Motivation
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 courses and in industry

Specifically, how to write code of
• Higher quality
• Increased complexity

We will discuss tools and techniques to help with this and the concepts and ideas behind them
  – There are timeless principles to both
  – Widely used across the industry
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...

... when **people** are involved?

People have been known to

- walk into windows
- drive away with a coffee cup on the roof
- drive away still tied to gas pump
- lecture wearing one brown shoe and one black shoe
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 defective OS could sink the whole fleet
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
People Do Build Great Software

Full scope of the challenge:
- software is built by people, who make mistakes all the time
- surprisingly difficult to get even a small program to work
- needed to write hundreds of millions of lines of code
- each line gets harder to write as the program scale

Despite those challenges, we have lots of software that works
- hundreds of millions of lines of working programs
- products rarely fail because the software is too buggy

How do we do it?
How do we ensure correctness...

... when **people** are involved?

People have been known to
- walk into windows
- drive away with a coffee cup on the roof
- drive away still tied to gas pump
- lecture wearing one brown shoe and one black shoe

**Key insights:**
- Can’t stop people from making mistakes
- Can stop mistakes from getting to users
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%
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
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