Week 5: Dataflow testing

- Dataflow models and diagrams
- Tracing pseudo execution
- "White box" testing

Dataflow versus logic flow

- **Dataflow diagrams** (DFDs) are a central element of structured systems analysis.
  - They specify what happens to data items as they pass among processes.
  - The don't necessarily specify strict time sequence
  - If you're curious, take COMP 320 or see http://en.wikipedia.org/wiki/Data_flow_diagram

- Note that they are entirely different from **flow charts** (or logic diagrams)

Flowcharts

- Flowcharts are an old-fashioned tool for:
  - Analyzing logic flow before coding.
    - Clear the programmer's thoughts if logic is complicated.
    - Easier to create, to read, and to change than low-level (e.g. assembly language) code.
  - Documenting a complicated module.
    - Help other programmers understand what the module does (or was intended to do)

- **Why do we rarely use them today?**
  - Are they helpful in planning tests?

Tracing pseudo execution

```
while(y!=0) {
    tmp = x%y;
    x = y;
    y = tmp;
}
```

Greatest common divisor function (iterative algorithm)
### Tracing pseudo execution
(continued)

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>tmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>while(y!=0) {</td>
<td>15</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>tmp = x%y;</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>x = y</td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>y = tmp;</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Tracing pseudo execution
(one more step--are we done?)

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>tmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>while(y!=0) {</td>
<td>6</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>tmp = x%y;</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>x = y</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>y = tmp;</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### A simpler approach to tracing pseudo execution

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>tmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>while(y!=0) {</td>
<td>6</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>tmp = x%y;</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>x = y</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>y = tmp;</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### White-box testing

- We recall that black-box test cases rely only on the specification of the MUT.
  - You can (some say should) write the tests before developing the module.
  - The tests can exercise all positive and negative aspects of the MUT's behavior.
- In contrast white-box test cases are inspired by inspecting the code.
  - Try to exercise all possible paths through the module.
  - Focus on tricky boundary values.
Example:

- A C++ function to count the number of 1 bits in an integer.
  - For example `countBits(13)` yields 3 (1101)

MUT v.1

```cpp
void countBits(char ch) {
  int count = 0;
  if (ch & 1) ++count;
  if (ch & 2) ++count;
  if (ch & 4) ++count;
  if (ch & 8) ++count;
  if (ch & 16) ++count;
  if (ch & 32) ++count;
  if (ch & 64) ++count;
  if (ch & 128) ++count;
  print . . . (etc.)
}
```

Where's the bug?
How many test cases would you need to find it?

MUT v.2

```cpp
void countBits(char ch) {
  int count = 0;
  for (int mask=1; mask <= 128; mask *=2);
  if (ch & mask) ++count;
  print . . . (etc.)
}
```

What's wrong now?
How would you test it?

MUT v.3

```cpp
int countBits(char ch) {
  int count = 0;
  for (int mask=1; mask <= 128; mask *=2);
  if (ch & mask) ++count;
  return count;
}
```

Is that better for testing?
Is it also better as general design?
MUT v.4

```c
int countBits(char ch) {
    int count = 0;
    for (char mask=1; mask !=0; mask <<=1);
    if (ch & mask) ++count;
    return count;
}
```

What will that do that the previous one won’t?

MUT v.5

```c
int countBits(char ch) {
    static const int bitCountTable[] =
    {0, 1, 2, 2, 1, 2, 2, 3, // 0-7
     1, 2, 3, 3, 2, 3, 2, 4, // 8-15
     1, 2, 3, 3, 2, 3, 2, 4, // 16-23
     2, 3, 4, 4, 3, 4, 3, 5, // 24-31
     1, 2, 3, 3, 2, 3, 2, 4, // 32-39
     ,
     5, 6, 7, 7, 6, 7, 7, 8}; // 248-255
    return bitCountTbl[ch];
}
```

What about this?

---

**Rule #1**

- The ease and thoroughness of testing a module are first influenced by the design and coding of that module.
  - No amount of brilliant testing can salvage an atrocious module.
- Repetition is the enemy!
- Computational modules shouldn’t print
  - Never?
  - OK for temporary debugging output

---

**Further clarification on mutability**

- Both are excellent books by very knowledgeable authors. I enthusiastically recommend them.
  - The disagreement is less about Bloch versus Meyers than about Java versus C++.

*Why should that make a difference?*
More terminology

- Bloch refers frequently to **value classes** or **value types**. What does he mean?
- We instantiate **objects** that
  - have a **value**, the values of the member data items.
  - have **meaning** in the **application domain**
- If the program can change the value, the object is **mutable**.
- In the early days of **OOP**, almost all classes were value types. That what OOP was about.
- What other kinds of classes and objects are there?
  - **Pseudo classes** (just packaging artifices)
  - **Program structures** (windows, ports, etc.)

C#, Java, and C++

- **Similarities:**
  - All 3 belong to the **C family**.
  - All 3 support mixed paradigm (object-oriented and procedural)
- **Differences**
  - C++ has maintained the original emphasis on value classes, while Java has departed from it. **Why?**
  - In particular:
    - Stroustrup (principal designer of C++) advises: "Make your objects as much like built-in types as possible"
    - But Java forces us to make them almost as different as possible!

What's wrong with Java for numeric computing?

- Almost everything
  - can't define operators
  - object reference semantics and overhead
- Java is really two languages:
  - one for primitive data (built-in types)
  - one for everything else (references)

Example

- This is legal in both C++ and Java:
  ```
  double creditLimit;
  double unitPrice = 49.95;
  double totalPrice = 0;
  int quantityOrdered;
  ...
  totalPrice += quantityOrdered * unitPrice;
  if (totalPrice > creditLimit) ...
  ```

  *Which lines would have to be changed to use a Money class in C++? in Java?*
**Changed to use Money class**

- **C++ version:**
  ```cpp
  Money creditLimit;
  Money unitPrice = 49.95;
  Money totalPrice = 0;
  int quantityOrdered;
  
  totalPrice += quantityOrdered * unitPrice;
  if (totalPrice > creditLimit) . . .
  ```

- **Java version:**
  ```java
  Money creditLimit;
  Money unitPrice = new Money(49.95);
  Money totalPrice = new Money(0);
  int quantityOrdered;
  
  totalPrice.addSet(unitPrice.mpy(quantityOrdered));
  if (totalPrice.greaterThan(creditLimit)) . . .
  ```

**Why doesn't Java support operator definition?**

> "... the language designers decided (after much debate) that overloaded operators were a neat idea, but that code that relied on them became hard to read and understand."

David Flanagan: *Java in a Nutshell*

- **Thus**
  ```java
  total += unitPrice * quantityOrdered
  ```

  must be harder to read and understand than

  ```java
  total.addSet(unitPrice.mpy(quantityOrdered))
  or
  total.setValue(total.getValue() + unitPrice.getValue() * quantityOrdered)
  ```

**What do Java programmers do?**

- Many Java programmers don't bother with numeric objects. They either:
  - ignore OOP altogether for numeric computing, as if they were coding in Fortran, or
  - use awkward wrapper classes, with accessor functions that violate OOP

- **Conclusion:**
  - Java is a horrible choice for computation.
  - But if you have to use Java, use numeric objects anyway.
    - Everything we've shown is possible (although ugly) in Java.
So what is Java good for?

- Java is very well suited to:
  - User interface ("front end") programming (console dialogs, forms data entry, window manipulation, etc.)
  - Internet deployment
  - Processor independence
  - International applications, esp. with non-Roman alphabets

What else?