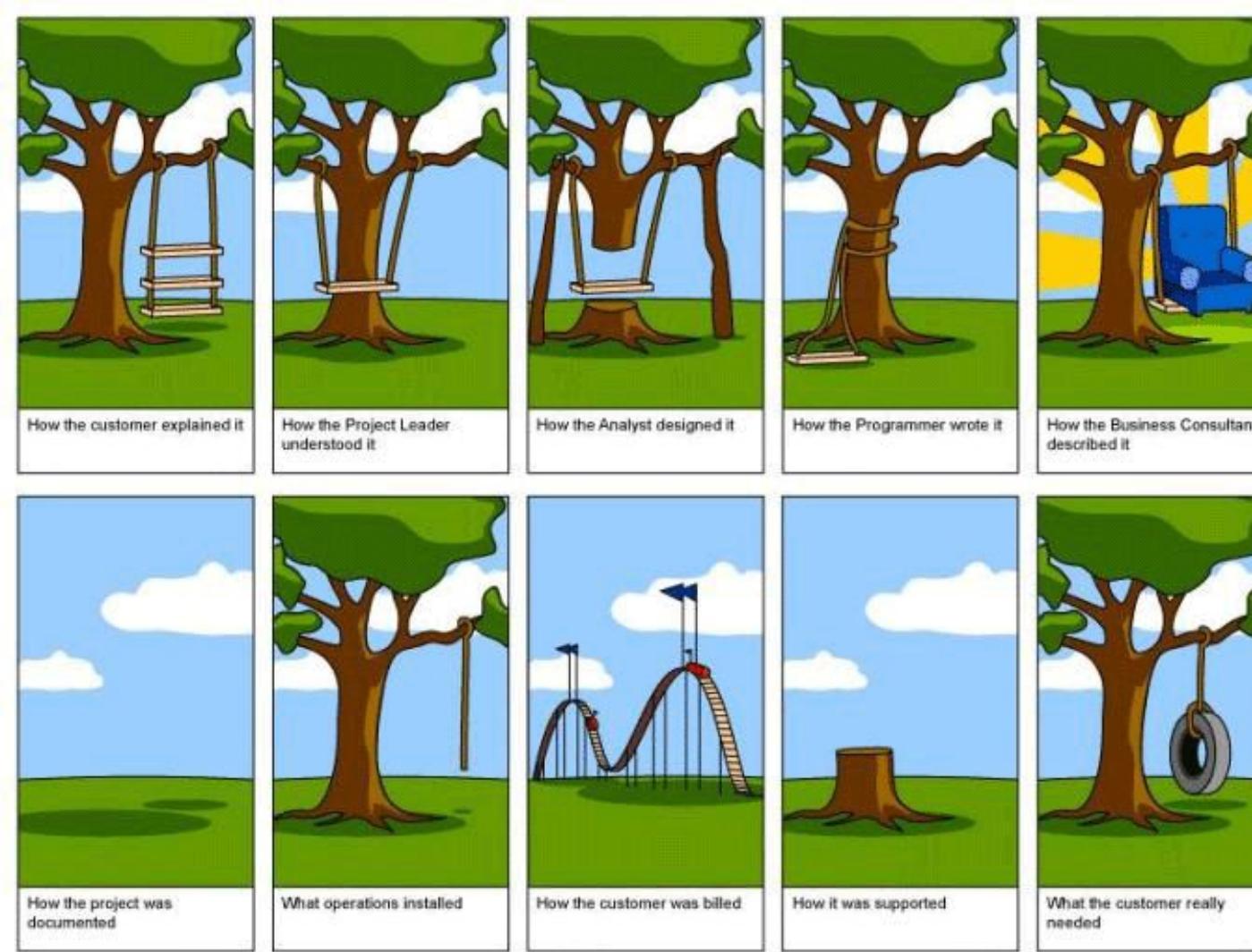
- High-level understanding of requirements challenges
- Use basic UML notation to communicate designs
- Identify the key abstractions in a domain, model them as a **domain model**
- Identify the key interactions within a system, model them as **system sequence diagram**
- Discuss benefits and limitations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

User needs  
(Requirements)



Code

# REQUIREMENTS



# 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



17-214/5

# Requirements

- What
  - What
  - Cust
  - Difficult to define requirements?
  - (Are
- 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

# 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

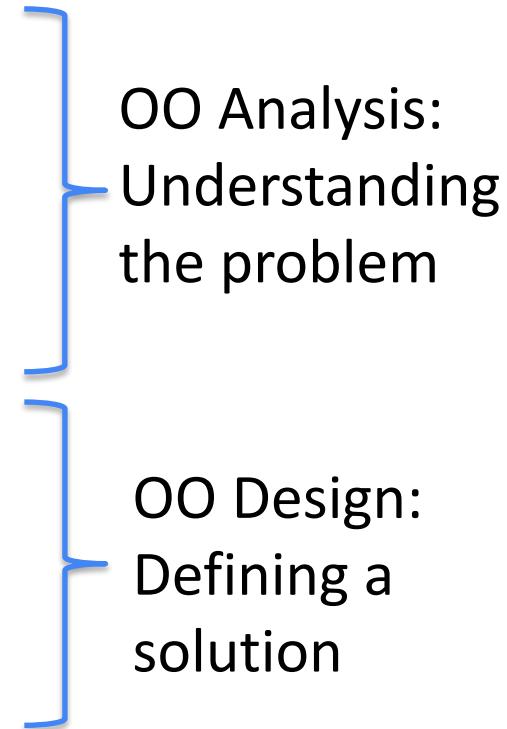
- **System sequence diagram**
- **System behavioral contracts**

Assign object responsibilities, define interactions

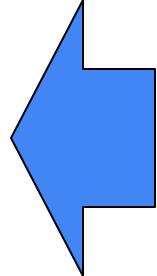
- **Object interaction diagrams**

Model / diagram a potential solution

- **Object model**



# 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

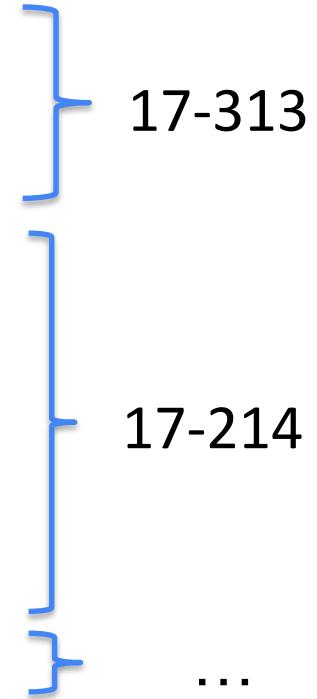
- Identify software classes and their relationships with class diagrams
- Assign responsibilities (attributes, methods)
- Explore behavior with interaction diagrams
- Explore design alternatives
- Create an object model (aka design model) and interaction models

## Implementation

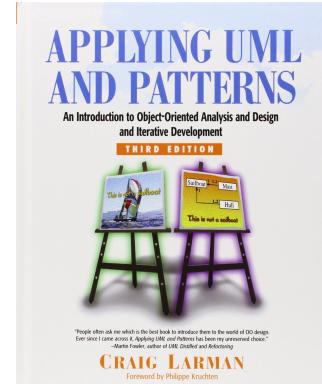
- Map designs to code, implementing classes and methods

# 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**

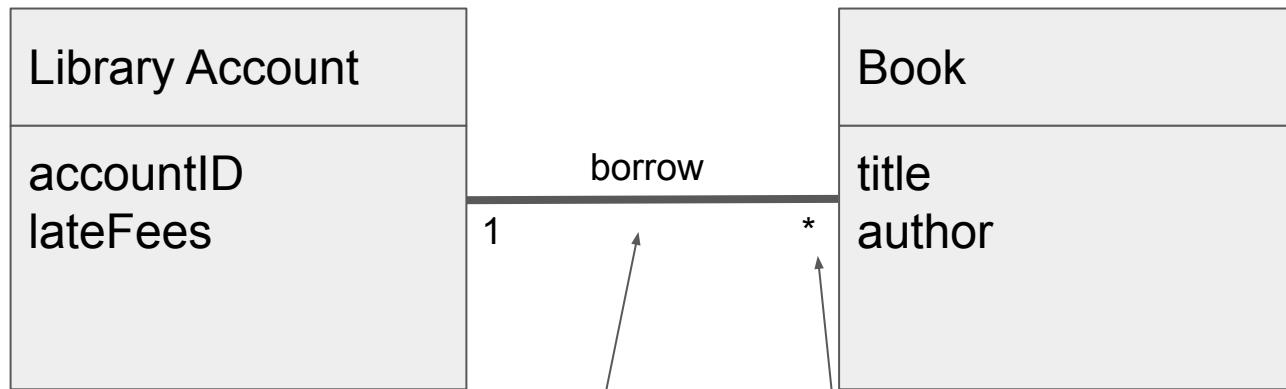
Define potentially ambiguous concepts

No need to expand on obvious concepts

# Visual notation: UML

Name of  
real-world  
concept  
(not software class)

Properties  
of concept

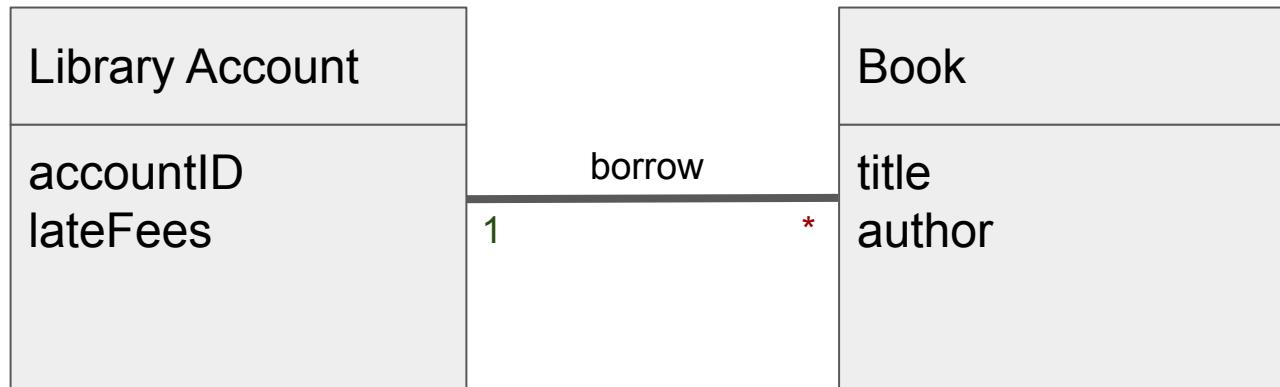


Associations  
between  
concepts

Multiplicities/cardinalities  
indicate “how many”

