Regression Testing

Software Testing
What is regression testing?

Regression:
"when you fix one bug, you introduce several newer bugs."
Regression Testing

• Introduction
• Test Selection
• Test Minimization
• Test Prioritization
• Summary
What is regression testing?

Informally:

- Regression testing is the execution of a set of test cases on a program in order to ensure that its revision does not produce unintended faults, does not "regress" - that is, become less effective than it has been in the past.

- In other words, Is the process of re-testing software that has been modified.
What is Regression Testing?

• A testing activity to make sure that:
  – Not only the newly added or modified code behaves correctly,
  – But also code carried over unchanged from the previous version continues to behave correctly.
## Develop-Test-Release Cycle

<table>
<thead>
<tr>
<th>Version 1</th>
<th>Version 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Develop P</td>
<td>1. Modify P to P’</td>
</tr>
<tr>
<td>2. Test P</td>
<td>2. Test P’ for new functionality</td>
</tr>
<tr>
<td>3. Release P</td>
<td>3. Perform regression testing on P’ to ensure that the code carried over from P behaves correctly</td>
</tr>
<tr>
<td></td>
<td>4. Release P’</td>
</tr>
</tbody>
</table>

*Version 1 & 2 refer to different versions of a program.*
Regression-Test Process

1. Test revalidation/selection/minimization/prioritization
2. Test setup
3. Test execution
4. Output comparison
5. Fault Mitigation
Idea 1 (A Simple Approach)

• Can we simply re-execute all the tests that are developed for the previous version?

• Potential problems:
  – May not have time to execute all the tests.
  – Some tests may be no longer valid, e.g., if you change the number of input parameters or input data format. This may not be efficient, as some tests may have nothing to do with the changes.
Idea 2

• Select a subset $T_r$ of the original test set $T$ such that successful execution of the modified code $P'$ against $T_r$ implies that all the functionality carried over from the original code $P$ to $P'$ is intact.

• Finding $T_r$ is accomplished by test revalidation, test selection, minimization, and test prioritization.
Major Tasks

• **Test revalidation** refers to the task of checking which tests for $P$ remain valid for $P'$.
• **Test selection** refers to the identification of tests that traverse the modified portions in $P'$.
• **Test minimization** refers to the removal of tests that are seemingly redundant with respect to some criteria.
• **Test prioritization** refers to the task of prioritizing tests based on certain criteria.
• Consider a web service ZipCode that provides two services:
  – ZtoC: returns a list of cities and the state for a given zip code
  – ZtoA: returns the area code for a given zip code

• Assume that ZipCode only serves the US initially, and then is modified as follows:
  – ZtoC is modified so that a user must provide a given country as well as a zip code.
  – ZtoT, a new service, is added that inputs a country and a zip code and return the time-zone.
Example (2)

• Consider the following two tests used for the original version:
  – t1: <service = ZtoC, zip = 47906>
  – t2: <service = ZtoA, zip = 47906>

• Can the above two tests be applied to the new version?
The RTS Problem (1)

Regressing testing as a test selection problem.
A subset $T_r$ of set $T$ is selected for retesting the functionality of $P$ that remains unchanged in $P'$
The RTS Problem (2)

• The RTS problem is to find a minimal subset $\mathcal{T}_r$ of non-obsolete tests from $\mathcal{T}$ such that if $P'$ passes tests in $\mathcal{T}_r$ then it will also pass tests in $\mathcal{T}_u$.

• Formally, $\mathcal{T}_r$ shall satisfy the following property: $\forall t \in \mathcal{T}_r$ and $\forall t' \in \mathcal{T}_u \cup \mathcal{T}_r$, $P(t) = P'(t) \Rightarrow P(t') = P'(t')$.

