Programming Languages and Techniques (CIS120)

Lecture 1

January 13, 2016

Welcome
Introduction to Program Design

Introductions

- Instructor: Dr. Stephanie Weirich*
  - Levine 510
  - sweirich@cis.upenn.edu
  - http://www.cis.upenn.edu/~sweirich
  - Office hours: Mondays 3:30-5 PM or by appointment, except:
    - TODAY (1/13) at 3:30-5
    - Tuesday (1/19) at 3:30-5

- Course Administrator: Laura Fox
  - Levine 308
  - lffox@cis.upenn.edu

*Pronounced phonetically as: "why rick". I won't get upset if you mispronounce my name (really!). I will answer to anything remotely close, or, you can just call me Stephanie. Whatever you feel comfortable with.

Introductions

- Instructor: Dr. Steve Zdancewic*
  - Levine Hall 511
  - stevez@cis.upenn.edu
  - http://www.cis.upenn.edu/~stevez/
  - Office hours:
    Mondays 3:30 – 5:00pm (may change!)
    or by appointment

- Course Administrator: Laura Fox
  - Levine 308
  - lffox@cis.upenn.edu

*Pronounced phonetically as: "zuh dans wick". I won't get upset if you mispronounce my name (really!). I will answer to anything remotely close, or, you can call me Steve, just Professor, or Professor Z. Whatever you feel comfortable with.

Teaching Assistant Staff*

- Becky Abramowitz
- Bethany Davis
- Brian Hirsh
- Danica Fine
- Dylan Mann
- Enrique Mitchell
- Graham Mosley
- Helena Chen
- Jacob Hultman
- Jenny Chen
- Jorge Liang
- Joyce Lee
- Julia Olsen

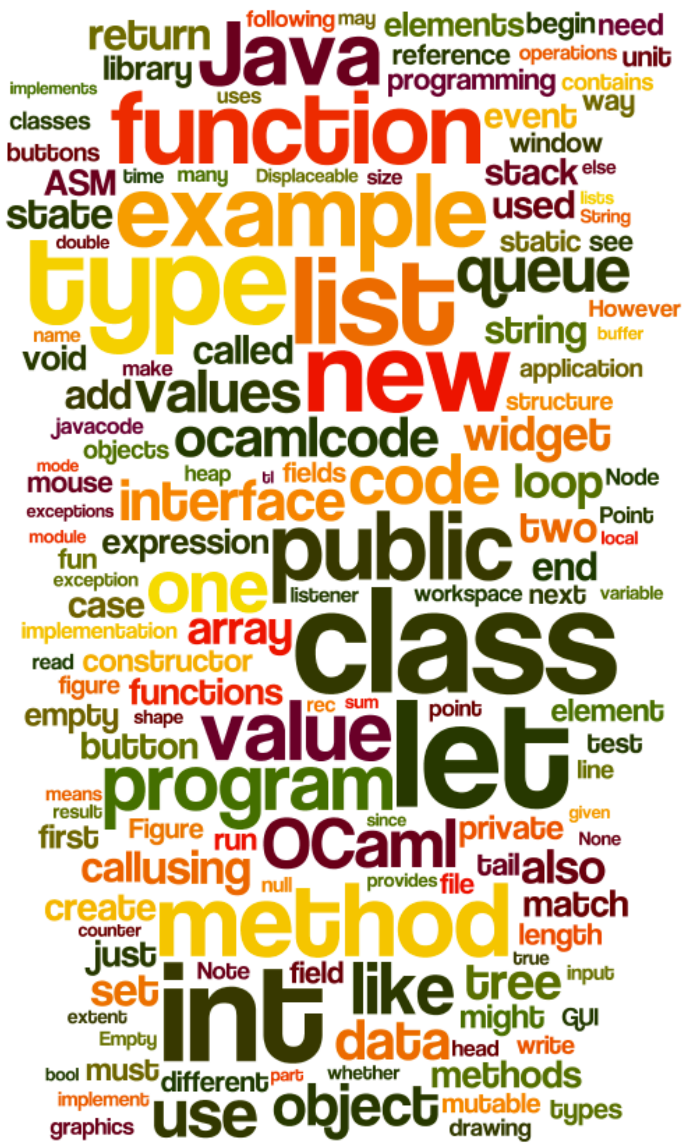
- Kaylin Raby
- Liam Gallagher
- Matt Chiaravalloti ★
- Matt Howard
- Max McCarthy ★
- Pia Kochar
- Sahil Ahuja
- Samantha Chung
- Sierra Yit
- Thomas Delacour
- Tony Mei
- Vivek Raj
- Zane Stiles

★ Head TAs

*AKA: CIS 120 spirit guides, student champions, and all-around defenders of the universe.

What is CIS 120?

