Princeton University

Computer Science 217: Introduction to Programming Systems

Testing

Fall 2017

Software engineering method 1

1. Write program
2. Upload program to customer

3. . . . program has bugs, useless ...

Software engineering method 2

1. Be vewy, vewy careful

   in writing program

2. Upload program to customer

(This is not really an engineering method, of course)

Software engineering method 3

```
do  {
     write_program;
     OK = test_program ( );
}  while (!OK);

upload program to customer;
```

(This isn't really much of an engineering method, either.)

Goals of this Lecture

Help you learn about:

- External testing
- Unit testing
- Internal testing
- Test coverage

Why?

- It' s hard to know if a (large) program works properly
- Software engineers spend **at least as much time building test code** as writing the program
- You want to spend that time efficiently!

EXTERNAL TESTING

Example: “upper1” program

```
/* Read text from stdin. Convert the first character of each
   "word" to uppercase, where a word is a sequence of
   letters. Write the result to stdout. Return 0. */

int main(void)
{
   . . .
}
```

How do we test this program?
Run it on some sample inputs?

```
$ ./upper1
heLLo there...
^D
HeLLo There...
$
```

Ok if you only had to do it once; tedious to repeat every time you make a change to the program.

Organizing your tests

```
/* Read text from stdin. Convert the first character of each
   "word" to uppercase, where a word is a sequence of
   letters. Write the result to stdout. Return 0. */
```

```
$ cat inputs/001
heLLo there...
$ cat correct/001
HeLLo There...
$ cat inputs/002
84weird e. xample
$ cat correct/002
84Weird E. Xample
```

Running your tests

```
/* Read text from stdin. Convert the first character of each
   "word" to uppercase, where a word is a sequence of
   letters. Write the result to stdout. Return 0. */
```

```
$ cat run-tests
./upper1 <inputs/001 >outputs/001
cmp outputs/001 correct/001
./upper1 <inputs/002 >outputs/002
cmp outputs/002 correct/002
$ sh run-tests
outputs/002 correct/002 differ: byte 5, line 1
```

this is a “shell script” or “bash script”

This *barely* qualifies as “engineering”

Limitations of whole-program testing:

- Works on noninteractive one-right-answer programs
- Requires knowing what the right answer is at the whole-program input-output level
  - If you already knew the right answer, you wouldn’t need a program!
- Can only test “surface” behavior, can’t examine internals
- Can never be sure when you have “enough” tests
- When you change the specification of the program, all your tests are obsolete

Shell scripting

```
/* Read text from stdin. Convert the first character of each
   "word" to uppercase, where a word is a sequence of
   letters. Write the result to stdout. Return 0. */
```

```
$ cat run-tests
for A in inputs/* ; do
 ./upper1 <inputs/$A >outputs/$A
 cmp outputs/$A correct/$A
done
$ sh run-tests
outputs/002 correct/002 differ: byte 5, line 1
```

this is a “shell script” or “bash script”

You can also write these scripts in **python** instead of bash. If you know some **python** already, this is probably a better idea than learning bash.

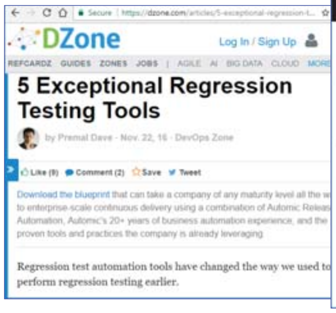
Regression testing

re·gres·sion
rəˈgreSH(ə)n/
noun
1. a return to a former or less developed state.
2. . . .

```
for (;;) {
   test program; discover bug;
   fix bug, in the process break something else;
}
```

re·gres·sion test·ing
Rerun your entire test suite after each change to the program. When new bugs are found, add tests to the test suite that check for those kinds of bugs.

Regression testing tools


https://dzone.com/articles/5-exceptional-regression-t...

