Software Engineering Techniques

• Low level design issues for *programming-in-the-large*.

• Software Quality

• Design by contract
  - Pre- and post conditions
  - Class invariants

• Ten do

• Ten do nots

• Another type of summary
Software Quality

- **Correctness**: Is the ability of software to exactly perform their tasks, as defined by the requirements and specifications.

- **Robustness**: Is the ability of software to function even in abnormal conditions.

- **Extendibility**: Is the ease with which software may be adapted to changes of specifications.

- **Reusability**: Is the ability of software to be reused, in whole or in part for new applications.

- **Compatible**: Is the ease with which software may be combined with others software.
Other Software Quality

• Efficiency: Is the good use of hardware resources.

• Portability: Is the ease with which software may be transferred to various hardware and software environments.

• Verifiability: Is the ease of preparing acceptance procedures, e.g., test data and methods for finding bugs and tracing the bugs.

• Integrity: Is the ability of software to protect its components against unauthorized access and modification.

• Ease of use: Is the ease of learning how to use the software, operating it, preparing input data, interpreting results and recovering from errors.
Design By Contract

• Purpose: To increase software quality by giving each part of a software product certain obligations and benefits.

• Without contract
  ▪ All parts of a program take a huge responsibility
  ▪ All parts of a program check for all possible error possibilities (called defensive programming).
  ▪ This makes a large program larger and more complicated

• With contracts
  ▪ Methods can make assumptions
  ▪ Fewer checks for errors possibilities
  ▪ This makes a large program simpler.
Design By Contract, Example

• A stack example the *push* method.

• Client programmer
  ▪ Obligation: Only call *push(x)* on a non-full stack
  ▪ Benefit: Gets *x* added on top of stack.

• Class programmer
  ▪ Obligation: Make sure that *x* is pushed on the stack.
  ▪ Benefit: No need to check for the case that the stack is already full

• Think Win-Win!
Pre and Postconditions

• A *precondition* expresses the constraints under which a method will function properly.
  ▪ The responsibility of the caller to fulfill the precondition.

• A *postcondition* expresses properties of the state resulting from a method's execution.
  ▪ The responsibility of the method to fulfill the postcondition.

• Both preconditions and postconditions are expressed using *logical expressions* also called *assertions*.

• Other issues
  ▪ Class invariants
  ▪ Loop invariants
Java 1.4's `assert` Keyword

• An *assertion* is a boolean expression that a developer specifically proclaims to be true during program runtime execution [Source: java.sun.com].

• New to Java 1.4.

• Used for expressing both pre- and postconditions.

• Syntax:

```
assert expression1;
assert expression1 : expression2;
```
Java 1.4's `assert` Keyword, cont.

- Evaluation of an `assert` statement.

Evaluate `expression1`
if true
   no further action
else
   if `expression2` exists
      Evaluate `expression2` and use the result in a single-parameter form of the `AssertionError` constructor
   else
      Use the default `AssertionError` constructor
assert, Examples

assert 0 <= value;
assert 0 <= value : "Value must be positive " + value;

assert ref != null;
assert ref != null : "Ref is null in myFunc";

assert newCount == (oldCount + 1);

assert myObject.myFunc(myParam1, myParam1 );
import java.util.*;
public class AStack{
    private LinkedList stck = new LinkedList();
    private final int no = 42;

    public boolean full() {
        if (stck.size() >= no) return true;
        else return false;
    }

    public boolean empty() {
        return !full();
    }

    public void push(Object v) {
        // precondition
        assert !full(): "Stack is full";
        stck.addFirst(v);
        // postconditions
        assert !empty();
        assert top().equals(v);
        // check no of elements increase by one
    }
}
Pre- and Postcondition, Example

```java
public Object top() {
    assert !empty();
    return stck.getFirst();
    // no post conditions
}
public Object pop() {
    assert !empty();
    return stck.removeFirst();
    assert !full();
    // check no of elements decrease by one
}
public static void main(String[] args) {
    AStack as = new AStack();
}
```
assert and Inheritance

class Base{
   public void myMethod (boolean val){
      assert val : "Assertion failed: val is " + val;
      System.out.println ("OK");
   }
}

class Derived extends Base {
   public void myMethod (boolean val){
      assert val : "Assertion failed: val is " + val;
      System.out.println ("OK");
   }
   public static void main (String[] args){
      try {
         Derived derived = new Derived();
         ...
      }
   }
}
assert and Inheritance, cont