- CIS 120 is a course in **program design**
- Practical skills:
  - ability to write larger (~1000 lines) programs
  - increased independence ("working without a recipe")
  - test-driven development, principled debugging
- Conceptual foundations:
  - common data structures and algorithms
  - several different programming idioms
  - focus on modularity and compositionality
  - derived from first principles throughout
- It will be fun!



Prerequisites

- We assume you can already write 10 to 100-line programs in some imperative or OO language
  - Java experience is *strongly recommended*
  - CIS 110 or AP CS is typical
  - You should be familiar with using a compiler, editing code, and running programs you have created

- CIS 110 is an alternative to this course
  - If you have doubts, come talk to me or one of the TAs to figure out the right course for you

CIS 120 Tools

- OCaml
  - Industrial-strength, statically-typed *functional* programming language
  - Lightweight, approachable setting for learning about program design
- Java
  - Industrial-strength, statically-typed *object-oriented* language
  - Many tools/libraries/resources available
- Eclipse
  - Popular open-source integrated development environment (IDE)

Installation: http://www.seas.upenn.edu/~cis120/current/ocaml_setup.shtml

Why *two* languages?

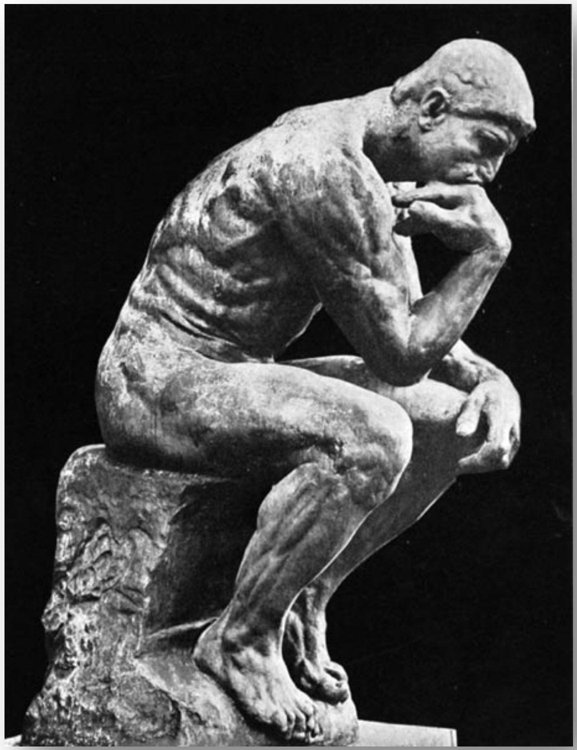
- Pedagogic progression
- Disparity of background
- Confidence in learning new tools
- Perspective

"[The OCaml part of the class] was very essential to getting fundamental ideas of comp sci across. Without the second language it is easy to fall into routine and syntax lock where you don't really understand the bigger picture."
---Anonymous CIS 120 Student

"[OCaml] made me better understand features of Java that seemed innate to programming, which were merely abstractions and assumptions that Java made. It made me a better Java programmer."
--- Anonymous CIS 120 Student

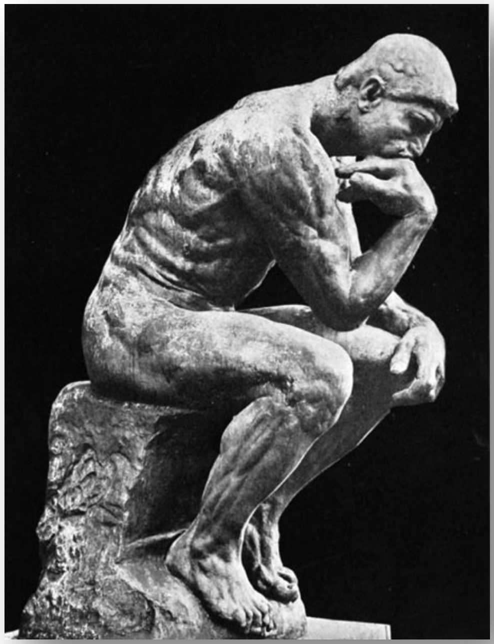
Teaching Philosophy

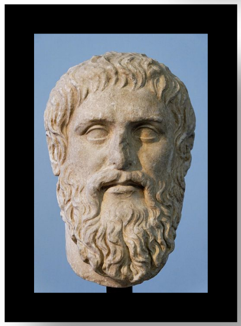
- Introductory computer science
  - Start with basic skills of "algorithmic thinking" (AP/110)
  - Develop systematic design and analysis skills in the context of larger and more challenging problems (120)
  - Practice with industrial-strength tools and design processes (120, 121, and beyond)
- Role of CIS120 and *program design*
  - Start with foundations of programming using the rich grammar and precise semantics of the OCaml language
  - Transition (back) to Java *after* setting up the context needed to understand why Java and OO programming are good tools
  - Give a taste of the breadth and depth of CS