• Make sure testing $\mathcal{T}_r$ is sufficient to ensure existing features continue to work.
• Testing $\mathcal{T}_r$ against $P'$ is equivalent to testing $\mathcal{T}$ against $P$.
• Basically we do not have to test $\mathcal{T}_u$ which is redundant.
Regression Testing

• Introduction
• Test Selection
• Test Minimization
• Test Prioritization
• Summary
Test Selection using Execution Trace

• Step 1: Given $P$ and test set $T$, record 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 a subset of $T$ appropriate for regression testing of $P'$.
• 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.

  – 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.
Main Idea

- The goal is to identify test cases that traverse the modified portions.
- Phase 1: \( P \) is executed and the trace is recorded for each test case in \( T_{no} = T_u \cup T_r \).
- Phase 2: \( T_r \) is isolated from \( T_{no} \) by a comparison of \( P \) and \( P' \) and an analysis of the execution traces
  - Step 2.1: Construct CFG and syntax trees
  - Step 2.2: Compare CFGs and select tests
Obtain Execution Traces

1. `main () {
2.   int x, y, p;
3.   input (x, y);
4.   if (x < y)
5.      p = g1(x, y);
6.   else
7.      p = g2(x, y);
8.   endif
9.   output (p);
10. end
11. }

1. `int g1 (int a, b) {
2.   int a, b;
3.   if (a + 1 == b)
4.      return (a*a);
5.   else
6.      return (b*b);
1. }`

1. `int g2 (int a, b) {
2.   int a, b;
3.   if (a == (b + 1))
4.      return (b*b);
5.   else
6.      return (a*a);
1. }

Consider the following test set:
- `t1: <x=1, y=3>`
- `t2: <x=2, y=1>`
- `t3: <x=1, y=2>`
Recall that each node represents a basic block.
## Execution Trace

<table>
<thead>
<tr>
<th>Test (t)</th>
<th>Execution Trace (trace(t))</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>main.Start, main.1, main.2, g1.Start, g1.1, g1.3, g1.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.End, main.2, main.4, main.End</td>
</tr>
</tbody>
</table>
A test vector for node $n$, denoted by $\text{test}(n)$, is the set of tests that traverse Node $n$ in the CFG. For the previous program, we obtain the following test vectors:

<table>
<thead>
<tr>
<th>Function</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>$t_1$, $t_2$, $t_3$</td>
<td>$t_1$, $t_3$</td>
<td>$t_2$</td>
<td>$t_1$, $t_2$, $t_3$</td>
</tr>
<tr>
<td>g1</td>
<td>$t_1$, $t_3$</td>
<td>$t_3$</td>
<td>$t_1$</td>
<td>-</td>
</tr>
<tr>
<td>g2</td>
<td>$t_2$</td>
<td>$t_2$</td>
<td>None</td>
<td>-</td>
</tr>
</tbody>
</table>

*Columns are node numbers in each function.*
A semicolon indicates left to right sequencing of two or more statements within a node.

In a syntax tree, a function call is represented by parameter nodes, one for each parameter, and a call node.
Selection Strategy

• The CFGs for $P$ and $P'$ are compared to identify nodes that differ in $P$ and $P'$.
  – Two nodes are considered equivalent if the corresponding syntax trees are identical.
  – Two syntax trees are considered identical when their roots have the same labels and the same corresponding descendants.

• Tests that traverse those nodes are selected.
Procedure SelectTestsMain

Input: (1) G and G’, including syntax trees; (2) Test vector test(n) for each node n in G and G’; and (3) Set T of non-obsolete tests

Output: A subset T’ of T

Procedure SelectTestsMain
- Step 1: Set T’ = ∅. Unmark all nodes in G and in its child CFGs
- Step 2: Call procedure SelectTests (G.Start, G’.Start’)
- Step 3: Return T’ as the desired test set

