Objects Analysis Threads Design

15-214

Principles of Software Construction: Objects, Design, and Concurrency

Course Introduction

toad

Fall 2012

Jonathan Aldrich Charlie Garrod

School of Computer Science
institute for SOFTWARE RESEARCH

Construction of

Software Systems

at Scale

Libraries
Reuse
Design
Analysis
Concurrency

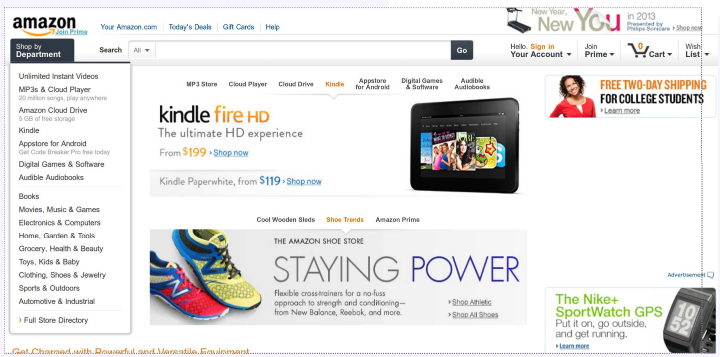
primes graph search

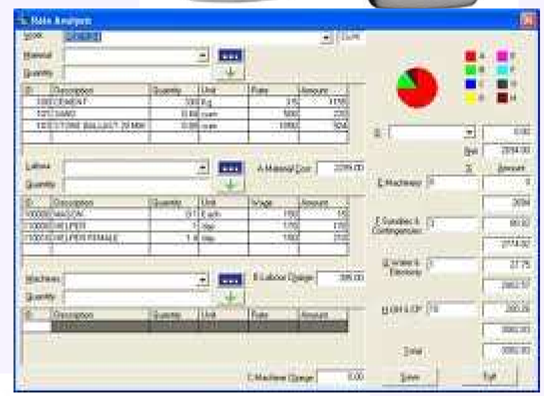
binary tree

GCD