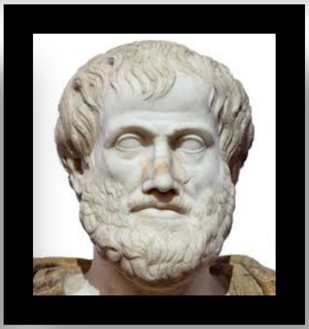
Philosophy

- Teaching introductory computer science
  - Start with basic skills of "algorithmic thinking" (AP/110)
  - Develop systematic design and analysis skills in the context of larger and more challenging problems (120)
  - Practice with industrial-strength tools and design processes (120, 121, and beyond)
- Role of CIS120 and *program design*
  - Start with foundations of programming using the rich grammar and precise semantics of the OCaml language
  - Transition (back) to Java *after* setting up the context needed to understand why Java and OO programming are good tools
  - Give a taste of the breadth and depth of CS





Plato



Aristotle

Al-Kwarizmi

Administrative Matters

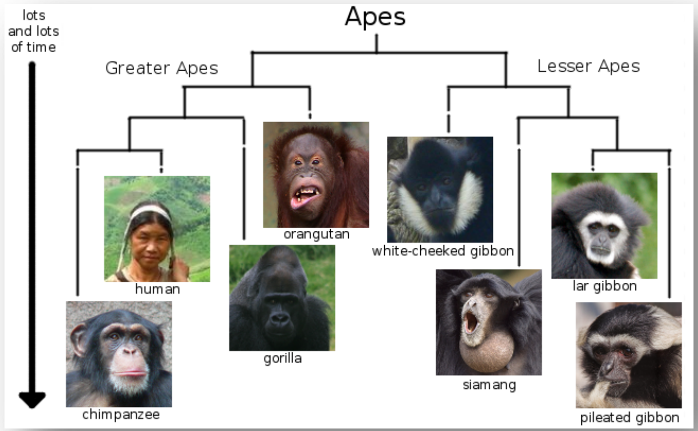
http://www.seas.upenn.edu/~cis120/

Course Components

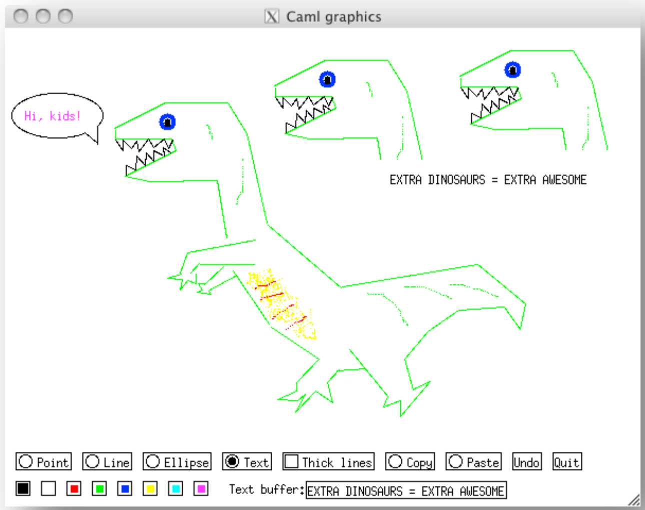
- Lectures (2% of final grade)
  - Presentation of ideas and concepts, interactive demos
  - Grade based on participation using "clickers"
  - Lecture notes & screencasts available on course website.
- Recitations / Labs (6% of final grade)
  - Practice and discussion in small group setting
  - Grade based on participation
- Homework (50% of final grade)
  - Practice, experience with tools
  - Exposure to broad ideas of computer science
  - Grade based on automated tests + style
- Exams (42% of final grade)
  - Evening exams, pencil and paper
  - Do you understand the terminology? Can you reason about programs? Can you synthesize solutions?

Warning: This is a *challenging* and *time consuming* (but rewarding) course!

Some of the homework assignments…



Computing with DNA


Caml graphics
Hi, kids!
EXTRA DINOSAURS = EXTRA AWESOME
Point
Line
Ellipse
Text
Thick lines
Copy
Paste
Undo
Quit
Text buffer: EXTRA DINOSAURS = EXTRA AWESOME