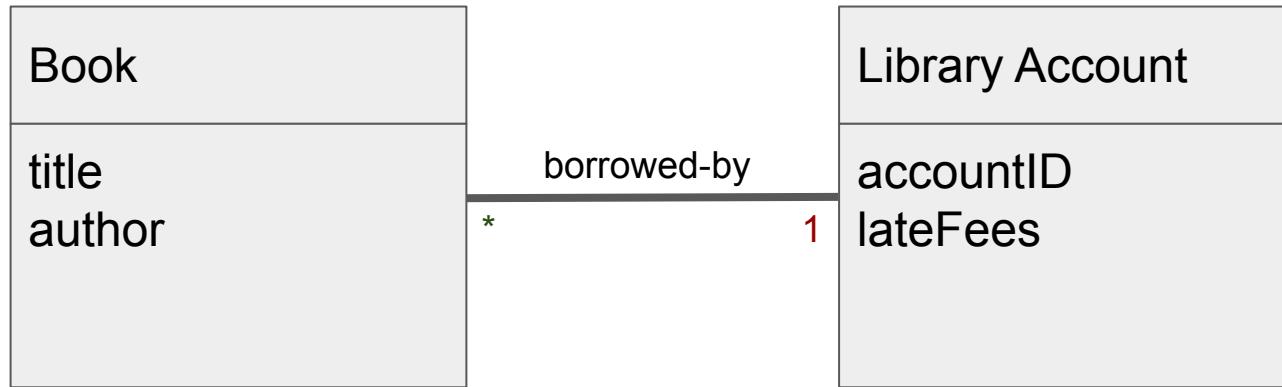
# Reading associations

One library account can borrow *many* books

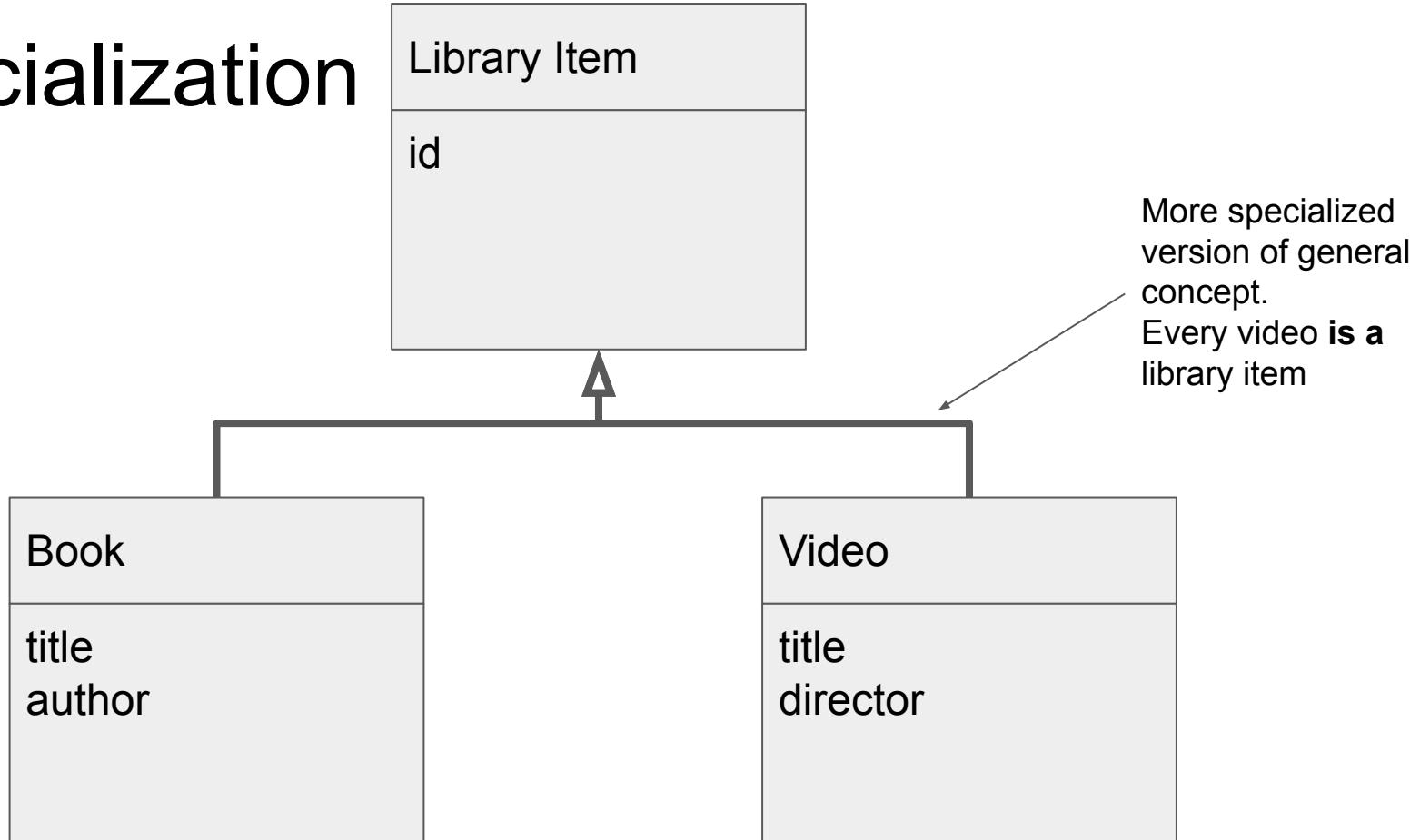


One book can be borrowed by *one* library account

# Reading associations



# Specialization

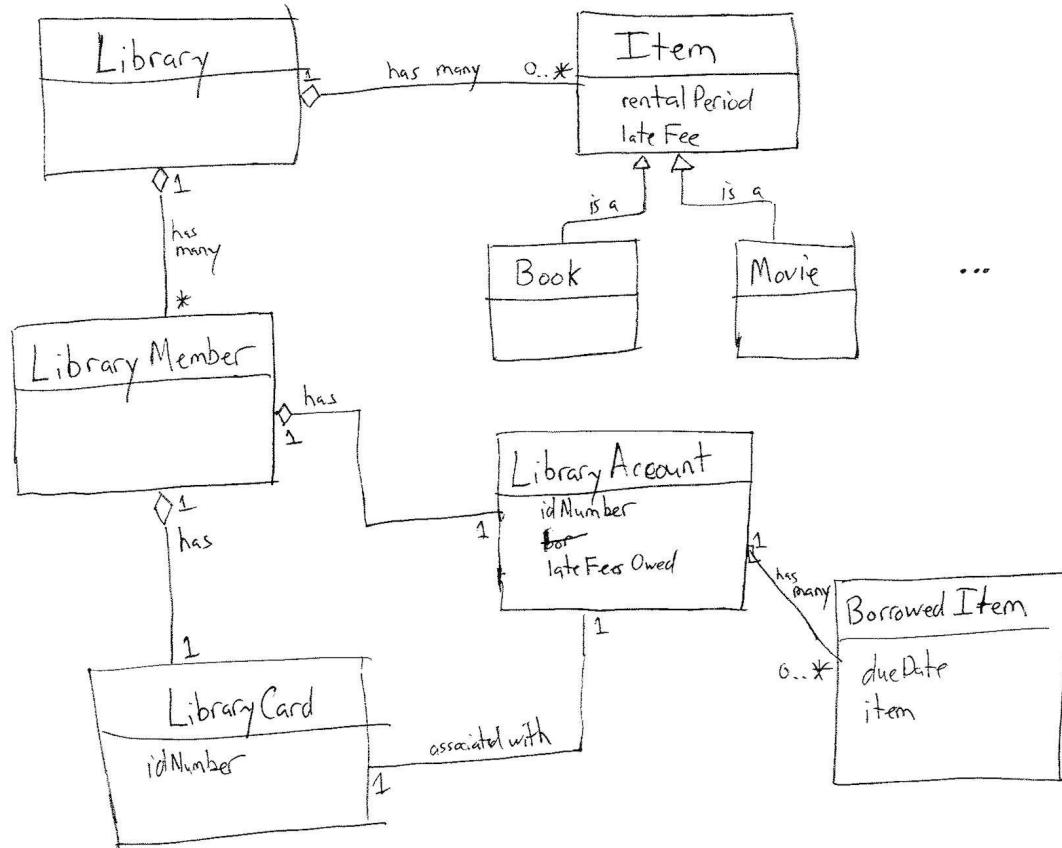


# Concepts vs. Attributes



- "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

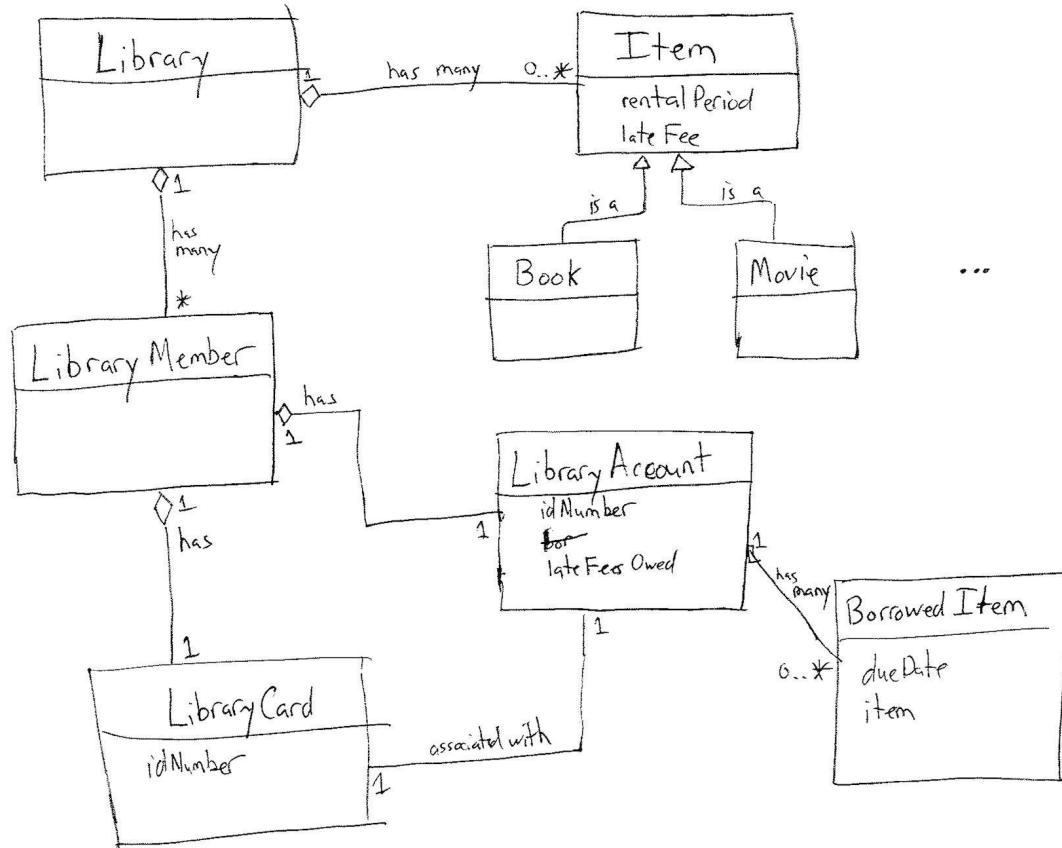
- tangible things: cars, telemetry data, terminals, ...
- roles: mother, teacher, researcher
- events: landing, purchase, request
- interactions: loan, meeting, intersection, ...
- structure, devices, organizational units, ...

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](http://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  on the Ocean Board , using the long and short tabs on the Cliff Pedestal to guide assembly.
- 2 Place the Island Board  on top of the Cliff Pedestal , again using the long and short tabs to guide assembly.
- 3 The youngest player is the Start Player, who begins by placing 2 Workers  of their chosen color into any unoccupied spaces on the board. The other player(s) then places their Workers .

## 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.

## Winning the Game

1 If one of your Workers moves up on top of level 3 during your turn, you instantly win!

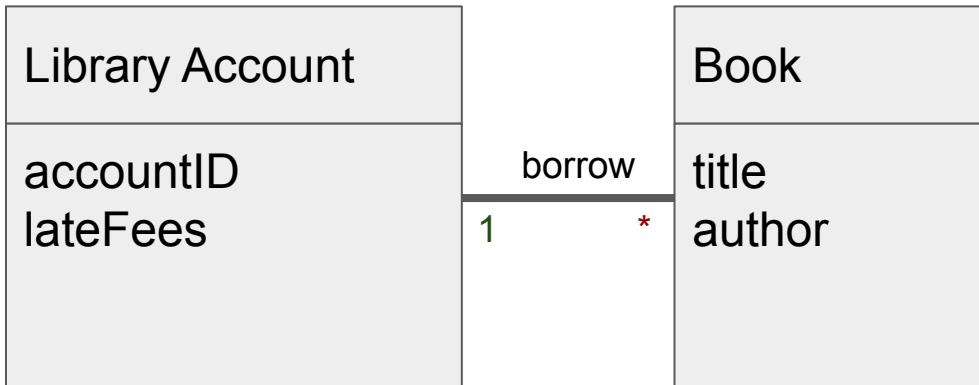
2 You must always perform a move then build on your turn. If you are unable to, you lose.

## Components

# Outlook: Low Representational Gap

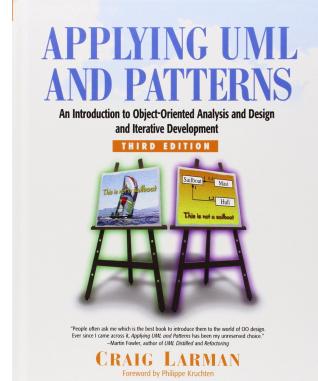
Identified concepts provide inspiration for classes in the implementation

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**

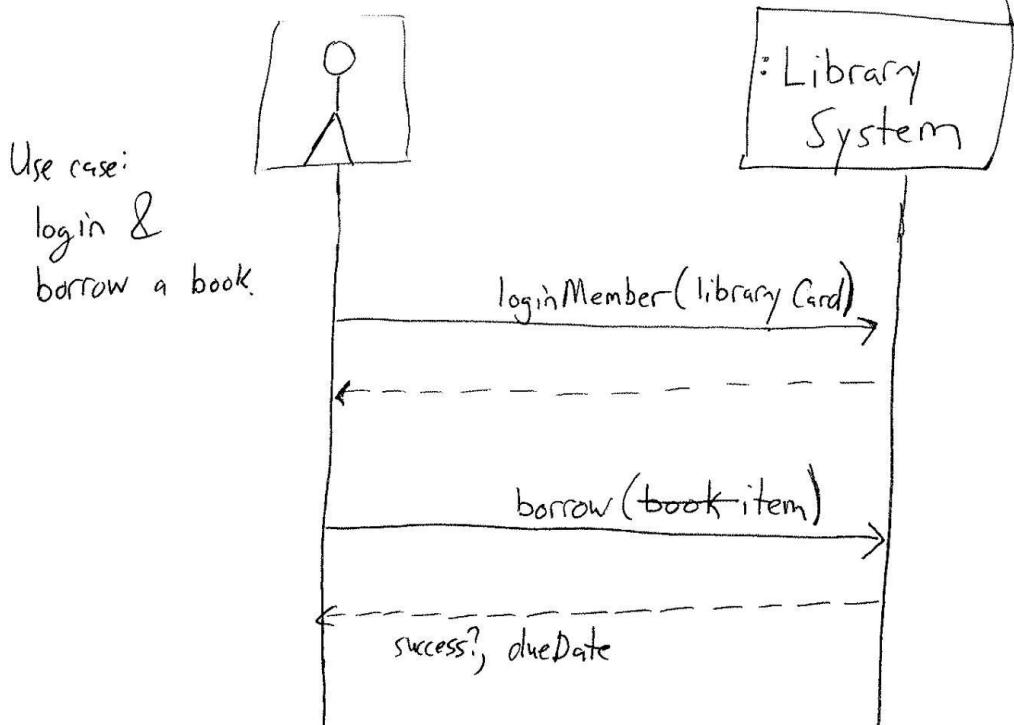
- System-level components only: e.g., A user and the overall system

# One example for the library system

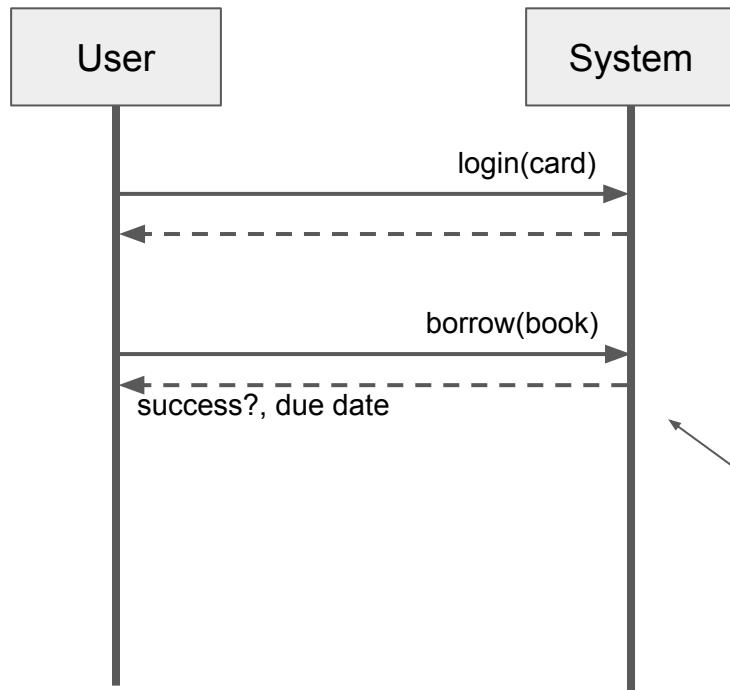
Use case scenario: A library member should be able to use her library card to log in at a library system kiosk and borrow a book. After confirming that the member has no unpaid late fees, the library system should determine the book's due date by adding its rental period to the current day, and record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



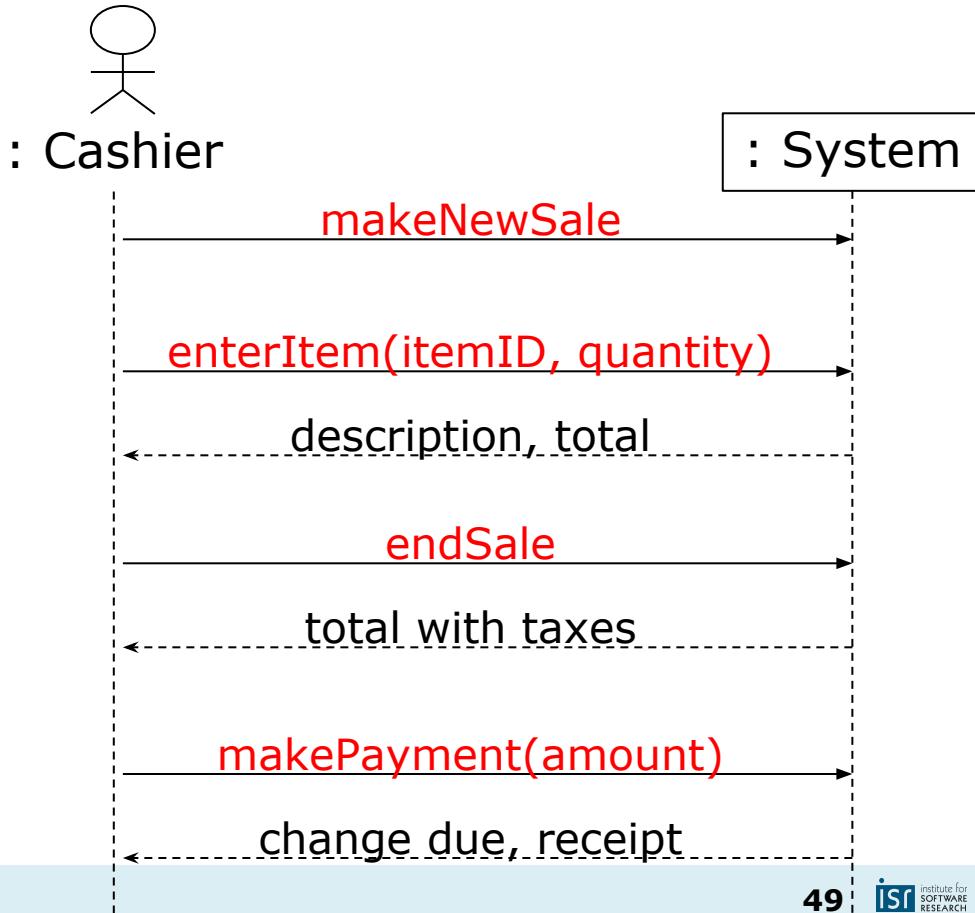
Actors in this use case (systems and real-world objects/people)

Time proceeds from top to bottom

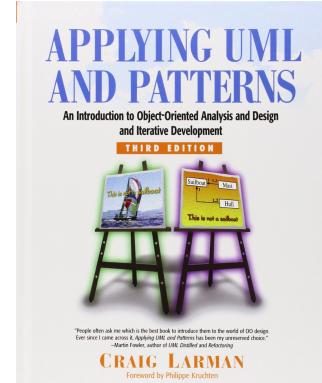
Messages and responses for interactions, text describes what happens conceptually

# 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

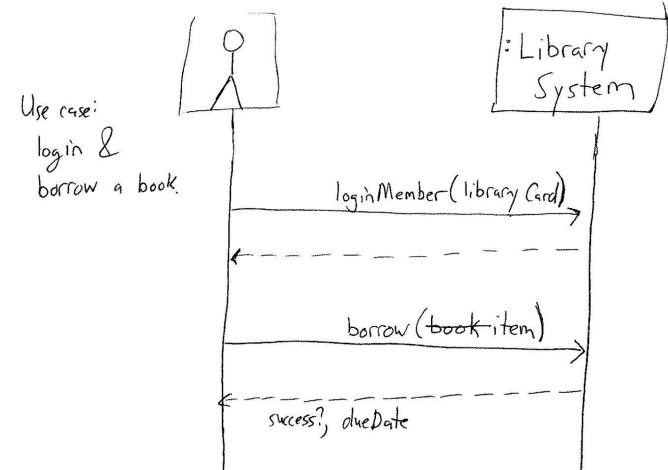


Chapter 11

# 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

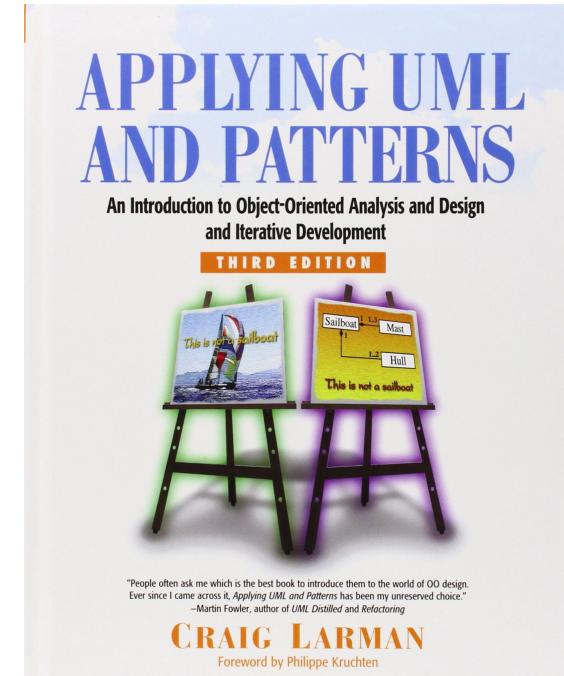
- Domain-level concepts:
  - Almost anything with a real-world analogue
- Implementation-level concepts:
  - Implementation-like method names
  - Programming types
  - Visibility modifiers
  - Helper methods or classes
  - Artifacts of design patterns

# Recommended Reading: Applying UML and Patterns

Detailed coverage of modeling steps

Explains UML notation

Many examples



"People often ask me which is the best book to introduce them to the world of OO design. Ever since I came across it, *Applying UML and Patterns* has been my unreserved choice."  
—Martin Fowler, author of *UML Distilled* and *Refactoring*

CRAIG LARMAN  
Foreword by Philippe Kruchten

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

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

## Assigning Responsibilities

**Christian Kästner** Vincent Hellendoorn



# Reading Quiz:



<https://bit.ly/2VUhx3B>

## Aside: Program Core without a GUI?

```
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() { ... }  
    }  
}
```

## Aside: Program Core without a GUI?

```
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)
}

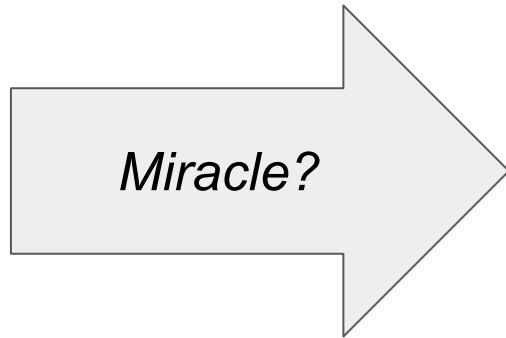
function newBoard(state, nextPlayer) {
    return {
        play: function(x, y) {
            state[y*3+x] = nextPlayer
            nextPlayer = 1 - nextPlayer
        },
        winner: function() { ... }
    }
}
```



# Learning Goals

- Apply GRASP patterns to assign responsibilities in designs
- Use UML notation for sequence and object models
- Reason about tradeoffs among designs
  - Discuss tradeoffs in terms of coupling and cohesion

