Questions

Bookmark this page

testing basics

1/1 point (graded)

In the 1990s, the Ariane 5 launch vehicle, designed and built for the European Space Agency, self-destructed 37 seconds after its first launch.

The reason was a control software bug that went undetected. The Ariane 5's guidance software was reused from the Ariane 4, which was a slower rocket. When the velocity calculation converted from a 64-bit floating point number (a `double` in Java terminology, though this software wasn't written in Java) to a 16-bit signed integer (a `short`), it overflowed the small integer and caused an exception to be thrown. The exception handler had been disabled for efficiency reasons, so the guidance software crashed. Without guidance, the rocket crashed too. The cost of the failure was $1 billion.

What ideas does this story demonstrate? Check all that apply.

- [x] Even high-quality safety-critical software may still have residual bugs.
- [ ] Testing all possible inputs is the best solution to this problem.
- [x] Software exhibits discontinuous behavior, unlike many physically-engineered systems.
- [ ] Static type checking could have detected this bug.

✔

Explanation

Testing all inputs is not feasible, because even just one 64-bit floating point number has 2^64 possible values, which is more than the age of the universe in microseconds.

Static type checking wouldn't have detected this bug, because the code intentionally converted a 64-bit `double` into a 16-bit `short`.

Submit

Show Answer

Answers are displayed within the problem

Questions

Bookmark this page

partitioning

1/1 point (graded)

Consider the following specification:

```
/**
 * Reverses the end of a string.
 *
 *                          012345                    012345
 * For example: reverseEnd("Hello, world", 5) returns "Hellodlrow ,"
 *                              <----->                   <----->
 *
 * With start == 0, reverses the entire text.
 * With start == text.length(), reverses nothing.
 *
 * @param text    non-null String that will have its end reversed
 * @param start   the index at which the remainder of the input is reversed,
 *                requires 0 <= start <= text.length()
 * @return input text with the substring from start to the end of the string reversed
 */
public static String reverseEnd(String text, int start)
```

Which of the following are reasonable partitions for the start parameter? Check all that apply.

- [ ] start = 0, start = 5, start = 100
- [ ] start < 0, start = 0, start > 0
- [x] start = 0, 0 < start < text.length(), start = text.length()
- [ ] start < text.length(), start = text.length(), start > text.length()

✔

Explanation
0, 5, 100 is not a partition. A partition should be a division of the whole space of possible start values, not specific test cases.
start < 0 and start > text.length() are not legal inputs for the function, so it isn't reasonable to use them as part of a partition.

Submit Show Answer

Answers are displayed within the problem

partitioning, part 2

1/1 point (graded)

Which of the following are reasonable partitions for the text parameter? Check all that apply.

- [ ] text contains some letters; text contains no letters, but some numbers; text contains neither letters nor numbers
- [x] text.length() = 0; text.length() > 0
- [x] text.length() = 0; text.length()-start is odd; text.length()-start is even
- [ ] text is every possible string from length 0 to 100

✔

Explanation
Letters and numbers aren't important to the behavior of this function, so it isn't reasonable to partition on that property.
Length is a useful partition, since it can interact with the start parameter.
Partitioning on even and odd length is also reasonable, because reversing an odd-length substring has different behavior (since it leaves the middle element in place) than an even-length string (where all elements swap).
Partitioning into all possible length 0 to 100 strings will produce far too many test cases.

Submit

Show Answer

Answers are displayed within the problem

Previous Next

Questions

Bookmark this page

blackbox and whitebox testing

1/1 point (graded)

Consider the following function:

```
/**
 * Sort a list of integers in nondecreasing order.  Modifies the list so that
 * values.get(i) <= values.get(i+1) for all 0<=i<values.length()-1
 */
public static void sort(List<Integer> values) {
    // choose a good algorithm for the size of the list
    if (values.length() < 10) {
        radixSort(values);
    } else if (values.length() < 1000*1000*1000) {
        quickSort(values);
    } else {
        mergeSort(values);
    }
}
```

Which of the following test cases are likely to be boundary values produced by white box testing?

- [ ] the empty list
- [ ] 1, 2, 3
- [x] 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
- [ ] 0, 0, 1, 0, 0, 0, 0

✔

Explanation

When the list is length 10, the function switches from one sorting algorithm (radix sort) to another (quick sort). This makes length-10 lists a boundary value in the behavior of the function. But this boundary isn't visible from the function's specification (black box), only from its implementation (white box).
The empty list, on the other hand, is a boundary value for every list, so white box testing isn't required to discover it.
The other lists are not boundary values by any visible evidence in this code.

Submit

Show Answer

Answers are displayed within the problem

Questions

Bookmark this page

coverage

3/3 points (graded)

Install EclEmma in Eclipse.
Then create a new Java class called Hailstone.java containing this code:

```
public class Hailstone {
  public static void main(String[] args) {
    int n = 3;
    while (n != 1) {
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
  }
}
```

Run this class with EclEmma code coverage highlighting turned on, by choosing Run / Coverage As / Java Application.

By changing the initial value of n, you can observe how EclEmma highlights different lines of code differently.

When n=3 initially, what color is the line n = n/2 after execution?

green ✔ **Answer:** green

Explanation
Run EclEmma and find out!
If it's hard to perceive the difference between the red and green highlighting, you can change the color of the coverage highlighting by going to Preferences / General / Appearance / Editors / Text Editors / Annotations. The annotations to change are called Full Coverage, Partial Coverage, and No Coverage. You might, for example, make Full Coverage white, Partial Coverage light gray, and No Coverage dark gray.