Build a GUI Framework

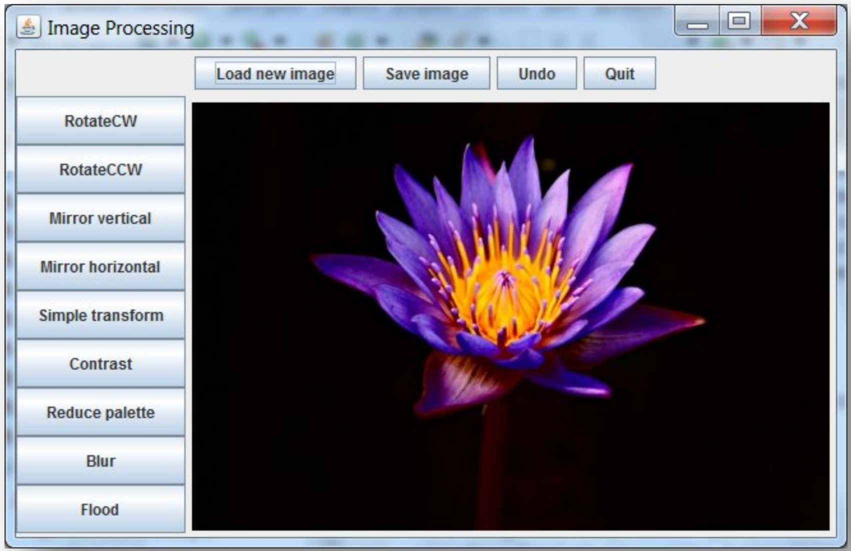
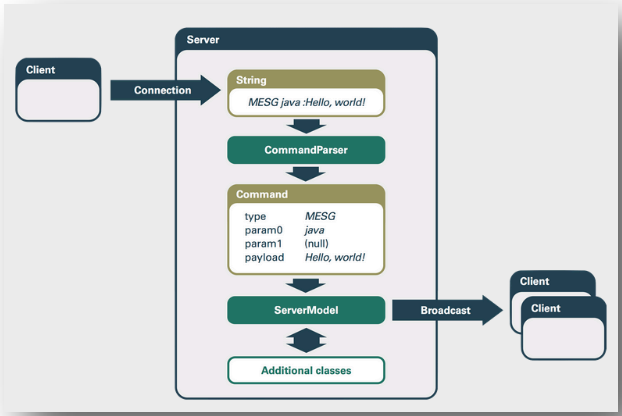
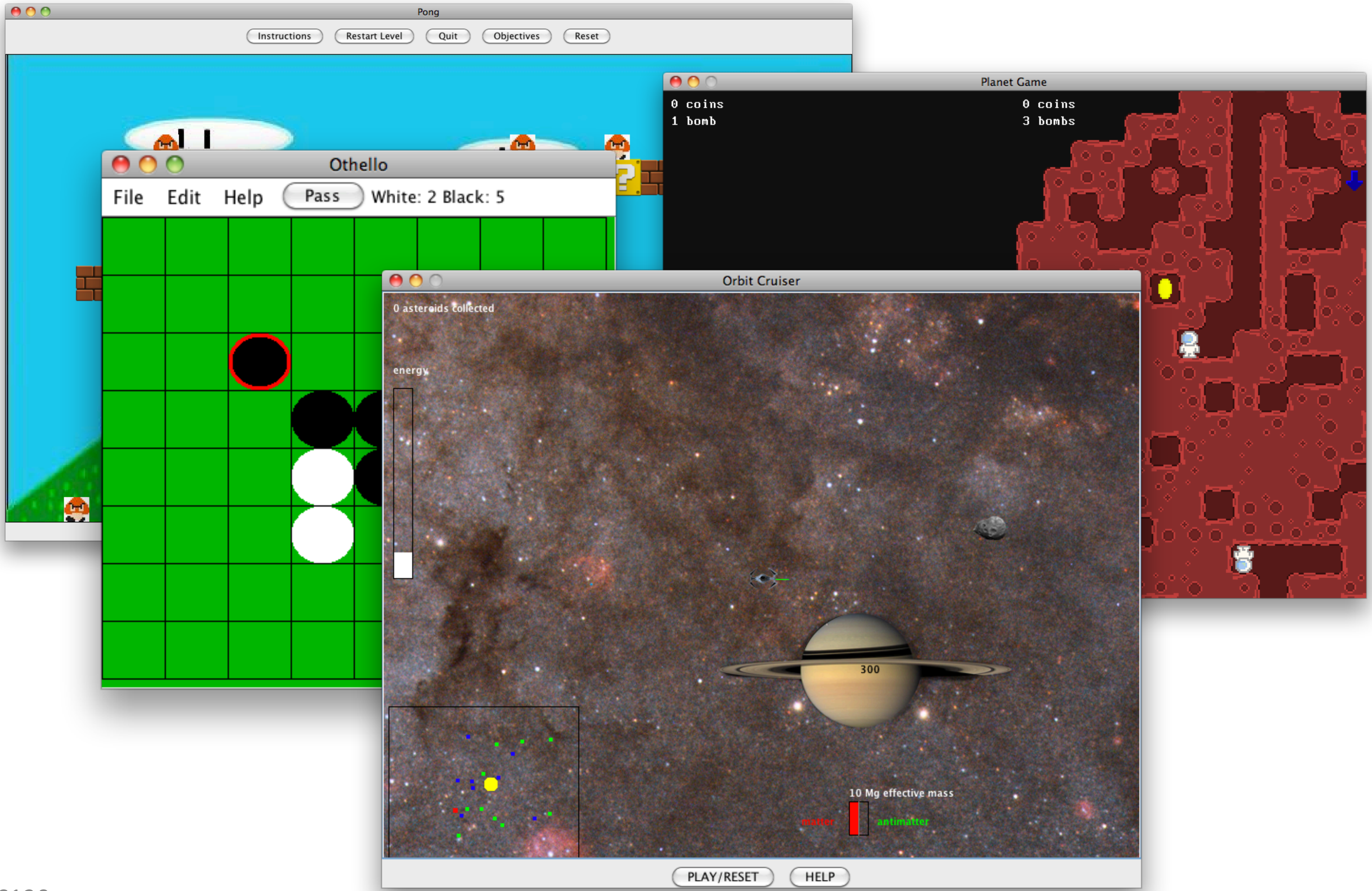