User needs  
(Requirements)



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

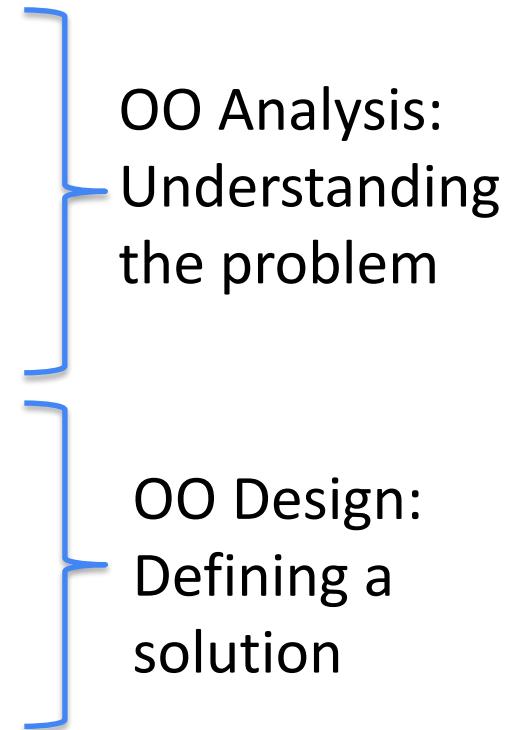
- **System sequence diagram**
- **System behavioral contracts**

Assign object responsibilities, define interactions

- **Object interaction diagrams**

Model / diagram a potential solution

- **Object model**



# **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

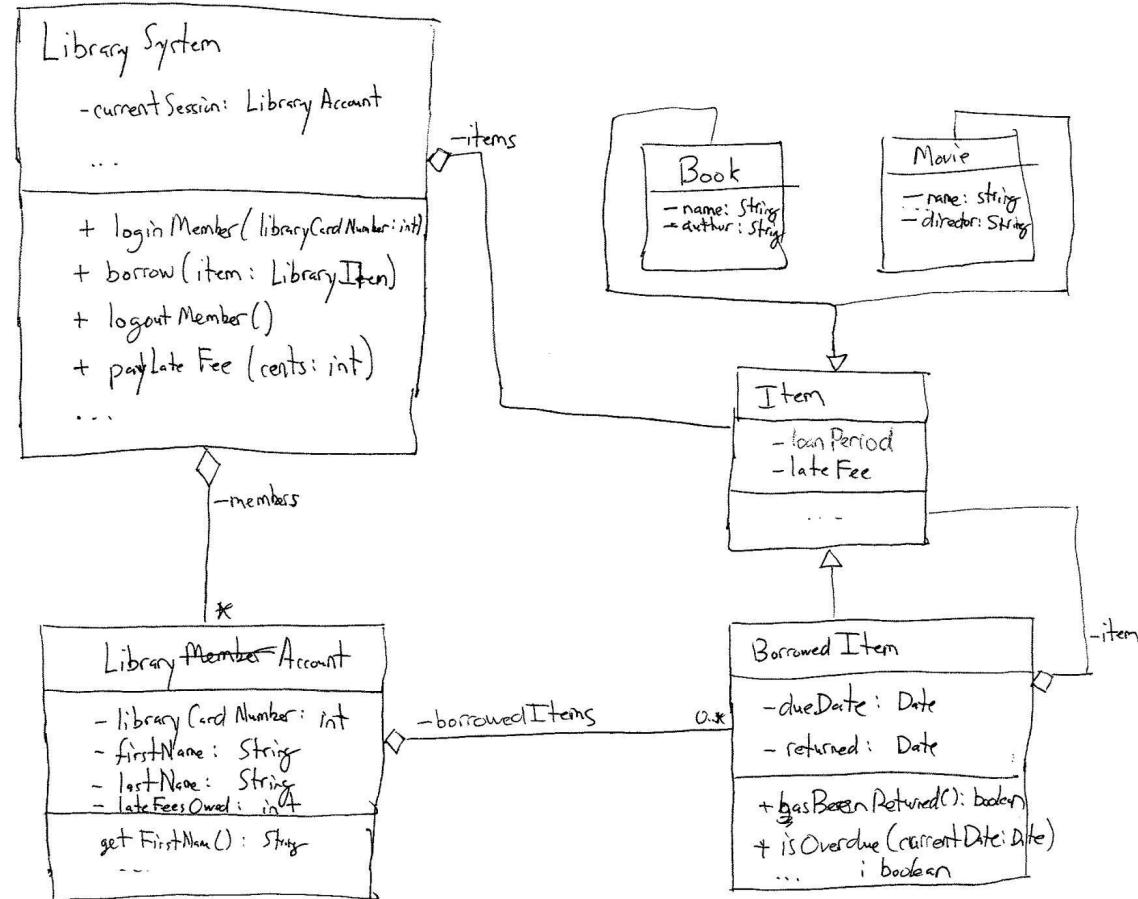
*In this course:* Use UML for communication; follow notation somewhat rigorously, but won't care about all details

# Object Diagrams

Objects/classes with fields and methods

Interfaces with methods

Associations, visibility, types

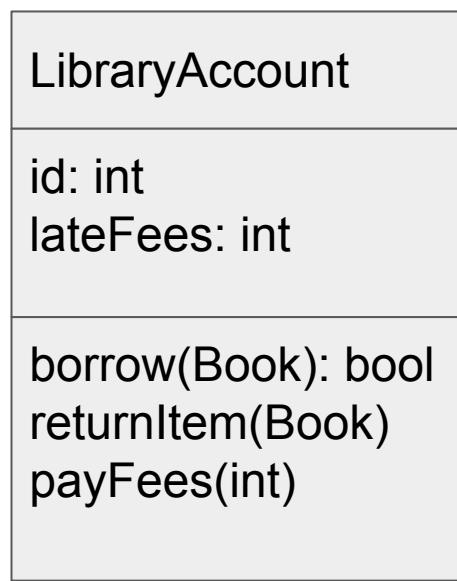


# Object Diagram Notation: Classes/Objects

Classname  
(lowercase  
name of  
objects)

Fields

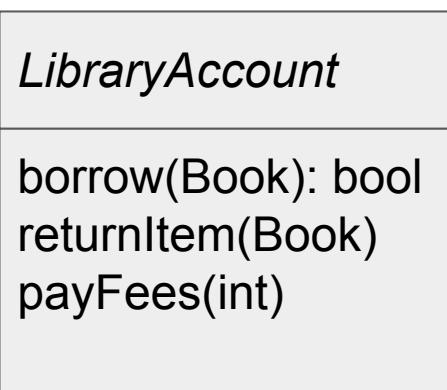
Methods



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

# Object Diagram Notation: Interfaces

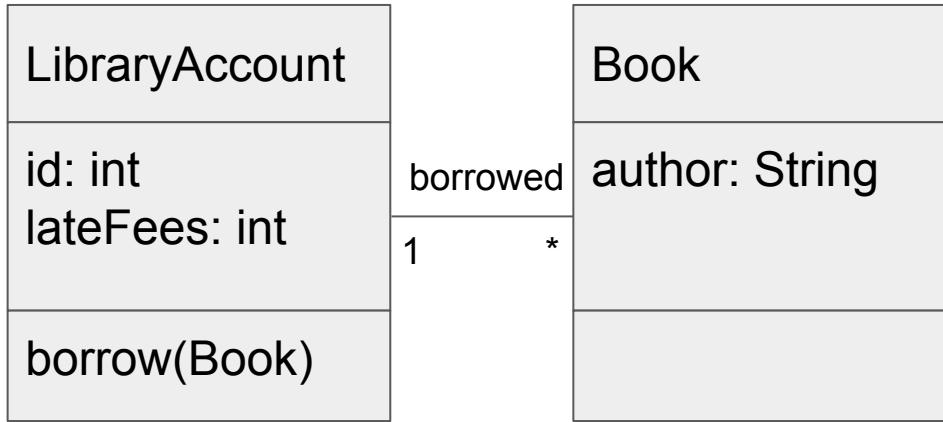
Interface name →



Methods →

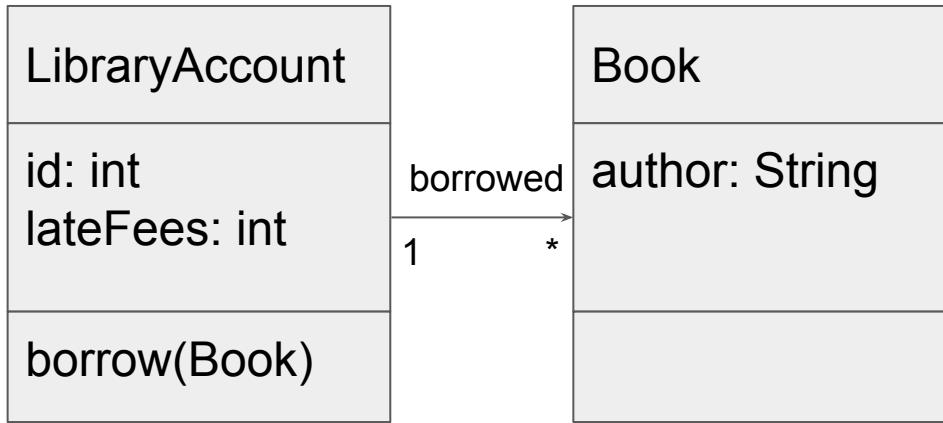
```
interface LibraryAccount {  
    boolean borrow(Book b);  
    void returnItem(Book b);  
    void payFees(int payment);  
}
```

# Object Diagram Notation: Associations



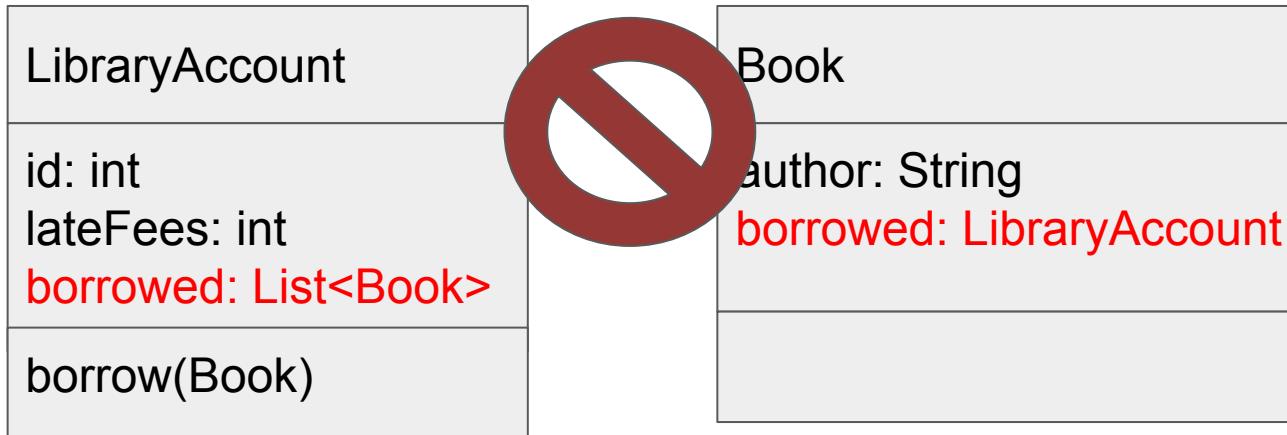
```
class LibraryAccount {  
    ...  
    List<Book> borrowedBooks;  
}  
  
class Book {  
    ...  
    LibraryAccount borrowedBy;  
}
```

# Object Diagram Notation: Associations



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

# Object Diagram Notation: Associations



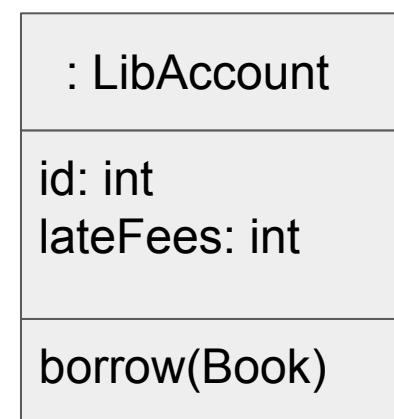
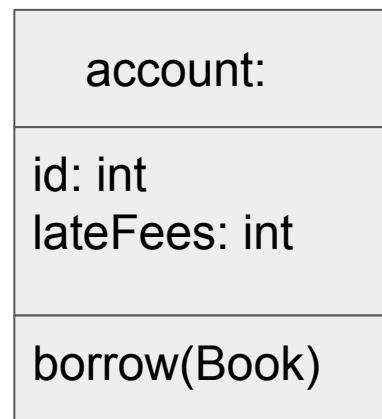
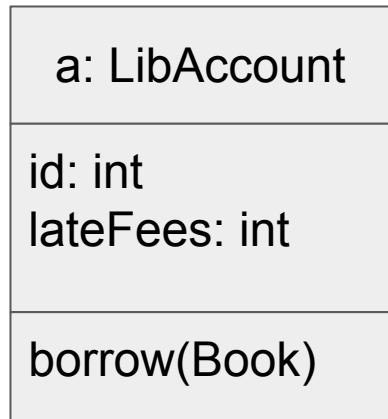
Don't use fields instead or in addition to associations. Use fields only for basic types

# 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

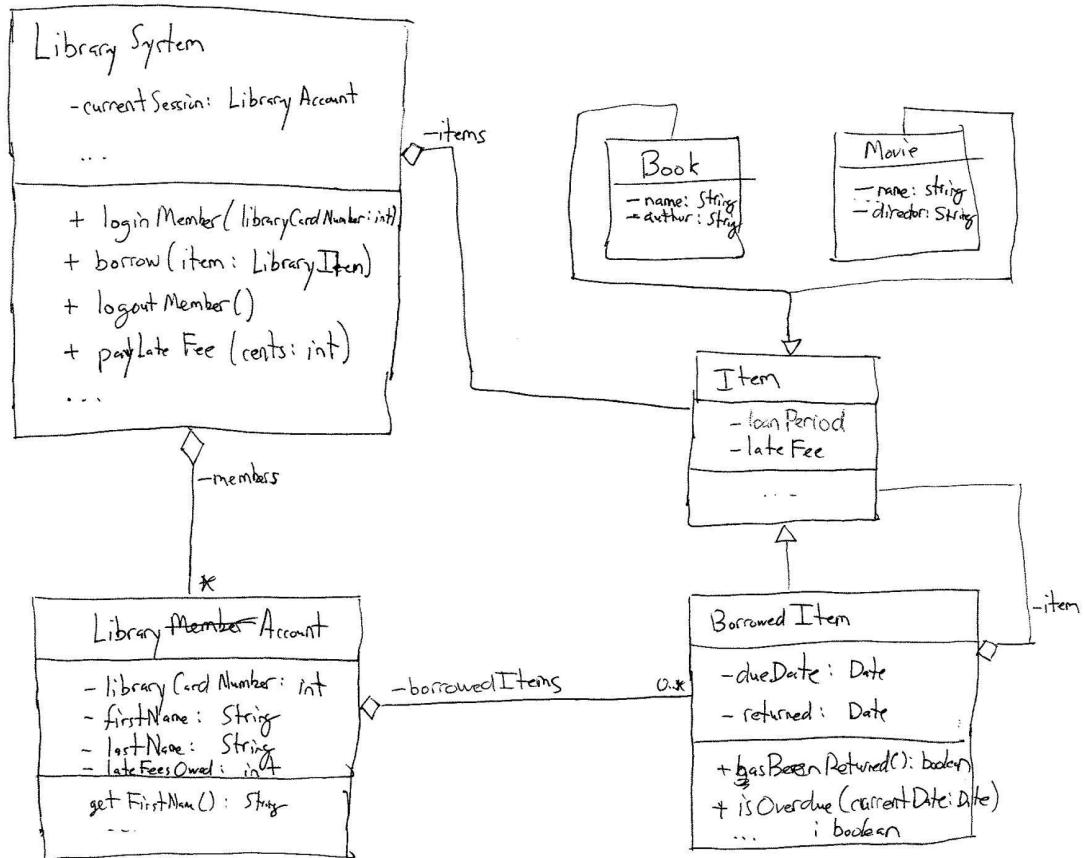
lateFees: int

borrow(Book): bool

returnItem(Book)

payFees(int)

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



# Object diagram notation requirements

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

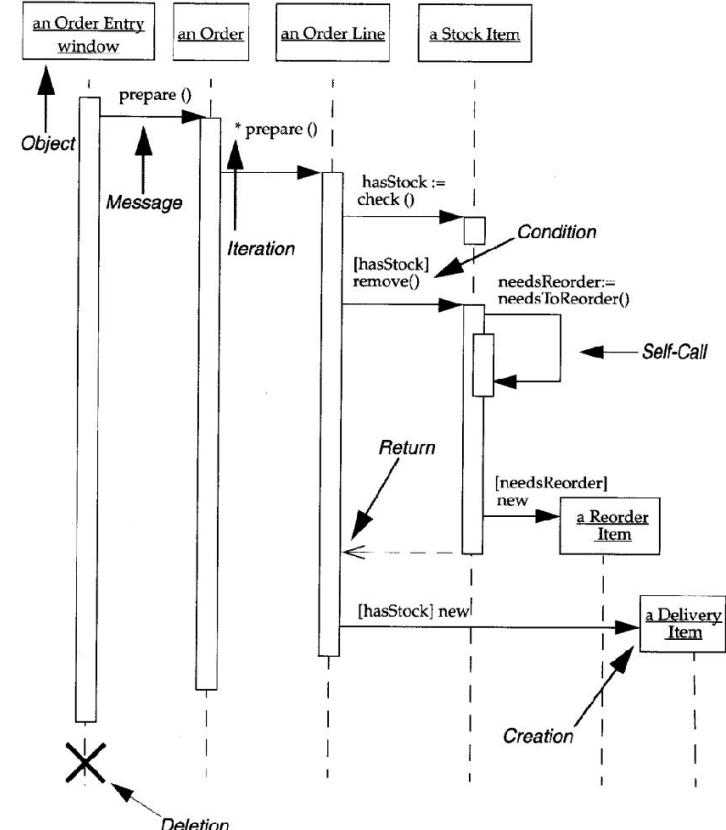
- Use boxes with 2 or 3 parts for fields, methods as appropriate for classes/objects, interfaces, concepts
- Include types for fields and methods
- Use associations, not fields, where appropriate
- Use association names and cardinalities (we don't care about arrow types, except "is-a")

