Testing: a form of program validation

pratical method validating program’s correctness

real data

satisfies specification

Testing method(step-by-step):generate Test Cases -> Develop& Run Tests

- unit testing: each module in isolation
- integration testing: a group of modules
- regression testing: after modifications -> re-run tests

Test Case (TC): combination {input data values} (of given test unit)

- exhaustive testing: *impractical*
- small representative set of Test Cases
- succeed with TCs as well as input

Test stand-alone procedures

- ✓ generate Test Cases: black-box white-box
- ✓ develop & run Test: Junit

generate Test Cases

- generate TDSs define input range + a TDS per range
- Combine TDSs -> form **Test Cases**

(using representative data values)

BBT black-box testing → generated from *specification*

GBT glass-box or white-box testing → code

develop & run Tests

repeat Testing task

1. difficultly automate ←→ specification not always precise
2. automated: Junit

develop Test Drivers -> ***realise*** Test Cases -> ***automate*** tests execution

Test run = run test driver

Test Drivers (small program) – may form a type hierachy

data abstractions -> name convention: ***Test

- initialize TDSs + TCs
- expected test result
- test each unit

Implement: **JUnit** – third-party package

- JUnit 3.5: test drivers as sub-types
- JUnit 4.0 (jdk >= 1.5): test drivers as annotated procedures

Assertion: boolean statement validated automatically (by run-time environment)

➔ validate *test results* *+ denfensive programming*

java keyword: assert

variants: *assertEquals* *assertArrayEquals*

```
// initialise s
IntSet s = …
// assert that s.repOK true
// otherwise throws an AssertionError
assert (s.repOK() == true);

// assert that s has 2 elements
// otherwise throws AssertionError with message “invalid size …”
```

```
assert (s.size() == 2) : "invalid size " + s.size();
```

Test Driver for procedure

named after procedure

no abstract properties

may be parameterized for each Test Case

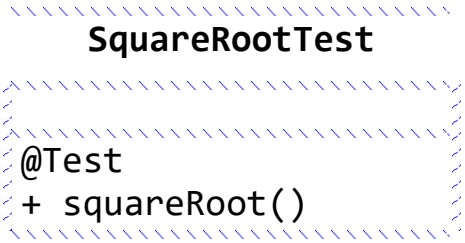
@Test

throws AssertionError

Use arrays → initialise Test Cases & Results

loop → run each Test Case

Assert.assertEquals (static method) → test assertion easily



```java
/**
* @overview      A Test Driver for Num.sqrt method
*/
public class SquareRootTest {

      /**
      * @modifies System.out
      * @effects
      *  for each test case TC = <x, e, r>
      *  if  | Num.sqrt(x, e)^2 – r^2 | > e
      *          throws AssertionError
      *  else
      *          displays result on the std output
      *
      */
      @Test      // (timeout = 5000)
      public void squareRoot() throws AssertionError {
            // … (code omitted) …
```

```
// test cases
float[] tcEps = { 0.00002f, 0.0001f, 0.009f }


float[] tcX = { 0f, 0.001f, 0.01f, 0.09f, 0.5f, 1f, 2f, 10f, 100f,
2147483600f };



// test results
float[] results = new float[tcX.length];
for (int i = 0; i < tcX.length; i++)
      results[i] = (float) Math.sqrt(tcX[i]);
float x, e, r;
for (int i=0; i< tcX.length; i++) {
      x = tcX[i];
      r = results[i];
      for (int j=0; j < tcEps.length; j++) {
            Sysout.println(“>>Test Case ” + ((i * tcEps.length) + j));
            e = tcEps[j];
            float result = Num.sqrt(x, e);


            // assume same delta error b/w trwo results
            assertEquals(r*r, result*result, 2*e);


            System.out.printf(“sqrt(%f, %f) = %f” + “(expected = %f)
            %n”, x, e, result, r);
      }
}
} // end squareRoot
```

Parameterised Test Driver

@RunWith(Parameterized.class)

Constructor -> initialise rep (with suitable args)

→ defines a Test Case + expected output

Method → Test Cases

@Parameters

static

return: Collection

Test method operates on rep directly

Defensive Programming: insert checking → detect errors

3 additional 'checks'

- representation(rep_invariant) → implement repOK
- @requires → check input values against pre-condition
- exhaustive testing (all conditionals) → all possible cases (incl. unspecified)

```
String s = Com.receive();

if (s.equals("deliver")) {
      // carry out deliver request
else if (s.equals("examine"))
      // carry out examine request
else
      // handle error case: can never happen!
      assert false;
```

Assertion: disabled (default) → use JVM's option – ea (- da)

enable : java -ea MyProgram

disable: java -da MyProgram

Debugging

uncover + correct bugs(errors) – (affected code regions)

examine intermediate states

Test Cases → produce bugs

efficiency: design (design diagram, design specification)