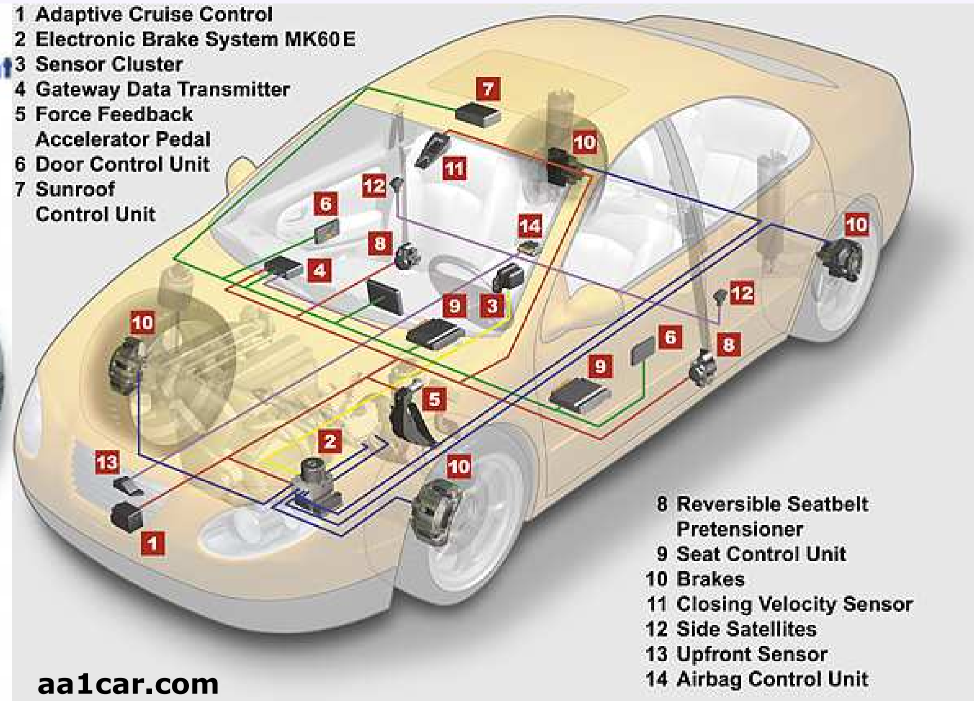
sorting

BDDs



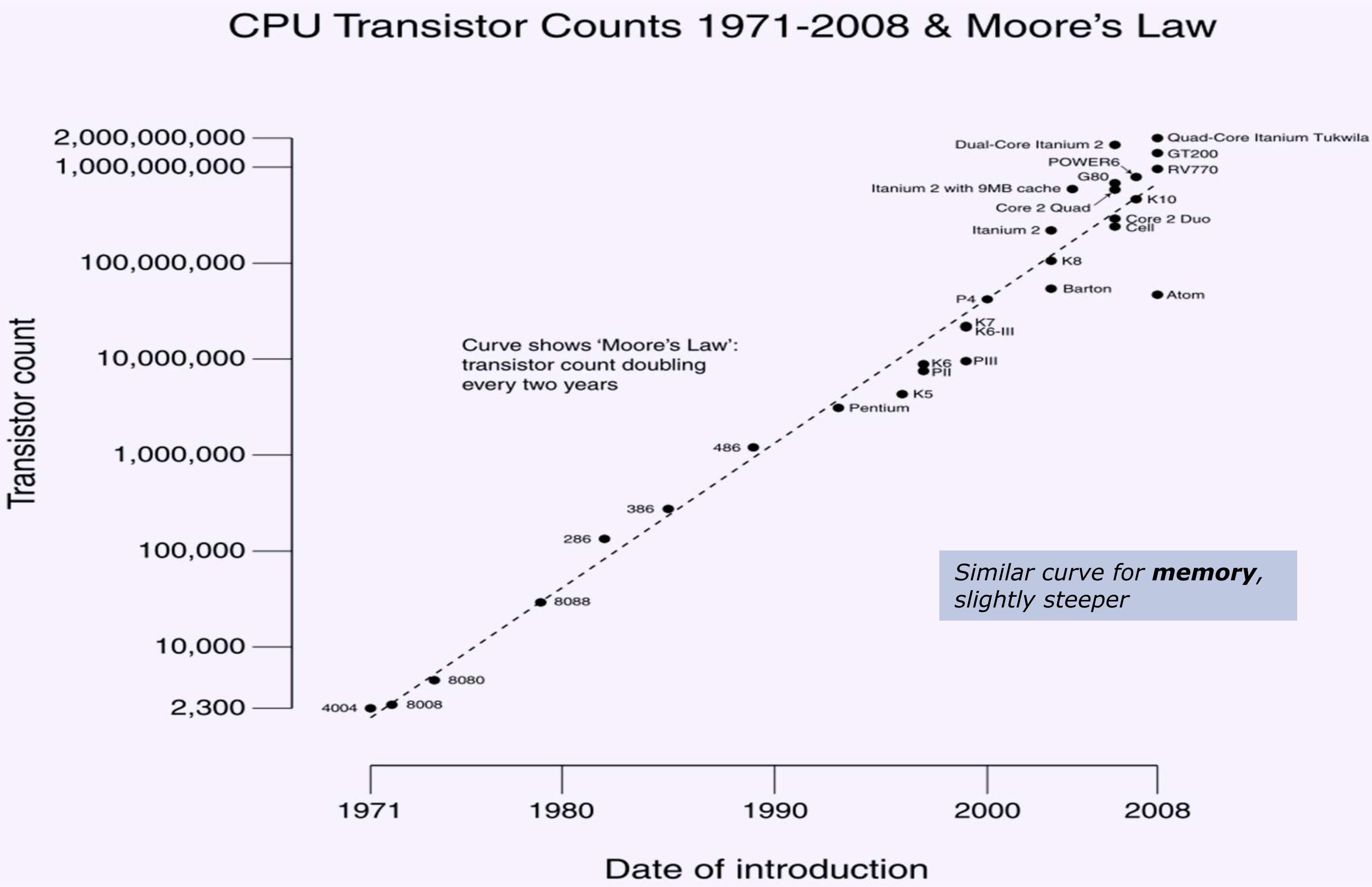


Software and automobiles



Air-bag system	Antilock brakes	Automatic transmission
Alarm system	Climate control	Collision-avoidance system
Cruise control	Communication system	Dashboard instrumentation
Electronic stability control	Engine ignition	Engine control
Electronic-seat control	Entertainment system	Navigation system
Power steering	Tire-pressure monitoring	Windshield-wiper control

