CISC327 - Software Quality Assurance

Lecture 19

Regression Testing

Regression Testing

- Today we look at regression testing
- Purpose of regression testing
- Method
  - Establishing a regression test set
  - Maintaining a regression test set
  - Observable artifacts

Regression Testing

- regression:

1. a return to a former or less developed state

Regression Testing

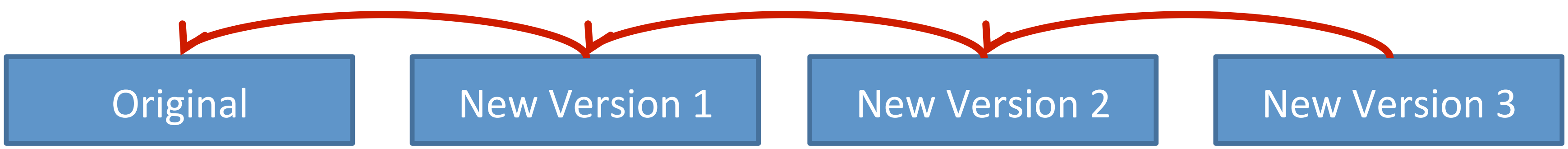
- Purpose
  - Ensure that existing functionality and behaviour is not broken by changes in new versions
  - Ensure that intended changes to functionality and behaviour are actually observed
  - Catch accidental or unintentional changes in functionality and behaviour before deployment, reducing costs

Regression Testing

- Method
  - Maintain a regression set of test inputs designed to exhibit existing functionality and behaviour
  - Choose a set of observable artifacts of computation that demonstrate desired aspects of functionality and behaviour (not just output!)
  - Maintain a history of the observable artifacts for each version of the software
  - Compare observable artifacts of each new version of software to previous version to ensure that differences are intentional

Regression Testing

- Regression Series
  - It's really called regression testing because we incrementally compare the results (functionality and behaviour) of tests for each new version of the software only to the previous version
  - And that one was compared to the one before it, and so on, forming a regression series based on the original software



  - It's a sort of inductive proof that we still have the behaviour we want to maintain

Regression Testing

- It's also called regression testing because to keep the total number of tests to be run at a practical level, we replace old tests with new ones to "cover" the same cases but to include testing of new/changed functionality
- This sequence of replaced tests covering previous tests also forms a (more complex) regression series of test cases based on the original test set
  - Where old tests are retired from the set as new tests are added to "cover" them
- The reasoning that the tests have not lost anything is also an induction:
  - New tests cover retired old tests, which in turn cover previous older tests, and so on, back to the original validated test set

Establishing a Baseline

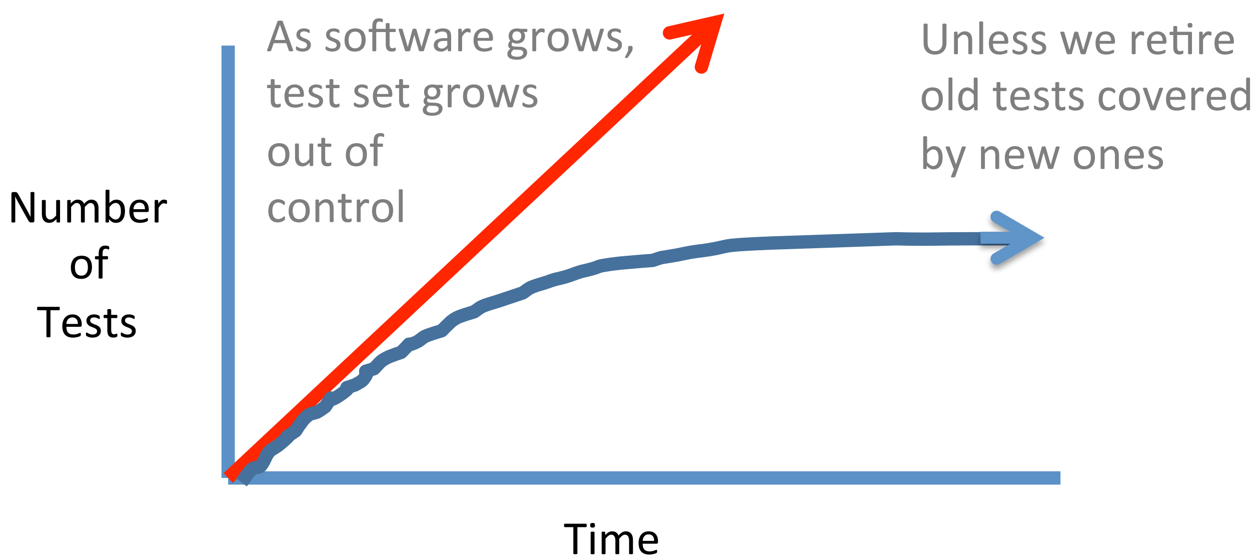
- Begin with the original functionality test suite, plus early failure tests (if any), plus first operational tests
- Validate that these tests all run correctly
- Choose the set of observable artifacts to be tracked
  - These should characterize the functionality and behaviour we want to maintain across versions (more on this later)
- Run these first tests and save the observable artifacts in an easy to compare form (more on this later also)