Procedure SelectTests (N, N’)
- Step 1: Mark node N
- Step 2: If N and N’ are not equivalent, T’ = T’ ∪ test(N) and return, otherwise go to the next step.
- Step 3: Let S be the set of successor nodes of N
- Step 4: Repeat the next step for each n ∈ S.
  - 4.1 If n is marked then return else repeat the following steps:
    - 4.1.1 Let l = label(N, n). The value of l could be t, f or ε
    - 4.1.2 n’ = getNode(l, N’).
    - 4.1.3 SelectTests(n, n’)

Step 5: Return from SelectTests
Example

• Consider the previous example. Suppose that function g1 is modified as follows:

```c
1. int g1 (int a, b) {
2.   int a, b;
3.   if (a - 1 == b) \(\Leftarrow\) Predicate modified
4.     return (a*a);
5.   else
6.     return (b*b);
}
```

As G.main.2 contains a call to g1, the equivalence needs to be checked with respect to the CFGs of g1 and g2. N and N' are not equivalent due to the modification in g1. Hence, \(T' = \text{tests}(N) = \text{tests}(G\text{.main.2}) = \{t1, t3\}\)
Regression Testing

• Introduction
• Test Selection
• Test Minimization
• Test Prioritization
• Summary
Motivation

• The adequacy of a test set is usually measured by the coverage of some testable entities, such as basic blocks, branches, and du-paths.

• Given a test set $T$, is it possible to reduce $T$ to $T'$ such that $T' \subseteq T$ and $T'$ still covers all the testable entities that are covered by $T$?
Example (1)

```c
1. main () {
2.   int x, y, z;
3.   input (x, y);
4.   z = f1(x);
5.   if (z > 0)
6.      z = f2(x);
7.   output (z);
8.   end
9. }

1. int f1(int x) {
2.   int p;
3.   if (x > 0)
4.      p = f3(x, y);
5.   return (p);
6. }
```
Consider the following test set:

- **t1**: main: 1, 2, 3; f1: 1, 3
- **t2**: main: 1, 3; f1: 1, 3
- **t3**: main: 1, 3; f1: 1, 2, 3

Recall that each node represents a basic block.
The Set-Cover Problem

- Let $E$ be a set of entities and $TE$ a set of subsets of $E$.

- A set cover is a collection of sets $C \subseteq TE$ such that the union of all entities of $C$ is $E$. The set-cover problem is to find a minimal $C$. 
Example

- Consider the previous example:
  \[ E = \{\text{main.1}, \text{main.2}, \text{main.3}, f1.1, f1.2, f1.3\} \]

  \[ \text{TE} = \{\{\text{main.1}, \text{main.2}, \text{main.3}, f1.1, f1.3\}, \{\text{main.1}, \text{main.3}, f1.1, f1.3\}, \{\text{main.1}, \text{main.3}, f1.1, f1.2, f1.3\}\} \]

- The solution to the set cover problem is:
  \[ C = \{\{\text{main.1}, \text{main.2}, \text{main.3}, f1.1, f1.3\}, \{\text{main.1}, \text{main.3}, f1.1, f1.2, f1.3\}\} \]
A Greedy Algorithm

• Find a test $t$ in $T$ that covers the maximum number of entities in $E$.
• Add $t$ to the return set, and remove it from $T$ and the entities it covers from $E$.
• Repeat the same procedure until all entities in $E$ have been covered.
Procedure CMIMX

**Input:** An \( n \times m \) matrix \( C \), where each column corresponds to an entity to be covered, and each row to a distinct test. \( C(i,j) \) is 1 if test \( t_i \) covers entity \( j \).

**Output:** Minimal cover \( \text{minCov} = \{i_1, i_2, \ldots, i_k\} \) such that for each column in \( C \), there is at least one nonzero entry in at least one row with index in \( \text{minCov} \).

**Step 1:** Set \( \text{minCov} = \emptyset \), \( \text{yetToCover} = m \).

**Step 2:** Unmark each of the \( n \) tests and \( m \) entities.

**Step 3:** Repeat the following steps while \( \text{yetToCover} > 0 \)

3.1. Among the unmarked entities (columns) in \( C \) find those containing the least number of 1s. Let \( LC \) be the set of indices of all such columns.

