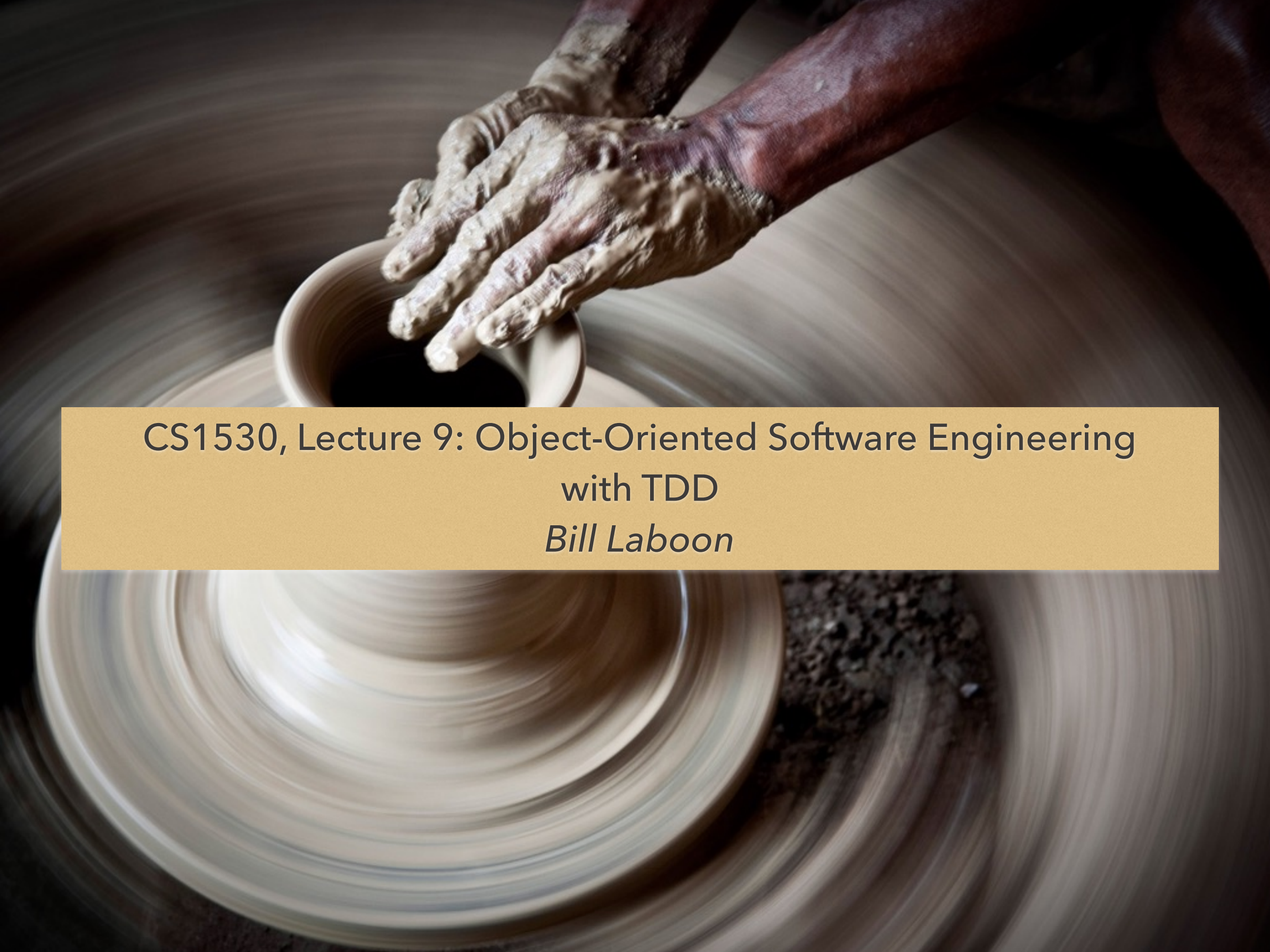


CS1530, Lecture 9: Object-Oriented Software Engineering with TDD

Bill Laboon

STARTING OUT: THE WALKING SKELETON

- Why make a walking skeleton?
  - Ensure that tools, frameworks, etc. work
  - Ensure everyone is on same page and can work together
  - Focus on the foundation of the application before worrying about details

WHY A WALKING SKELETON?

