Software Engineering

Or how to build useful, reliable, flexible systems
What is Software Engineering?

- The architecture or design or structure of software.

- Think of bridges and factories and...
  (In fact you’ll see those names again.)

- Similar to data structures
  - which worry about the correct way to design, represent, and manipulate data.
  - here we worry about correct way to design, represent, and manipulate the whole system.
What Are the Tools of Software Engineering?

Object-oriented design
- we’ll review, but I assume you know basic OO.

UML
- Unified Modeling Language
  - A “meta-language” to describe OO architectures
- Booch, Jacobson, and Rumbaugh in early 90’s, together at Rational Rose (“Three Amigos”).
What Are the Tools of Software Engineering? (cont.)

Design Patterns
- Suites of effective OO designs that can be used over and over and...
- Popularized in mid 90’s Design Patterns book by Gamma, Helm, Johnson, and Vlissides (“Gang of Four”).

Software Development Process (“Life Cycle”)
- requirements, to software implementation, to quality assurance – management!
- Waterfall model, spiral model...
What Tool is Missing?

Mathematics!

- No good mathematical construct for designing and testing code.
- Want a structured set of math rules that easily ensure success.

Homework: Make your fame and fortune by doing this.

Due date: Jan 1, 2015.

Note: Aspects of this are impossible – “halting problem” from Theory of Computation. But perhaps can get “close”...

- no way to test if arbitrary program ends successfully.
- no way to test if arbitrary program ever prints “Yo!”
- can prove if particular program prints “Yo!” (see end of semester)
What is a Well-Engineered Project?

1. Useful and usable
   - Makes life easier or better

2. Reliable
   - Not too many bugs

3. Flexible
   - Must be able to fix bugs
   - Must be able to add new features
Examples of Poorly Engineered Software

Code with millennium bug
- **Problem: Flexibility.** (Legacy issues, fixing bugs)
- Impact: Fortunes spent fixing code.

DIA Baggage Handling
- **Problem: Reliability.** So complicated that it had too many bugs
- Impact: Airport opened late. Software development 50% over budget (a cool $100 million extra!).
Examples of Poorly Engineered Software (cont.)

Therac-25

- In mid 80’s, several people died when software failure caused x-ray machine to deliver fatal doses.
- Problem: Flexibility. Tried to add new features to an old model.
- Impact: Obvious.

London Ambulance System

- Early 90’s, dispatch software failed
- Problem: Unusable and unreliable.
- Impact: Ambulances did not reach people, and people died.

YOUR WORK CAN HAVE AN IMPACT ON PEOPLE’S LIVES!
What Practices Help Make Good Engineering?

Coding practice:

- Modularity
  - “Plug and play” components (e.g., add features – flexible!)
  - Easily replaced/removable components (e.g., fix bugs – reliable!)
  - How modularize?
    - OO. UML. More to follow.

- Use Common Design Patterns
  - So others can figure out your code
  - To guide/streamline your thinking process

Business practice:

- Software life-cycle
  - Minimize risk
  - Quality assurance finds bugs
Example of a Good Design: Modular Computer

Communications between modules

i/o ports  cd rom
cooling  hard drv
power  memory
processor  ETC.

Modular computer: replace component as soon as it is obsolete.
Modules

- Replaceable
  - for repairs, updates, new features

- Reusable
  - Write code once, then use again in a later project next year.
  - (Ok, yeah, sure. Good idea. But reality hasn’t lived up to hype.)
Creating Good Modules: Coupling

- **Coupling (or dependency or connectivity)**
  - The number of connections between modules.
    - Like wires between modules in the “modular computer” example.

- **Want **low coupling**
  - Each module is connected to very few others.
  - In “modular computer” example, power supply has high coupling (bad), but fan has low coupling (good).

- **Why?**
  - So module is easily replaced without messing up the other modules to which it is connected.
  - So a change in one module effects few others.
  - So a change in the code “here” has no effect in the code “over there”

(No interconnected spaghetti code!)
Creating Good Modules: Cohesion

Cohesion
- How closely related are the internal workings of a module.

Want **high cohesion!**
- A “Car” module *should* have steering wheel routines.
- A “Car” module *should not* have routines that deal with gasoline prices.
- In modular computer example, power supply should have a transformer but not any RAM.

Why?
- If replace a module, then know what new version should contain.
- Imagine replacing the power supply module and discovering that you no longer had any RAM? Who would have known that your new power supply needed RAM?
Creating Good Modules: Encapsulation

- **Encapsulate**
  - Hide sensitive or unnecessary info.

- **Want controlled access**
  - Module has **interface** that everything else uses to access it.
  - In code: hide internal workings, privatize variables, getter/setter methods.

  - Example: electrical outlet. Always takes the same kind of plug, no matter what. Never see the wires or worry about polarity, etc. – the stuff in the wall is encapsulated. (The interface is the outlet.)

- **Why?**
  - So no one can mess up the internal workings of your module.
  - In “modular computer” example, imagine if the power supply’s output voltage wasn’t encapsulated. A new fan module could decide to change the voltage from 5 volts to 12 volts. But poof! Now your processor blows up.
Module Upshot

Good Modules (think “pluggability”)

- Low coupling
- High Cohesion
- Encapsulate (interface)

Remember

- **Definition: Cohesion** – How closely the operations in a routine are related.
- **Definition: Coupling** – The strength of the connection between two routines.
Modules Through OO

- OO does not inherently give low coupling, etc.
- OO *does* facilitate good modules

- We will review OO.
Good Software Design: Summary

**Good software**
- Useable
- Reliable
- Flexible

**Tools**
- OO
- UML
- Life-cycle
- Design patterns

**Coding Approach**
- Modular code!
- Design patterns

**Business Approach**
- Life-cycle

So, on to modularity...