3.2. Among all the unmarked tests (rows) in \( C \) that also cover entities in \( LC \), find those that have the max number of nonzero entries that correspond to unmarked columns. Let \( s \) be any one of those rows.

3.3. Mark test \( s \) and add it to \( \text{minCov} \). Mark all entities covered by test \( s \). Reduce \( \text{yetToCover} \) by the number of entities covered by \( s \).
Consider the previous example:

Suppose program P has been executed against the five test in test suit T. A total of six entities are covered by the tests as shown in the following table; 0(1) in a column indicates that the corresponding entity is not covered (covered). The entities could be basic blocks in the program, functions, def-uses, or any other testable element of interest. Further, a testable entity in P not covered by any of the five test is not included in the table.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>t2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>t3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>t4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>t5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Step 1: $\text{minCov}= \phi$, $\text{yetToCover}= 6$

Step 2: All five tests and six entities are unmarked

Step 3: as $\text{yetToCover} > 0$, we execute the loop

///Loop 1

Step 3.1: Among the unmarked entities, 4 and 6 each contain a single 1 and hence qualify as the highest priority entities. Thus $\text{LC} = \{4, 6\}$

Step 3.2: among the unmarked tests, $t_2$ covers entities 1 and 4, and $t_4$ covers entities 3 and 6. Both tests have identical benefits of 2 each in terms of the number of entities they cover. We randomly select $t_2$. Thus $s=2$.

Step 3.3: $\text{minCov} = \{2\}$. Test $t_2$ is marked. Entities 1 and 4 covered by test $t_2$ are also marked. $\text{yetToCover} = 6-2 = 4$

///End Loop

Cont. next slide ->
Example (3)

///Loop 2
Step 3.1: We continue with the second iteration of the loop as yetToCover > 0. Among the remaining unmarked entities, t6 is the one with the least cost (=1), Hence, LC = {6}
Step 3.2: only t4 covers entity 6 and hence s=4
Step 3.3: minCov = {2,4}. Test t4 and entities 3 and 6 are marked. yetToCover = 4-2 =2
/// End Loop

///Loop 3
Step 3.1: we continue with the third iteration of the loop as yetToCover > 0. The remaining entities, 2 and 5, have identical costs.
Hence LC = {2,5}
Step 3.2: Entities 2 and 5 are covered unmarked by tests t1 and t3, and t5 Of theses tests, t3 has maximum benefit of 2. Hence s=3.
Step 3.3: minCov = {2,4,3}. Test t3 and entities 2 and 5 are marked. yetToCover = 2-2 =0
/// End Loop
Finally, we will have: t2, t4, t3
Regression Testing

- Introduction
- Test Selection
- Test Minimization
- Test Prioritization
- Summary
Motivation

• In practice, **sufficient resources** may not be available to execute all the tests.

• One way to solve this problem is to **prioritize** tests and only execute those high-priority tests that are allowed by the budget.

• Typically, **test prioritization** is applied to a reduced test set that are obtained, e.g., by the **test selection and/or minimization** process.
Residual coverage refers to the number of elements that remain to be covered w.r.t. a given coverage criterion.

One way to prioritize tests is to give higher priority to tests that lead to a smaller residual coverage.
Procedure PrTest

Input: (1) $T'$: a regression test set to be prioritized;
(2) $\text{entitiesCov}$: set of entities covered by tests in $T'$;
(3) $\text{cov}$: Coverage vector such that for each test $t \in T'$, $\text{cov}(t)$ is the set of entities covered by $t$.

Output: $\text{PrT}$: A prioritized sequence of tests in $T'$

Step 1: $X' = T'$. Find $t \in X'$ such that $|\text{cov}(t)| \geq |\text{cov}(u)|$ for all $u \in X'$.