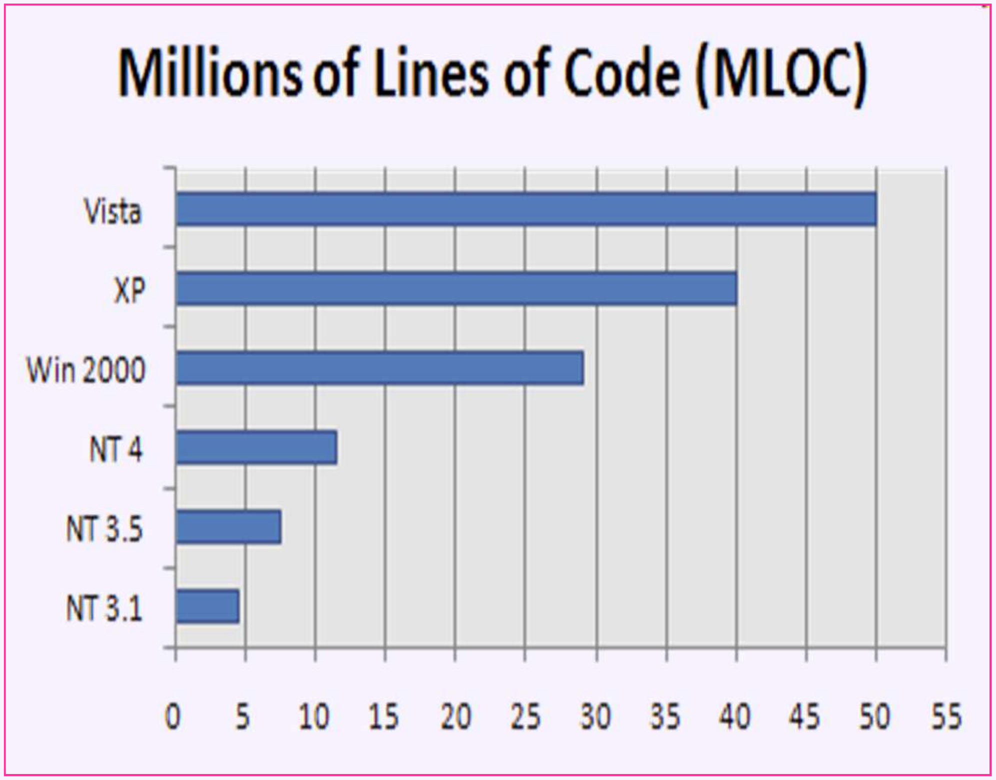
Moore's Law: transistors per chip



Similar curve for ***memory****, slightly steeper*

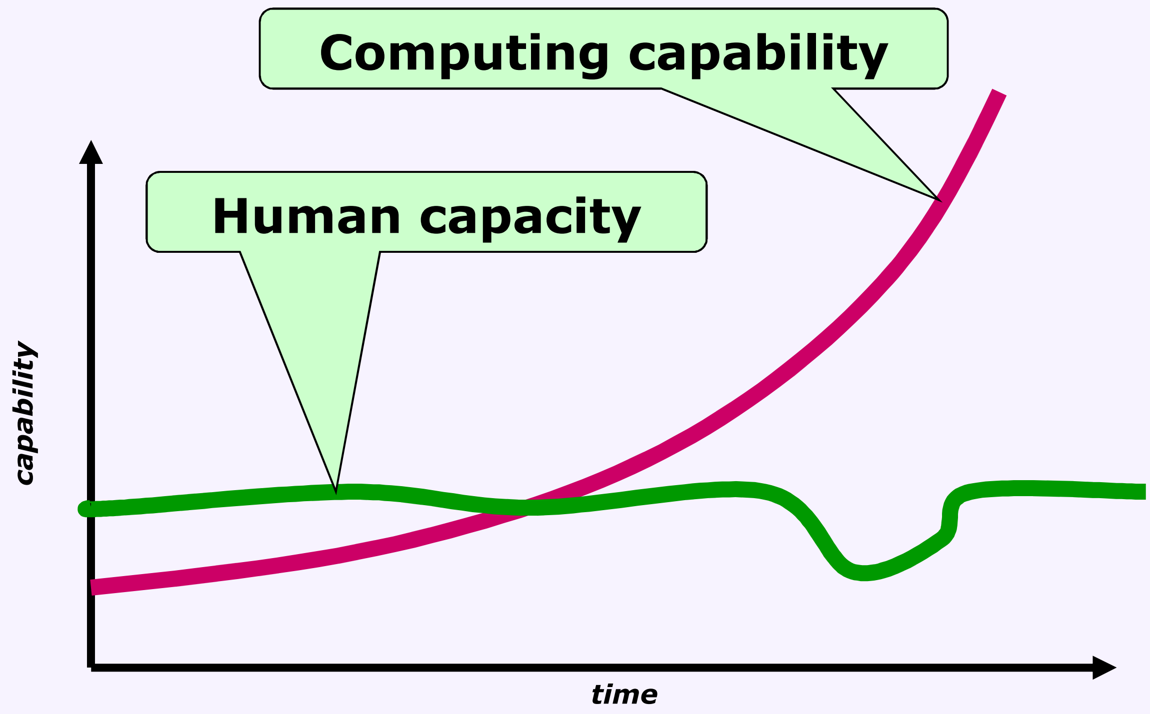
How much software?