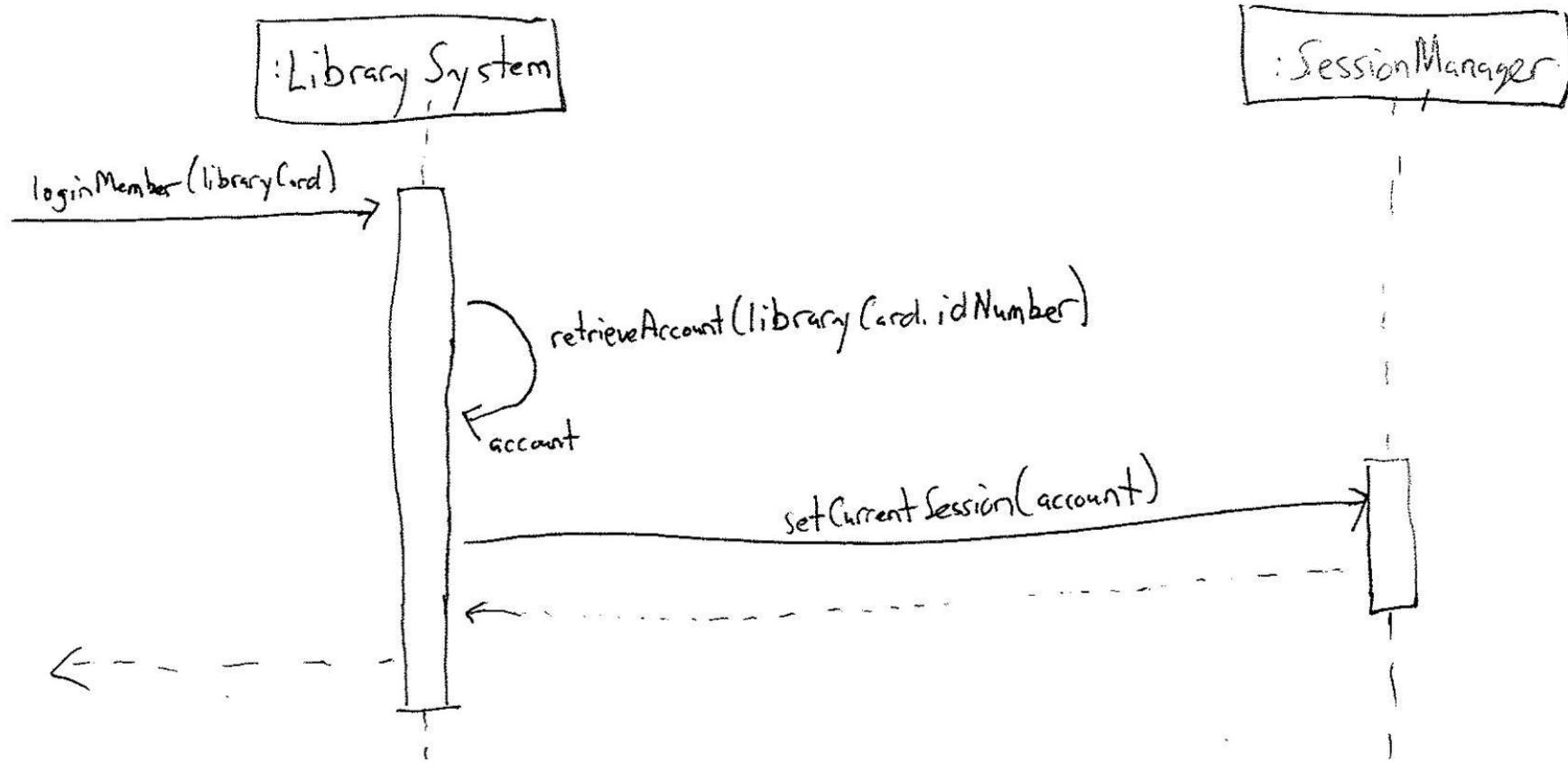
# Interaction Diagrams

Interactions between objects

Two common notations: sequence diagrams and communication diagrams

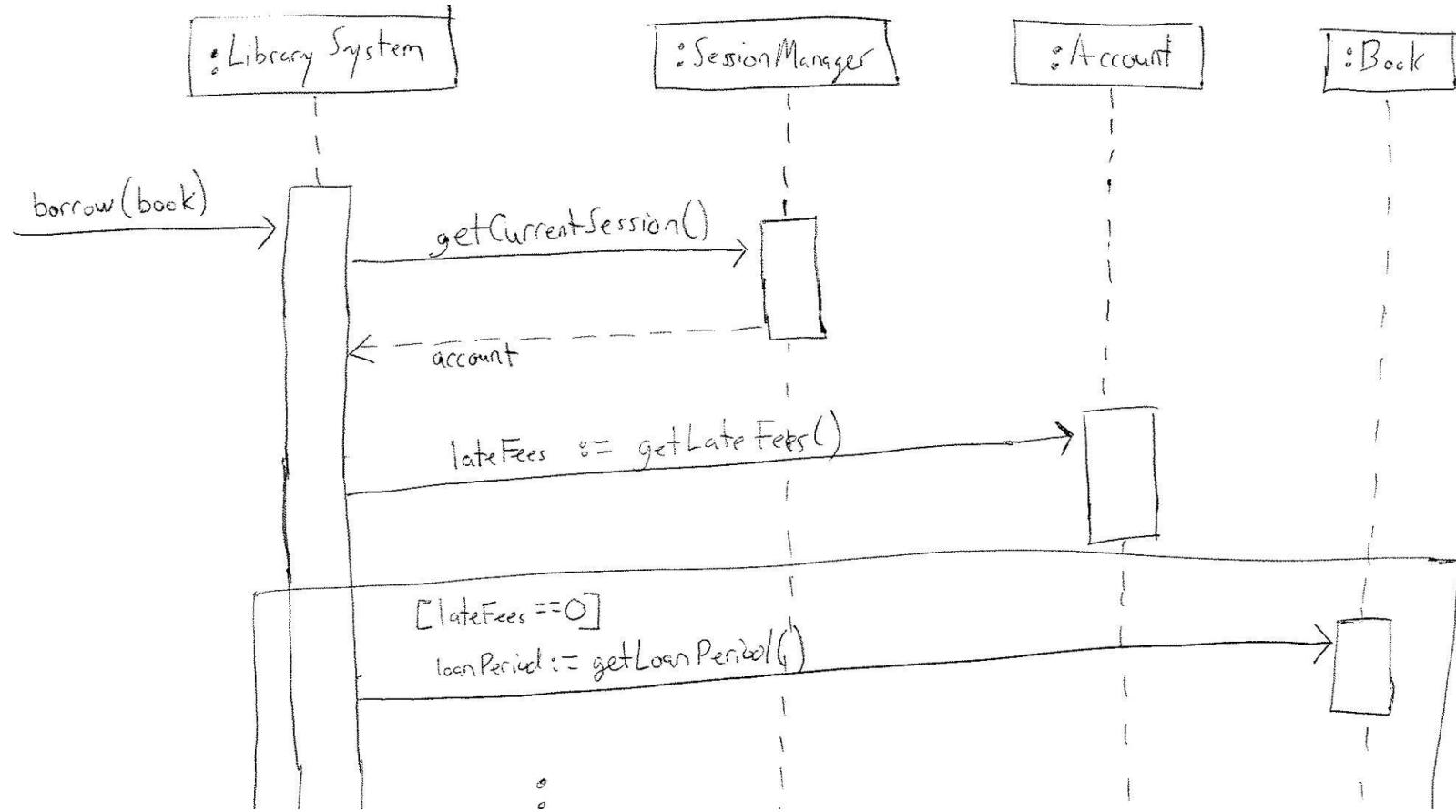
Sequence diagrams like system sequence diagrams, but depicting interactions between objects/classes





# Interaction Diagram Practice:

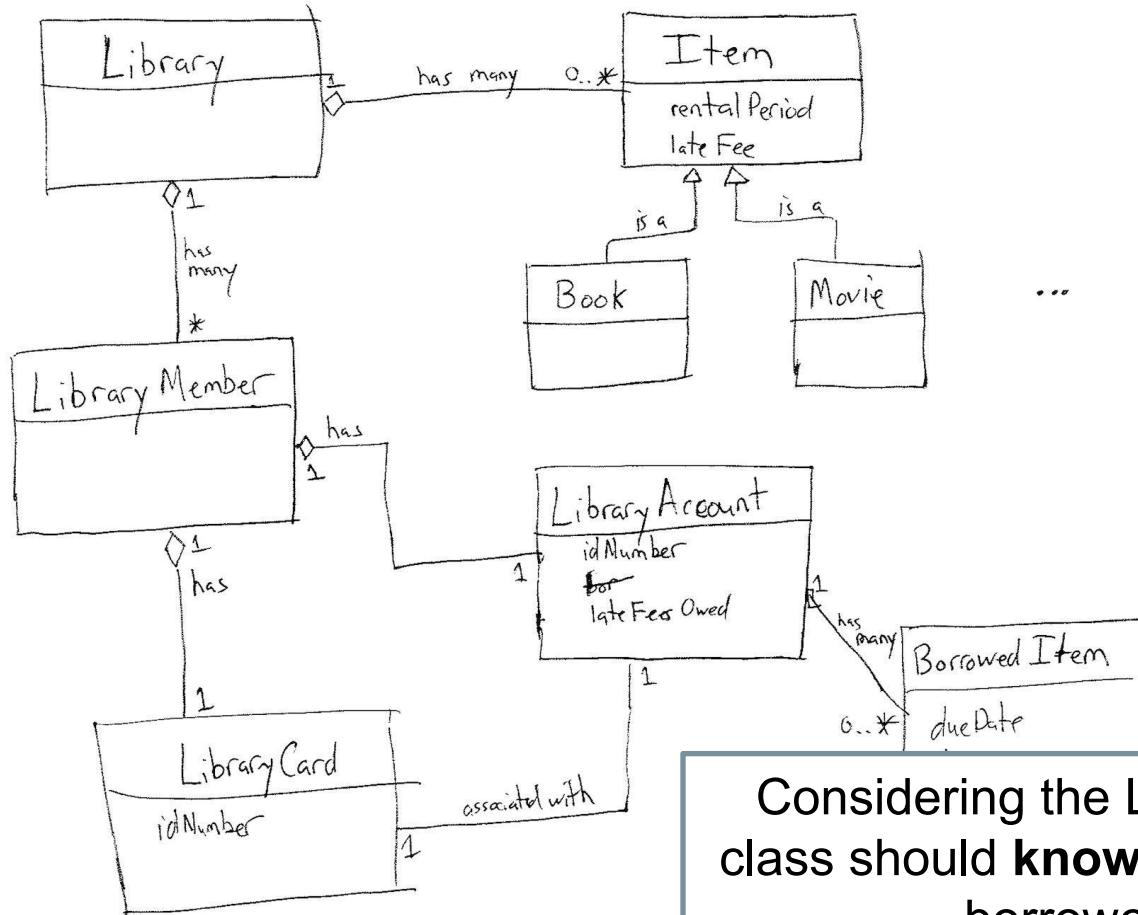
Use case scenario: ...and borrow a book. After confirming that the member has no unpaid late fees, the library system should determine the book's due date by adding its loan period to the current day, and record the book and its due date as a borrowed item in the member's library account.



# 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

# Object-Level Design



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*

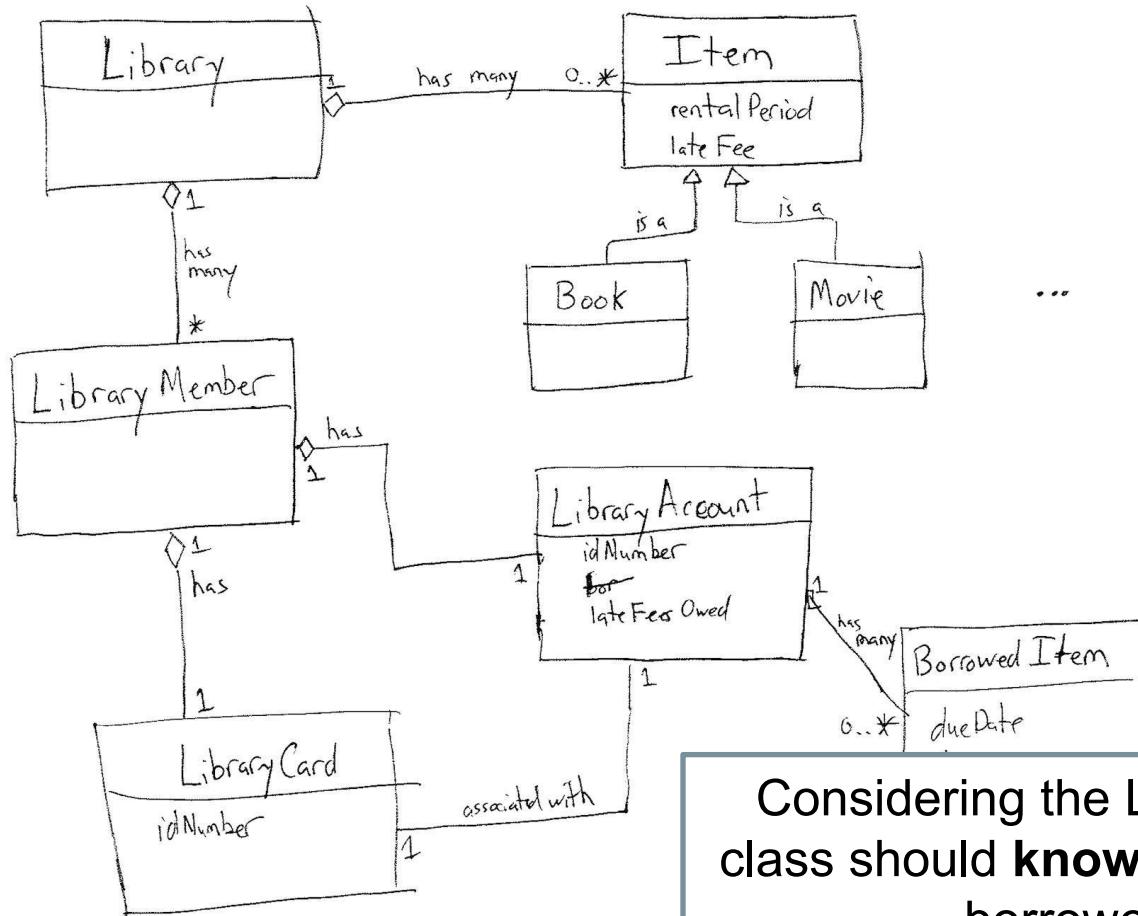
Responsibilities are related to the obligations of an object in terms of its behavior.

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:

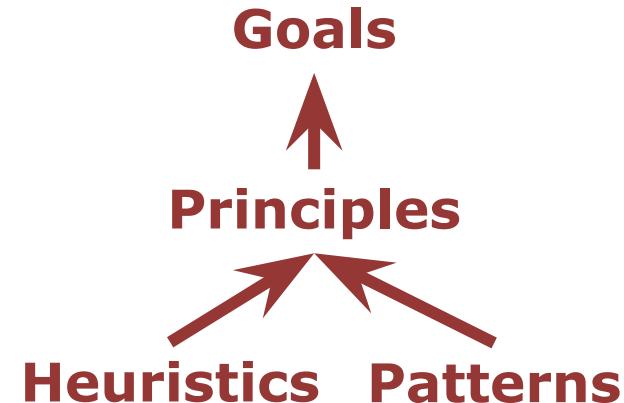
- knowing about private encapsulated data
- knowing about related objects
- knowing about things it can derive or calculate



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

# Design Goals, Principles, and Patterns

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

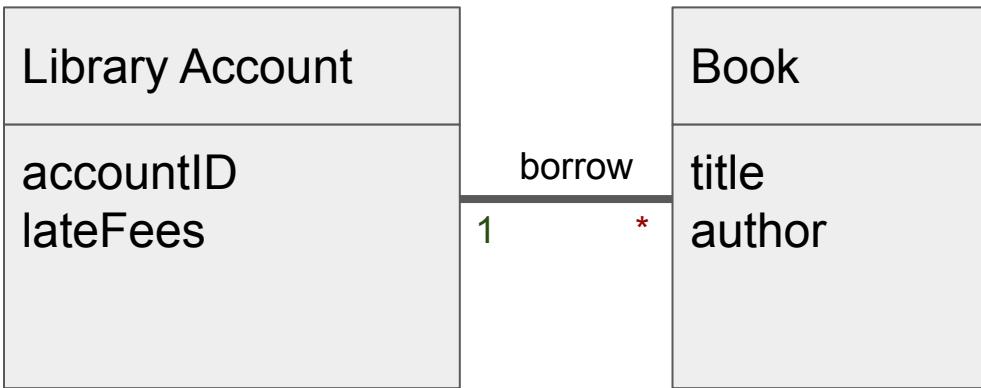


# **Design Heuristic: Low Representational Gap**

# 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();

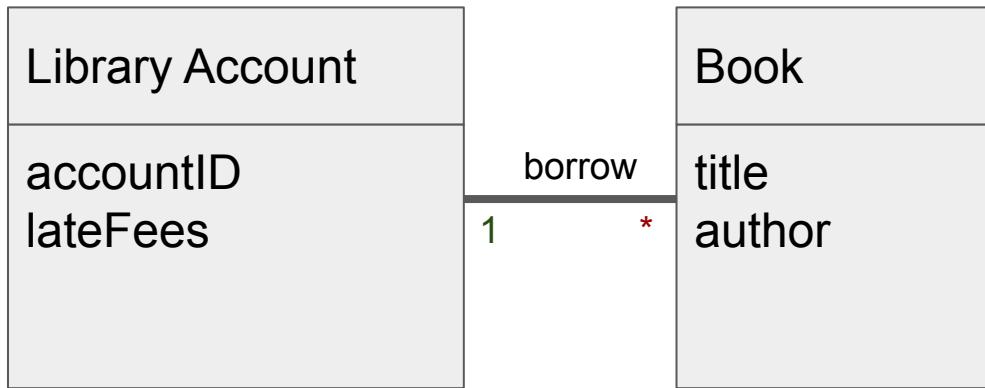
}

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 { ... }
```

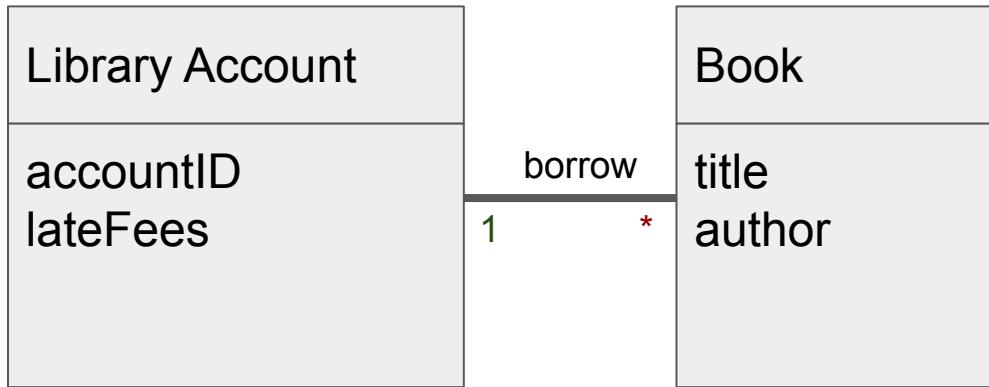
# Designs with Low Representational Gap

- 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

# Problem Space

(Domain Model)

Intuitive  
(low representational gap)



# Solution Space

(Object Model)