DZone

Log In / Sign Up

REFCARDZ GUIDES ZONES JOBS | AGILE AI BIG DATA CLOUD MORE

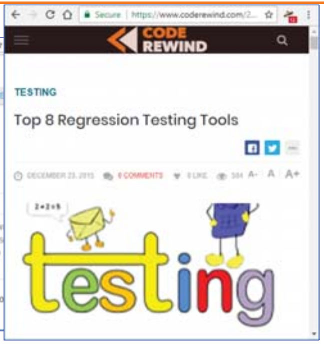
5 Exceptional Regression Testing Tools

by Premal Dave · Nov. 22, 16 · DevOps Zone

Like (9) Comment (2) Save Tweet

Download the blueprint that can take a company of any maturity level all the w to enterprise-scale continuous delivery using a combination of Automic Releas Automation, Automic's 20+ years of business automation experience, and the proven tools and practices the company is already leveraging.

Regression test automation tools have changed the way we used to perform regression testing earlier.


https://www.coderewind.com/2...

CODE REWIND

TESTING

Top 8 Regression Testing Tools

DECEMBER 23, 2015 0 COMMENTS

Q: Are we allowed to use regression testing tools in the homeworks?
A: Yes, but if you do,
- You're on your own, don't ask the preceptors or Lab TAs for help with the tool
- Describe in your README how you used the tool.

UNIT TESTING

Testing modular programs

Any program (that's not a toy) is broken up into *modules,* or *units.*

Example:

Homework 2.

Homework 2

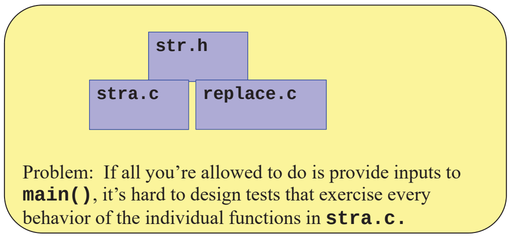
str.h (excerpt)
```
/* Return the length of src */
size_t Str_getLength(const char *src);
/* Copy src to dest. Return dest.*/
char *Str_copy(char *dest, const char *src);
/* Concatenate src to the end of dest.  Return dest. */
char *Str_concat(char *dest, const char *src);
```

stra.c (excerpt)
```
#include "str.h"
size_t Str_getLength(co
 ... you write this code ...
}
char *Str_copy(char *de
 ... you write this code ...
}
char *Str_concat(char *
 ... you write this code ...
}
```

replace.c (excerpt)
```
#include "str.h"
/* Write line to stdout with each occurrence
   of from replaced with to. */
size_t replaceAndWrite(
    char *line, char *from, char *to) {
 ... you write this code ...
 calls  Str_getLength, Str_copy,
        Str_concat,  etc.
}
int main(int argc, char **argv) {...}
```

Whole-program testing

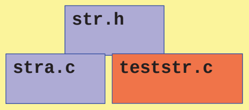
```
$ a.out ain eign
The rain in Spain is mainly in the plain.
^D
The reign in Speign is meignly in the pleign.
```



Problem: If all you're allowed to do is provide inputs to `main()`, it's hard to design tests that exercise every behavior of the individual functions in `stra.c`.

Unit testing

Put `stra.c` in a *test harness.*



teststr.c

/* Test the `Str_getLength()` function. */

```
static void testGetLength(void) {
 size_t result;
 printf("   Boundary Tests\n");
 {  char src[] = {'\0', 's'};
    result1 = Str_getLength(acSrc);
    ASSURE(result == 0);
 }
 printf("   Statement Tests\n");
 {  char src[] = {'R', 'u', 't', 'h', '\0', '\0'};
    result1 = Str_getLength(src);
    ASSURE(result == 4);
 }
 {  char src[] = {'R', 'u', 't', 'h', '\0', 's'};
    result1 = Str_getLength(src);
    ASSURE(result == 4);
 }
 {  char src[] = {'G', 'e', 'h', 'r', 'i', 'g', '\0', 's'};
    result1 = Str_getLength(src);
    ASSURE(result == 6);
}}
```

Stress Testing

Should stress the program with respect to:

- **Quantity** of data
  - Large data sets
- **Variety** of data
  - Textual data sets containing non-ASCII chars
  - Binary data sets
  - Randomly generated data sets

Should use computer to generate input sets

- Avoids human biases

Stress testing

```
enum {STRESS_TEST_COUNT = 10};
enum {STRESS_STRING_SIZE = 10000};

static void testGetLength(void) {

 . . .

 printf("   Stress Tests\n");
 {int i;
  char src[STRESS_STRING_SIZE];
  for (i = 0; i < STRESS_TEST_COUNT; i++) {
     randomString(src, STRESS_STRING_SIZE);
     result1 = Str_getLength(acSrc);
     ASSURE(result1 == strlen(acSrc));
  }
 }
}
```

Is this cheating?
Maybe, maybe not.