System	Year	% of Functions Performed in Software
F-4	1960	8
A-7	1964	10
F-111	1970	20
F-15	1975	35
F-16	1982	45
B-2	1990	65
F-22	2000	80



(informal reports)

The limits of exponentials



Scaling Up: From Programs to Systems

- You’ve written small- to medium-size programs in 15-122

- This course is about managing software complexity
  - **Scale** of code: KLOC -> MLOC
  - Worldly **environment**: external I/O, network, asynchrony
  - Software **infrastructure**: libraries, frameworks, components
  - Software **evolution**: change over time

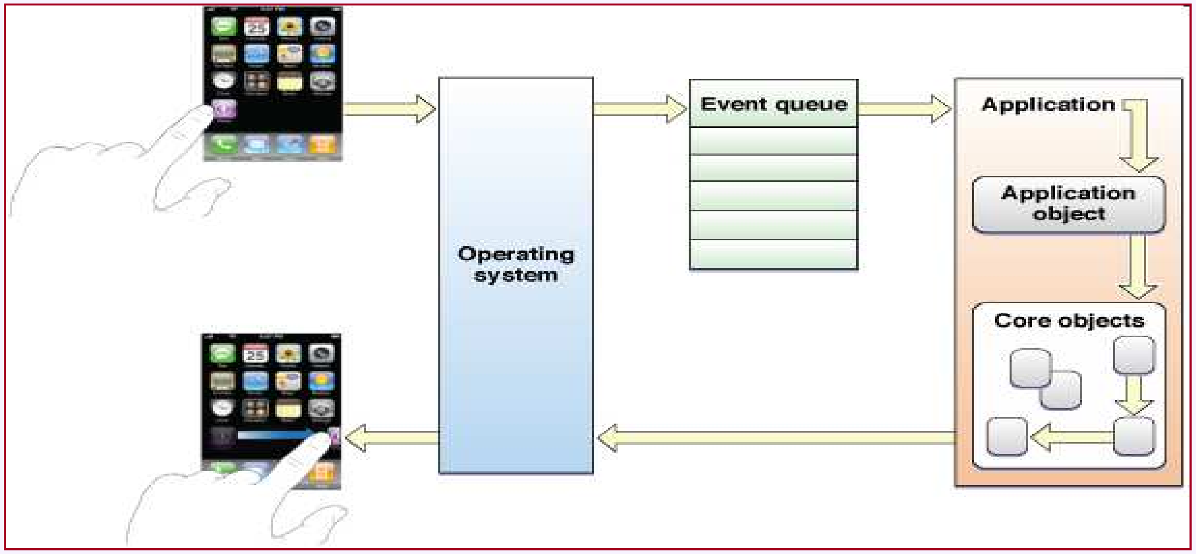
  - Contrast: algorithmic complexity
    - Not an emphasis in this course