```
lateFees: Int;
borrowed: List<Book>;
boolean borrow(Book) { ... }

void save();

}

class Book { ... }
```

# **DESIGN PRINCIPLE: LOW COUPLING**

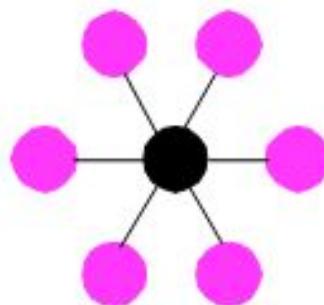
# Design Principle: Low Coupling

A module should depend on as few other modules as possible

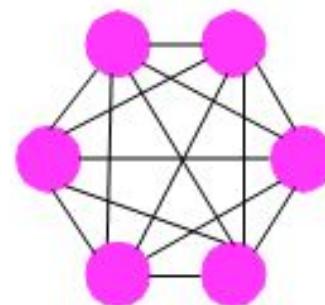
- Enhances understandability (design for underst.)
  - Limited understanding of context, easier to understand in isolation
- Reduces the cost of change (design for change)
  - Little context necessary to make changes
  - When a module interface changes, few modules are affected (reduced rippling effects)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

# Topologies with different coupling

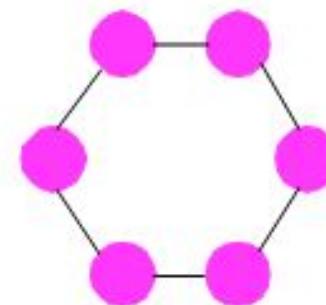
*Types of module  
interconnection  
structures*



(A)



(B)



(C)

# High Coupling is undesirable

- Element with low coupling depends on 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

```
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 classes are coupled?  
How can coupling be improved?**

**Draw an interaction diagram to illustrate what's happening.**

```
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); }  
}
```

**How can coupling be improved?**

# 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() ...
```

# 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 week:) 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

# Coupling to “non-standards”

- Libraries or platforms may include non-standard features or extensions
- Example: JavaScript support across Browsers
  - <div id="e1">old content</div>
- In JavaScript...
  - MSIE: e1.innerText = “new content”
  - Firefox: e1.textContent = “new content”

**W3C-compliant  
DOM standard**

# Design Goals

- Explain how low coupling supports
  - design for change
  - design for understandability
  - design for division of labor
  - design for reuse
  - ...

# Design Goals

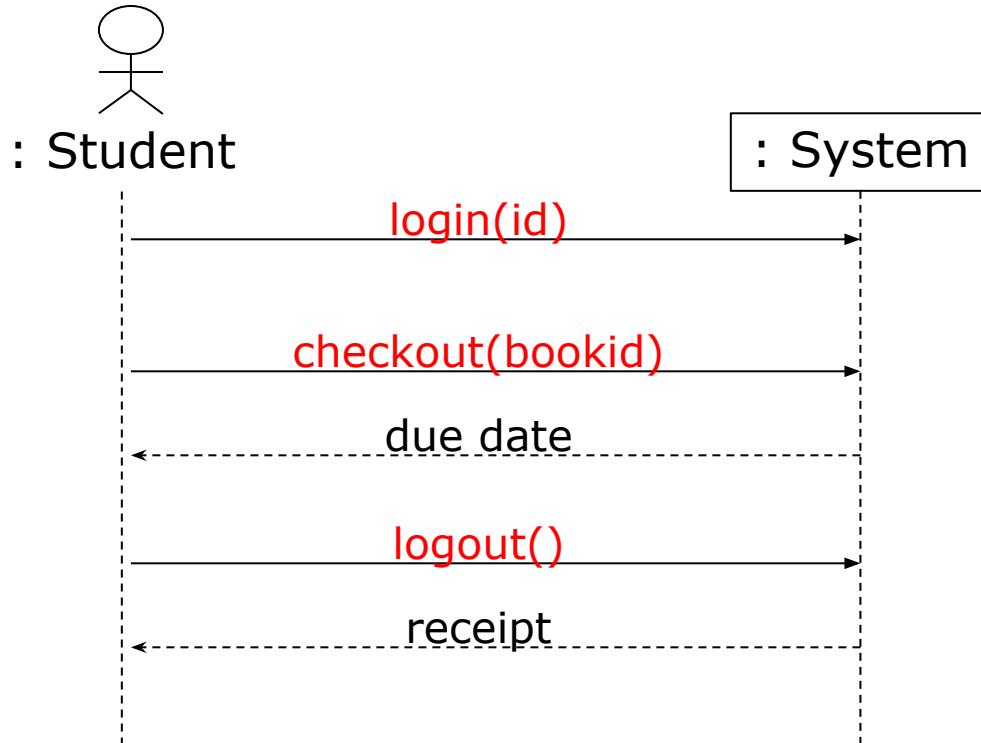
- design for change
  - changes easier because fewer dependencies on fewer other objects
  - changes are less likely to have rippling effects
- design for understandability
  - fewer dependencies to understand (e.g., a.getB().getC().foo())
- design for division of labor
  - smaller interfaces, easier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

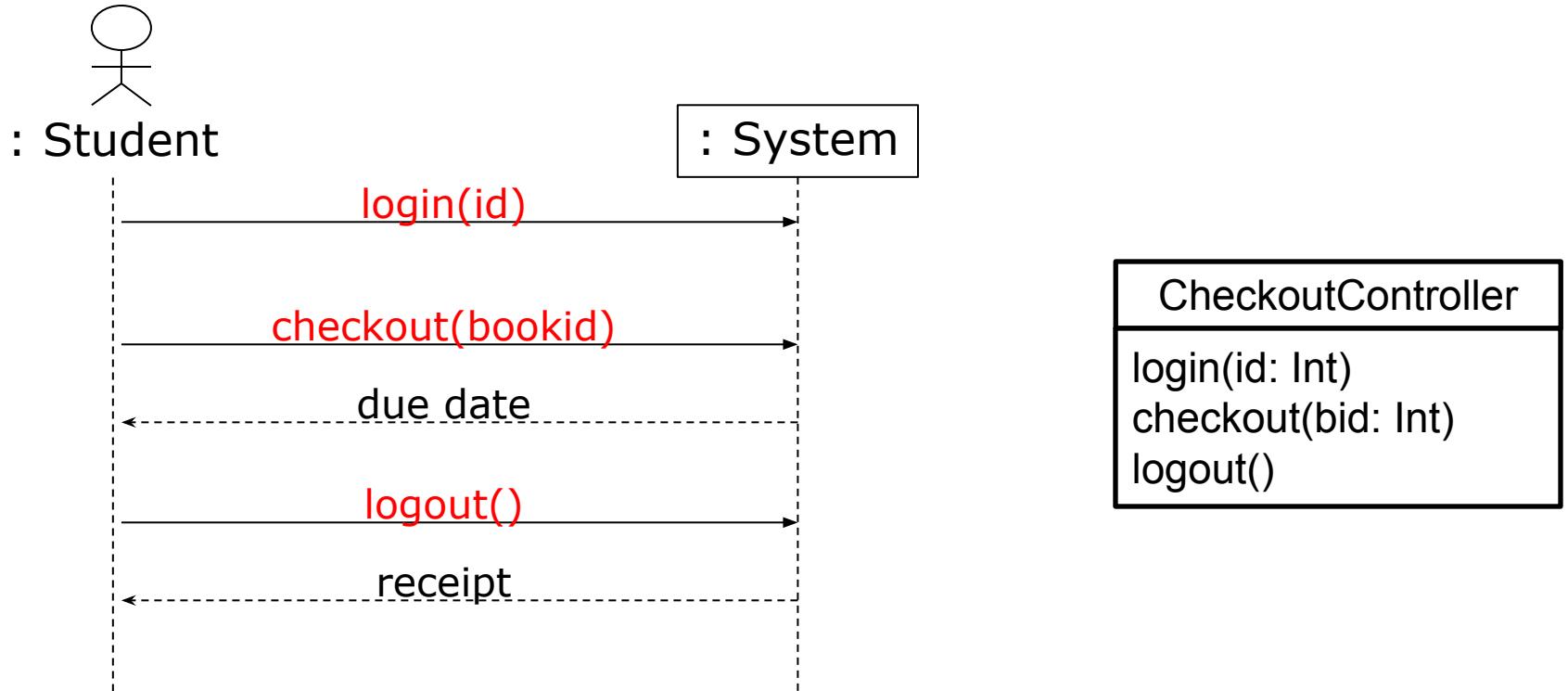
# **Design Heuristic: CONTROLLER**

**(also DESIGN PATTERN: FAÇADE )**

# Controller (Design Heuristic)

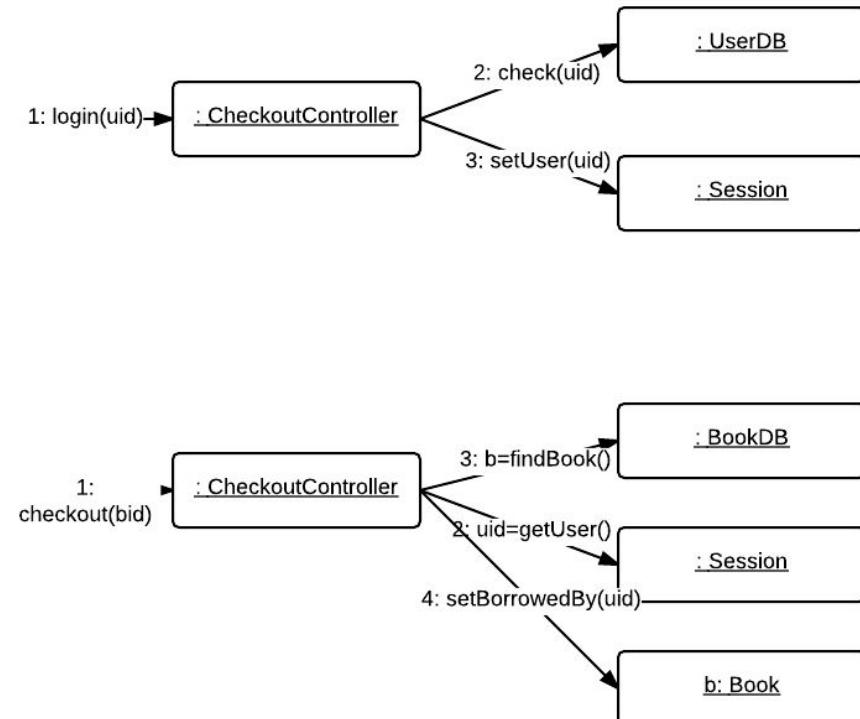
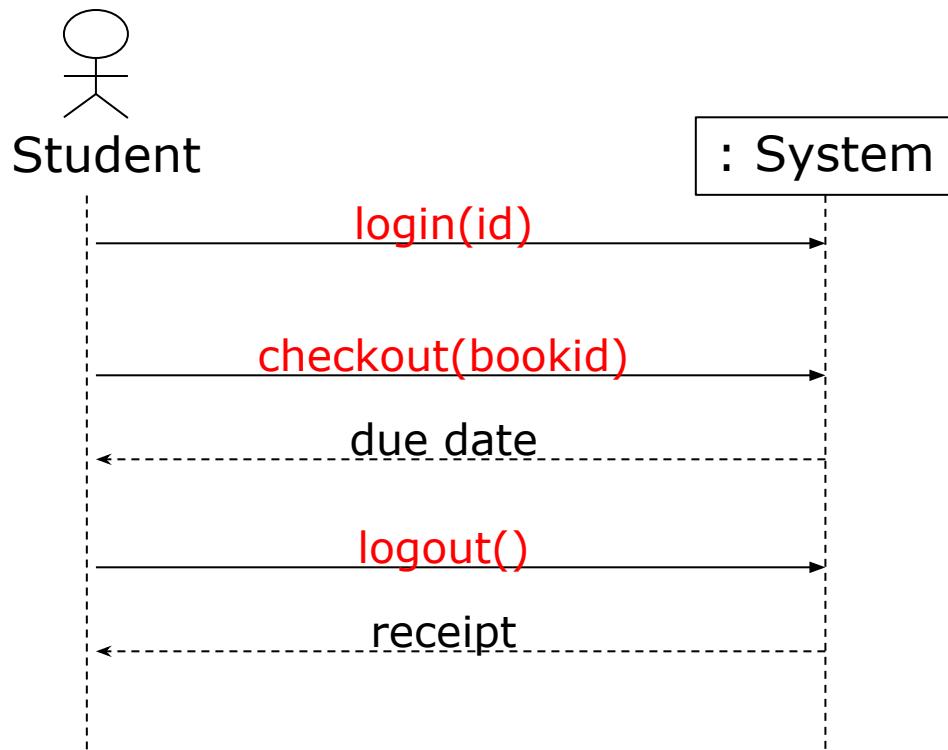
- Problem: What object receives and coordinates a system operation (event)?
- Solution: Assign the responsibility to an object representing
  - the overall system, device, or subsystem (façade controller), or
  - a use case scenario within which the system event occurs (use case controller)
- Process: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





# 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

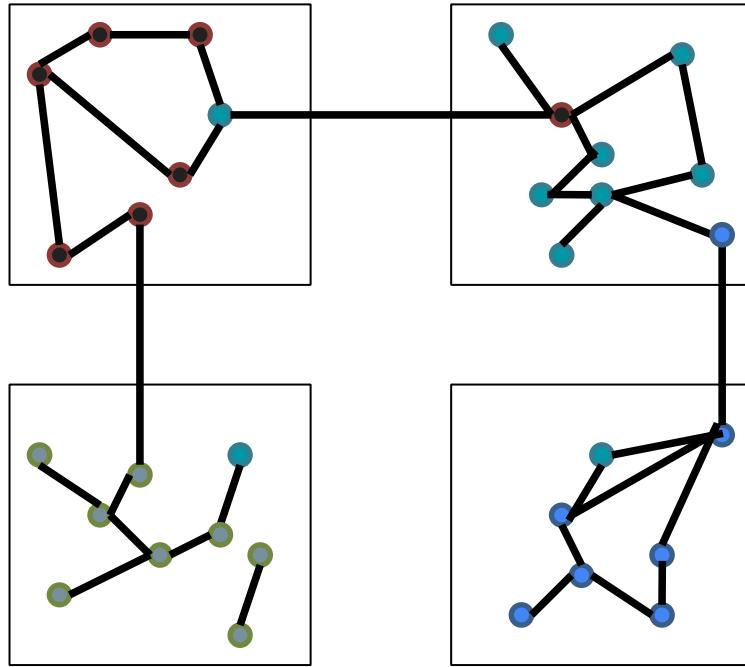
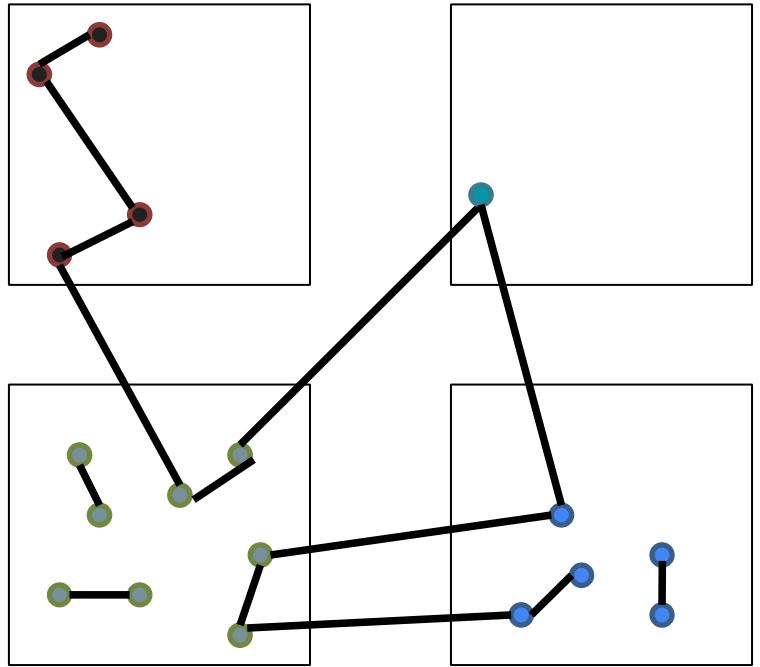
# Controller in Flash Cards Project?

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

```
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;
```

# Façade vs God Object?

# 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 is a “concept”
- 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)

Think about extreme cases:

- Very low coupling?
- Very high cohesion?

