CSC 143

Software Design Principles (part 1)

Object Oriented Principles

- Abstraction
  - Can you drive a car?
  - Do you know how an internal combustion engine works?
  - Other examples?
  - What experience have you had with data abstraction? Procedural abstraction?
- Encapsulation
  - Given the datatype **int**, what are some of the operations
- Inheritance, Polymorphism **(more on these later)**

The Software Development Lifecycle

Typical stages (iterate as needed):

- Analysis and Specification
- Design
- Coding
- Testing
- Production
- Maintenance

Documentation throughout

Which stage is the largest?

Let's discuss some key stages…

What Constitutes a Good Design?

- Correctness (of course!)
- Modularity
  - Module: a piece which has some independence
- Ease of maintenance / extendability
- Fail-safe programming
- Style

Beginning an OO Design

Programming can be viewed as building a **model** of a real or imaginary world within the computer

✓**Objects** can model *things*
Examples?
✓ **Objects** have
Responsibilities – what you can ask them to do
Properties – what they know

Start with a dialogue

- Nouns will generate ideas for classes and properties
- Verbs will generate ideas for responsibilities (methods)

Role play to refine

Public Interface

- One way to aid evolution is to define good **public interfaces** independent from the implementation
- An interface specifies to clients (users of the class) what are the operations (methods) that can be invoked; anything else in the class is hidden
- What are the benefits of a simpler, public interface and hidden implementation?

Coupling and Cohesion

- **Coupling** – the degree to which a class interacts with or depends on another class
- **Cohesion** – how well a class encapsulates a single notion
- A system is more robust and easier to maintain if
  - Coupling between classes/modules is minimized
  - Cohesion within classes/modules is maximized

Documentation: Contract/Specification

- Clients and suppliers of an abstraction (e.g. a method or class) agree on the **public interface**
  - **a *contract* between the 2 parties**
  - **gives rights and responsibilities of each**
- A **specification** is a more complete contract, that also indicates
  - **any restrictions on the allowed argument values**
  - **what the return value must be, in terms of the argument values**
  - **any changes in state that might happen, and when**
  - **when any exceptions might be thrown**

Preconditions and Postconditions

- Precondition: something that must be true before a method can be called
  - constraint on client : assumed by method implementation
- Postcondition: something that is guaranteed to be true after the method finishes
  - constrain on implementor: assumed by client
  - only guaranteed if precondition was true when method called

Examples?

Invariants

- An invariant is a condition that is always true
- Special case: class invariant – an invariant regarding properties of class instances; something that is always true for all instances of the class.
  - Examples
    0 <= size <= capacity
    price > 0
  - Note: a class invariant might not hold momentarily while a method is updating related variables, but it must ***always*** be true by the time a constructor or method terminates

Documentation Saves Time (really!)

Preconditions, postconditions, and invariants are incredibly useful

- ✓Include all non-trivial ones as comments in the code
  (e.g. using @param, @return, @throws)
  - These are ***essential*** parts of the design and a reader must understand them to understand the code
  - If you don't write them down, the reader (who may be you) will have to reconstruct them as best he/she can
- ✓Whenever you update a variable, double check any invariants that mention it to be sure the invariant still holds
  - May need to update related variables to make this happen
  - May need to add preconditions (e.g. no negative deposits) or explicit checks (e.g. overdraft) to ensure they hold
  - Helps you write your code!

Fail-safe Programming: precondition failures

- Principle: Check and stop early!
  - The sooner a precondition failure is detected the better
- Who is responsible for checking?
  - Most logical place is at the beginning of the called method
  - What step(s) to take?
- How aggressive should we be about checking?
  - Can overdo it

Focus on checking preconditions that wouldn't crash already, and that would lead to obscure behavior if they weren't detected

Testing & Debugging

- Testing Goals
  - A controlled experiment to verify that the program works as intended
  - Be able to recheck this as the software evolves
- Debugging
  - Strategies and questions:

    What's wrong?

    What do we know is working? How far do we get before something isn't right?

    What changed? Even if the changed code didn't produce the bug, it's fairly likely that some interaction between the changed code and other code did.

Unit Tests

Idea: create small tests that verify individual properties or operations of objects

- Do constructors and methods do what they are supposed to?
- Do variables and value-returning methods have the expected values?
- Is the right output produced?

- **Lots of small unit tests, each of which test something specific, not big, complicated tests**
  - If something breaks, the broken test is a clue about where the problem is

Writing Tests

When?

Before you write the code!!!

Why?

- Helps you understand the problem and the public interface
- Makes you think about code design and implementation
- Gives you immediate feedback once the code is written

Where to put the tests?

- **Good: an interactions window**
  - **Great way to prototype tests**
  - **Way too tedious to do any extensive testing**
- **Better: static methods**
  - **First step to automated testing**
  - **Can have too many to do a thorough job, or**
- **Drawback: someone has to check the output**

- **Ideal: automate this by writing self-checking tests**

JUnit

- Test framework for Java Unit tests
- Idea: implement classes that extend the JUnit TestCase class
- Each test method in the class is named testXX (starting with "test" is the key)
- Each test performs some computation and then checks the result
- Optional: setUp() method to initialize instance variables or otherwise prepare before each test
- Optional: tearDown() to clean up after each test
  - Less commonly needed than setUp()

Example (from DrJava help)

```
import junit.framework.TestCase;
public class CalculatorTest extends TestCase {

  public void testAddition() {
     Calculator calc = new Calculator();
     int expected = 7;
     int actual =  calc.add(3, 4);
     assertEquals("adding 3 and 4", expected, actual);
  }

   public void testDivisionByZero() {
     Calculator calc = new Calculator();
     try {                         // exception handling – coming attraction
        calc.divide(2, 0);
         fail("should have thrown an exception");
     } catch (ArithmeticException  e) {
           // do nothing – this is what we expect
     }
  }
  …
}
```

You know there's a problem, now what?

Avoid changing code randomly and seeing if that helps!!

Debugging techniques to help identify problem code:

- ✓ System.out statements
- ✓ Integrated debugger tool
  - Step by step execution
  - Watch state of objects, values of local variables
- ✓ Others?

Software Engineering and Practice

- Building good software is not just about getting it to produce the right output
- Many other goals may exist
- "Software engineering" refers to practices which promote the creation of good software, in all its aspects
  - Some of this is directly code-related: class and method design
  - Some of it is more external: documentation, style
  - Some of it is higher-level: system architecture
- Attention to software quality is important in this class
  - as it is in the profession