Image Processing



Chat Client/Server

Final project: Design a Game



Registration / Recitations

- Registration is currently closed
  - Add your name to the wait list if you are not registered
  - We will be accepting students off the wait list as space is available (18 students currently…)
  - If you are on the wait list, you must keep up with the course
  - Sign in after class so I know that you were here
- If you need to switch recitations, fill out the online *change request form* linked from the course web page
  - If you don't have a recitation, leave the first one blank
- Recitations start *next week*:
  - Make sure you set your laptop up BEFORE the first recitation
  - See instructions on course website, ask a TA for help

CHANGE OF LOCATION

- Lab Section 213
  - Thurs. 5-6pm
- Moore 207

Office Hours

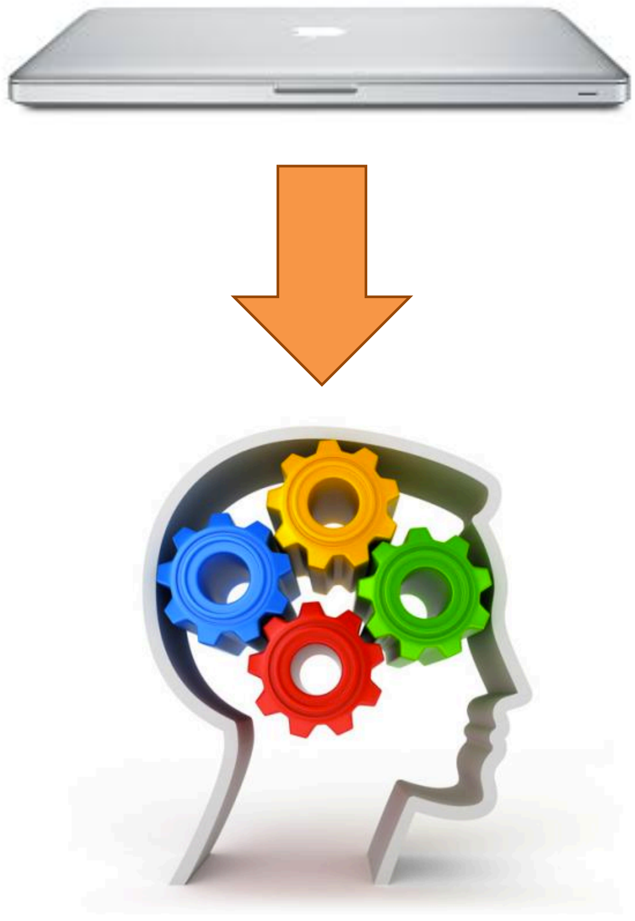
- We will try to offer office hours starting soon!
  - Look for announcements
- Use them to help get the course infrastructure set up

Clickers

- We will use TurningPoint ResponseCards (clickers) for interactive exercises during lectures.
  - wrong answers do not count against your grade
- Please buy one at the bookstore (textbook section)
  - You can sell it back at the end of the semester
- Bring it to lecture every day, beginning Friday
  - Participation grades start Friday 1/22/2016

Lecture Policy

- Laptops *closed*… minds *open*
  - Although this is a computer science class, the use of electronic devices – laptops, cell phones, mobile devices, iPads, etc., in lecture is *prohibited*.
- Why?
  - Laptop users tend to surf/chat/e-mail/game/text/tweet/etc.
  - They also distract those around them
  - You will get plenty of time in front of your computers while working on the course projects :-)



Academic Integrity

- Submitted homework must be *your individual work*

Talk all you want about any level of detail of the HW, but don't look at anyone else's code and don't share yours.