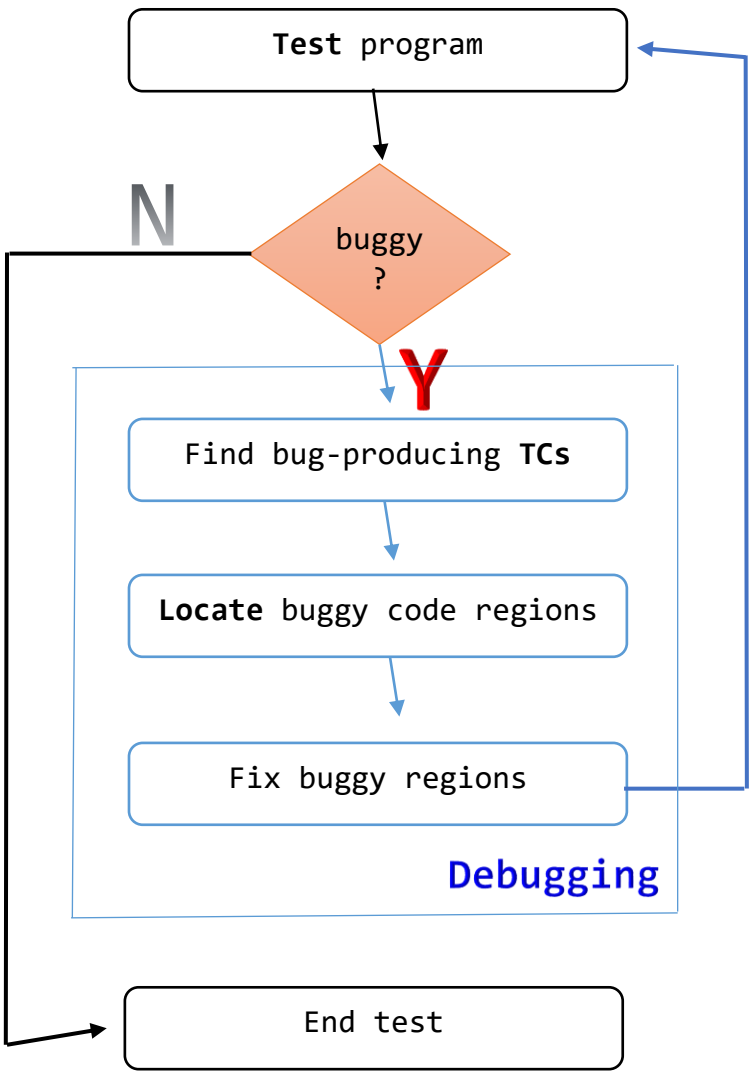
implementation

documentation (specification…)

Steps:

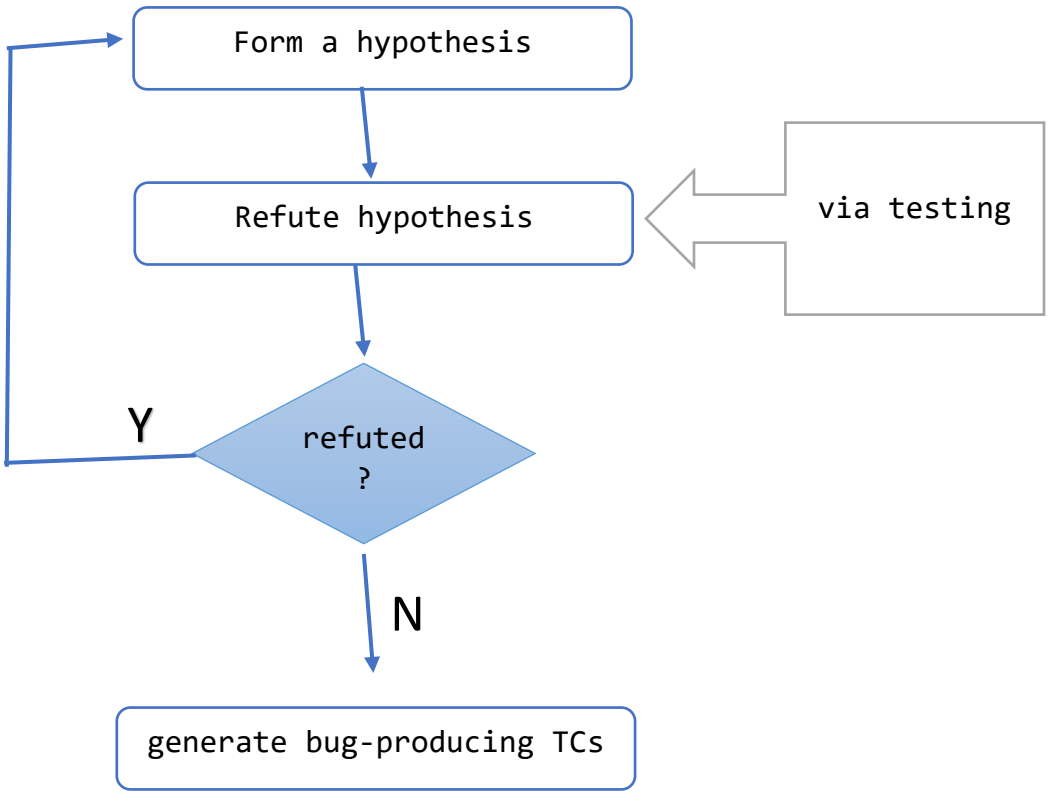
1. *Find* bug-producing TCs
2. *Locate* buggy code regions
3. *Fix* buggy code regions
4. *Retest* program (regression testing)