```
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;  
    }  
}
```

# 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

# **Design Heuristic: INFORMATION EXPERT**

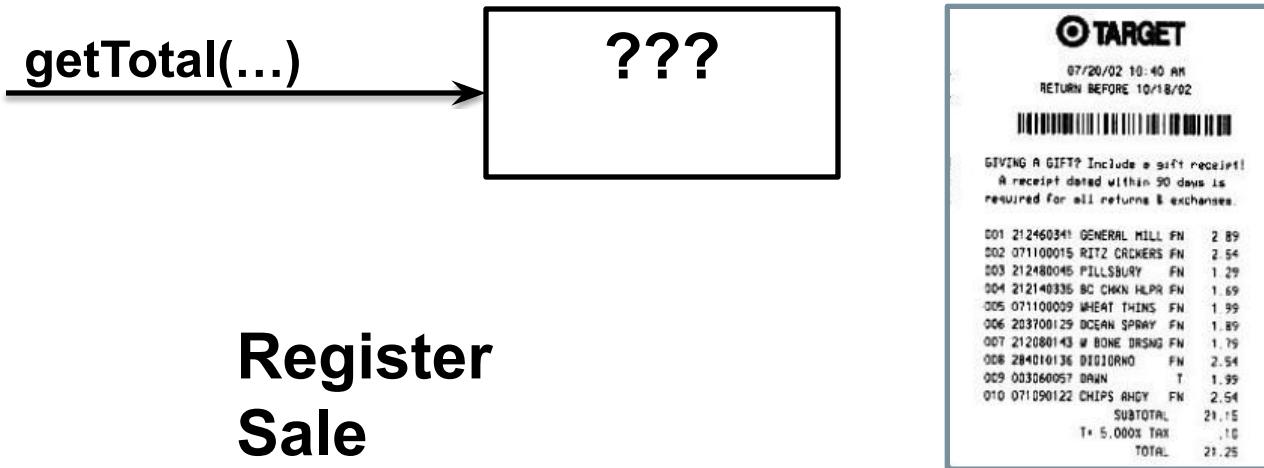
# Information Expert (Design Heuristic)

- 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;  
    }  
}
```

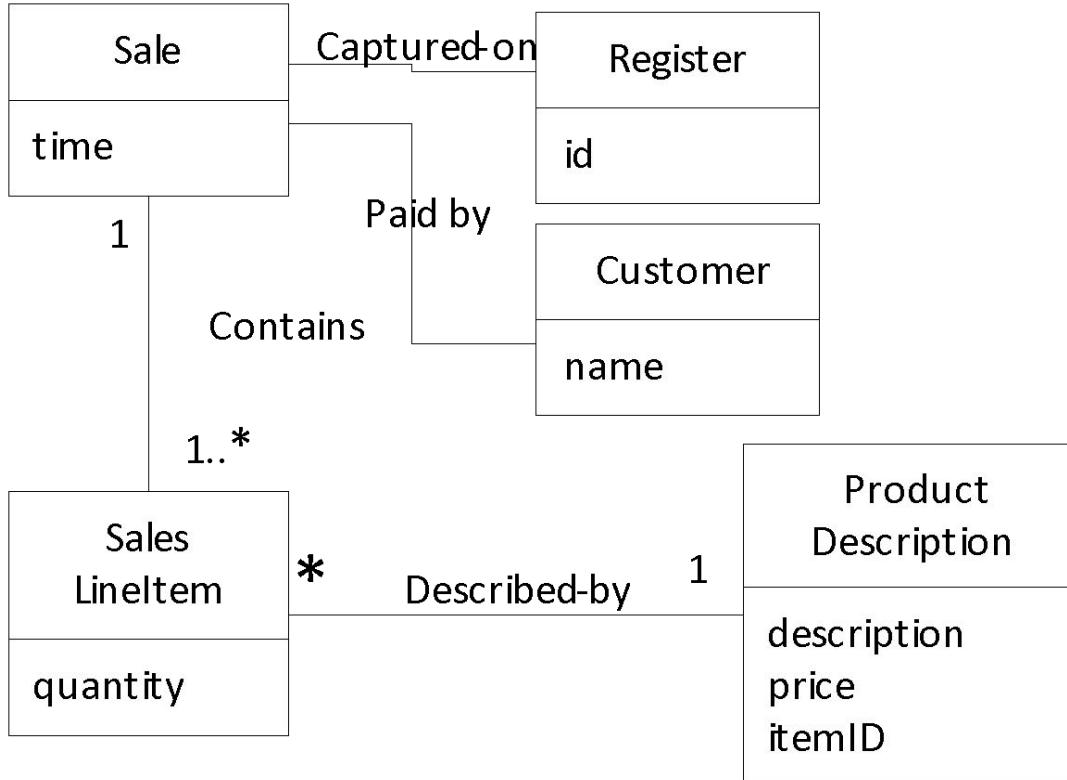
**Which class has all the information  
to compute the shipment's weight?**

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

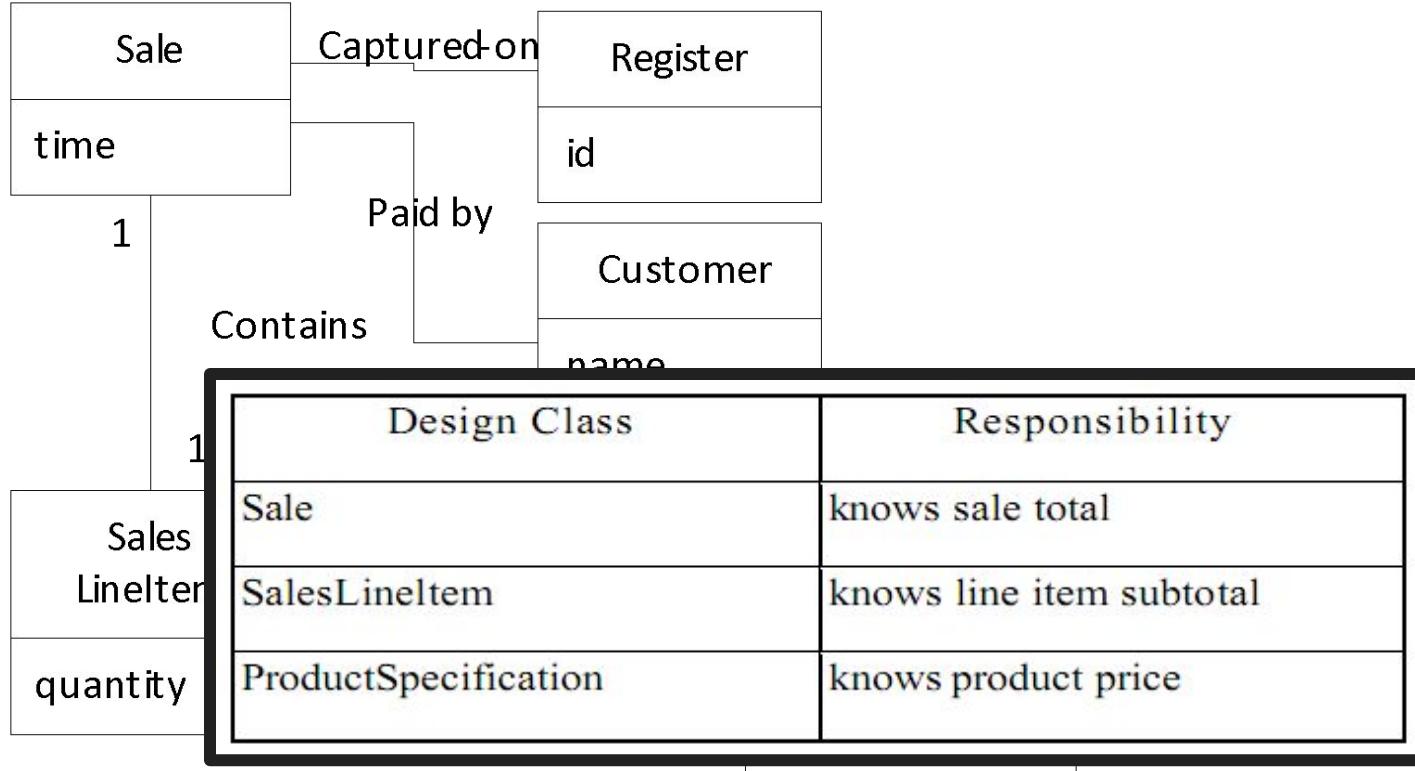


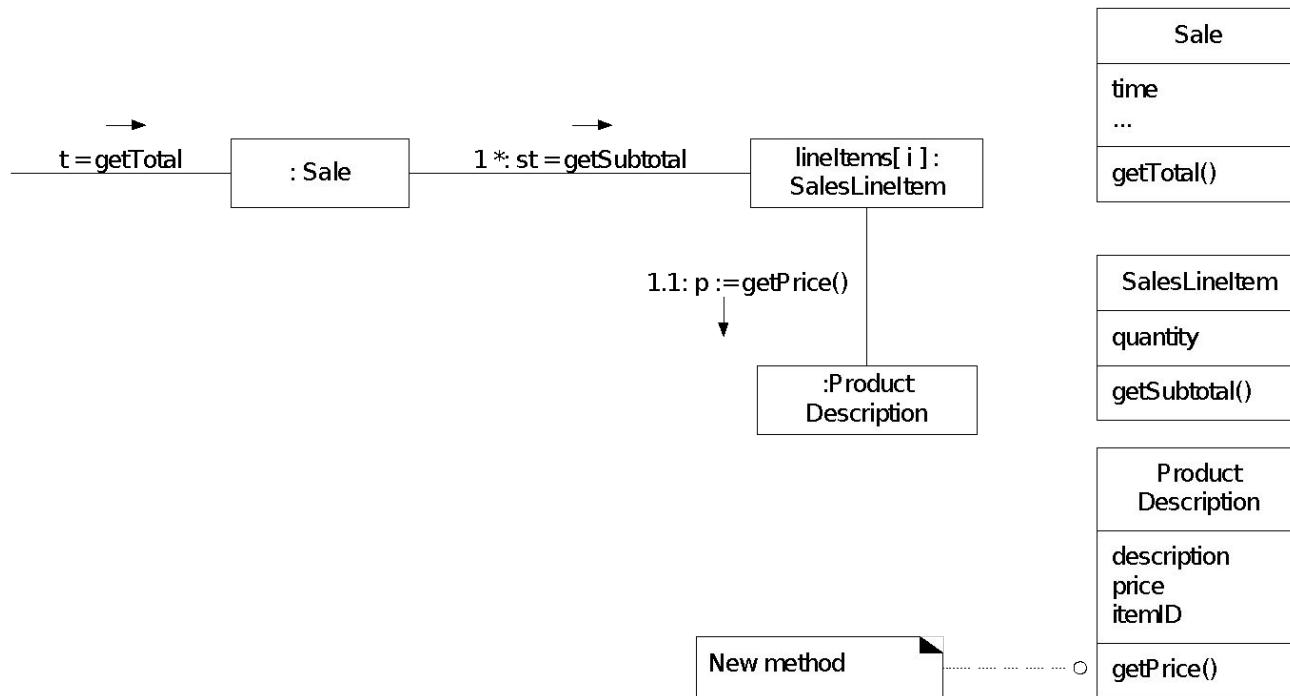
**Register  
Sale  
LineItem  
Product Descr.**

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



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





# 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 Expert in Flash Cards Prj.

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

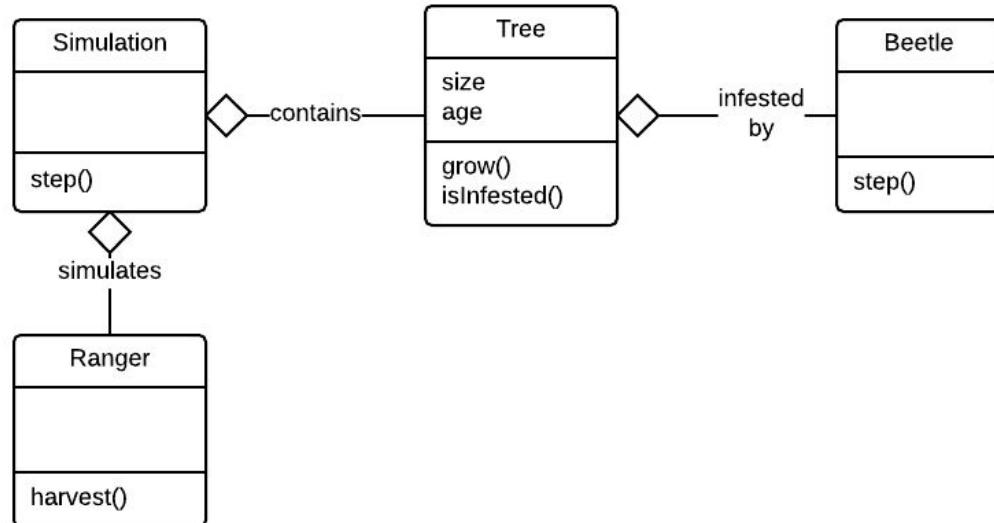
**Process:** Extract from domain model, interaction diagrams (key principles: Low coupling and low representational gap)

# Creator heuristic

- Design process: Extract from domain model, interaction diagrams
  - Key principles: Low coupling and low representational gap

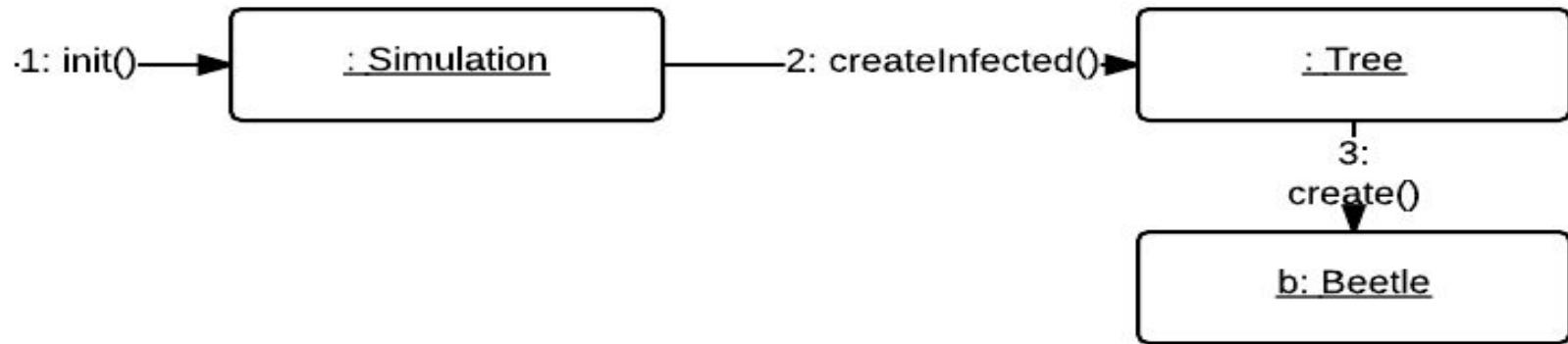
# Creator (GRASP)

- Who is responsible for **creating** Beetle objects?  
Tree 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*

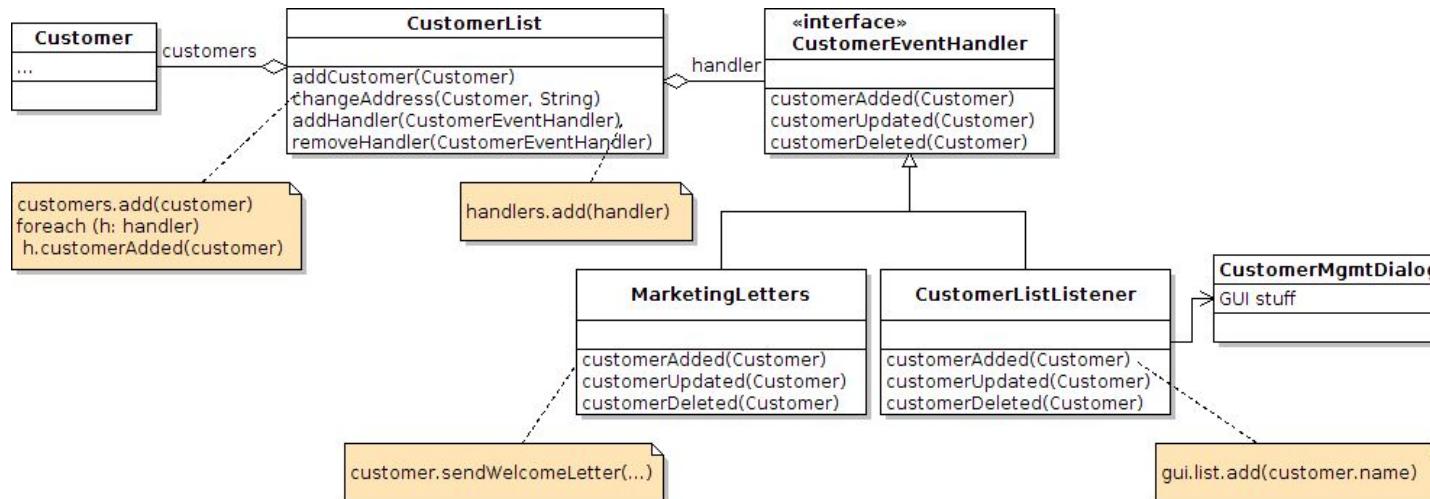
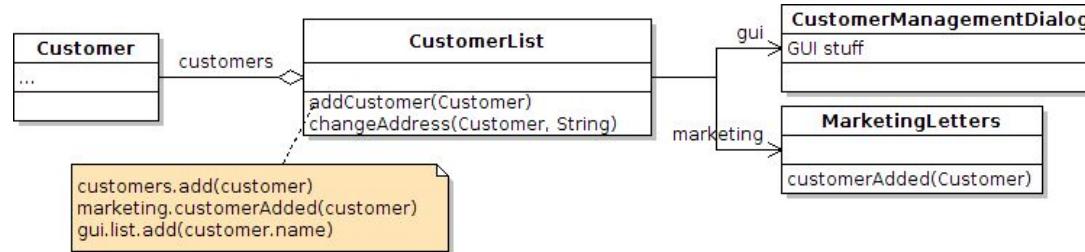
# Creator in Flash Cards Project