- Not OK:
  - Copying or otherwise looking at someone else's code
  - Sharing your code in any way (internet, copy-paste, by hand)
- OK / encouraged:
  - Discussions of concepts
  - Discussion of debugging strategies
  - Verbally sharing experience

Rationale

- HW is intended to be doable *individually* in the time allowed.
  - With help/clarification from the course staff
- Learning to debug your code is a *very important* skill!
  - Getting too much help hinders this learning process

- There is a bit of a gray area here…
  - Hard to delineate OK from not-OK behavior
  - We need a simple, clear rule
  - Use good judgment

Enforcement

- Course staff *will* check for copying.
  - We use plagiarism detection tools on your code
- If you have significant discussions with a TA or another student in the class, *acknowledge them* in comments in the submitted code.

Violations will be treated seriously!

- *Question? See the course FAQ. If in doubt, ask.*

Penn's code of academic integrity:
http://www.vpul.upenn.edu/osl/acadint.html

Academic Integrity

- Not OK

  A: I still can't figure out this problem on HW06. How do you write checkboxes?

  B: Oh, I'm done already. Yeah, that problem took forever…

  A: Wait, you're done? Can I look at your code?

  B: Sure (*shows code*)

- OK

  A: I still can't figure out this problem on HW06. How do you write checkboxes?

  B: Oh, I'm done already. Yeah, that problem took forever….

  A: Wait, you're done? Can I look at your code?

  B: Well, what are you stuck on?

  A: (*points to own code*) I don't get this thing about listeners...

  B: Oh! Those things are weird. So think of it this way...

  A: Ok, I understand now. Thanks!

- Bottom line: *your homework should come from your brain, as well as your fingers.*

Program Design

Fundamental Design Process

Design is the process of translating informal specifications ("word problems") into running code.

1. Understand the problem
   What are the relevant concepts and how do they relate?
2. Formalize the interface
   How should the program interact with its environment?
3. Write test cases
   How does the program behave on typical inputs? On unusual ones? On erroneous ones?
4. Implement the required behavior
   Often by decomposing the problem into simpler ones and applying the same recipe to each

5. Revise / Refactor / Edit

A design problem

Imagine that you own a movie theater. The more you charge, the fewer people can afford tickets. In a recent experiment you determined a relationship between the price of a ticket and average attendance. At a price of $5.00 per ticket, 120 people attend a performance. Decreasing the price by a dime ($.10) increases attendance by 15.

However, increased attendance also comes at increased cost; each attendee costs four cents ($0.04). Every performance also has a base cost of $180. At what price do you make the highest profit?

Step 1: Understand the problem

- What are the relevant concepts?
  - *(ticket) price*
  - *attendees*
  - *revenue*
  - *cost*
  - *profit*
- What are the relationships among them?
  - profit = revenue – cost
  - revenue = price * attendees
  - cost = $180 + attendees * $0.04
  - attendees = *some function of the ticket price*
- Goal is to determine profit, given the ticket price

So profit, revenue, and cost also depend on price.

Step 2: Formalize the Interface

*Idea: we'll represent money in cents, using integers**

comment documents the design decision

type annotations declare the input and output types**

```
(* Money is represented in cents. *)
let profit (price : int) : int = …
```

* Floating point is generally a *bad* choice for representing money: bankers use different rounding conventions than the IEEE floating point standard, and floating point arithmetic isn't as exact as you might like. Try calculating 0.1 + 0.1 + 0.1 sometime in your favorite programming language…

**OCaml will let you omit these type annotations, but including them is *mandatory* for CIS120. Using type annotations is good documentation; they also improve the error messages you get from the compiler. When you get a type error message from the compiler, the first thing you should do is check that your type annotations are there and that they are what you expect.

Step 3: Write test cases

- By looking at the design problem, we can calculate specific test cases

```
let profit_500 : int =
   let price     = 500 in
   let attendees = 120 in
   let revenue   = price * attendees in
   let cost      = 18000 + 4 * attendees in
   revenue - cost
```

Writing the Test Cases in OCaml

- Record the test cases as assertions in the program:
  - the *command* run_test executes a test

a *test* is just a function that takes no input and returns true if the test succeeds

```
let test () : bool =
   (profit 500) = profit_500

;; run_test "profit at $5.00" test
```

the string in quotes identifies the test in printed output (if it fails)

note the use of double semicolons before commands

Step 4: Implement the Behavior

Profit is easy to define:

```
let attendees (price : int) = ...

let profit (price : int) =
  let revenue = price * (attendees price) in
  let cost = 18000 + 4 * (attendees price) in
  revenue - cost
```

Apply the Design Pattern Recursively

attendees* requires a bit of thought:

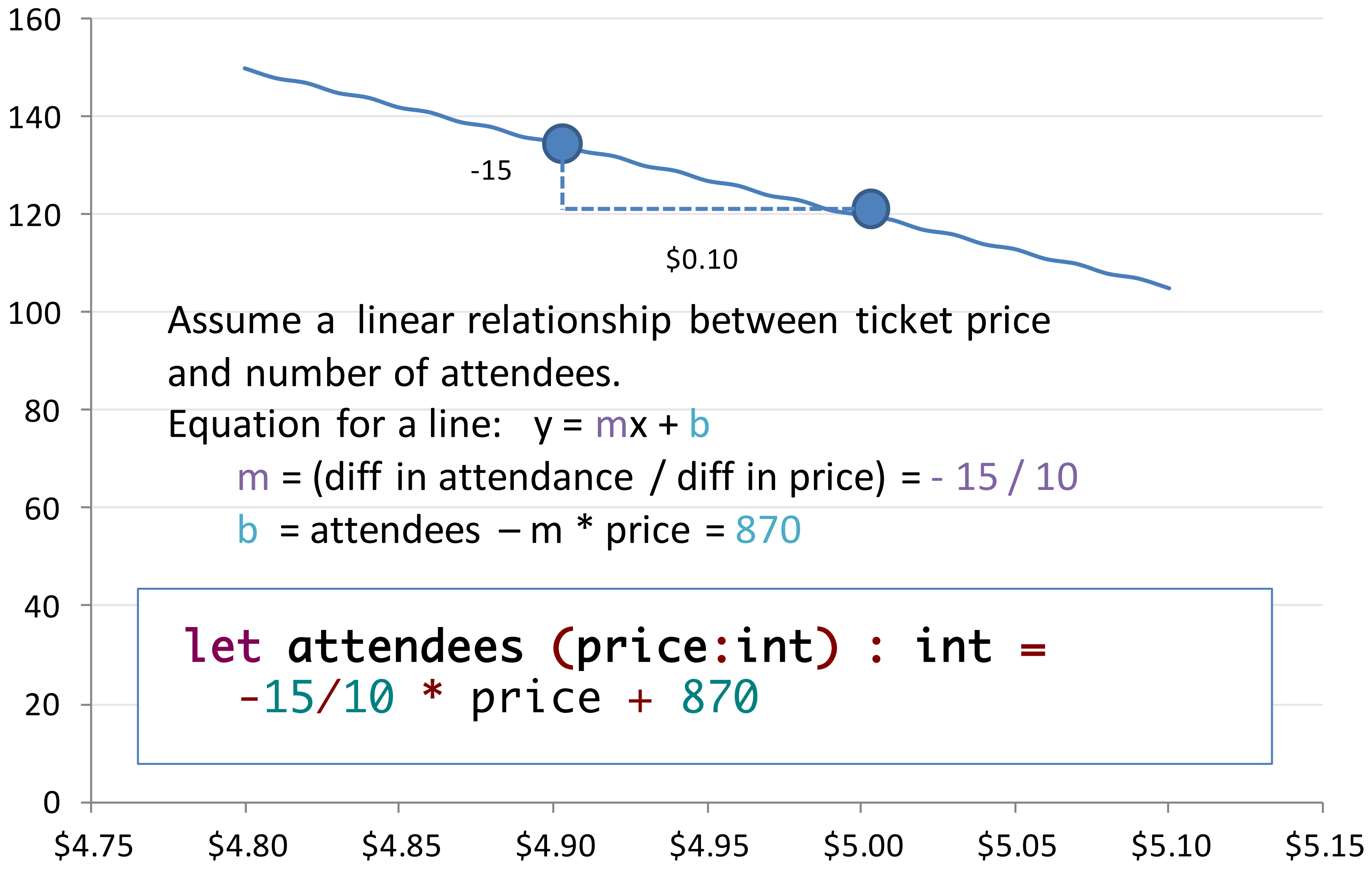
```
let attendees (price : int) : int =
   failwith “unimplemented”

let test () : bool =
   (attendees 500) = 120
;; run_test "attendees at $5.00" test

let test () : bool =
   (attendees 490) = 135
;; run_test "attendees at $4.90" test
```

“stub out” unimplemented functions

generate the tests from the problem statement *first*.

*Note that the definition of attendees must go *before* the definition of profit because profit uses the attendees function.

Attendees vs. Ticket Price



Assume a linear relationship between ticket price and number of attendees.

Equation for a line: y = mx + b

m = (diff in attendance / diff in price) = - 15 / 10

b = attendees – m * price = 870

```
let attendees (price:int) : int =
  -15/10 * price + 870
```

Run!

Run the program!

- One of our test cases for attendees failed…
- Debugging reveals that integer division is tricky*
- Here is the fixed version:

```
let attendees (price:int) :int =
    (-15 * price) / 10 + 870
```

*Using integer arithmetic, -15 / 10 evaluates to -1, since -1.5 rounds to -1. Multiplying -15 * price before dividing by 10 increases the precision because rounding errors don't creep in.

