EECS4313 week 3

Jerry Wu

2024-01-22

Contents

Lecture 1

Test automation

1.1 Why automated testing?

- Manual testing is time consuming and not economical
- Automation is **unattended** and **can happen overnight**; thus not requiring human intervention
- Automation increases the speed of test execution
- Manual testing is error prone due to its recurring nature

1.1.1 Executing automated tests

- Test automation is the use of software separate from the software being tested to control the execution of tests
  - Generating test inputs and expected results
  - Running test suites without manual intervention
  - Evaluating pass/no pass
- Testing must be automated to be **effective and repeatable**

1.1.2 What can be automated?

Each one of these steps in software development can be automated, and each step down is easier to automate than the last. Of course, there has to be some human intervention to formulate the requirements for automation at the start.

- **Analyze** - intellectual (performed once)
- **Design** - mostly intellectual
- **Construct** - clerical and intellectual
- **Testing** - mostly clerical
- **Deploy** - clerical (repeated many times)

The more clerical something is, the more worth automating it is.

1.2 Record & playback

- Usually for websites, mobile, UIs, etc.
- Test scripts are recorded on the initial version of the application
- The same scripts are executed on the next version
- The scripts need **some modification** (quite costly at high number of modification) for the changes happening on the application during every version of the application
- The test scripts repository keeps growing as the application goes through changes. This makes this kind of test suite very hard to maintain

1.2.1 Procedure

- Automation tool generates scripts by recording user actions
- The generated scripts can be played back to reproduce the exact user actions

1.2.2 Advantages

- Less effort for automation
- Quick returns
- Does not require expertise on tools

1.2.3 Limitations

- High dependency on the GUI of AUT (application under test)
- Difficult to maintain the scripts

1.2.4 Problems

- Very fragile
  - A single change in UI can cause the whole system to break
- Hard to maintain
  - An abundance of test scripts causes the test suite to be hard to maintain
- No modularity or reuse
  - System must be ready before automation can start

1.3 Testing terminology contd'

"You don't want to make assumptions. That is not unit testing" - H.V. Pham 2024

- A **unit test** is a test of a single class/method
- A **test case** tests the respose of a single method to a particular set of inputs
- A **test suite** is a collection of test cases
- A **test runner** is software that runs tests and reports results
- A **flaky test** is a test case where it is sometimes triggered and sometimes not depending on specific conditions, seemingly randomly.

1.3.1 Std. structure of a JUnit test class

- Test a class called `Fraction`
- Create a test class called `FractionTest`

```
1  import org.junit.;
2  import static org.junit.Assert.;
3
4  public class FractionTest{
5      @Test
6      public void addTest(){...}
7
8      @Test
9      public void testToString(){...}
10
11     //useful if we have a counter during the tests to set in SetUp() and
           reset in TearDown()
12     @Before
13     public void SetUp(){...}
14
15     @After
16     public void TearDown(){...}
17 }
```

1.3.2 More fixtures for test classes

- There is also a `@BeforeClass` annotation that will execute once before **all** test cases
- Similarly, there is also an `@AfterClass` annotation that executes a method after every test case
- Usually done for expensive allocations for resources like connecting/disconnecting a database.

1.3.3 Assert method: Equal, Null and fail

- `assertEquals(Object expected, Object actual)` - Compare two objects to see if they're identical. Use `.equals(Object other)` for compare

- `assertArrayEquals(int[] expecteds, int[] actuals)` or `assertArrayEquals(long[] expecteds, long[] actuals)` - Compares two arrays
- `assertSame(Object expected, Object actual)` - asserts that two rerferences are the same object (using `==`). Useful for testing a find method for data structures
- `assertNull(Object object)` - asserts that a reference is not null
- `fail()` - causes the test to fail and throw an exception (`AssertionError`). Useful as a result of a complex test, or when testing for exceptions.

1.3.4 Exception testing

- If a test case is expected to raise an exception, it can be written as follows:

```
1 @Test(expected = Exception.class)
2 public void testException(){
3     //some code that should raise an exception
4
5     fail("Should raise an exception");
6 }
```

- if code doesn't throw an exception in this case, test fails. Else test passes.

1.3.5 Assert statement

- `assert boolean_condition;` - throws an `AssertionError` if `boolean_condition == false`. Can be used in place of `assertTrue()` in JUnit.

1.3.6 Ignoring test cases

- Test cases that are not finished being written yet can be ignored using the `@Ignore` annotation.
- JUnit will not execute the test cases marked with this annotation, and will report the number of test cases that were ignored

1.4 Types of test generation

- **Random testing**

- Pure random
  - Easiest way to do automatic test case generation
  - Doesn't require any preparation and is easy to implement
  - However, there may be semantically redundant inputs. E.g., for a program that computes $\frac{10}{x}$, providing any onput except 0 means the same thing.
  - For example:

```
1 //Simple useful test
2 Set s = new HashSet();
3 s.add("hi");
4 assertTrue(s.equals(s));
```

  - Redundant test:

```
1 //Redundant test
2 Set s = new HashSet();
3 s.add("hi");
4 s.add("hi");
5 assertTrue(s.equals(s));
```

  - Another example

```
1 //Simple useful test
2 Date d = new Date(2006,2,14);
3 assertTrue(d.equals(d));
```