When you don't have a reference implementation to give you "the answer"

```
printf("   Stress Tests\n");
{int i,j;
 char src[STRESS_STRING_SIZE];
 for (i = 0; i < STRESS_TEST_COUNT; i++) {
    randomString(src, STRESS_STRING_SIZE);
    result1 = Str_getLength(acSrc);

    ASSURE (0 <= result1);
    ASSURE (result1 < STRESS_STRING_SIZE);
    for (j=0; j<result1; j++)
      ASSURE (src[j]!=0);
    ASSURE (src[result1]==0);
  }
 }
}
```

Think of as many properties as you can that the right answer must satisfy.

You can . . .

. . . combine unit testing and regression testing!

. . . write your unit tests (teststr.c) before you write your client code (replace.c)

. . . write your unit tests (teststr.c) before you begin writing the code that they will test (stra.c)

. . . use your unit-test design as a way to refine your interface specifications (i.e., what's described in comments in the header file) another reason to write the unit tests before writing the code!

. . . avoid relying on the COS 217 instructors to provide you all the unit tests in advance. (We have more unit tests in our grading system than we give you in the homework assignments. It's your job to test your own code!)

Unit testing tools

WIKIPEDIA
The Free Encyclopedia

List of unit testing frameworks

From Wikipedia, the free encyclopedia

This page is a list of tables of code-driven unit testing frameworks for various programming languages. Some but not all of these are based on xUnit.

Contents [hide]

Q: Are we allowed to use unit testing tools in the homeworks?

A: Yes, but if you do,

- You're on your own, don't ask the preceptors or Lab TAs for help with the tool
- Describe in your README how you used the tool.

Internal testing

ASSERTIONS

The `assert` Macro

`assert(int expr)`

- If `expr` evaluates to TRUE (non-zero):
  - Do nothing
- If `expr` evaluates to FALSE (zero):
  - Print message to stderr "assert at line x failed"
  - Exit the process

Useful for internal testing

The `assert` Macro

Disabling `asserts`

- To disable asserts, define `NDEBUG`…
- In code:

```
/*------------------------------------*/
/* myprogram.c                        */
/*------------------------------------*/
#include <assert.h>

#define NDEBUG
…
/* Asserts are disabled here. */
…
```

- Or when building:

```
$ gcc217 –D NDEBUG myprogram.c –o myprogram
```

Validating Parameters

(1) Validate parameters

- At leading edge of each function, make sure values of parameters are valid

```
int f(int i, double d)
{
   assert(i has a reasonable value);
   assert(d has a reasonable value);
   …
}
```

Validating Parameters

- Example

```
/* Return the greatest common
   divisor of positive integers
   i and j. */

int gcd(int i, int j)
{
   assert(i > 0);
   assert(j > 0);
   …
}
```

Checking Invariants

(2) Check invariants

- At function entry, check aspects of data structures that should not vary; maybe at function exit too

```
int isValid(MyType object)
{  …
   /* Code to check invariants goes here.
      Return 1 (TRUE) if object passes
      all tests, and 0 (FALSE) otherwise. */
   …
}

void myFunction(MyType object)
{  assert(isValid(object));
   …
   /* Code to manipulate object goes here. */
   …
   assert(isValid(object));
}
```

Checking Invariants

- Example
  - "Balanced binary search tree insertion" function
  - At leading edge:
    - Are nodes sorted?
    - Is tree balanced?
  - At trailing edge:
    - Are nodes still sorted?
    - Is tree still balanced?

Checking Return Values

(3) Check function return values

- Check values returned by called functions

```
f(someArgs);
…
```

Bad code (sometimes)

```
someRetValue = f(someArgs);
if (someRetValue == badValue)
   /* Handle the error */
…
```

Good code

```
if (f(someArgs) == badValue)
   /* Handle the error */
…
```

Good code

Checking Return Values

- Example:
  - scanf() returns number of values read
  - Caller should check return value

```
int i, j;
…
scanf("%d%d", &i, &j);
```

Bad code

```
int i, j;
…
if (scanf("%d%d", &i, &j) != 2)
   /* Handle the error */
```

Good code

Checking Return Values

- Example:
  - printf() returns number of chars (not values) written
  - Can fail if writing to file and disk quota is exceeded
  - Caller should check return value???

```
int i = 1000;
…
printf("%d", i);
```

Bad code???