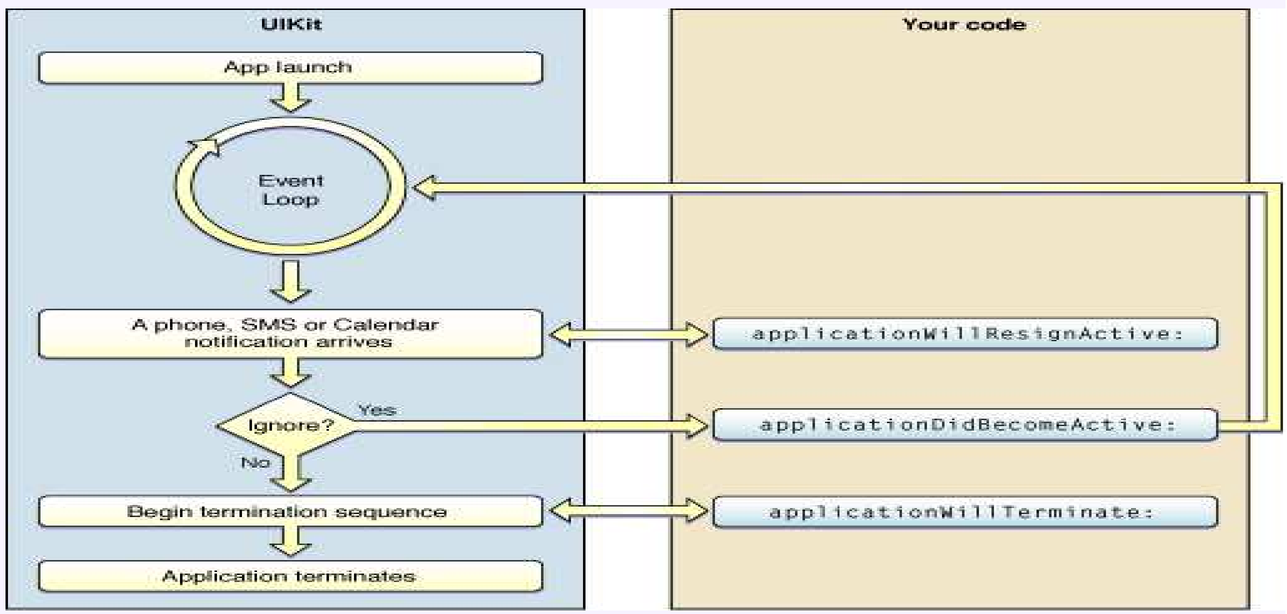
From Programs to Systems

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 **principles** for construction of software systems at scale

A framework for mobile app software (IOS)