Adding and Retiring Tests

- Whenever functionality is added or changed in the software, add and validate new tests for the new or changed functionality, and retire the tests for the replaced old functionality
- Some practitioners retire failure tests after a fixed number of new versions do not exhibit the failure, as a way to keep the number of failure tests from growing too large

Adding and Retiring Tests

- Operational tests must also be maintained, and retired or replaced when they no longer reflect current functionality



Choosing Observable Artifacts

- Observable artifacts include at least the direct outputs of the software, but also other indicators of behaviour
- Because many program have multiple kinds, streams, or files of output, we normally include all of them together in the observable artifacts
- Because subtle unintended changes in behaviour may not be immediately visible in direct test output, we normally turn on all debugging, tracing, and instrumenting flags the software may have when running regression tests, in order to have more detail in observable artifacts

Choosing Observable Artifacts

- Because performance is part of the user-visible behaviour of software, we usually measure time and space when running regression tests
  - Add these to the observable artifacts in order to observe unintended changes in performance
- Most systems provide some kind of external performance measuring tools
  - E.g., Unix "time" command, which can be used to give us this information
- To allow easy differencing, we normally translate all observable artifacts to text in the stored test results

Combining Observable Artifacts

- To allow easy differencing and archival, the entire set of observable artifacts resulting from running all of the tests in the entire set of regression tests is often combined into a single text file
- This file includes the direct and indirect output, tracing, and debugging information, time and space statistics, and all other observable artifacts resulting from running each test, all concatenated together in a fixed order into one text file
- This file forms a kind of behavioural signature for the version of the software, storing every observable characteristic of behaviour on the test set in one file

Comparing Signatures

- The actual regression aspect of the test is implemented by looking at the difference between the signature files for the previous version and the new version
- If we're careful, this difference can be implemented by simple tools such as Unix's "diff"

```
diff -b OldSignatureFile NewSignatureFile
314c314
< 0.3u 0.0s 0:00 97% 359+781k 0+0io 0pf+0w
---
< 0.7u 0.0s 0:01 95% 361+770k 0+0io 0pf+0w
2721c2721,2722
< End of run - goodbye!
---
> *** Error: invalid command 'create'
> End of run - goodbye!
```

Normalizing Signatures

- To allow easy differencing, it is important that irrelevant or intentional differences between versions be factored out
- Since the signature file is all text, this can be automated using editor scripts to normalize signature files to reduce or eliminate non-behavioural or intended differences
  - Example: If the previous version of the software did all output in upper case and the new version (intentionally) outputs mixed case instead, the new signature can be normalized to upper case before differencing

Establishing the Baseline

- The baseline is the signature file of the version used to establish regression testing (the "original" version)
- The baseline signature must be carefully examined line by line by hand to ensure that every artifact is as it should be (a lot of work)
- Once established, only differences need to be examined for future versions

Regression Test Harness

- The regression test harness is the implementation of a procedure for automating the running, collection of observable artifacts, and differencing of versions for regression testing a product
- Should be developed such that it adapts automatically to addition or deletion of test cases or individual tests
- Again, requires care in planning and implementation, but once established, requires very little work

Regression Test Signatures

- The output of the entire run of the regression test script, including all test output, diagnostic output, and time and memory resource usage is saved in a single (large) signature file named for the version of the program being tested
- The signature file is diff'ed against the previous version's signature file to check for differences in behaviour, and saved for comparison with the next version

Example: Service Numbers

- I could change the requirements about length of service numbers
- Make "new" tests by adjusting old tests to meet the new requirements
- Existing tests that should pass would become **failure** tests
- **Some** failure tests might become tests that should pass

Regression Testing: TXL

- The TXL interpreter is a software product that implements the ***TXL*** programming language (http://www.txl.ca)
- It takes as input a TXL program "foo.Txl" and an input file to the program "bar.foo", and compiles and runs the program on the input
- It produces two output streams:

1. compiler and run time error messages on the **standard** **error** stream,
2. output of the program on the **standard** **output** stream