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



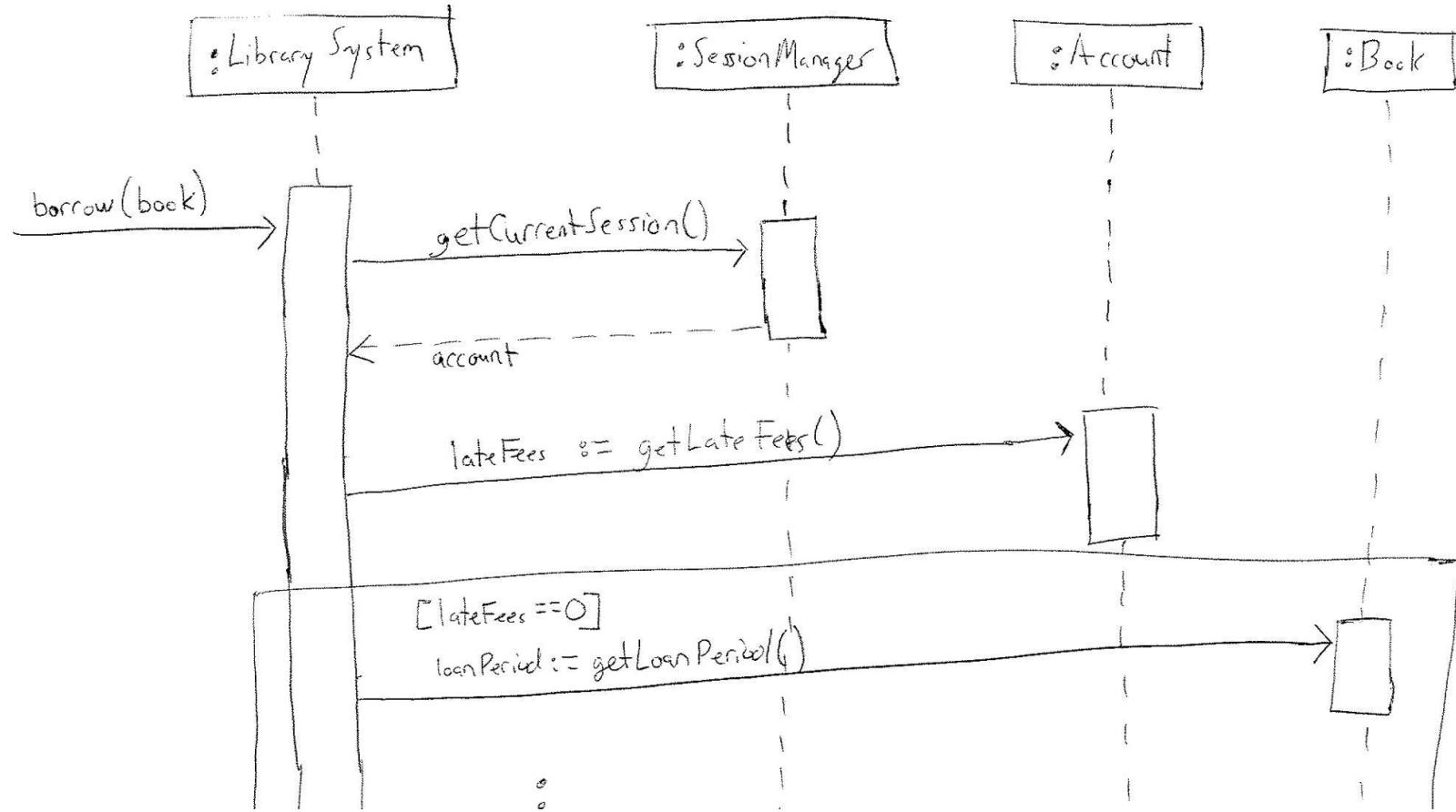
# Other Design Heuristics

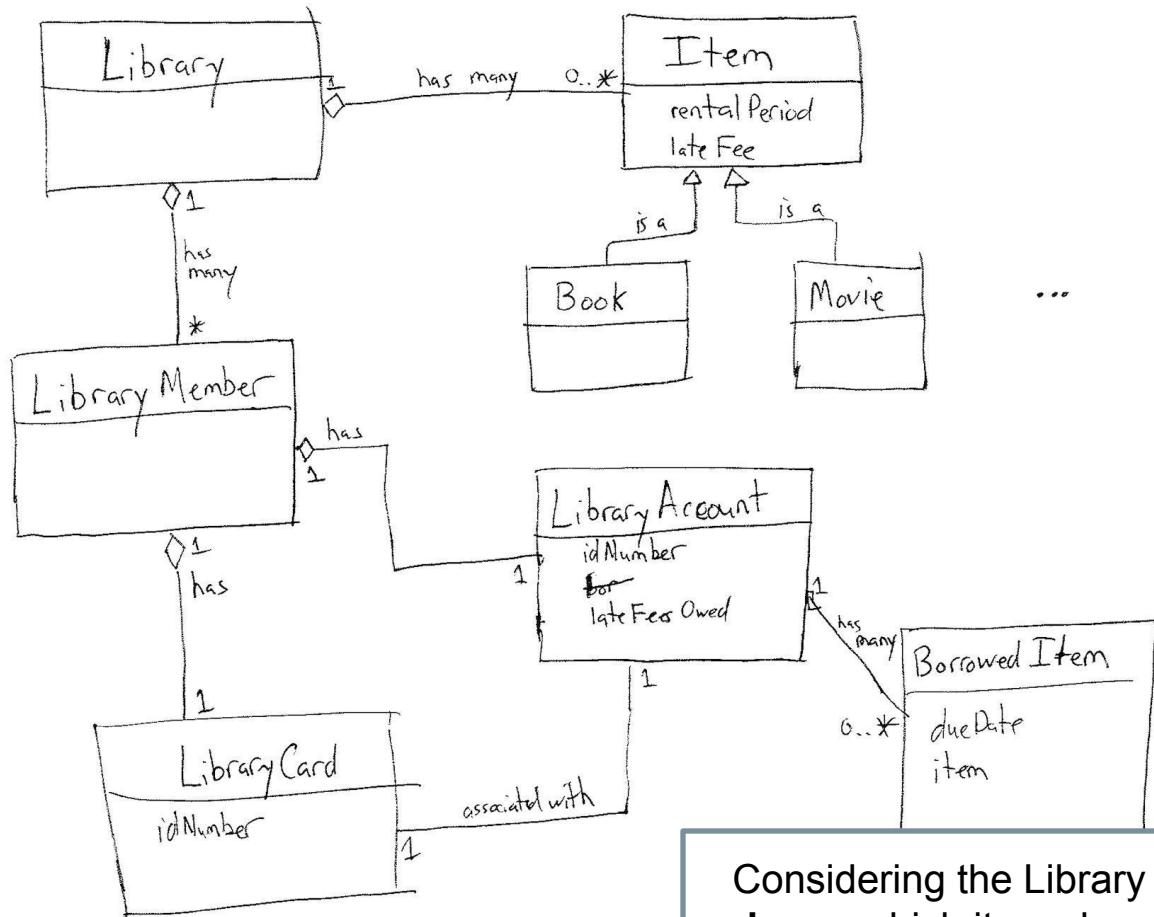
In future lectures:

- 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

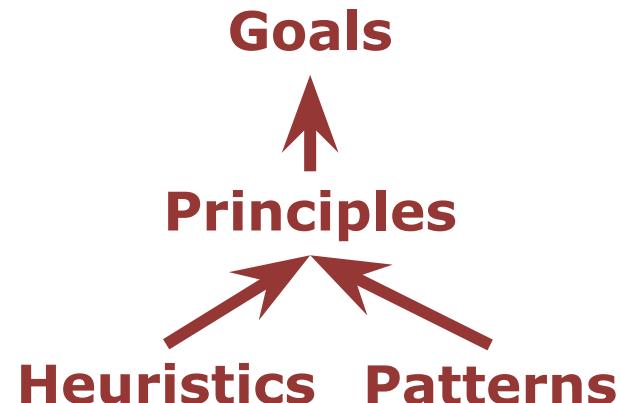




Considering the Library problem, which class should know which items have been borrowed by a user?  
 Which should compute late fees?

# Design Goals, Principles, and Patterns

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



# HW 2 Feedback

We try to improve...

Feedback optional,  
but appreciated

Participation bonus points



<https://bit.ly/3nPIqBx>

# 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, ...

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

## Design Patterns

Christian Kästner      Vincent Hellendoorn



# Administrativa

HW1 grades out, use Gradescope regrade request mechanism if needed  
(with concrete reference to rubric)

Midterm next week Thursday during class

- Covers all lecture, recitation, and reading content with emphasis on topics you had opportunities to practice
- Coding, design, and discussion questions
- Bring up to 4 pages of notes (both sides, handwritten or printed)

Reading on design patterns

# HW2 Feedback

Thanks for your responses!

Ambiguous specifications? Clarifications needed on Piazza? What do real specifications look like?

Too many tests? For EVERY method? Testing constructors and trivial methods?  
How many tests would you write in practice?

“Were our expectations clearly described in the rubric? (*if not, please suggest improvements*)”

Other: More interactions in recitations. Solutions for recitations. Classes in Java.  
Heavy Readings.



# One design scenario

- Amazon.com processes millions of orders each year, selling in 75 countries, all 50 states, and thousands of cities worldwide. These countries, states, and cities have hundreds of distinct sales tax policies and, for any order and destination, Amazon.com must be able to compute the correct sales tax for the order and destination.

# Another design scenario

- A vision processing system must detect lines in an image. For different applications the line detection requirements vary. E.g., for a vision system in a driverless car the system must process 30 images per second, but it's OK to miss some lines in some images. A face recognition system can spend 3-5 seconds analyzing an image, but requires accurate detection of subtle lines on a face.

# A third design scenario

- Suppose we need to sort a list in different orders...

```
const ASC = function(i: number, j: number): boolean {
    return i < j;
}

const DESC = function(i: number, j: number): boolean {
    return i > j;
}

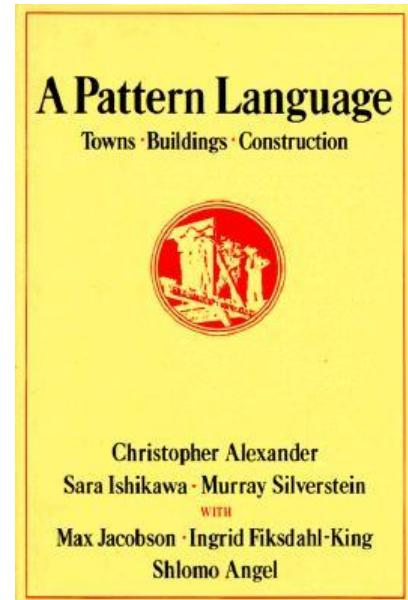
function sort(
    list: number[],
    order: (number, number) => boolean) {
    // ...
    boolean mustSwap = order(list[j], list[i]);
    // ...
}
```

# Design Patterns

# *Design patterns*

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”

– Christopher Alexander,  
Architect (1977)



# How not to discuss design (from Shalloway and Trott)

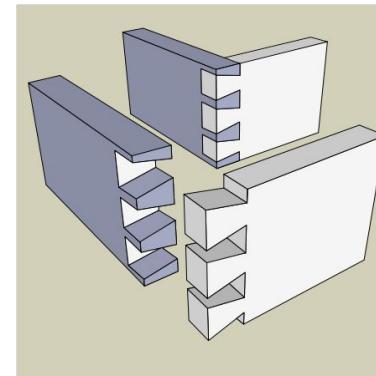
- Carpentry:
  - How do you think we should build these drawers?
  - Well, I think we should make the joint by cutting straight down into the wood, and then cut back up 45 degrees, and then going straight back down, and then back up the other way 45 degrees, and then going straight down, and repeating...

# How not to discuss design (from Shalloway and Trott)

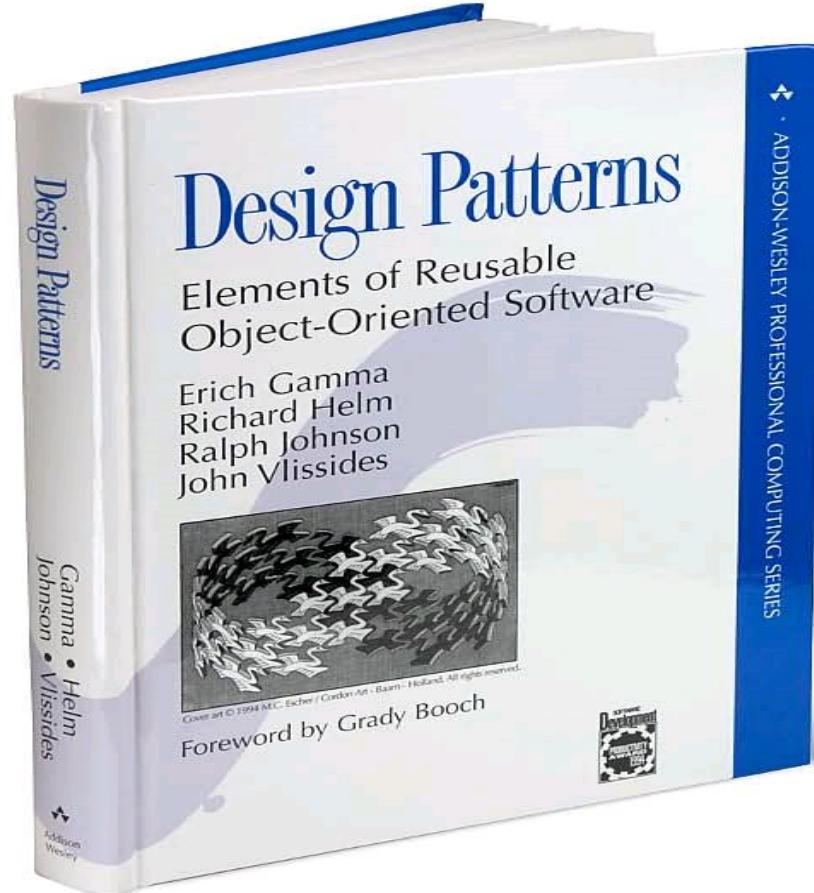
- Software Engineering:
  - How do you think we should write this method?
  - I think we should write this if statement to handle ... followed by a while loop ... with a break statement so that...

# Discussion with design patterns

- Carpentry:
  - "Is a dovetail joint or a miter joint better here?"
- Software Engineering:
  - "Is a strategy pattern or a template method better here?"