  - Illegal test:

```
1 //Simple useful test
2 Date d = new Date(2006,2,14);
3 d.setMonth(-1); // not allowed!
4 assertTrue(d.equals(d));
```

  - Illegal test:

```
1 //Simple useful test
2 Date d = new Date(2006,2,14);
3 d.setMonth(-5); // ILLEGAL!
4 assertTrue(d.equals(d));
```

  - Rule-based - uses information like past crashes, constraints, etc. to make test cases
    * Use rules like boundary cases: $x \in [-2^{32}, 2^{32}]$
    * Use distributions like normal, bimodal, χ^2, etc. Ask Jason.
  - Feedback guided (Radoop)
    * Question: It is easy to generate random values for primitive types like `int`. How do we generate random objects like linked lists, trees, etc.?
    * Use `id = -1, content = "", child = null`

```
1  class TreeNode{
2      int id; String content; Child child;
3      public TreeNode(String str, Child child){
4      this.id = Global.id ++;
5      this.content = "node:" + str;
6      this.child = child;
7      }
8      public do(){
9      String content = this.content.substring(4); //
           NullPointerException! (at random)
10     this.child.do();
11     ...
12     }
13 }
```

  - Build test inputs incrementally
    * New test inputs extend previous ones
    * In this context, a test input would be a method sequence
  - As soon as new test input is created, execute it
  - Use execution result to guide future test case generations
    * away from redundant or illegal method sequences
    * towards sequences that create new object states
    * do not use duplicate and null objects
    * do not use objects generated from exceptions

- **Search based testing**
  - EvoSuite
  - Deem test case generation as an optimization problem

- Based on random testing, and focuses on the input domains
- Uses code coverage as guidance
- Find input values that can achieve best coverage such as statement coverage, logic coverage, input, etc.
- Try to find maximum or minimum value of a certain function
- Numerous practical problems can be viewed as optimization problems
  * Least cost to travel to a number of cities
  * Least camera to cover an area
  * Distribution of stores to attract most customers
  * Design pipe of systems with least material
  * Put items into a backpack (with limited volume) of the highest value

1.4.1 Solutions of optimization

- Hill climbing
  * Start from random point
  * Try all neighbouring points and choose neighbour with highest value until all neighboring points have a lower value than the current point
  * Easy to find **local** maxima
- Random restart hill climbing
  * Restart hill climbing multiple times to avoid local maxima and get to global maxima
- Annealing simulation
  * Improved hill climbing
  * Has probability to move (i.e. restart) after reaching local peak
  * The probability drops as time goes by
- Genetic algorithm (search based SE)
  * Simulate the process of evolution
  * Start with random points
  * Select a number of best points
  * Combine and mutate these points until no more improvements can be made

1.4.2 Symbolic execution based test generation use cases

- Making a target at a certain code coverage
- Understanding how the code works to use this method
- Analyze the code structure to find out a path to go to a certain statement & the constraints of certain inputs to let the program follow the path
- Analyzing a program to determine what inputs cause a part of a program to execute

1.4.3 The main idea

- If a statement hasn't been covered yet, try and provide an input to go over that statement
- A statement is only covered once a path that goes to said statement is covered
- So what input will cause the program to go through certain paths? Only if the input satisfies all if conditions along the path!

1.4.4 Static symbolic execution

- Do not execute the software at all
- Construct a constraint for each statement, the statement will be executed when the constraint is satisfied
- Solve the constraints
- The parameters (variables) in te constraint are inputs of the software to cover a specific statement

- **Basic example**:
  Here, `T` is the condition for the statement to reach this block. `y=s` is the relationship of all variables to the inputs after the statement is executed. `s` is a symbolic variable for input

```
1 y=read();            //T                                        (y=s)
2 y=2*y;               //T                                        (y=2*s)
3 if(y <= 12)          //T                                        (y=2*s)
4     fails();         //T && y <= 12                             (y=2*s)
5 else
6     success();       //T && !(y <= 12)                          (y=2*s)
7 print("OK");         //T && y <= 12 (y=2*s) || T && !(y<=12)    (y=2*s)
```

1.4.5 Problems in static symbolic execution

- Path explosion
  - Remember n branches will cause $2n$ paths
  - Infinite paths for unbounded (infinite) loops
  - Calculating constraints on all paths is infeasible for real software
- Overcomplicated constraints
  - Constraints get very complex for large programs (ESPECIALLY when loops are involved)
  - Not to mention resolving the constraint is an NPC (nondeterministic polynomial complete, or NP complete) problem.
- **Moral of the story**: we only want to usse SSE for functional/method level testing. Anything beyond that will be impractical.

1.4.6 Dynamic symbolic execution

- One of the revolutionary progress milestones of SWE in the 21st century

1.4.7 Main idea

- Actually run the program
- Generate the constraints as the program runs.
- **Flip the constraints to reach other statements**

1.4.8 Symbolic execution challenges

- Scalability
  - Key challenge!
  - Path space of a large program is massive
- Complex non linear constraints
- Testing web apps and security problems
  - String constraints
  - Mixed numeric and string constraints
- Java Pathfinder for java
- PEX
  - Visual studio 2010 power tool
- KLEE
  - Most popular symbolic execution tool for test generation in C/C++