Is this too much?

```
int i = 1000;
…
if (printf("%d", i) != 4)
   /* Handle the error */
```

Good code???

Checking array subscripts

Out-of-bounds array subscript is the cause of vast numbers of security vulnerabilities in C programs!

```
#include <stdio.h>
#include <assert.h>

#define N 1000
#define M 1000000
int a[N];

int main(void) {
  int i,j, sum=0;
  for (j=0; j<M; j++)
    for (i=0; i<N; i++) {
      assert (0 <= i && i < N);
      sum += a[i];
     }
  printf ("%d\n", sum);
}
```

Checking array subscripts

```
#include <stdio.h>
#include <assert.h>

#define N 1000
#define M 1000000
int a[N];

int main(void) {
  int i,j, sum=0;
  for (j=0; j<M; j++)
    for (i=0; i<N; i++) {
      assert (0 <= i && i < N);
      sum += a[i];
     }
  printf ("%d\n", sum);
}
```

Doesn't that slow it down?

How much slower is this program with the assertion?

$ gcc –O2 test.c; time a.out

0.385 seconds ± .02

$ gcc –O2 –D NDEBUG test.c; time a.out

0.385 seconds ± .02

Why?

Leave Testing Code Intact!

Examples of testing code:

- unit test harnesses (entire module, **`teststr.c`**)
- **`assert`** statements
- entire functions that exist only to support asserts (**`isValid()`** function)

Do not remove testing code when program is finished

- In the "real world" no program ever is "finished"

If testing code is inefficient:

- Embed in calls of **`assert()`**, or
- Use **`#ifdef…#endif`** preprocessor directives
  - See Appendix

TEST COVERAGE

Statement Testing

(1) **Statement** testing

- "Testing to satisfy the criterion that each statement in a program be executed at least once during program testing."

From the *Glossary of Computerized System and Software Development Terminology*

Statement Testing Example

Example pseudocode:

```
if (condition1)
   statement1;
else
   statement2;
…
if (condition2)
   statement3;
else
   statement4;
…
```

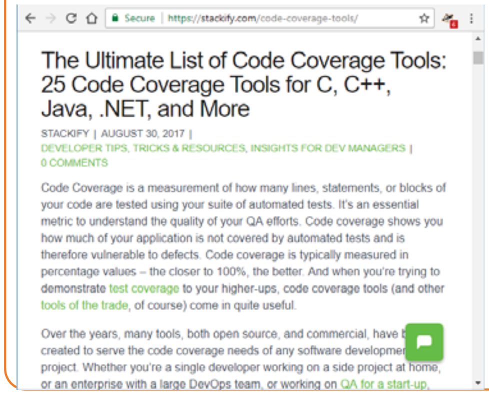
Statement testing:

Should make sure both **`if`** statements and all 4 nested statements are executed

How many passes through code are required?

How can you measure code coverage?

Use a tool!



The Ultimate List of Code Coverage Tools: 25 Code Coverage Tools for C, C++, Java, .NET, and More

STACKIFY | AUGUST 30, 2017 |
DEVELOPER TIPS, TRICKS & RESOURCES, INSIGHTS FOR DEV MANAGERS | 0 COMMENTS

Code Coverage is a measurement of how many lines, statements, or blocks of your code are tested using your suite of automated tests. It's an essential metric to understand the quality of your QA efforts. Code coverage shows you how much of your application is not covered by automated tests and is therefore vulnerable to defects. Code coverage is typically measured in percentage values – the closer to 100%, the better. And when you're trying to demonstrate test coverage to your higher-ups, code coverage tools (and other tools of the trade, of course) come in quite useful.

Over the years, many tools, both open source, and commercial, have b created to serve the code coverage needs of any software developme project. Whether you're a single developer working on a side project at home, or an enterprise with a large DevOps team, or working on QA for a start-up,

Q: Are we allowed to use code coverage tools in the homeworks?

A: Yes, but if you do,
- You're on your own, don't ask the preceptors or Lab TAs for help with the tool
- Describe in your README how you used the tool.

