Announcements

- Course evaluation: [https://uw.iasystem.org/survey/179905](https://uw.iasystem.org/survey/179905)

- Final review material on the web site
  - concepts that are fair game for the final
  - pay particular attention to 16su

- Working through readability reviews (good so far!)
Review from Lecture 1
What is the goal of CSE 331?

In short: to help you become better programmers

Specifically, to teach you how to write code of
• higher quality
• increased complexity

We will discuss tools and techniques to help with these
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**
   - e.g., type checking compiler, @Override
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%
   - none of these can be left out
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

In summary, we want our code to be:
1. correct
2. easy to change
3. easy to understand
4. modular
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% are errors are btw modules [Seaman, ‘08]
  – corner cases are more important with more users
• Comm. costs dominate schedules [Brooks, ‘75]

Corollary: quality must be even higher, per line, in order to achieve overall quality in a large program
What we covered in CSE 331

- Everything we covered relates to the 4 goals
- We used Java but the principles apply in any setting

**Correctness**
1. Tools
   - Git, Eclipse, 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

**Understandability**
- specifications
- Adv. Java: exceptions
- subtypes

**Modularity**
- module design & design patterns
- listeners & callbacks
- event-driven programming, MVC, GUIs
Advice
Write Less Code

- The best way to reduce bugs is to write less code.
  - more lines of code usually means more bugs

- The best way to improve your productivity is to write less code.
  - your time is valuable!
  - don’t waste it on unnecessary work
Promise as Little as Possible

- I.e., make your method specifications as **weak** as possible

- That means less work for you
  - see the previous slide!
  - don’t promise to solve problems you don’t actually have

- That makes your code easier to change in the future

- **Exception:** you can’t have preconditions in widely used libraries
  - clients will try out your code on every input
  - whatever you do becomes the specification no matter what you say about it
Limit the Use of Abstraction

• Only introduce abstraction if it will **pay for itself**

• Abstractions usually make certain kinds of changes easier
  - e.g., interpreter vs procedural design patterns
    - one makes it easier to add operations, the other to add types
  - ADTs make it easy to change the data representation
    - the latter is common when optimizing for efficiency

• Adding abstraction is usually more work
  - see the earlier slide!
  - still pays for itself if it makes the code easier to understand

• Adding abstraction *can* make the code harder to understand
Prefer Correctness to Efficiency

• We are notoriously bad at guessing what will be inefficient
  - if you guess wrong, you’ll waste time optimizing
    • see the earlier slide!

• On the other hand, we can be pretty certain that users won’t like it when the program crashes

• First, make it correct. Then, find out what is slow and optimize

• Example: copying mutable inputs and outputs
  - you can remove these copies later if it turns out to be slow
Don’t Trust Other Programmers

• Write assertions to check preconditions on code they call
  - they should read the comments carefully, but they won’t

• Avoid representation exposure so they can’t break your code.

• Copy mutable inputs and outputs
  - better yet, prefer **immutable** types

• Don’t let other programmers extend your classes
  - relationship between sub- and super-class is often *intimate*
  - either design for subclassing or disallow it
  - prefer **composition** over inheritance
Don’t Trust Yourself Either!

- The first step is recognizing you have a problem…

- You will make mistakes — you can’t help that
  - but you can stop those mistakes (bugs) from getting to users

- Write assertions to check your assumptions
  - if you can have mistakes in your code, you can have them in your proofs of correctness as well

- Write assertions to check that your loop invariants hold.
- Write assertions to check that your representation invariants hold.
Fail Fast

• When you detect that something is wrong, just crash
  - (... if you can get away with it. Hide failures in client code.)

• This will make debugging much easier
  - search from the failure to the defect (bug) is shorter if the
    failure occurs close to the defect

• This limits additional damage from the defect
  - once we know there’s a mistake in our reasoning,
    it’s hard to know what else could go wrong
  - it could be very bad…
Write Tests before the Code

• It’s easier to have the energy for good testing beforehand
  - finishing the code feels like crossing the finish line

• Thinking through the tests often makes the code easier to write
  - forces you to think through all the cases you have to handle
  - helps you realize which cases are the same

• Confirmation bias makes it hard to realize the cases you missed after you’ve written the code

• Write tests before the code... then write more tests after
  - add tests for any special cases you missed
Test Code Should Be Obviously Right

• If your tests are wrong, they may not be testing anything at all

• For tests, correctness matters much more than anything else
  - throw elegance and efficiency out the window
  - throw changeability out the window (most of the time)

• It’s kind of fun to write brain-dead code
  - take a break from style, efficiency, etc.

• Any code that is not obviously correct needs its own tests
Have Fun

• Programming should be fun

• You get to…
  – create solely with the power of your imagination
  – positively affects the lives of large numbers of people