Week 8 - Wednesday

COMP 3100

Last time

- What did we talk about last time?
- Construction techniques
  - Bought and customized
  - Built systems
- Kinds of programming languages
- Programming style

Questions?

More on Construction Techniques

Data organization

- Programs often include data, but how should it be organized?
- Data structures store the data in the program, but the data also needs to be stored between program runs or sent to someone else to use
  - Internal data vs. external data
- Common data organization approaches
  - Markup languages
  - Databases

Markup languages

- Markup languages format text using tags so that it's clear what the text means
  - XML (Extensible Markup Language) is a general purpose language for describing any kind of hierarchical data
  - HTML (Hypertext Markup Language) describes structured documents
  - JSON (JavaScript Object Notation) uses a key-value pair structure and some people like it more than XML because it has less overhead
- Many languages have libraries for automatically converting data structures to and from markup language versions

JSON

```
{
"place": "Boston",
"country": "USA",
"state": "MA",
"date": "31 Oct 2018",
"units": "F",
"high": 61,
"low": 54
}
```

XML

```
<temperatures>
    <place>Boston</place>
    <country>USA</country>
    <state>MA</state>
    <date>31 Oct 2018</date>
    <units>F</units>
    <high>61</high>
    <low>54</low>
</temperatures>
```

Databases

- Databases are such a huge topic that we can't meaningfully talk about them here
  - But many of you are taking COMP 3600 anyway
- Databases have many advantages over flat files (like markup files)
  - They can have rules for integrity
  - They are often stored on servers, allowing many different programs and users to interact with them
  - They're designed for efficiently retrieving information
  - Good backup techniques exist for databases
- Relational databases use tables to store **records** (rows) containing a list of different values called **fields** (columns) associated with each record
- Designing databases well is important

Hybrid systems

- Virtually all systems will be a combination of bought systems and built systems
- It's impossible to write a meaningful program without using library code
  - Java has an excellent standard library, with additional open-source libraries for almost anything you might want to do
  - Some libraries need to be bought
- Application frameworks are more than just libraries
  - They provide a way to structure applications around sets of functionality that many applications in a particular domain might need
  - Web application framework examples: Ruby on Rails, Angular JS, Django

Version control

- We already know the value of a **version control system (VCS)**
- Some details:
  - A VCS stores **items** (usually files)
  - A **version** is the set of items after one or more modifications
  - A **revision** is a version stored in a VCS
  - A **baseline** is the first revision
  - Storage for revisions is called a **repository**
  - Storing a version in the repository is called **checking in** or **committing**
  - Retrieving a version from the repository is called **checking out** or **updating**
  - A checked-out version of an item is a **working copy**

VCS choices

- How do we deal with two or more different people working on the same file and trying to commit them to the same repository?
  - **File locking:** When a files are checked out for modification, they are locked, meaning that no one else can check them out for modification
  - **Concurrent modification and merge:** If someone tries to commit a file based on an older version of the file, the commit fails, forcing the person to merge the newer repository file with the file they're working on
- Before you start modifying a file, it's wise to pull down the latest changes first
- A centralized VCS has one central repository
- A distributed VCS has many repositories that are peers

Build automation

- **Build automation** is recompiling, relinking, and retesting systems automatically
- This is not tremendously important for programs of the size you work on in school
- Large programs, however, can take hours or days to build
- Tools that can automatically build them and test them are critical
  - Many systems prevent your code from being pushed into the main repository unless it passes all automated tests
- **DevOps** is a modern buzzword for systems and practices that automate the building and testing of software

Quality Assurance in Construction

Static analysis and dynamic analysis

- **Static analysis** is looking at code without running it
  - Code reviews
  - Syntax checking
  - Style checking
  - Usage checking
  - Model checking
  - Data flow analysis
  - Symbolic evaluation
- **Dynamic analysis** is running code to test it
  - Unit testing
  - Debugging
  - Performance optimization and tuning
- Both static and dynamic analysis are valuable and have different strengths
  - Static analysis doesn't require a fully working program
  - Dynamic analysis can give real data about things like performance

Code reviews

- Desk checking is one form of code review
  - Looking over the code
  - Executing it by hand (actually computing values)
- Formal inspections (discussed earlier) are another
- Formal review guidelines
  - Don't read more than 200 lines of code per hour when preparing alone
  - Don't cover more than 150 lines of code when doing a team inspection
  - Use a checklist
- Examples from a Java inspection checklist
  - All variables and constants are named in accord with naming conventions
  - There are no variables or attributes with confusingly similar names
  - Every variable and attribute has the correct data type
  - Every method returns the correct value at every return point
  - All methods and attributes have appropriate access modifiers (**`private`**, **`protected`**, or **`public`**)
  - No nested **`if`** statements should be converted into a **`switch`** statement
  - All exceptions are handled appropriately