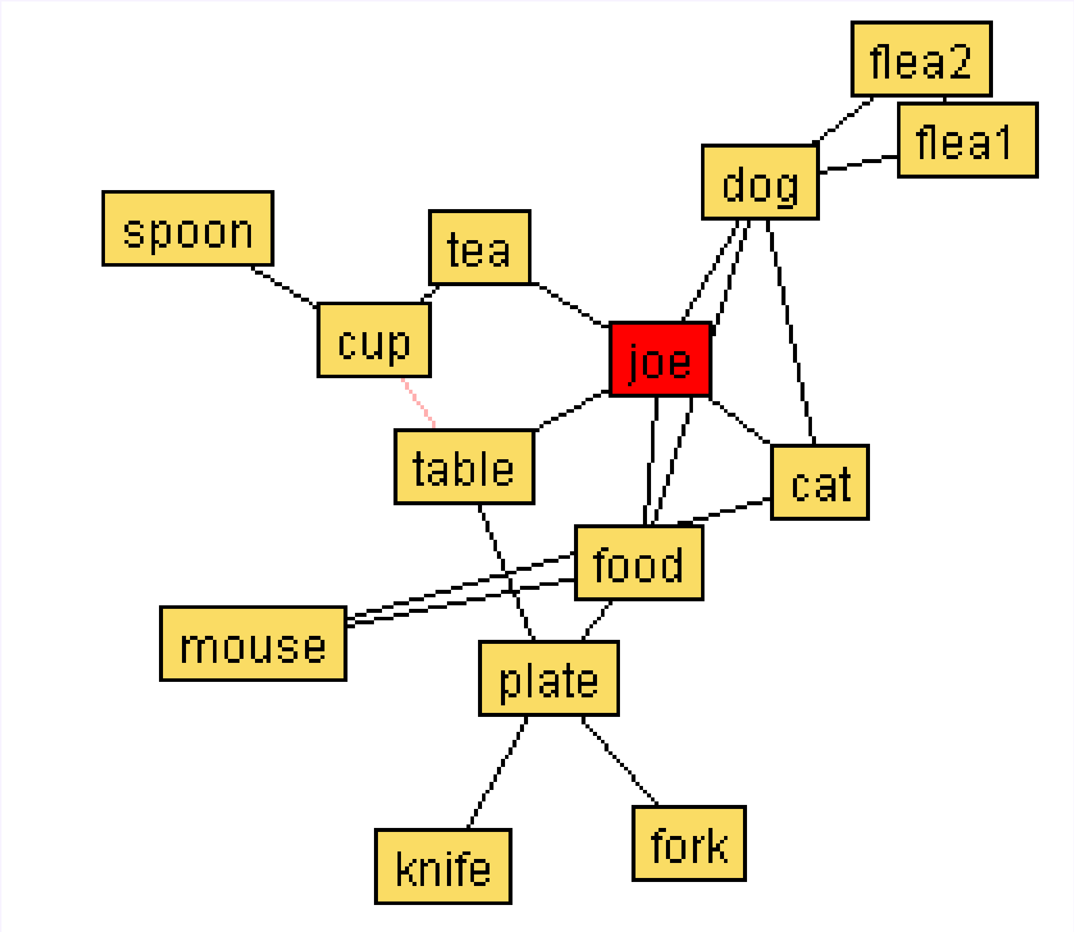




The four course themes

- **T**hreads and Concurrency
  - Concurrency is a crucial system abstraction
    - E.g., background computing while responding to users
  - Concurrency is necessary for performance
    - Multicore processors and distributed computing
  - *Our focus*: application-level concurrency
    - Cf. functional parallelism (150, 210) and systems concurrency (213)
- **O**bject-oriented programming
  - For flexible designs and reusable code
  - A primary paradigm in industry – basis for modern frameworks
  - Focus on Java – used in industry, some upper-division courses
- **A**nalysis and Modeling
  - *Practical* specification techniques and verification tools
  - Address challenges of threading, correct library usage, etc.
- **D**esign
  - Proposing and evaluating alternatives
  - Modularity, information hiding, and planning for change
  - Patterns: well-known solutions to design problems

Motivating example #1: GraphLayout



Source code: http://java.sun.com/applets/jdk/1.4/demo/applets/GraphLayout/example1.html

Screenshot from http://stackoverflow.com/questions/1318770/impressive-examples-in-java

Discussion: GraphLayout

- What does the design of GraphLayout look like, conceptually?

- What is most important about the design?

- How should the GUI be organized? Why?

Motivating example #2: Virtual Worlds

Discussion: Virtual Worlds

- How can the virtual world to scale to thousands of users?
- How can we organize the system to easily add new things?
- How can we support different kinds of things, while taking advantage of their similarities? (can you think of an example?)

Considering the examples

- **T**hreads and Concurrency
  - In the GUI-based app
  - On game clients
  - On the game servers
- **O**bject-oriented programming
  - Organizing by object types, then actions
- **A**nalysis and Modeling
  - How to gain confidence regarding *all* possible executions
- **D**esign
  - How to organize systems that grow and evolve
  - How to define the interfaces between infrastructure and our code