Using Tests

Modern approaches to software engineering advocate *test-driven development*, where tests are written very early in the programming process and used to drive the rest of the process.

We are big believers in this philosophy, and we'll be using it throughout the course.

In the homework template, we may provide one or more tests for each of the problems. They will often not be sufficient. You should *start* each problem by making up *more* tests.

How *not* to Solve this Problem

```
let profit price =
   price * (-15 * price / 10 + 870) -
   (18000 + 4 * (-15 * price / 10 + 870))
```

This program is bad because it

- hides the structure and abstractions of the problem
- duplicates code that could be shared
- doesn't document the interface via types and comments

Note that this program still *passes all the tests!*

Summary

- *To read:* Chapter 1 of the lecture notes and course syllabus. Both available on the course website
- *To buy:* Turning Point clicker. Bring to every class, and register your ID number on the course website
- *To do:* Try to install OCaml and Eclipse on your laptops, following the setup instructions on the course website. TAs will hold office hours this week to help.
- *If on the waitlist*: sign in at the front of class

Welcome!

Who uses OCaml?

intel

facebook

JANE STREET

LexiFi

Google

CITRIX

Microsoft

MLstate

cea

Microsoft Visual F#

mylife

SimCorp

DASSAULT AVIATION

Course goal

Strive for beautiful code.

- Beautiful code
  - is simple
  - is easy to understand
  - is likely to be correct
  - is easy to maintain
  - *takes skill to develop*
- Beautiful code stems from a good design *process*

Evolving/Refactoring Code

- For this simple problem, this design methodology may seem like overkill.
  - The real benefits show up in bigger programs
  - But even *small* programs evolve over time…

- Suppose that, based on the problem description, we decided to define cost in terms of the number of attendees directly, rather than calling the attendees from within cost.
  - How do our tools and this design methodology help?

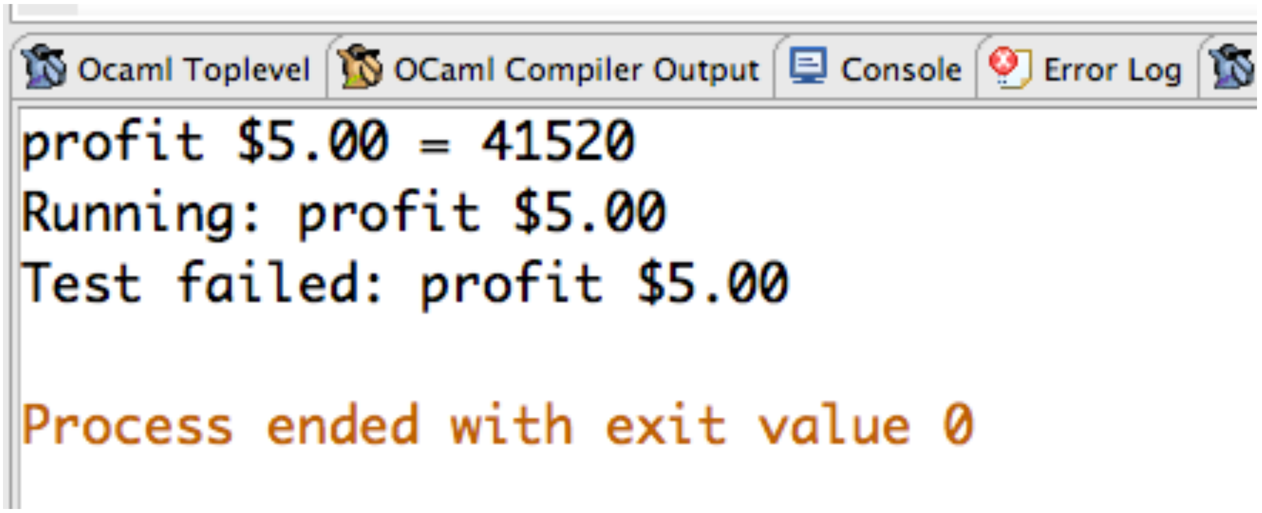
Example Refactoring: Change ‘cost’

cost is simplified:

```
(* atts is the number of attendees *)
let cost (atts:int) : int =
  18000 + 4 * atts
```

… but suppose we forget to change profit, which calls cost. (As might easily happen in a big program.)

Test Case for Profit Fails



```
profit $5.00 = 41520
Running: profit $5.00
Test failed: profit $5.00

Process ended with exit value 0
```

We need to fix profit like this:

```
let profit (price:int) : int =
  (revenue price) - (cost (attendees price))
```

Textbook

- Textbook (free download)
  - http://www.seas.upenn.edu/~cis120/current/notes/120notes.pdf
  - written by the course instructors, closely follows the lectures
  - updated throughout the semester