When n=16 initially, what color is the line n = 3 * n + 1 after execution?

red ✔ **Answer:** red

Explanation
Run EclEmma and find out!

What initial value of n would make the line while (n != 1) yellow after execution?

1 ✔ **Answer:** 1

1

Explanation
Yellow is used for a line that contains a branch (e.g., an if or while predicate) that is only taken in one direction during the program -- either the predicate is false every time the program gets there, or the predicate is true every time. In this case, the way to make the while predicate yellow is to make n != 1 false, so that the predicate executes just once and the loop is skipped entirely. n=1 does that.

Submit Show Answer

Answers are displayed within the problem

Questions

Bookmark this page

unit testing

3/3 points (graded)

The RSA cryptography algorithm needs two prime numbers, p and q, to encrypt and decrypt messages. The encryption key is the product n=pq, and the decryption key is p and q.

Suppose you are building an RSA implementation with several modules in it:

```
long randomPrimeNumber();
String encrypt(long n, String message);
String decrypt(long p, long q, String message);
```

Construct a suitable **unit test** for decrypt() below by choosing code to replace code1, code2, and code3. Assume the resulting test is correct.

```
String originalMessage = "hello!";
code1
code2
code3
assertEquals(originalMessage, decryptedMessage);
```

code1

- ( ) long p = randomPrimeNumber(); long q = randomPrimeNumber();
- (•) long p = 17; long q = 37;

✔

code2

- (•) String encryptedMessage = "iy@94RFe"
- ( ) String encryptedMessage = encrypt(p*q, originalMessage);

✔

code3

- (•) String decryptedMessage = decrypt(p, q, encryptedMessage);
- ( ) String decryptedMessage = "hello!";

✔

Explanation

A good unit test for decrypt() isolates it, calling **only** decrypt() rather than the other modules of the program.

How did we come up with the encryptedMessage without calling encrypt()? For small p and q, we may have done it by hand. Or we may have run encrypt() and stored its output once we were confident that it worked, so that this code could detect regressions.

Submit

Show Answer

Answers are displayed within the problem

Questions

Bookmark this page

unit testing

3/3 points (graded)

The RSA cryptography algorithm needs two prime numbers, p and q, to encrypt and decrypt messages. The encryption key is the product n=pq, and the decryption key is p and q.

Suppose you are building an RSA implementation with several modules in it:

```
long randomPrimeNumber();
String encrypt(long n, String message);
String decrypt(long p, long q, String message);
```

Construct a suitable **unit test** for decrypt() below by choosing code to replace code1, code2, and code3. Assume the resulting test is correct.

```
String originalMessage = "hello!";
code1
code2
code3
assertEquals(originalMessage, decryptedMessage);
```

code1

- ( ) long p = randomPrimeNumber(); long q = randomPrimeNumber();
- (•) long p = 17; long q = 37;

✔

code2

- (•) String encryptedMessage = "iy@94RFe"
- ( ) String encryptedMessage = encrypt(p*q, originalMessage);

✔

code3

- (•) String decryptedMessage = decrypt(p, q, encryptedMessage);
- ( ) String decryptedMessage = "hello!";

✔

Explanation

A good unit test for decrypt() isolates it, calling **only** decrypt() rather than the other modules of the program.
How did we come up with the encryptedMessage without calling encrypt()? For small p and q, we may have done it by hand. Or we may have run encrypt() and stored its output once we were confident that it worked, so that this code could detect regressions.

Submit Show Answer

Answers are displayed within the problem

Questions

Bookmark this page

regression testing

1/1 point (graded)

Which of the following best defines regression testing?

- (●) Changes should be tested against all inputs that elicited bugs in earlier versions of the code.
- ( ) Every component in your code should have an associated set of tests that exercises all the corner cases in its specification.
- ( ) Tests should be written before you write the code as a way of checking your understanding of the specification.
- ( ) When a new test exposes a bug, you should run it on all previous versions of the code until you find the version where the bug was introduced.

✔

Explanation

The first choice is correct because regression test cases are inputs that once triggered bugs, and the idea of regression testing is to test new changes to the code against these test cases.
The second choice is a good practice -- black-box unit testing -- but it isn't regression testing.
The third choice is also good practice -- test-first programming -- but again, not regression testing.
The fourth choice may be useful for tracking down a difficult bug. It is a variant of delta debugging, and is never done as exhaustively as *all* previous versions, but more typically with a binary search. Still, it is not regression testing.

Submit Show Answer

Answers are displayed within the problem

running automated tests

1/1 point (graded)

Which of the following are good times to rerun all your JUnit tests? Check all that apply.

- [x] Before doing git add/commit/push
- [x] After rewriting a function to make it faster
- [x] When using a code coverage tool
- [x] After you think you fixed a bug

✔

Explanation

Pushing your code to git sends it to the rest of your team, so rerun the tests first to make sure you're not pushing broken code.
Rewriting a function may introduce bugs, so rerun your tests to find them.
Rerunning tests is an essential part of using a code coverage tool, because you want to see the code lines that your tests don't reach.
Fixing a bug is a change to your program, and you should rerun your tests after every change.

Submit Show Answer

Answers are displayed within the problem

testing techniques

1/1 point (graded)

Which of these techniques are useful for choosing test cases in test-first programming, before any code is written? Check all that apply.

- [x] black box
- [ ] regression
- [ ] static typing
- [x] partitioning
- [x] boundaries
- [ ] white box
- [ ] coverage

✔

Explanation
Regression testing creates test cases from bugs, so it only happens after (buggy) code has been written.
Static typing doesn't generate test cases.
White box testing and coverage both require looking at the written code.

Submit

Show Answer

Answers are displayed within the problem

Previous Next