Objects Analysis Threads Design
15-214

toad

Fall 2012

School of Computer Science
institute for SOFTWARE RESEARCH

Principles of Software Construction: Objects, Design, and Concurrency

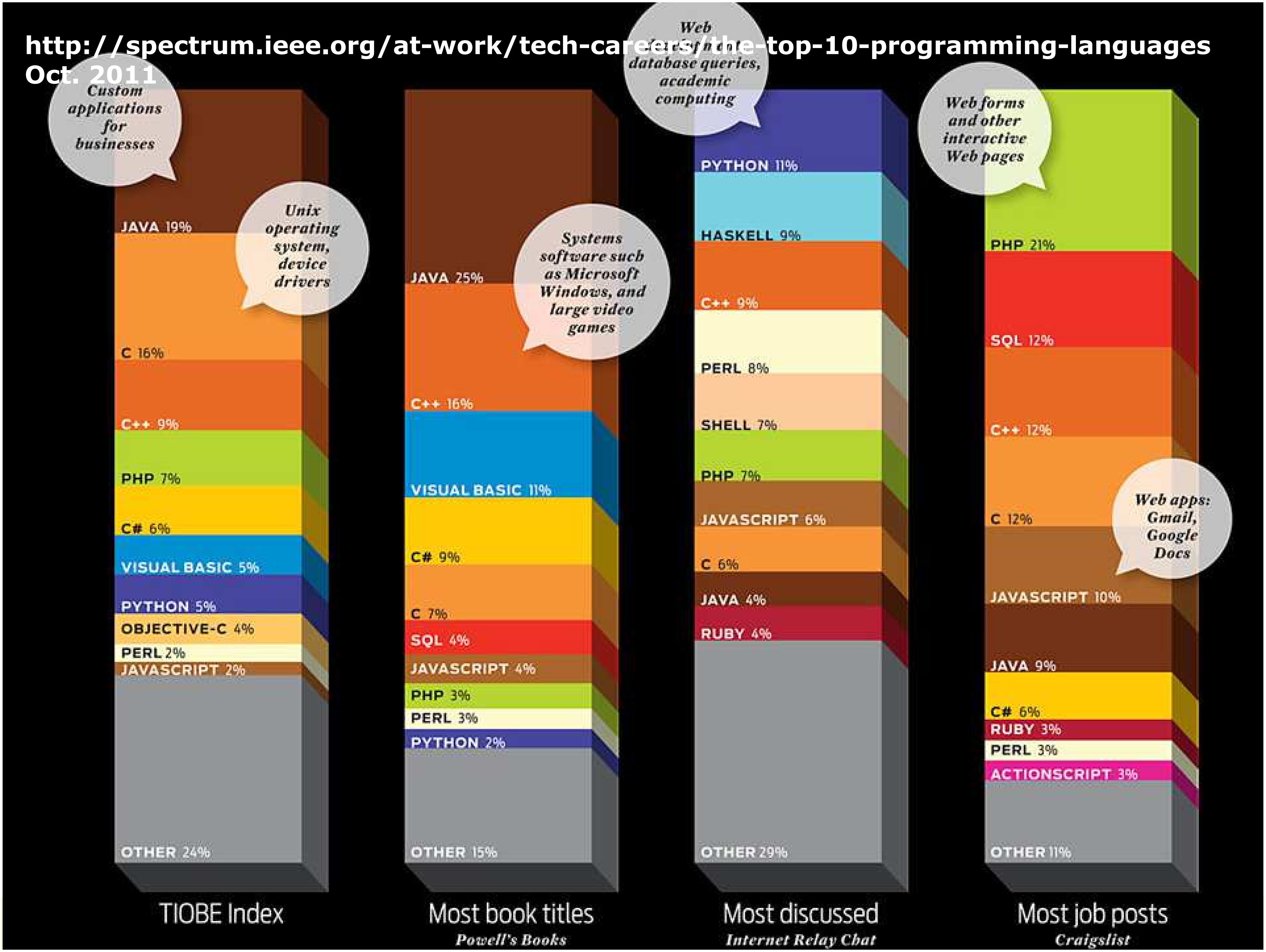
Objects

Jonathan Aldrich Charlie Garrod

Object-Oriented Programming Languages

- C++
- Java
- C#
- Smalltalk
- Scala
- Objective-C
- JavaScript
- Ruby
- PHP5
- Object Pascal/Delphi
- OCaml
- …

http://spectrum.ieee.org/at-work/tech-careers/the-top-10-programming-languages
Oct. 2011



Object orientation (OO)

- History
  - Simulation – Simula 67, first OO language
  - Interactive graphics – SmallTalk-76 (inspired by Simula)

- Object-oriented programming (OOP)
  - Organize code bottom-up rather than top-down
  - Focus on **concepts** rather than **operations**
  - Concepts include both **conventional data types** (e.g. List), and **other abstractions** (e.g. Window, Command, State)

- Some benefits, informally stated
  - Easier to reuse concepts in new programs
    - Concepts map to ideas in the target domain
  - Easier to extend the program with new concepts
    - E.g. variations on old concepts
  - Easier to modify the program if a concept changes
    - **Easier** means the changes can be **localized** in the code base

Objects

- **Object**
  - A package of state (data) and behavior (actions)
- Data and actions
  - **Fields** in the object hold data values
    - Like the fields of a struct in C
    - Access to fields can be restricted
  - **Methods** describe operations or actions on that data
    - Like functions associated with an abstract data type
    - They have access to the all fields
    - Method calls can be thought of as "**messages**" to the object
- Thus…
  - **Methods** can control access to the fields
    - Best practice: Don't allow fields to be seen from outside
  - The **object** can be thought of as a *service* that is accessed through a managed interface. The **class** described a family of similar services.
    - E.g., a particular button (object) vs. buttons in general (class)

Example: Concept of a Rectangle

- What do you need to **know** about a rectangle?

- What might you want to **do** with a rectangle?

Example: Points and Rectangles

```
class Point {
    int x, y;
    int getX() { return x; } // a method; getY() is similar
    Point(int px, int py) { x = px; y = py; }  // constructor for creating the object
}
class Rectangle {
    Point origin;
    int width, height;
    Point getOrigin() { return origin; }
    int getWidth() { return width; }
    void draw() {
            drawLine(origin.getX(), origin.getY(),        // first line
                    origin.getX()+width, origin.getY());
            … // more lines here
    }
    Rectangle(Point o, int w, int h) {
            origin = o; width = w; height = h;
    }
}
```