- Building
- Deployment
- Testing
- … all can be difficult, more difficult than code!
- Much easier to figure them out now when there is just a baseline, instead of figuring out if there is an issue with your code or your configuration

WHY A WALKING SKELETON?

- Much easier to do this now than later!
- Ensures that you have continuous, good test coverage and deployment possibilities from the start
- Allows for better “savepoints”
- Exposes uncertainty earlier

WALKING SKELETON AS THE FUTURE

- Many languages/frameworks are gaining popularity because of their ability to quickly make a walking skeleton: Rails, Django, Gradle, Maven, Yesod…
- Big jump in complexity from line-by-line code to system architectures
- “Convention over configuration”

YAGNI IS NOT A "GET OUT OF JAIL FREE" CARD

- YAGNI does not mean "don't think about the future"
- YAGNI means not to unnecessarily add features or overengineer
- In some methodologies (e.g. Waterfall, Cleanroom), YIGNI ("you is gonna need it") - you plan everything out ahead of time
- YAGNI is not an excuse not to prepare for issues or not to plan

TDD IN A NESTED LOOP

- Come up with a systems-level test (can be manual or automated) for a feature or user story
- Use TDD to reach that point
- Once done, figure out next feature/story to work on and loop
- This is a variant on something called ATDD (Acceptance Test-Driven Development)

```
// includes Walking Skeleton
Project p = new Project();

for (int j=0; j < NUM_FEATURES; j++) {
    at = writeFailingTest(FEATURE_LEVEL);
    while (moreCodeToWrite) {
        Test t = writeFailingTest(UNIT_LEVEL);
        Code c = writeCodeToMakeTestPass(t);
        Code rc = refactorCode(c);
        addCodeToProject(rc, p);
        if (p.test(at) == true) {
            continue; // loop to next feature;
        } else {
            /// keep on keepin' on
        }
    }
}
```

ACTUALLY DOING TDD

- What should I start with?
- For Scrum, I prefer the happy path first - ensure that your method can do the basic functionality you need it to do
- Ideally, write at least one test per equivalence class

NOT ALWAYS A GREAT IDEA!

- Like most other topics in this class, very little is set in stone - our field is young (computer -> calc -> watch)
- For mission-critical work, start with what can go wrong before even assuming things can go right!

WRITE TESTS YOU'D WANT TO READ

- Most developers spend more time **reading code** than **writing code**
- Be clear and concise
- Explain what the test is doing - comments are your friends
- Test one specific thing per test - almost always better to have more unit tests than fewer
- Why? Specific, focused tests are better at telling you what is failing

REMEMBER TO CHECK THAT TESTS SHOULD ALWAYS FAIL FIRST

- A test that doesn't fail may be either a superfluous test (checks something you've already tested) or a tautological test (will always pass, e.g. assertTrue(true) or missing assertions)
- These add bloat to your test suite and may give you a false sense of security

