CSE 331
Software Design & Implementation

Dan Grossman

Autumn 2019

Lecture 1 – Introduction & Overview

(Based on slides by Mike Ernst, Hal Perkins, and many others)

What is the goal of CSE 331?

How to build harder-to-build software

- Move from CSE 143 problems toward what you'll see in upper-level CSE courses and in industry

Specifically, how to write code of

- Higher **quality**
- Increased **complexity**

We will discuss *tools* and *techniques* to help with this

  - There are *timeless principles* to both

What is high quality?

Code is high quality when it is

1. **Correct**
   - Everything else is of secondary importance

2. Easy to **change**
   - Most work is making changes to existing systems

3. Easy to **understand**
   - Needed for 1 & 2 above

How do we ensure correctness?

Best practice: use three techniques (we'll study each)

1. **Tools**
   - Type checkers, test runners, etc.
2. **Inspection**
   - Think through your code carefully
   - Have another person review your code
3. **Testing**
   - Usually >50% of the work in building software

Each removes ~2/3 of bugs. Together >97%

What is increased complexity?

Analogy to building physical objects:

- 100 well-tested LOC = a nice cabinet
- 2,500 LOC = a room with furniture
- 2,500,000 LOC = 1000 rooms ≈

North Carolina class WW2 battleship

≈

the entire British Naval fleet in WW2

Actually, software is more complex…

- Every bit of code is unique, individually designed
  - US built 10 identical Essex carriers

  - Software equivalent would be one carrier 10 times as large:

- Defects can be even more destructive
  - A defect in one room can sink the ship
  - But a defect OS could sink the *whole fleet*
- And more reasons we will see shortly…

How do we cope with complexity?

We tackle complexity with **modularity**

- Split code into pieces that can be built independently
- Each must be documented so others can use it
- Also helps understandability and changeability

Scale makes everything harder

Modularity makes scale **possible** but it's still **hard**…

- Time to write N-line program grows faster than linear
  - Good estimate is $O(N^{1.05})$ [Boehm, '81]
- Bugs grow like $\Theta(N \log N)$ [Jones, '12]
  - 10% of errors are between modules [Seaman, '08]
- Communication costs dominate schedules [Brooks, '75]
- Small probability cases become high probability cases
  - Corner cases are more important with more users

Corollary: quality must be even higher, per line, in order to achieve overall quality in a *large* program

What is high quality code?

In summary, we want our code to be:

1. Correct
2. Easy to change
3. Easy to understand
4. Easy to scale (modular)

These qualities also allow for increased complexity

What we will cover in CSE 331

- Everything we cover relates to the 4 goals
- We'll use Java but the principles apply in any setting

Correctness

1. Tools
   - Git, IntelliJ, JUnit, Javadoc, …
   - Java libraries: equality & hashing
   - Adv. Java: generics, assertions, …
   - debugging
2. Inspection
   - reasoning about code
   - specifications
3. Testing
   - test design
   - coverage

Changeability

- specifications, ADTs
- listeners & callbacks

Understandability

- specifications, ADTs
- Adv. Java: exceptions
- subtypes

Modularity

- module design & design patterns
- event-driven programming, MVC, GUIs

Administrivia

Who: Course staff

- Lecturer:
  - Dan Grossman
- TAs:
  - Ten great TAs, mix of veterans and new

- Office hours posted soon

Get to know us!

  - We're here to help you succeed

Staying in touch

- Message Board
  - Using Google groups this quarter (link on course home page; set preferences for email, display name, etc.)
  - Join in! Use for most discussions; staff will read/contribute
    - Use your @uw.edu Google identity for this – not CSE
  - Help each other out and stay in touch outside of class

- Course staff: **`cse331-staff@cs.washington.edu`**
  - For things that don't make sense to post on message board
  - Please do *not* send messages to individual TAs – we need to get the traffic centrally so we can route it appropriately

- Course email list: **`cse331a_au19@u.washington.edu`**
  - Students and staff already subscribed (your UW email address)
  - You must get announcements sent there
  - Fairly low traffic – one way (from staff to everyone)

Prerequisites

- Knowing Java is a prerequisite
  - We assume you have mastered CSE142 and CSE143

Examples:

- Difference between **`int`** and **`Integer`**
- Distinction between **`==`** and **`equals()`**
- Aliasing: multiple references to the same object, what does assignment (**`x=y;`**) *really* mean?
- Subtyping via **`extends`** (classes) and **`implements`** (interfaces)
- Method calls: inheritance and overriding; dynamic dispatch
- Difference between compile-time and run-time type

Lecture and section

- Both required

- All materials posted, but they are visual aids
  - Arrive punctually and pay attention (& take notes!)
  - If doing so doesn't save you time, one of us is messing up (!)

- Section will often be more tools and homework-details focused
  - And there may be short quizzes at the beginning of the hour!

Homeworks

- Biggest misconception (?) about CSE331

  "Homework was programming projects that seemed disconnected from lecture"

  - If you think so, you are making them harder!
    - Reconsider
    - Seek out the connections by thinking-before-typing
    - Approaching them as CSE143 homework won't work well
    - Don't keep cutting with a dull blade

- First couple assignments are "more on paper", followed by software development that is increasingly substantial

Late Policy

- Assignments must be submitted by deadline. Full stop.
- But, stuff happens (bugs, computer crashes, …)
- So:
  - Up to **4** times this quarter you can turn in a homework assignment **one** (**1**) day late **=>*max*<=**
  - That's it. Not accepted for credit after that.
  - Late days are 24-hour chunks
- Why?
  - Keep you on schedule (most important)
  - Allow staff to get feedback to you before next deadline
- This is almost certainly different from what you're used to. No excuses for not knowing what the policy is.