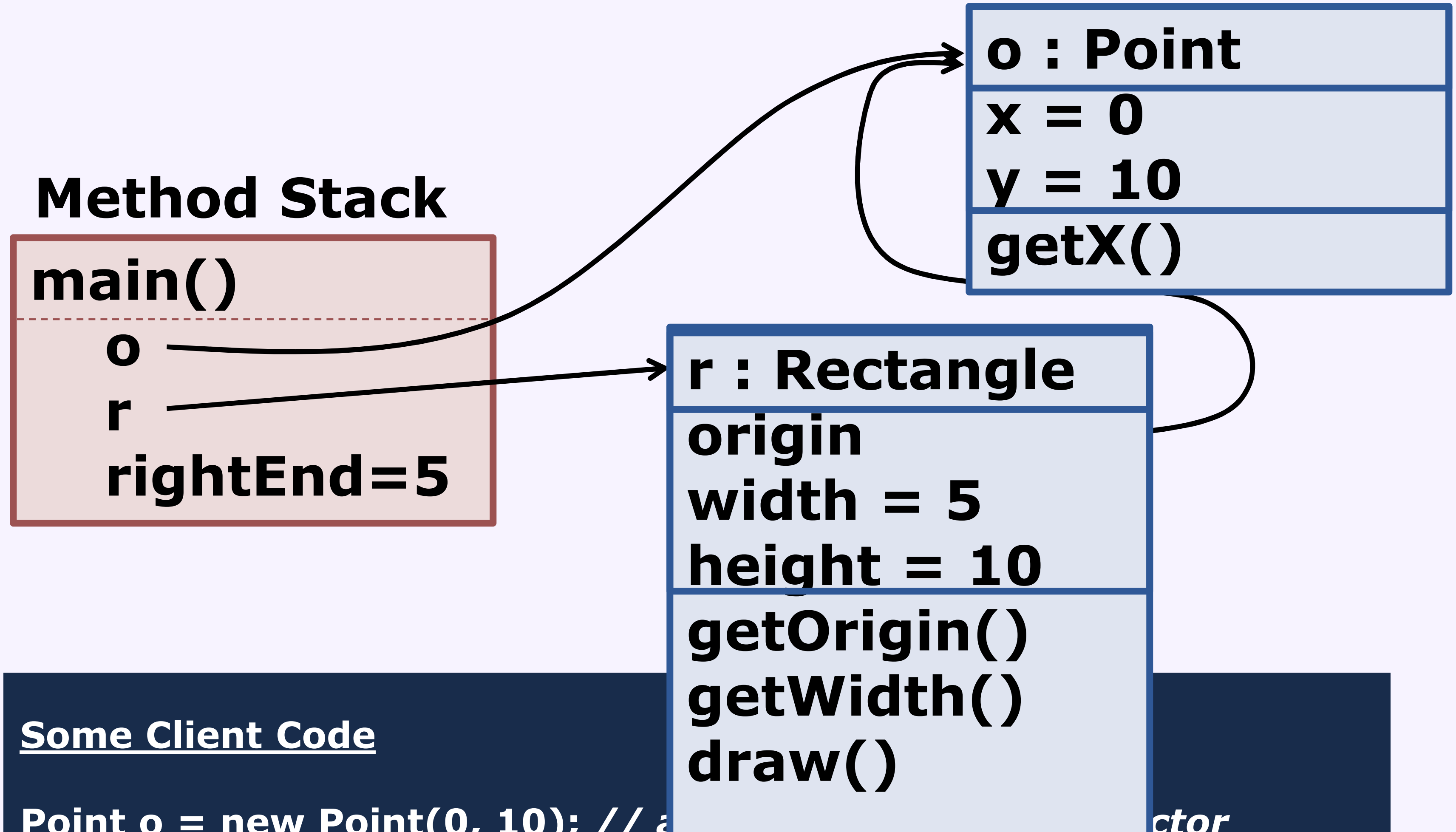
Example: Points and Rectangles

```
class Point {
    int x, y;
    int getX() { return this.x; } // a method; getY() is similar
    Point(int px, int py) {this.x = px; this.y = py; }  // constructor for creating the object
}
class Rectangle {
    Point origin;
    int width, height;
    Point getOrigin() { return this.origin; }
    int getWidth() { return this.width; }
    void draw() {
            this.drawLine(this.origin.getX(), this.origin.getY(),    // first line
                    this.origin.getX()+this.width, this.origin.getY());
```

Some Client Code

```
Point o = new Point(0, 10); // allocates memory, calls ctor
Rectangle r = new Rectangle(o, 5, 10);
r.draw();
int rightEnd = r.getOrigin().getX() + r.getWidth(); // 5
```

What's really going on?



Some Client Code

```
Point o = new Point(0, 10); // a ... ctor
Rectangle r = new Rectangle(o, 5, 10);
r.draw();
int rightEnd = r.getOrigin().getX() + r.getWidth(); // 5
```

Toad's Take-Home Messages

- 214: managing complexity, from programs to systems
  - Threads and concurrency
  - Object-oriented programming
  - Analysis and modeling
  - Design
- GraphLayout and virtual worlds illustrate some challenges
- Object-oriented programming organizes code around **concepts**
  - Methods capture behavior, fields capture state
  - As we will see, this organization allows
    - Greater reuse of concepts
    - Better support for change when concepts vary