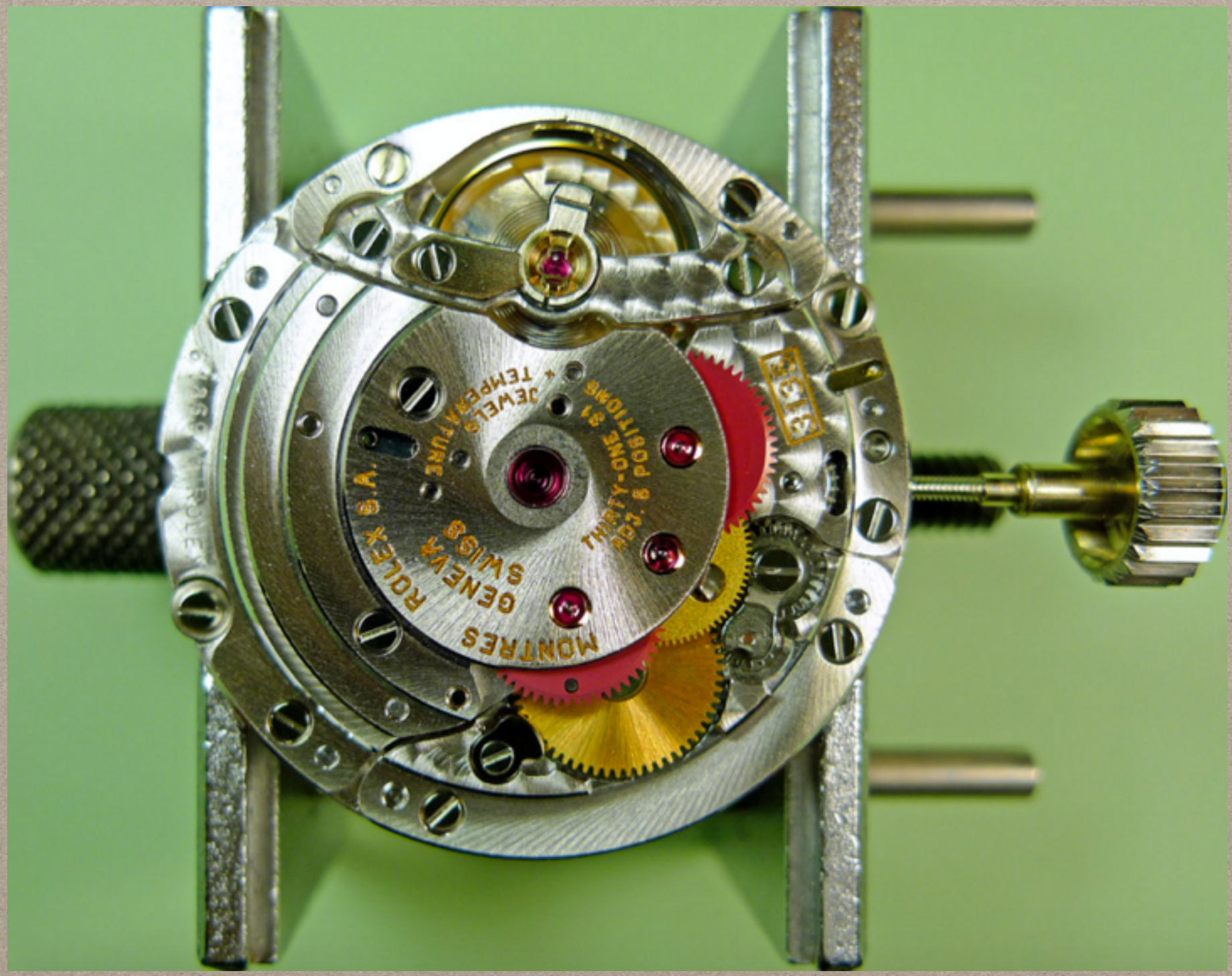
REMEMBER YOU ARE CHECKING BEHAVIOR

- Your unit tests are calling methods but really what you are concerned about is the expected behavior:
  - What values are returned?
  - What state is changed (e.g. attribute values)?
  - What side effects occur (e.g. writing to a file, printing to screen, etc.)

HOW MANY TESTS TO WRITE?

- Just like asking “what is the best flavor of ice cream?” - It will vary by domain (and taste)
- Good heuristic for this class: at least one test per method, additional tests for failure cases - ideally one test per "kind of expected input-output mapping”.
- This is a great thing to discuss during sprint retrospectives / planning! Are people finding defects that unit tests could have caught? Or are you overtesting?

A WATCH AS GUIDE FOR OBJECT-ORIENTED STYLE



From *"Rolex Caliber 3135 – Still worthy of the crown after all these years?"* http://www.chronometrie.com/rolex3135/rolex3135.html

A BEAUTIFUL EXAMPLE OF OBJECT-ORIENTED DESIGN!