The TXL Regression Tests

- The regression tests for the TXL interpreter are organized into one large directory in which subdirectories contain test cases
- Test case directories are named to indicate the kind and source of the test case they cover (functionality tests, failure tests or operational tests)
- Each test case directory contains a number of test inputs, each named beginning with the letters "eg" (standing for "example") to make them easy to find automatically, as well as a README file explaining the original source and intentions of the test case

The TXL Regression Tests

Regression Test Directory

Operational Tests

Functionality Tests

Failure Tests

```
drwxr--r--   4 cordy    penguin      512 Apr 01 17:11 ASDT/
drwxr--r--   3 cordy    penguin      512 Apr 01 17:11 ASDT2/
drwxr--r--   2 cordy    penguin      512 Nov 07 1997  ASTI-issue/
drwxr--r--   2 cordy    penguin      512 Nov 27 1997  ASTI_issue/
drwxr--r--   3 cordy                 512 Apr 01 17:11 Abacus/
drwxr--r--   2 cordy                 512 Dec 19 1996  Analyzer_Bug/
drwxr--r--   2 cordy                 512 Apr 13 1996  AndCondition/
drwxr--r--   2 cordy    penguin      512 Jun 02 1996  Andy/
drwxr--r--   2 cordy    penguin      512 Apr 29 1997  Apr97Bugs/
drwxr--r--   2 cordy    penguin      512 Apr 13 1996  Backtrack/
drwxr--r--   3 cordy    penguin      512 Apr 13 1996  Booster/
drwxr--r--   2 cordy    penguin      512 Jun 24 1996  C2T/
   . . .
```

```
./Abacus:
total 11
-rw-r--r--   1 cordy    penguin      898 Jun 30 1993  README
drwxr--r--   2 cordy    penguin      512 Dec 23 1994  Txl/
-rw-r--r--   1 cordy    penguin      487 Jun 30 1993  eg.Compound
-rw-r--r--   1 cordy    penguin       34 Jun 30 1993  eg1.Cascade
-rw-r--r--   1 cordy    penguin      375 Jun 30 1993  eg2.Cascade
-rw-r--r--   1 cordy    penguin     2102 Oct 16 1997  txltrace.out
```

Running the TXL Regression Tests

TXL Regression Test Harness

- The TXL regression tests are run by a C-shell script that walks through each subdirectory (test case) in the regression test directory, and runs each test input through TXL

```
#!/bin/csh
# NewTestAll - the TXL regression script
foreach i (*)
    if -d $i then
        echo "===== $i ====="
        cd $i
        foreach j (eg*.*)
            time newtxl -v $j
        end
        cd ..
    endif
end
```

Each Test Case Directory

Separator Message for each Test Case in Signature

Each Input in the Test Case Directory

Turn on All Verbose Diagnostic Messages

Run with Unix "time" command to Measure Time and Memory Use

Running the TXL Regression Tests

TXL Regression Test Signatures

- The output of the entire run of the regression test script, including all test output, diagnostic output, and time and memory resource usage, is saved in a single (large) signature file named for the version of TXL being tested
- The signature file is diff'ed against the previous version's signature file to check for differences in behaviour, and saved for comparison with the next version

```
# Run TXL regression tests
NewTestAll >& NTAout2.42
diff NTAout2.41 NTAout2.42
```

Run Putting All Direct and Diagnostic Output in Signature File

Compare to Previous Version

Example TXL Regression Signature

Separator Message for Test Case

```
===== Abacus =====
TXL Pro-LS 2.5d3b (22.7.98) Copyright 1995-1998 Legasys Corp.
Bootstrapping TXL ...
  ... used 348 trees and 229 kids.
Scanning the TXL program Txl/Compound.Txl
Parsing the TXL program
  ... used 1445 trees and 2270 kids.
Making the object language grammar tree
TXL ERROR : (Warning) Declaration of define 'choice'
previous declaration
  ... used 72 trees and 49 kids.
Making the rule table
  ... used 252 trees and 261 kids.
Scanning the input file eg.Compound
Parsing the input file
  ... used 158 trees and 266 kids.
Applying the transformation rules
Forced to copy 16 local vars (27%)
  ... used 93 trees and 158 kids.
Generating transformed output
Used a total of 2368 trees (0%) and 3233 kids (0%).
True = true ! True + setFalse ? False + setTrue ? True
False = false ! False + setTrue ? True + setFalse ? False
Negate = false ? Negate1
Negate1 = true ? Negate2 + setTrue ! nil
Negate2 = setFalse ! nil
And = false ? And1
And1 = true ? And2 + setFalse ! nil
And2 = true ? setTrue ! nil + false ? setFalse ! nil
[True & Negate]
0.0u 0.0s 0:00 109% 150+103k 0+0io 0pf+0w
```

