CMPSCI 187: Programming With Data Structures

Lecture 3: Software Engineering Overview
12 September 2011

Software Engineering Overview

- Leftover from Lecture 2: dog example, methods, the call stack, exceptions
- Software versus Programs
- Software Engineering versus Engineering
- Eight Goals of Software Quality

Dynamic Typing Example

```
public class Dog {
   public void bark {
      System.out.println(“Woof!”);}}
public class Terrier extends Dog {
   public void bark {
      System.out.println(“Yip!”);}
   public void dig {}}

Dog cardie = new Dog();
Dog duncan = new Terrier();
cardie.bark(); // Woof!
duncan.bark(); // Yip!
cardie.dig(); // won’t compile
duncan.dig(); // won’t compile
Terrier d = (Terrier) duncan;
d.dig(); // works
d.bark(); // Yip!
Terrier c = (Terrier) cardie; // compiles, ClassCastException
```

The Call Stack and Exceptions

- When one method calls another, the context of the first is saved to return to.
- If that second method calls a third, both must be saved. When we restore context, we return to the one **that was saved last.**
- When we save a bunch of things and only want to access the last-saved one first, we need a **stack**.
- An exception interrupts a method. If the exception can be thrown, it goes to the calling method, where it might stop execution or be thrown to the method calling that, and so on.
- The operating system prints the **call stack** when an exception stops execution -- which methods were in progress when the exception happened.

Back to Software Engineering

- What is **software**? How is it different from CMPSCI 121 or 187 programs?
- Wikipedia (this week): "Software is a conceptual entity which is a set of computer programs, procedures, and associated documentation concerned with the operation of a data processing system."
- A single program solves a single data processing problem, with specified input and output. The programs in a piece of software may have different users, different desired behaviors, different versions, etc.
- Note "procedures", "associated documentation" -- everything in the system but the hardware.
- Examples: SPIRE, Mac OS X, *Civilization,* Facebook, the Internet protocol...

Software Engineering versus Real Engineering

- Engineering is the application of science and mathematics to affect reality, particularly by constructing artifacts.
- If you wanted to build a bridge across the Connecticut from Hatfield to Hadley, a civil engineer could give you a variety of designs and cost and building time estimates for each.
- “Software engineers” apply science and engineering to build artifacts, but they are very bad at estimating cost or completion time.
- Real engineers usually apply known techniques and materials with known characteristics -- software engineers not so much.
- Breakthrough software, e.g. Facebook, changes the environment it exists in.

Aspects of Software Quality

- Lewis and Chase list eight in Section 1.1: **correctness, reliability, robustness, usability, maintainability, reusability, portability,** and **efficiency.**

- Quality is in the eye of the **stakeholder:** maybe a paying customer, maybe not. Developers and maintainers care about the internals, users about the observable behavior.

- Data structures are reusable components, so after this lecture we will focus most on reliability, robustness, reusability, and efficiency.

Correctness

- A program is correct if it meets its **specification**: in this course we'll have fairly simple specifications, and it will be fairly easy to determine correctness of your programs.

- In CMPSCI 320 you will see that specifying the desired behavior of software is at least as hard as creating software with that behavior.

- A truly complete specification would deal with all possible anomalous inputs to the system, and how it should deal with each. We won't worry that much about strange inputs in our programs here.

- In general we determine correctness by a combination of **testing** and **analysis**. One reason to study mathematical technique, as in CMPSCI 250, is to be able to formally prove things about the behavior of programs.

Reliability

- A **software failure** is “any unacceptable behavior that occurs within permissible operating conditions” -- reliability is the relative absence or rareness of software failures.
- Absolute reliability is usually impossible in the real world, with the limits on our ability to create correct programs, wildly varying conditions of use, and even malicious attempts to deny service or break data security.
- Real engineers, who have been designing and managing life-critical systems for centuries, tend to look down on software engineering’s performance.
- Fatal software failures are rare, but costly ones are fairly frequent and annoying ones are commonplace.

Robustness

- How well does the software handle conditions for which it was not specifically designed?
- Users can’t be expected to follow strict protocols, and they get annoyed with software that punishes small deviations from the expected.
- But making assumptions about what the user meant can also be dangerous.
- Error handling should minimize the disruption from bad input, e.g., ask the user to reenter it rather than crash the system or start erasing things.
- The need for robustness varies, and a good specification says how important it is and when.

Usability

- When computers are used by humans, good software design will take account of what the humans want, how they perceive things, and how they learn.
- There is a whole field of **human-computer interaction (HCI)**, which we introduce in CMPSCI 325. It integrates art with engineering, and focuses most on user interfaces.
- Most of our knowledge about HCI is empirical -- we look at what works in the marketplace, try to survey users, beta-test products, etc.
- The visionaries (Gates, Jobs, Berners-Lee, Zuckerberg) see this in advance.
- User interfaces are hard and not our focus in 187, so this isn't a priority for us.

Maintainability

- Software has a **life cycle** of specification, design, development, testing, maintenance, and obsolescence.
- As it is used, possibly for years, requirements change, improvements are thought of, and errors are corrected -- most likely not by the original developers.
- The more sense the code makes to someone who didn't write it, the easier it is to maintain.
- The key technique for maintainability is **modularity**. Pieces of the software must have clearly defined interfaces with each other and must be understandable as abstractions. We will see this concept a lot, and it is inherent in the Java language.

Reusability

- It is expensive and time-consuming to solve the same problem twice.
- Well-known, well-tested solutions to specific problems can and should be reused. Sometimes you can use **commercial, off-the shelf** software.
- To make something more reusable, write programs to solve the widest, most general problem possible.
- We can contrast **top-down** with **bottom-up** design. The former takes a big problem and breaks it into pieces until the pieces can be solved. The latter builds up a set of tools and puts them together to solve larger and larger problems. A good design process, even for a CMPSCI 187 project, usually uses both.

Portability

- Good software can be used in a variety of different environments.
- Different computers have different machine languages, but with high-level languages, compilers, and interpreters, the same program can run on each.
- Java was designed with the web in mind -- a Java program does not “know” what sort of computer it is running on.
- Commercial software usually has to access its operating system in ways more complicated than our programs do (e.g., input and output) so commercial programs have versions for Windows, Mac, etc. and **porting** a piece of software is a major undertaking.
- You can emulate one OS on another, but there is a cost in **efficiency**.

Efficiency

- Computer programs use resources, such as **time** and **memory**.
- You may know how to solve a problem, and even write correct code implementing the solution, but be unable to solve it in the real world because your code uses too many resources.
- Typically we look at a problem where the **input size** is a variable. As input size increases, time and memory usage increase, to where there is a limit on the practically feasible size.
- With reused components like data structures, understanding resource use is vital. In this course we'll begin the study of **computational complexity**, the science of how resource consumption, particularly running time, increase with input size.