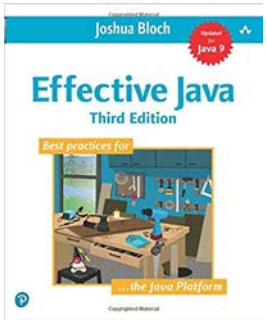
Academic Integrity

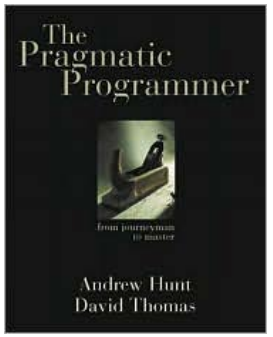
- Read the course policy carefully
  - Clearly explains how you can and cannot get/provide help on homework and projects
- Always explain any unconventional action
- I have promoted and enforced academic integrity for > 20 years
  - Great trust with little sympathy for violations
  - Honest work is the most important feature of a university (or engineering or business or life). Anything less disrespects your colleagues (including course staff) and yourself.

Books

Required textbooks

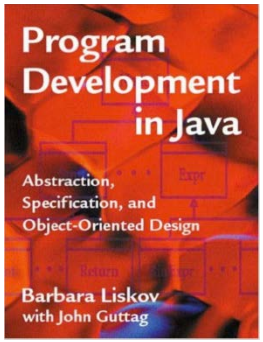
- *Effective Java* 3rd ed, Bloch (EJ)
- *Pragmatic Programmer*,
  will support both editions





Other useful books:

- *Program Development in Java*, Liskov & Guttag
  - would be the textbook if not from 2001
- *Core Java* Vol I, Horstmann
  - good reference on language & libraries



Readings (and quizzes)

- These are “real” books about software, approachable in 331
  - Occasionally slight reach: accept the challenge
- Overlap only partial with lectures
- Want to make sure you “do it”
  - Reading and thinking about software design is essential
    - Books may seem expensive given your budget, but very cheap as a time-constrained professional
  - Quizzes
  - Material is fair-game for exams

Books? In the 21st century?

- Why not just use Google, Stack Overflow, Reddit, Quora, …?
- Web-search good for:
  - Quick reference (What is the name of the function that does … in Java? What are its parameters?)
  - Links to a good reference
- (can be) Bad for
  - Why does it work this way?
  - What is the intended use?
  - How does my issue fit into the bigger picture?
- Beware:
  - Random code blobs cut-and-paste into your code (why does it work? what does it do?)
  - "This inscrutable incantation solved my problem on an unknown version for no known reason"

Exams

- Midterm, October 28
- Final, December 9
- All the concepts, different format than homework
  - More information later
  - Old exams posted

You have homework!

- Homework 0, due online by **10AM Friday** (no late days)
  - Write (don't run!) an algorithm to rearrange (swap) the elements in an array
  - in O(*n*) time (and preferably in a single pass)
  - And argue (prove) in concise, convincing English that your solution is correct!

- Purpose:
  - Great practice
  - Surprisingly difficult (and useful calibration on what's easy!)
  - So we can build up to reasoning about large designs, not just 5-10 line programs

Written homework submission

- Using Gradescope (programming projects will use gitlab)

- Later today you will get mail from gradescope.com with login details (id = UW email address)
  - (If not registered, send mail to cse331-staff with your name, UW email, and UW student # – not netid – so we can create an account)
  - Then click on the link or follow the one on the course resource page, upload your file, and identify which pages have answers to which questions

- You get email when assignments are graded

Back to Goals

- CSE 331 will teach you to how to write correct programs
- What does it mean for a program to be correct?
  - Specifications
- What are ways to achieve correctness?
  - Principled design and development
  - Abstraction and modularity
  - Documentation
- What are ways to verify correctness?
  - Testing
  - Reasoning and verification

Programming is hard

- It is surprisingly difficult to specify, design, implement, test, debug, and maintain even a simple program
- CSE331 will challenge you
- If you are having trouble, *think* before you act
  - Then, look for help
- We strive to create assignments that are reasonable if you apply the techniques taught in class…
  … but likely hard to do in a brute-force manner
    … and almost certainly impossible to finish if you put them off until a few days before they're due

CSE331 is hard! (but very rewarding)

- You will learn a lot!
- Be prepared to work and to think
- The staff will help you learn
  - And will be working hard, too

- So let's get going…
  - Before we create masterpieces we need to hone our ability to reason very precisely about code…

A Problem

"Complete this method such that it returns the largest value in the first **n** elements of the array **arr**."

```
int max(int[] arr, int n) {
    ...
}
```

A Problem

"Complete this method such that it returns the largest value in the first **n** elements of the array **arr**."

```
int max(int[] arr, int n) {
    ...
}
```

What questions do you have about the *specification*?

Given a (better) specification, is there exactly 1 *implementation*?

Moral

- You can all write the code

- More interesting in CSE331:
  - What if n is 0?
  - What if n is less than 0?
  - What if n is greater than array length
  - What if there are “ties”?
  - Ways to indicate errors: exceptions, return value, …
  - Weaker versus stronger specifications?
  - Hard to write English specifications (n vs. n-1)

Concise to-do list

Before next class:

1. Familiarize yourself with website, do readings
   - Lecture slides will be posted on web evening before class
2. Read syllabus and academic-integrity policy
3. Do Homework 0 (see web calendar), due by **10AM Friday**! (no late days this time)
   - (send mail to cse331-staff with name, ID #, and UW Email address if not registered so we can add you to the gradescope course roster to turn in the assignment)
4. Start working on software installation [over weekend may be okay]