Verbose Messages Showing TXL Internal Diagnostic Information

Direct Output of Test Run

Time and Space Stats from “time” Command

TXL Regression Differencing

```
2c2
< TXL Pro-LS 2.4d8 (9.4.98) Copyright 1995-1998 Legasys Corp.
---
> TXL Pro-LS 2.5d3b (22.7.98) Copyright 1995-1998 Legasys Corp.
314c314
< 0.3u 0.0s 0:00 97% 359+781k 0+0io 0pf+0w
---
> 0.3u 0.0s 0:00 83% 350+773k 0+0io 0pf+0w
316c316
< TXL Pro-LS 2.4d8 (9.4.98) Copyright 1995-1998 Legasys Corp.
---
> TXL Pro-LS 2.5d3b (22.7.98) Copyright 1995-1998 Legasys Corp.
2970,2971c2970,2971
< 1.1u 0.1s 0:01 100% 400+1395k 0+0io 0pf+0w
< TXL Pro-LS 2.4d8 (9.4.98) Copyright 1995-1998 Legasys Corp.
---
> 1.2u 0.1s 0:01 98% 395+1369k 0+0io 1pf+0w
> TXL Pro-LS 2.5d3b (22.7.98) Copyright 1995-1998 Legasys Corp.
7039,7040c7039,7040
< 1.7u 0.1s 0:01 100% 413+1289k 0+0io 0pf+0w
< TXL Pro-LS 2.4d8 (9.4.98) Copyright 1995-1998 Legasys Corp.
---
> 1.7u 0.1s 0:01 100% 410+1275k 0+0io 0pf+0w
> TXL Pro-LS 2.5d3b (22.7.98) Copyright 1995-1998 Legasys Corp.
9787,9788c9787,9788
< 1.8u 0.1s 0:01 100% 413+1427k 0+0io 0pf+0w
< TXL Pro-LS 2.4d8 (9.4.98) Copyright 1995-1998 Legasys Corp.
---
> 1.7u 0.1s 0:01 98% 410+1431k 0+0io 0pf+0w
> TXL Pro-LS 2.5d3b (22.7.98) Copyright 1995-1998 Legasys Corp.
   . . .
```

Version Message Difference

Performance Difference

TXL Regression Differencing

```
32514c32532
< Preprocessor directives    58
---
> Preprocessor directives   58
32516c32534
< Declarations    91
---
> Declarations   91
    . . .

15010c15010
< TXL Pro-LS 2.4d2 (9.12.97) Copyright 1995-1997 Legasys Corp.
---
> TXL Pro-LS 2.5d3b (22.7.98) Copyright 1995-1998 Legasys Corp.
27888c27888
< 8.1u 1.1s 0:09 99% 372+6375k 0+0io 11pf+0w
---
> 7.7u 0.4s 0:08 99% 421+6965k 0+0io 0pf+0w
27891c27891
< TXL Pro-LS 2.4d2 (9.12.97) Copyright 1995-1997 Legasys Corp.
---
> TXL Pro-LS 2.5d3b (22.7.98) Copyright 1995-1998 Legasys Corp.
27942c27942
<    ... used 425 trees and 519 kids.
---
>    ... used 423 trees and 519 kids.
41066c41066
<     Used a total of 490839 trees (16%) and 998275 kids (22%).
---
>     Used a total of 490837 trees (16%) and 998275 kids (22%).
    . . .
```

Output Spacing Difference (Bug!)

Significant Performance Difference (But an Improvement)

Internal Diagnostic Difference

Regression Testing

- Advantages
  - Previous functionality never accidentally lost
  - Previously fixed bugs never reappear in production
  - Virtually all accidental bugs are caught before deployment
  - Virtually no unintentional changes in behaviour slip into production
  - Users observe very high level of quality

Regression Testing

- Disadvantages
  - Regression set must be maintained with a high degree of discipline and care
    - At least as carefully as the software itself
  - Establishing the baseline and regression testing harness requires significant effort
    - But it pays off in ease of use later
- Bottom Line
  - High-quality software shops all do it, because the difference in confidence and observed quality is easily worth it

Summary

- Regression Testing
  - Ensure that existing functionality and behaviour is not broken by changes in new versions
  - Maintain regression set of tests designed to exhibit existing functionality and behaviour
  - Compare observable artifacts of each new version of software to previous version to ensure that differences are intentional
- Then
  - Some material on inspection—enough for A4
  - Security