Syntax and style checking

- Syntax checking is now mostly done by editors and IDEs
- Be careful about the errors and warnings IDEs and compilers give
  - As computers, they can only guess about why the syntax is wrong
- Language-specific style guides are required on most projects
- Automated style checkers also exist
  - In addition to formatting, they can check semantic issues like variables that are declared and not used
  - Some features like this are included in modern compilers as warnings

Usage checking and idiom checking

- For broader semantic issues, usage and idiom checkers (which can be combined with a style checker) look for:
  - Suspicious or error-prone constructs
  - Non-portable constructs
  - Memory allocation inconsistencies
  - Language-specific issues
    - Loops that never execute
    - Loops that never terminate
    - Using types together that are legal but unusual

Formal methods

- **Formal methods** use mathematical models to do static analysis
- **Model checking** uses analysis to determine if a program meets requirements, usually if certain preconditions are met, it's guaranteed that certain postconditions will be met
- **Data flow analysis** represents a program as a graph and uses that knowledge to calculate the possible values at various points in the graph
  - Modern languages like Java use data flow analysis to complain, for example, that a variable might not have been initialized
- **Symbolic evaluation** traces through the execution of a program with symbolic values instead of concrete values

Unit Testing

Unit testing

- Testing is an important form of dynamic analysis
- **Unit testing** is testing individual units or sub-programs (classes or methods in Java) in isolation
- A **test case** has one value for every input and an expected value for every output
- A **false negative** happens when there's a problem with your code but you don't write a test that catches it
  - This almost always happens, since it's very hard to test everything
- A **false positive** happens when your code is fine but your test is bad
  - For example, you did the math wrong when coming up with your expected answer

Developing test cases

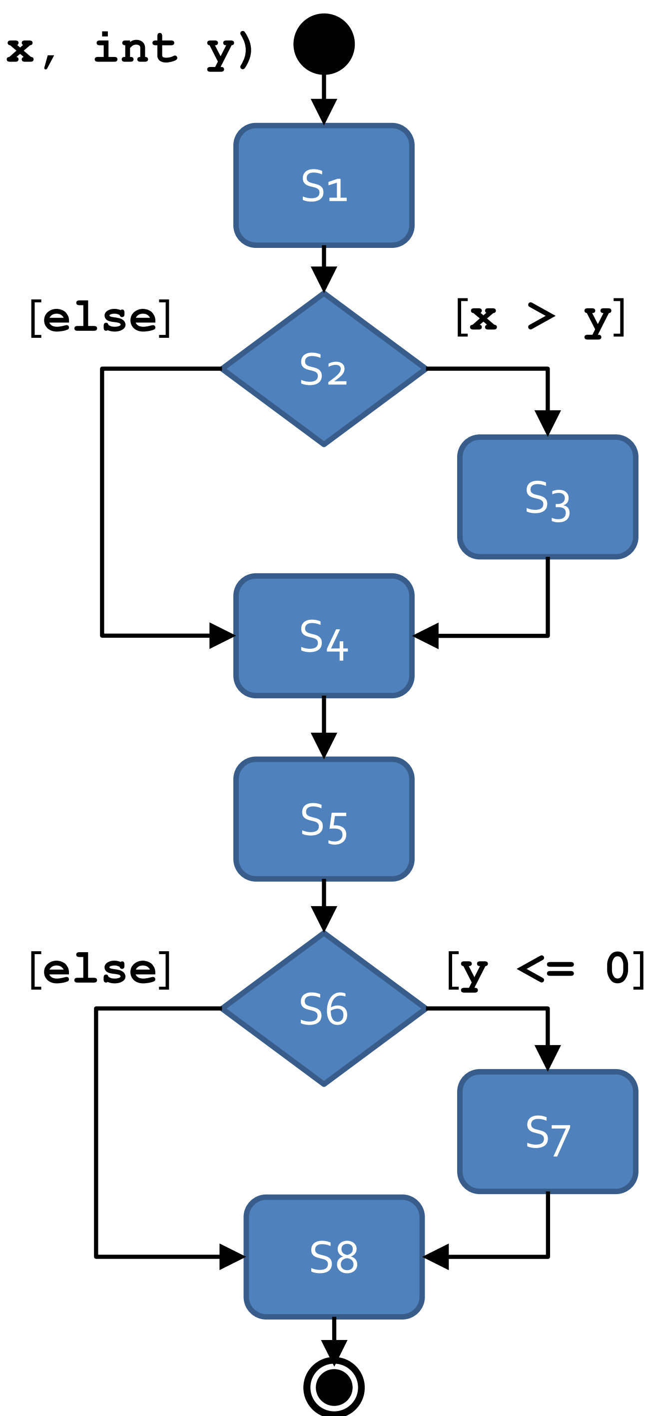
- Picking good test cases is an art form
- **Black box testing** is a strategy that assumes no knowledge of what happens inside the system
  - Only what the input and matching output should be are known
  - Black box testing is easily done by someone who had nothing to do with developing the code
  - Black box testing isn't affected by assumptions about how an algorithm should work
- **Clear box** (or white box or open box) **testing** uses knowledge of the system to generate good tests
- Both kinds of testing are needed to be thorough

