The process of re-testing software that has been modified
Most software today has relatively little new development
- Correcting, perfecting, adapting, or preventing problems with existing software
- Composing new programs from existing components
- Applying existing software to new situations
- Because of the deep interconnections among software components, changes in one method can cause problems in methods that seem to be unrelated
- Regression testing is intended to reduce the chance that existing properties are harmed by a change
- Large regression test suites may accumulate as programs age

Too many tests to be run by hand
- Tests must be run and evaluated quickly
- Often overnight, or more frequently for web applications
- Testers do not have time to view the results by inspection
- Types of tools include
  - Capture / Replay – Capture values entered into a GUI and replay those values on new versions
  - Version control – Keeps track of collections of tests, expected results, where the tests came from, the criterion used, and their past effectiveness
  - Scripting software – Manages the process of obtaining test inputs, executing the software, obtaining the outputs, comparing the results, and generating test reports
- Tools are plentiful and inexpensive (often free)

Test suites accumulate new tests over time
- Test suites are usually run in a fixed, short, period of time – often overnight, sometimes more frequently, sometimes less
- At some point, the number of tests can become unmanageable
  - We cannot finish running the tests in the time allotted
  - We can always add more computer hardware
- But is it worth it? Does it solve the problem? How many of these tests really need to be run?

Which tests to keep can be based on several policies
- Add a new test for every problem report
- Ensure that a coverage criterion is always satisfied
- Sometimes harder to choose tests to remove
  - Remove tests that do not contribute to satisfying coverage
  - Remove tests that have never found a fault (risky!)
  - Remove tests that have found the same fault as other tests (also risky!)
- Reordering strategies
  - If a suite of N tests satisfies a coverage criterion, the tests can often be reordered so that the first N-k tests satisfies the criterion – so the remaining tests can be removed
  - This is often called test selection
  - If the criterion is approximated, not guaranteed, this is often called test prioritization
Aside

- When we talked about white box testing, we talked about coverage criteria—statement, edge, path, etc.
- Criterion coverage in regression testing is somewhat different—among other things, we have two programs and a test suite, rather than one program and a test suite
- We'll see examples of how these coverage criteria differ

When a Regression Test Fails

- Regression tests are evaluated based on whether the result on the new program $P'$ is equivalent to the result on the previous version $P$
  - If they differ, the test is considered to have failed—this is called a regression
- Regression test failures represent three possibilities:
  - The software has a fault—must fix the fix
  - The test values are no longer valid on the new version—must delete or modify the test
  - The expected output is no longer valid—must update the test
- But which?

Choosing Which Regression Tests to Run

- Change impact analysis: how does a change impact the rest of the software?
- When a small change is made in the software, what portions of the software can be impacted by that change?
- More directly, which tests need to be re-run?
- Conservative approach: run all tests
- Cheap approach: run only tests whose test requirements relate to the statements that were changed
- Analytic approach: consider how the changes propagate through the software
- Clearly, tests that never reach the modified statements do not need to be run—is this true?
- Lots of clever algorithms to perform change impact analysis have been invented

Rationales for Selecting Tests to Re-Run

- Inclusive: A selection technique is inclusive if it includes tests that are “modification revealing”
- Unsafe techniques have less than 100% inclusiveness
- Precise: A selection technique is precise if it omits regression tests that are not modification revealing
- Efficient: A selection technique is efficient if deciding what tests to omit is cheaper than running the omitted tests
- This can depend on how much automation is available

Overview of a test selection method

Step 1: Given $P$ and test set $T$, find the execution trace of $P$ for each test in $T$.

Step 2: Extract test vectors from the execution traces for each node in the CFG of $P$

Step 3: Construct syntax trees for each node in the CFGs of $P$ and $P'$. This step can be executed while constructing the CFGs of $P$ and $P'$.