# History: *Design Patterns* (1994)



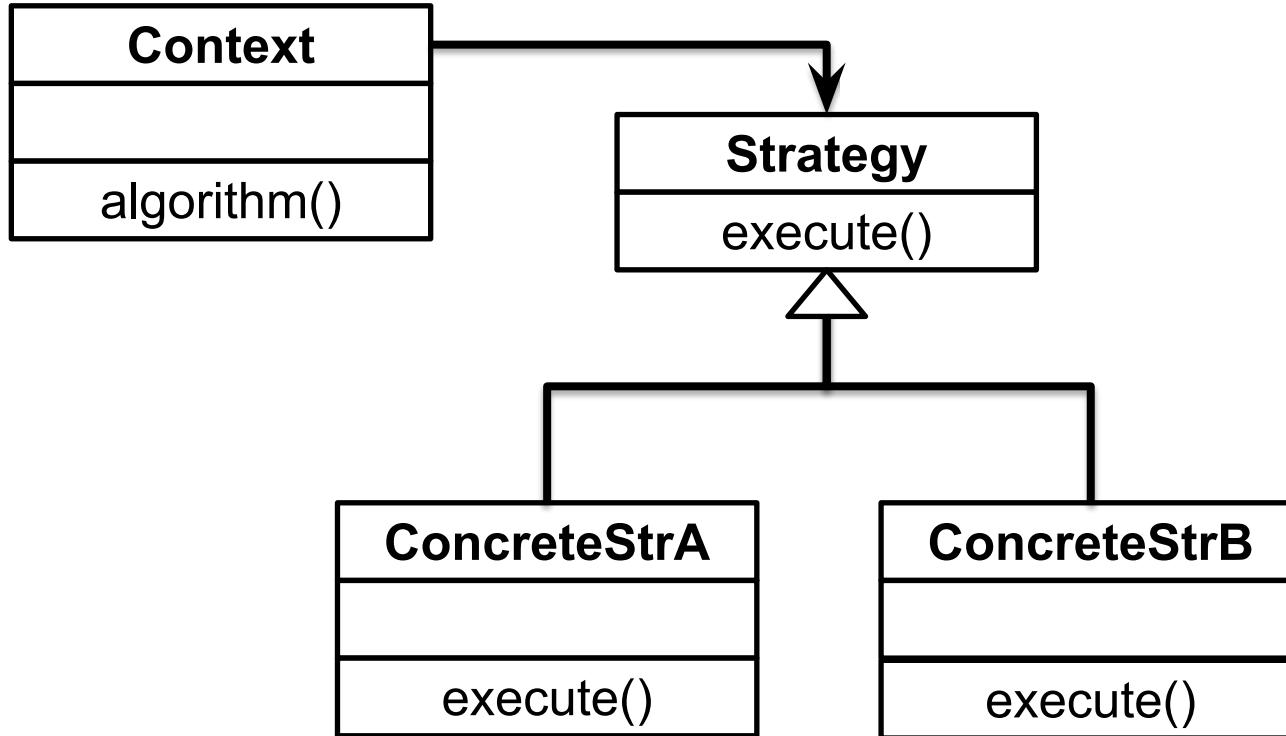
# Elements of a design pattern

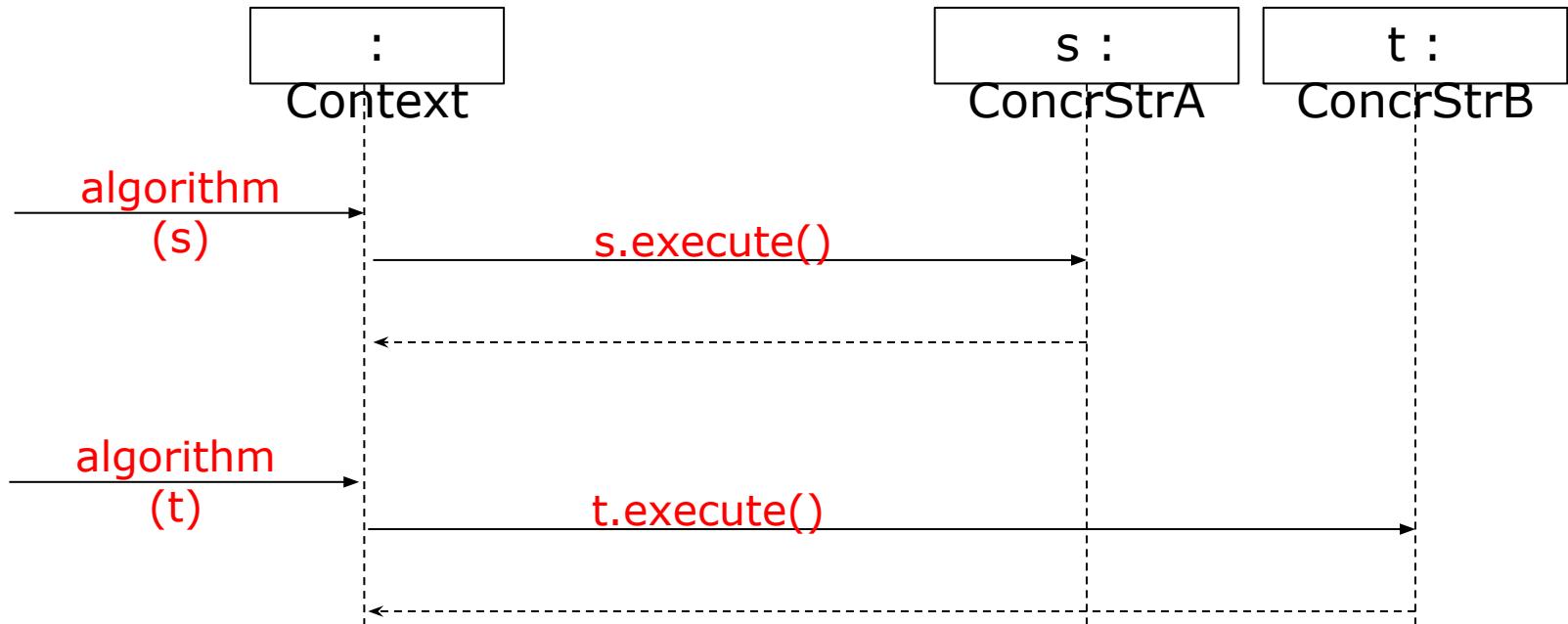
- Name
- Abstract description of problem
- Abstract description of solution
- Analysis of consequences

# Strategy Pattern

# Strategy pattern

- Problem: Clients need different variants of an algorithm
- Solution: Create an interface for the algorithm, with an implementing class for each variant of the algorithm
- Consequences:
  - Easily extensible for new algorithm implementations
  - Separates algorithm from client context
  - Introduces an extra interface and many classes: (1) Code can be harder to understand, (2) Lots of overhead if the strategies are simple





*Strategy can be provided in method call or in any other way to context*

# One design scenario

- Amazon.com processes millions of orders each year, selling in 75 countries, all 50 states, and thousands of cities worldwide. These countries, states, and cities have hundreds of distinct sales tax policies and, for any order and destination, Amazon.com must be able to compute the correct sales tax for the order and destination.

# Another design scenario

- A vision processing system must detect lines in an image. For different applications the line detection requirements vary. E.g., for a vision system in a driverless car the system must process 30 images per second, but it's OK to miss some lines in some images. A face recognition system can spend 3-5 seconds analyzing an image, but requires accurate detection of subtle lines on a face.

# Design Patterns and Programming Languages

Design patterns address general design challenges

Some patterns address problems with built-in solutions

Example: Strategy pattern vs higher-order functions

```
const ASC = function(i: number, j: number): boolean {
    return i < j;
}
const DESC = function(i: number, j: number): boolean {
    return i > j;
}
```

# Strategy Pattern vs Higher-Order Function

```
interface Order {  
    boolean lessThan(int i, int j);  
}  
  
class AscendingOrder implements Order {  
    public boolean lessThan(int i, int j) {  
        return i < j; }  
}  
  
class DescendingOrder implements Order {  
    public boolean lessThan(int i, int j) {  
        return i > j; }  
}  
  
void sort(int[] list, Order order) ;
```

```
const ASC =  
    function(i: number, j: number): boolean {  
        return i < j;  
    }  
const DESC =  
    function(i: number, j: number): boolean {  
        return i > j;  
    }  
  
function sort(  
    list: number[],  
    order: (number, number) => boolean) ...;
```

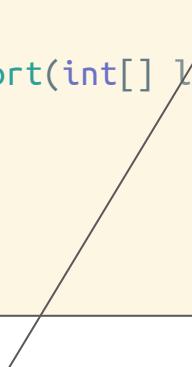
# Strategy Pattern vs Higher-Order Function

```
interface Order {  
    boolean lessThan(int i, int j);  
}  
  
class AscendingOrder implements Order {  
    public boolean lessThan(int i, int j) {  
        return i < j; }  
}  
  
class DescendingOrder implements Order {  
    public boolean lessThan(int i, int j) {  
        return i > j; }  
}  
  
void sort(int[] list, Order order) ;
```

```
const ASC = function(i, j) { return i < j; }  
const DESC = function(i, j) { return i > j; }  
  
function sort(list, order) ...;
```

# New Java Syntax for “Functions”

```
interface Order {  
    boolean lessThan(int i, int j);  
}  
  
final Order ASCENDING = (i, j) -> i < j;  
final Order DESCENDING = (i, j) -> i > j;  
  
static void sort(int[] list, Order order);
```



```
const ASC =  
    function(i: number, j: number): boolean {  
        return i < j;  
    }  
const DESC =  
    function(i: number, j: number): boolean {  
        return i > j;  
    }  
  
function sort(  
    list: number[],  
    order: (number, number) => boolean) ...;
```

Convenient syntax (introduced for lambdas) to create objects of interface with single method.

# Module Pattern

# Module pattern: Hide internals in closure

```
(function () {  
    // ... all vars and functions are in this scope only  
    // still maintains access to all globals  
}());
```

Function provides local scope, internals not accessible

Function directly invoked to execute it once

Wrapped in parentheses to make it expression

Discovered around 2007, became very popular, part of Node

# Using closures to hide methods and fields

```
function createPolarPoint(len, angle) {  
    let xcache = -1;  
    let internalLen=len;  
    function computeX() {...}  
    return {  
        getX: function() {  
            computeX(); return xcache; },  
        getY: function() {  
            return len * sin(angle); }  
    };  
}
```

# Module pattern: Decide what to export

```
var MODULE = (function () {
    var my = {},
        privateVariable = 1;

    function privateMethod() {
        // ...
    }

    my.moduleProperty = 1;
    my.moduleMethod = function () {
        // ...
    };

    return my;
}());
```

# Java: Module Pattern?

Public/private built in, problem does not exist

Fully qualified names (“edu.cmu.cs17214.FlashCard”) as convention/pattern to solve naming clashes

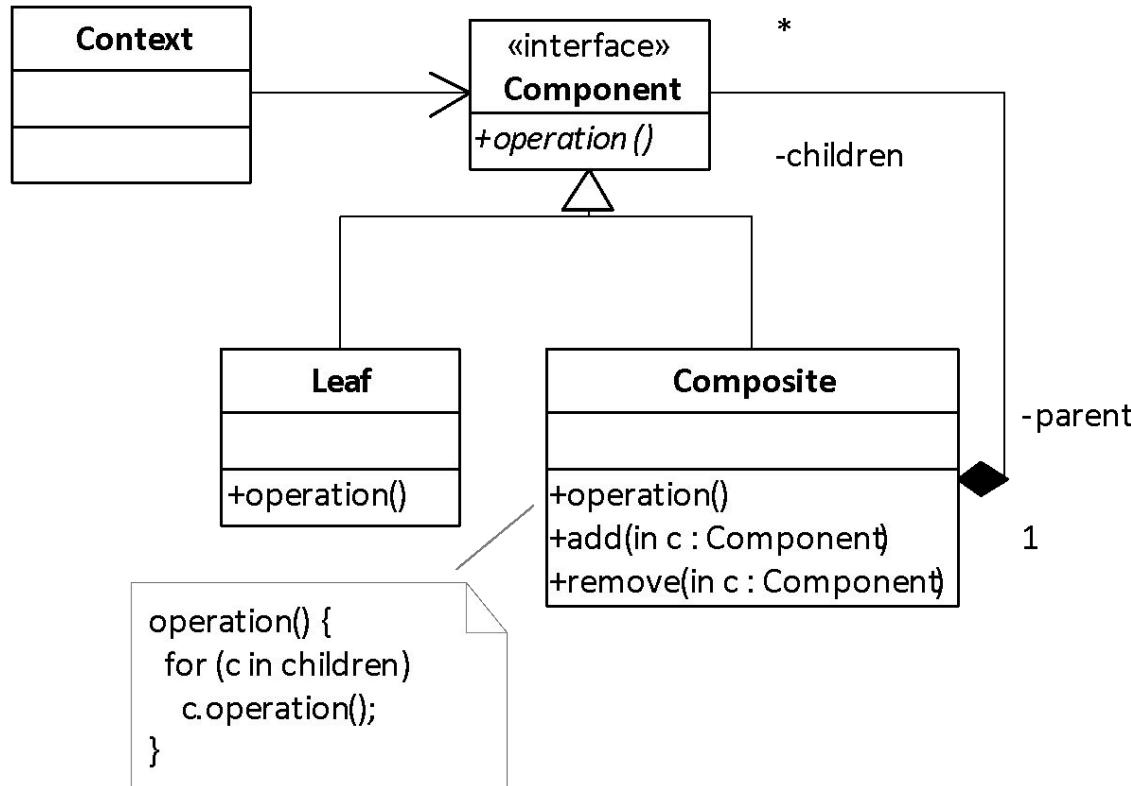
Never JavaScript/TypeScript features make it less important (ES6 modules, classes, public/private)

# Composite Pattern

# Design Exercise (on paper)

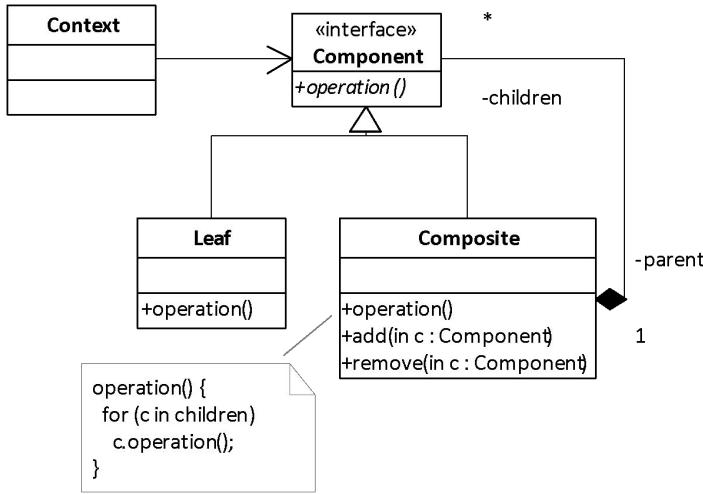
- You are designing software for a shipping company.
- There are several different kinds of items that can be shipped: letters, books, packages, fragile items, etc.
- Two important considerations are the **weight** of an item and its **insurance cost**.
  - Fragile items cost more to insure.
  - All letters are assumed to weigh an ounce
  - We must keep track of the weight of other packages.
- The company sells **boxes** and customers can put several items into them.
  - The software needs to track the contents of a box (e.g. to add up its weight, or compute the total insurance value).
  - However, most of the software should treat a box holding several items just like a single item.
- Think about how to represent packages; what are possible interfaces, classes, and methods? (letter, book, box only)

# The Composite Design Pattern



# The Composite Design Pattern

- Applicability
  - You want to represent part-whole hierarchies of objects
  - You want to be able to ignore the difference between compositions of objects and individual objects
- Consequences
  - Makes the client simple, since it can treat objects and composites uniformly
  - Makes it easy to add new kinds of components
  - Can make the design overly general
    - Operations may not make sense on every class
    - Composites may contain only certain components



# We have seen this before

```
interface Point {  
    int getX();  
    int getY();  
}  
  
class MiddlePoint implements Point {  
    Point a, b;  
    MiddlePoint(Point a, Point b) {this.a = a; this.b = b; }  
    int getX() { return (this.a.getX() + this.b.getX()) / 2; }  
    int getY() { return (this.a.getY() + this.b.getY()) / 2; }  
}
```

# Composite Pattern and Flash Cards?

# We have seen this before

```
function newCombinedCardOrganizer (cardOrganizers: CardOrganizer[]): CardOrganizer {  
    return {  
        reorganize: function (cards: CardStatus[]): CardStatus[] {  
            let status = cards.slice()  
            for (const cardOrganizer of cardOrganizers) {  
                status = cardOrganizer.reorganize(status)  
            }  
            return status  
        }  
    }  
}
```

# Fluent APIs / Cascade Pattern

# Setting up Complex Objects

Long constructors, lots of optional parameters, long lists of statements

```
Option find = OptionBuilder
    .withArgName("file")
    .hasArg()
    .withDescription("search...")
    .create("find");
```

```
client.getItem('user-table')
    .setHashKey('userId', 'userA')
    .setRangeKey('column', '@')
    .execute()
    .then(function(data) {
        ...
    })
```

# Liquid APIs

Each method changes state,

then returns **this**

(Immutable version:  
Return modified copy)

```
class OptBuilder {  
    private String argName = "";  
    private boolean hasArg = false;  
    ...  
    OptBuilder withArgName(String n) {  
        this.argName = n;  
        return this;  
    }  
    OptBuilder hasArg() {  
        this.hasArg = true;  
        return this;  
    }  
    ...  
    Option create() {  
        return new Option(argName,  
                          hasArgs, ...)  
    }  
}
```

# Python: Named parameters

```
parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')
```

# JavaScript: JSON Objects

```
var argv = require('yargs/yargs')(process.argv.slice(2))
  .option('size', {
    alias: 's',
    describe: 'choose a size',
    choices: ['xs', 's', 'm', 'l', 'xl']
  })
  .argv
```

Notice the combination of cascading and complex JSON parameters

# Fluent APIs: Discussion and Tradeoffs

Problem: Complex initialization and configuration

Advantages:

- Fairly readable code
- Can check individual arguments
- Avoid untyped complex arguments

Disadvantages:

- Runtime error checking of constraints and mandatory arguments
- Extra complexity in implementation
- Not always obvious how to terminate
- Possibly harder to debug

# Iterator Pattern & Streams

# Traversing a collection

- Since Java 1.0:

```
Vector arguments = ...;
for (int i = 0; i < arguments.size(); ++i) {
    System.out.println(arguments.get(i));
}
```

- Java 1.5: enhanced for loop

```
List<String> arguments = ...;
for (String s : arguments) {
    System.out.println(s);
}
```

- Works for every implementation of `Iterable`

```
public interface Iterable<E> {
    public Iterator<E> iterator();
}

public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

- In JavaScript (ES6)

```
let arguments = ...
for (const s of arguments) {
    console.log(s)
}
```

- Works for every implementation with a “magic” function `[Symbol.iterator]` providing an iterator

```
interface Iterator<T> {
    next(value?: any): IteratorResult<T>;
    return?(value?: any): IteratorResult<T>;
    throw?(e?: any): IteratorResult<T>;
}

interface IteratorReturnResult<TReturn> {
    done: true;
    value: TReturn;
}
```

# The Iterator Idea

Iterate over elements in arbitrary data structures (lists, sets, trees) without having to know internals

Typical interface:

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
}
```

(in Java also remove)

# Using an iterator

Can be used explicitly

```
List<String> arguments = ...;  
for (Iterator<String> it = arguments.iterator(); it.hasNext(); ) {  
    String s = it.next();  
    System.out.println(s);  
}
```

Often used with magic syntax:

```
for (String s : arguments)  
for (const s of arguments)
```

# Java: Getting an Iterator

```
public interface Collection<E> extends Iterable<E> {  
    boolean add(E e);  
    boolean addAll(Collection<? extends E> c);  
    boolean remove(Object e);  
    boolean removeAll(Collection<?> c);  
    boolean retainAll(Collection<?> c);  
    boolean contains(Object e);  
    boolean containsAll(Collection<?> c);  
    void clear();  
    int size();  
    boolean isEmpty();  
    Iterator<E> iterator();  
    Object[] toArray();  
    <T> T[] toArray(T[] a);  
    ...  
}
```

*Defines an interface for creating an Iterator,  
but allows Collection implementation to decide  
which Iterator to create.*

# Iterators for everything

```
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) { ... }
```

# Iterator design 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

# Iterator and FlashCards?

# Streams

Stream ~ Iterator -- a sequence of objects

Typically provide operations to produce new stream from old stream (map, flatMap, filter) and operations on all elements (fold, sum) -- using higher-order functions/strategy

Often provide efficient/parallel implementations  
(subtype polymorphism)

Built-in in Java since Java 8; basics in Node libraries in JavaScript

```
List<String>results = stream.map(Object::toString)
    .filter(s -> pattern.matcher(s).matches())
    .collect(Collectors.toList());
```

```
int sum = numbers.parallelStream().reduce(0, Integer::sum);
```

```
for (let [odd, even] in numbers.split(n => n % 2, n => !(n % 2)).zip()) {
  console.log(`odd = ${odd}, even = ${even}`); // [1, 2], [3, 4], ...
}
```

```
Stream<Person>.filter({age: 23}).flatMap("children").map("firstName")
    .distinct().filter(/a.*/i).join(", ");
```

# Design pattern conclusions

- Provide shared language
- Convey shared experience
- Can be system and language specific