Path Testing

(2) **Path** testing

- "Testing to satisfy coverage criteria that each logical path through the program be tested. Often paths through the program are grouped into a finite set of classes. One path from each class is then tested."

From the *Glossary of Computerized System and Software Development Terminology*

Path Testing Example

Example pseudocode:

```
if (condition1)
    statement1;
else
    statement2;
…
if (condition2)
    statement3;
else
    statement4;
…
```

Path testing:

Should make sure all logical paths are executed

How many passes through code are required?

- Simple programs ⇒ maybe reasonable
- Complex program ⇒ combinatorial explosion!!!
  - Path test code fragments

Some code coverage tools can also assess path coverage.

Boundary Testing

(3) **Boundary** testing (alias **corner case** testing)

- "A testing technique using input values at, just below, and just above, the defined limits of an input domain; and with input values causing outputs to be at, just below, and just above, the defined limits of an output domain."

From the *Glossary of Computerized System and Software Development Terminology*

Boundary Testing Example

How would you boundary-test this function?

```
/* Where a[] is an array of length n,
   return the index i such that a[i]=x,
   or -1 if not found */
int find(int a[], int n, int x);
```

Boundary Testing Example

How would you boundary-test this function?

```
/* Where a[] is an array of length n,
   return the index i such that a[i]=x,
   or -1 if not found */
int find(int a[], int n, int x);
```

```
int a[10];
for (i=0;i<10;i++) a[i]=1000+i;
assert (find(a,10,1009)==9);
assert (find(a,10,1000)==0);
assert (find(a+1,8,1000)== -1);
assert (find(a,9,1009)== -1);
```

Stress Testing

(4) **Stress** testing

- "Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements"

From the *Glossary of Computerized System and Software Development Terminology*

Stress Testing Example 1

Specification:

- Print number of characters in stdin

Attempt:

```
#include <stdio.h>
int main(void)
{  char charCount = 0;
   while (getchar() != EOF)
      charCount++;
   printf("%d\n", charCount);
   return 0;
}
```

Does it work?

Stress Testing Example 2

Specification:

- Read a line from **stdin**
- Store as string (without **'\n'**) in array of length **ARRAY_LENGTH**

Attempt:

```
int i;
char s[ARRAY_LENGTH];
for (i = 0; i < ARRAY_LENGTH-1; i++)
{  s[i] = getchar();
   if ((s[i] == EOF) || (s[i] == '\n')) break;
}
s[i] = '\0';
```

Does it work?

49

Changing Code Temporarily

(4) Change code temporarily

- Temporarily change code to generate artificial boundary or stress tests
- Example: Array-based sorting program
  - Temporarily make array very small
  - Does the program handle overflow?

50

Bug-Driven Testing

(5) Let debugging drive testing

- Reactive mode…
  - Find a bug ⇒ create a test case that catches it
- Proactive mode…
  - Do **fault injection**
    - Intentionally (temporarily!) inject a bug
    - Make sure testing mechanism catches it
    - Test the testing!!!

51

Who Does the Testing?

Programmers

- **White-box** testing
- Pro: Know the code ⇒ can test all statements/paths/boundaries
- Con: Know the code ⇒ biased by code design

Quality Assurance (QA) engineers

- **Black-box** testing
- Pro: Do not know the code ⇒ unbiased by code design
- Con: Do not know the code ⇒ unlikely to test all statements/paths/boundaries

Customers

- **Field** testing
- Pros: Use code in unexpected ways; "debug" specs
- Cons: Often don't like "participating"; difficult to generate enough cases

52

Summary

Test coverage

- Statement coverage (measure with tools)
- Path coverage (measure with tools)
- Boundary testing
- Stress testing
- Regression testing

External testing (manage with tools)

Unit testing (manage with tools)

Internal testing

- Validate parameters
- Check invariants
- Check function return values
- Change code temporarily
- Leave testing code intact

Test the code—and the tests!

53

Appendix: #ifdef

Using **#ifdef…#endif**

```
…
#ifdef TEST_FEATURE_X
/* Code to test feature
   X goes here. */
#endif
…
```

myprog.c

- To enable testing code:

```
$ gcc217 –D TEST_FEATURE_X myprog.c –o myprog
```

- To disable testing code:

```
$ gcc217 myprog.c –o myprog
```

54