Step 4: Traverse the CFGs and determine the a subset of $T$ appropriate for regression testing of $P'$

Execution Trace [1]

Let $G=(N, E)$ denote the CFG of program $P$. $N$ is a finite set of nodes and $E$ a finite set of edges connecting the nodes. Suppose that nodes in $N$ are numbered $1, 2$, and so on and that Start and End are two special nodes as discussed in Chapter 1.

Let $T_{no}$ be the set of all valid tests for $P'$. Thus $T_{no}$ contains only tests valid for $P'$. It is obtained by discarding all tests that have become obsolete for some reason.
An execution trace of program P for some test t in Tno is the sequence of nodes in G traversed when P is executed against t. As an example, consider the following program.

```c
1 void main() { 1 int g1(int a, b) { 1 int g2(int a, b) {
2 int x, y, p; 2 int a, b;
3 input(x, y); 3 if (a == 1 - b) 3 if (a == b + 1);
4 if (x > y) 4 return(a + a); 4 return(b + b);
5 p = g1(x, y); 5 else 5 else;
6 else 6 return(b + b); 6 return(c + a);
7 p = g2(x, y); 7 } 7 ;
8 return(p); 8 end;
9 output(p); 9 end; 11 }
```

Here is a CFG for our example program.

Now consider the following set of three tests and the corresponding trace.

<table>
<thead>
<tr>
<th>Test (i)</th>
<th>Execution trace (trace(i))</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>main, Start, main, 1, main, 2, g1, Start, g1, 1, g2, End, main, 2, main, 4, main, End</td>
</tr>
<tr>
<td>t2</td>
<td>main, Start, main, 1, main, 3, g2, Start, g2, 1, g2, 2, g2, End, main, 3, main, 4, main, End</td>
</tr>
<tr>
<td>t3</td>
<td>main, Start, main, 1, main, 2, g1, Start, g1, 1, g1, 2, g1, 2, g1, End, main, 2, main, 4, main, End</td>
</tr>
</tbody>
</table>

A test vector for node n, denoted by test(n), is the set of tests that traverse node n in the CFG. For program P we obtain the following test vectors.

<table>
<thead>
<tr>
<th>Function</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>g1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>g2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

A syntax tree is constructed for each node of CFG(P) and CFG(P'). Recall that each node represents a basic block. Here sample syntax trees for the example program.

Given the execution traces and the CFGs for P and P', the following three steps are executed to obtain a subset T' of T for regression testing of P'.

1. Set T' = ∅. Unmark all nodes in G and in its child CFGs.
2. Call procedure selecttrace(G, Start, G', Start'), where G.Start and G'.Start are, respectively, the start nodes in G and G'.
3. T' is the desired test set for regression testing of P'.
Test selection [2]

The basic idea underlying the SelectTests procedure is to traverse the two CFGs from their respective START nodes using a recursive descent procedure.

The descent proceeds in parallel and the corresponding nodes are compared. If two two nodes \( N \) in \( \text{CFG}(P) \) and \( N' \) in \( \text{CFG}(P') \) are found to be syntactically different, all tests in test \( (N) \) are added to \( T' \).

Test selection example

Suppose that function \( g_1 \) in \( P \) is modified as follows.

```
1    int g1(int a, b){ // Modified g1.
2    int e, b;
3    if(a-<b) // Predicate modified,
4    if(e>a),
5    else
6    return(b=b).
7    }
```

Try the SelectTests algorithm and check if you get \( T' = \{t_1, t_3\} \).

Issues with SelectTests

Think:

What tests will be selected when only, say, one declaration is modified?

Can you think of a way to select only tests that correspond to variables in the modified declaration?

More clever algorithms...

- Beyond the scope of 331
- Prioritization – an example "greedy" approach
  - "Diff" \( P \) and \( P' \) to identify the basic blocks (sequences of statements always executed together)
  - Identify the statements involved in new or modified paths
  - Use the test vectors to find the test that covers the largest number of these statements
  - Repeat until no more of the statements can be covered by remaining tests

Regression testing is real

- But don’t forget that it specifically does not include tests of new aspects of a program – it is not common for test suites to get out of date in this regard