Code coverage

- Clear box testing is built around the idea of **coverage**, which is how much of the unit is tested
- Coverage can be explored with a **control-flow graph (CFG)** that shows the possible paths execution could take in a program
  - An **action node** in a CFG is straight-line code with one entry point and one exit point
  - A **decision node** in a CFG is code like an `if` statement or a loop with multiple exit points
  - Arrows show the flow of execution through nodes

Example CFG

```
int calculate(int x, int y)
{
     int a, b;
     a = 1;            // S1
     if (x > y)        // S2
     {
          a = 2;       // S3
     }
     x++;              // S4
     b = y * a;        // S5
     if (y <= 0)       // S6
     {
          b++;         // S7
     }
     return b;         // S8
}
```



Kinds of coverage

- We say a statement is **exercised** by a test or a suite of tests if it gets executed
- **Statement coverage** is the percentage of statements exercised by a set of tests
  - Example: (`x = 1`, `y = 2`) exercises everything except S3 and S7 in the previous CFG, giving a statement coverage of 75%
- **Branch coverage** is the percentage of branch directions taken by a set of tests
  - Example: (`x = 1`, `y = 2`) covers the else edge from S2 and the else edge from S6, giving a branch coverage of 50%
- **Path coverage** is the percentage of all execution paths that have been taken
  - Example: (`x = 1`, `y = 2`) takes only one of the four paths from S1 to S8, giving a path coverage of 25%
- More coverage is better
- It will usually take many tests to get good coverage

Complete enumeration

- Even with relatively high coverage, it's hard to be sure that everything is tested
- Complete enumeration is a test suite that contains all possible inputs
  - For `int` values, 2^{32} values for each one
- There are two reasons that complete enumeration is impractical
  - You would need to know the correct output for *all* of those inputs
  - Just a few inputs explodes the size of the tests to absurd levels: an input array with 10 `int` values would have $(2^{32})^{10} \approx 2 \times 10^{96}$ possible values, more than a quadrillion times the number of electrons in the Universe
- One approximation is to create many randomly generated input values (and figure out the right answer for each corresponding test case)
- Another approach is to think about which values will be treated the same as others, dividing the inputs into **equivalence classes**

Boundary value analysis

- **Boundary value analysis** uses values near the edges of legal limits
  - If input must be within a range, create tests just below, at, and just above the endpoints of the range
  - If output must be in a certain range, try to pick inputs that generate values around the minimum and maximum of that range
- Example: Boundary values for a method that's supposed to accept passwords if they're between 6 and 12 characters inclusive

Input	Length	Case	Valid
"goats"	5	Minimum – 1	False
"wombat"	6	Minimum	True
"wombats"	7	Minimum + 1	True
"abracadabra"	11	Maximum – 1	True
"hippopotamus"	12	Maximum	True
"administrator"	13	Maximum + 1	False

Other heuristics

- A number of other heuristics are commonly used because they often find errors
- For single input parameters
  - 0 (because people forget about 0 or because of division by 0)
  - Very large and very small numbers (because of underflow and overflow)
  - Character or string versions of numbers (which makes sense in a language like Python or JavaScript but not in Java where type checkers would prevent such things)
- For multiple input parameters
  - Equal values for the parameters
  - Different relative values (x larger than y, then x smaller than y)
- For arrays and collections
  - Very small and very large arrays and collections
  - Arrays or collections of length 0 and 1
  - Arrays or collections that are unsorted, ascending, and descending
  - Arrays or collections with duplicated values and with no duplicated values

Regression testing

- Something's wrong with your program, so you change your code, what happens?

	No New Fault Introduced	New Fault Introduced
Fault Corrected	Good	Bad
Fault Not Corrected	Bad	Very Bad

- Data suggests that
  - 30% of software changes result in one of the three bad outcomes
  - On average, bad outcomes occur about 10% of the time
  - Faults introduced during bug fixes are harder to find and remove than others
- One safeguard is **regression testing**, running *all* tests after any software change
  - Any time you find a bug, add the test you used to find the bug into your test suite

Unit testing tools

- Nowadays, running large test suites can be automated
- Tools such as JUnit and other testing tools allow us to:
  - Write clearly marked tests with special set-up and clean-up code if needed
  - Run the tests, sometimes with randomized values or in randomized orders
  - Record which tests pass and fail
  - Show coverage information to see which lines of code the tests covered

Quiz

Upcoming

Next time…

- JUnit, debugging, optimization, refactoring, and TDD next Monday

Reminders

- Keep reading Chapter 8: Quality Assurance in Construction for next Monday
- Work on the final version of Project 2
  - **Due Monday!**