- Preconditions cannot be strengthened in subclasses.
- Postconditions cannot be weakened in subclasses.
Class Invariant

• A *class invariant* is an expression that must be fulfilled by all objects of the class at all stable times in the lifespan of an object
  ▪ After object creation
  ▪ Before execution a public method
  ▪ After execution of a public method

• A class invariant is extra requirement on the pre and postconditions of methods.

• Class invariants can be used to express consistency checks between the data representation and the method of a class, e.g., after if a stack is empty then size of the linked list is zero.

• Class invariants cannot be weakened in subclasses.

• Not supported in Java.
Ten Dos

- Logical naming
  - Class name `p3452` vs. class name `Vehicle`
  - The foundation for reuse!

- Symmetry
  - If a `get` method then also a `set` method
  - If an `insert` method then also a `delete` method
  - Makes testing easier.
  - To avoid "surprises" for the clients.

- Add extra parameters to increase flexibility
  - `split (string str)` vs.
    - `split (string str, char ch default ' ')`
  - To anticipate "small" changes.
Ten Dos

• Set a maximum line size (80-100 characters)
  ▪ To avoid more than one thing being done in the same line of code
  ▪ To be able to print the code without wrapping. For code reviews

• Set the maximum of lines for a method
  ▪ What can be shown on a screen (30-60 lines)
  ▪ To increase readability
  ▪ To increase modularity

• Indent your code
  ▪ Increases readability

• Avoid side-effects
  ▪ If a method refers to an object in a database and the object does not exist then raise an error, do not create the object.
  ▪ Make program logic impossible to understand
Ten Dos

• Add comments in method
  ▪ Comment where you are puzzled yourself or is puzzled the day after you wrote the code
  ▪ Do not comment the obvious!

• Look at (and comment on) other peoples code
  ▪ Code reviews are a good investment
  ▪ Increases readability of code
  ▪ A good way to learn from each other

• Be consistent
  ▪ Can automate global changes with scripts
Ten Do Nots

• Make a method do more than one thing
  - `split_and_store (string str, char ch) vs. split (string str, char ch) and store (string_array)`
    - Makes the method more complicated
    - Decreases reuse

• Make a method take more than 7±2 parameters
  - Can parameters be clustered in objects?

• Make more than 4 level of nesting in a method
  - `if {if{if{if{if }}}}]]]`}
    - Decreases readability

• Make use of "magic" numbers
  - `WHERE employee.status = '1' vs WHERE employee.status = global.open`
Ten Do Nots

• Make use of Copy-and-Paste facilities
  ▪ Redundant code
  ▪ Make a new method or use inheritance
• Become mad and aggressive if some one suggest changes to your code.
• Have more than one return statement in a function
• Skip exception handling
• Skip testing
• Assume the requirement specification is stable
Summary

• Any fool can write code that a computer can understand. Good programmers write code that humans can understand. (Fowler)
• Debug only code - comments can lie.
• If you have too many special cases, you are doing it wrong.
• Get your data structures correct first, and the rest of the program will write itself.
• Testing can show the presence of bugs, but not their absence.
• The first step in fixing a broken program is getting it to fail repeatedly.
• The fastest algorithm can frequently be replaced by one that is almost as fast and much easier to understand.
Summary, cont.

- The cheapest, fastest, and most reliable components of a computer system are those that are not there.
- Good judgement comes from experience, and experience comes from bad judgement.
- Do not use the computer to do things that can be done efficiently by hand.
- It is faster to make a four-inch mirror then a six-inch mirror than to make a six-inch mirror.
  [Thompson's Rule for first-time telescope makers]
- If you lie to the computer, it will get you.
- Inside of every large program is a small program struggling to get out.