From ABlogToWatch, ***Rolex Submariner Review: 114060 & 116610*** http://www.ablogtowatch.com/rolex-submariner-114060-116610-watch-review/

COMPLEXITY DIFFERENTIAL

- Note how complex the internals are compared to the input (crown) / output (hands)
- An ordinary user of the watch does not need to adjust mainspring vs balance spring, balance wheel, BPM, etc. - only turn crown or press/depress it
- Keep your classes like this!
- Hide complexity - keep “intellectual humility”

INFORMATION HIDING AND ENCAPSULATION

- Unless you are a watchmaker (or a watch nerd), you'll never even know these things exist
- Allows you to work at a higher abstraction level - "what time is it?" instead of confirming how many times the balance wheel has rotated compared to a baseline
- Information hiding - no need for knowledge of internals
- Encapsulation - can only interact with system via well-defined interface

POSTEL'S PRINCIPLE (A/K/A THE ROBUSTNESS PRINCIPLE)

- "Be conservative in what you send, but liberal in what you accept."
- A paraphrase of what Jon Postel wrote in the specification for TCP
- In other words, code as closely to the specification as you can, but don't assume that others will do so.
- Be prepared for invalid, outdated, or incorrect data

NOT ALWAYS A GOOD IDEA

- ON ERROR RESUME NEXT - A VB command saying that if an error occurs in a line, just go to the next one

- The VB compiler is being very liberal in what it accepts… but is that helping to build better code?

- What if received data is malformed? Should we try to guess what the user meant? Where is the line?

LINKING CLASSES TOGETHER

- Hopefully, much of that is review from CS0401
- But we don’t talk about how classes interact with each other in lower-level courses
- This will become a large problem as programs and systems grow in size and complexity
- Consider the following a “sneak peek” at our lecture on Object-Oriented Design

COUPLING

- Classes are coupled if a change in one means a change in another

// Tightly coupled - A change in any of the objects
// would mean changing code all over the place
DogFactorySingleton.getDogInstance("Fido").getFace()
.getJawMuscleSet().contractMuscles(7, 13);

// Loosely coupled - Minimal changes necessary
dog.bite("Person");

COHESION

- A class should have one area of concern - no “god” classes which do all sorts of things
- Methods / attributes all relate to one specific concept or represent one particular object
- High cohesion = separation of concerns
- Low cohesion = Big blob o’ code

CLASS WITH HIGH COHESION

```
public class Dog {
    public void bark();
    public void move(Location l);
    public int getHeight();
    public int getWeight();
}
```

CLASS WITH LOW COHESION

```
public class Stuff {
    public void shutdownSystem();
    public int getNumMonkeys();
    public String printMenu();
    public void squeak();
}
```

GOAL: LOOSELY COUPLED, HIGHLY COHESIVE CLASSES

- Classes should do one thing and do it well
- Classes should be able to used by many different other classes easily
- Classes should be flexible
- System should be flexible

ABSTRACTIONS

- Systems are built on abstractions
- You write classes, which are an abstraction of Java code
- Java code is an abstration of JVM bytecode
- JVM bytecode runs on the JVM, which is an abstraction of the actual hardware
- etc.

ABSTRACTIONS ARE GOOD!

- You can think of the history of programming as piling up abstractions on top of each other
- I couldn’t get anything done if I worried about what x86 instructions were executing with every keystroke

LEAKY ABSTRACTIONS

- Abstractions should ideally abstract away all unnecessary implementation details of what they are hiding
- If they do not, this is called a “leaky abstraction”
- A leaky abstraction “leaks” details of its implementation / underlying details which those using the abstraction do not need to know

EXAMPLES OF LEAKY ABSTRACTIONS

- SQL: Performance characteristics can vary dramatically based on which columns are accessed, in which order joins occur, etc. Not obvious from SQL code!
- Accessing a file on a network share drive: many errors which can occur here (network failure, external server power failure, etc.) which would never happen for a local file.
- TCP over IP - “guarantees” a reliable connection, but you don’t always actually get one!

LEAKY ABSTRACTION CLASS

```
// Dog should be able to move and bite
// Implementation details (muscles) should
// not be visible!

public class Dog {
    public void bite(int strength) { … }
    public void move(Location l) { … }
    public void tenseMuscle(int muscleId){
      …
    }
    public void relaxMuscle(int muscleId){
      …
    }
}
```

SPOLSKY'S LAW OF LEAKY ABSTRACTIONS

"All non-trivial abstractions are, in some sense, leaky."

-Joel Spolsky

http://www.joelonsoftware.com/articles/LeakyAbstractions.html

WHY?

- You can never fully abstract everything away
- The abstraction is, at heart, implemented somehow
- For edge cases or performance, sometimes you will need to know how data is being abstracted