Step 2: $\text{PrT} = <t>, X' = x' \setminus \{t\}, \text{entitiesCov} = \text{entitiesCov} \setminus \text{cov}(t)$

Step 3: Repeat the following steps while $X' \neq \phi$ and $\text{entitiesCov} \neq \phi$.

3.1. $\text{resCov}(t) = |\text{entitiesCov} \setminus (\text{cov}(t) \cap \text{entitiesCov})|$

3.2. Find test $t \in X'$ such that $\text{resCov}(t) \leq \text{resCov}(u)$ for all $u \in X', u \neq t$.

3.3. Append $t$ to $\text{PrT}$, $X' = X' \setminus \{t\}$, and $\text{entitiesCov} = \text{entitiesCov} \setminus \text{cov}(t)$

Step 4: Append to $\text{PrT}$ any remaining tests in $X'$ in an arbitrary order.
Example

- Consider application P that consists of four classes C1, C2, C3, and C4. Each of these classes has one or more methods as follows: C1 = \{m_1, m_{12}, m_{16}\}, C2 = \{m_2, m_3, m_4\}, C3 = \{m_5, m_6, m_{10}, m_{11}\}, and C4 = \{m_7, m_8, m_9, m_{13}, m_{14}, m_{15}\}. The methods covered by each test in T' are listed in the following table.

| Test(t) | Methods covered (cov(t)) | |cov(t)| |
|---------|--------------------------|---|---|
| t1      | 1,2,3,4,5,10,11,12,13,14,16 | 11 |
| t2      | 1,2,4,5,12,13,15,16       | 8  |
| t3      | 1,2,3,4,5,12,13,14,16     | 9  |
| t4      | 1,2,4,5,12,13,14,16       | 8  |
| t5      | 1,2,4,5,6,7,8,10,11,12,13,15,16 | 13 |
Example (2)

1) : $X' = \{t_1,t_2,t_3,t_4,t_5\}$. $t_5$ covers the largest number of functions -13- in $X'$ and hence the least cost.

2) : $PrT = <t_5>$. $X' = \{t_1,t_2,t_3,t_4\}$. $entitiesCov = \{3,14\}$

3) : We continue the process as $X'$ and $entityCov$ are not empty.

3.1) : Compute residual coverage for each test in $X'$

\[
\begin{align*}
resCov(t_1) &= |\{3,14\}\setminus (\{1,2,3,4,5,10,11,12,13,14,16\} \cap \{3,14\})| = |\emptyset| = 0 \\
resCov(t_2) &= |\{3,14\}\setminus (\{1,2,4,5,12,13,15,16\} \cap \{3,14\})| = |\{3,14\}| = 2 \\
resCov(t_3) &= |\{3,14\}\setminus (\{1,2,3,4,5,12,13,14,16\} \cap \{3,14\})| = |\emptyset| = 0 \\
resCov(t_4) &= |\{3,14\}\setminus (\{1,2,4,5,12,13,14,16\} \cap \{3,14\})| = |\{3\}| = 1
\end{align*}
\]

3.2) // $t_1$ and $t_3$ have the least cost; we arbitrarily choose $t_3$.

3.3) $PrT = <t_5,t_3>$. $X' = \{t_1,t_2,t_4\}$, $entitiesCov = \emptyset$

3) There is no function remaining to be covered. Hence we terminate the loop.

4) $t_1$, $t_2$, and $t_4$ remain to be prioritized. As $entityCov$ is empty, the residual coverage criterion can not be applied to differentiate among the priorities of these remaining tests. In this case we break the tie arbitrarily (in a random order)

This leads to $PrT = <t_5,t_3,t_1,t_2,t_4>$
Regression Testing

- Introduction
- Test Selection
- Test Minimization
- Test Prioritization
- Summary
Summary

- Regression testing is about ensuring new changes do not adversely affect existing functionalities.
- Three techniques can be used to reduce the number of regression tests: modification-traversing selection, minimization, and prioritization.
- The prioritization is a practical choice when resources are limited.