Flow chart



Find bug-producing TCs

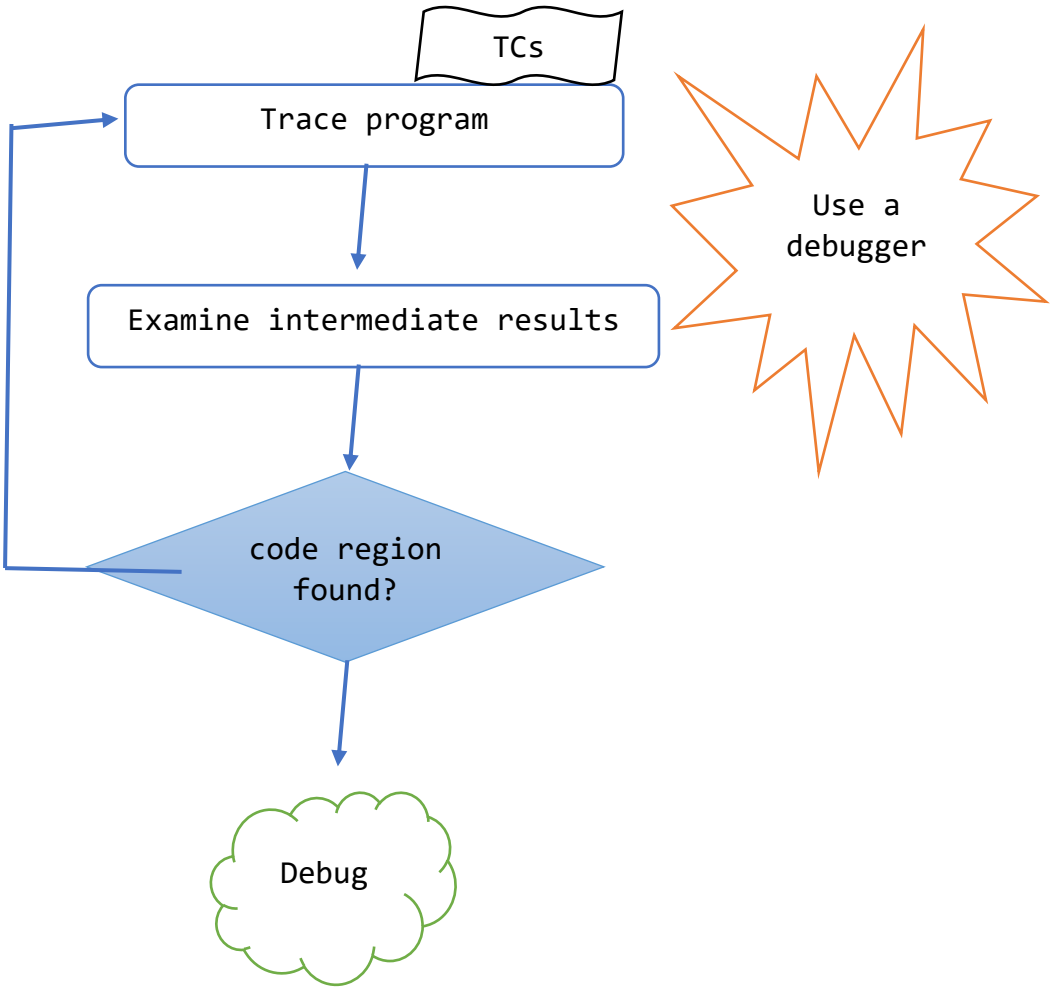
- Form a hypothesis consistent with test result
- Design + run tests ←→ refute hypothesis
- a hypothesis established
- hypothesis → generate bug-producing TCs (typical/ atypical value rule)



Locate buggy code regions

- trace program with TCs
- examine intermediate results top-down
  - ✓ check procedure groups → data abstraction
  - ✓ check @requires @effects → procedure
  - ✓ check variables → code region

- aided by a debugger (Java debugger)



Fix buggy code regions

analyse each region

common programming pitfalls:

- *syntac*tically correct typing errors
- reverse *order* of input arguments
- *loop* one index too *far*
- fail re*intialise* a variable
- *incomplete* code copy
- incorrect use of parentheses *()* in an expression

Debug guidelines

- use a debugger
- right source code
- toString methods (sensible)
- a bug (may) occur far from (its) manifestation
- program assumptions
- check input against @requires
- eliminate possible code regions
- get help from others ❤
- TAKE A BREAK!
- bug match symptoms
- aware bug occurred where it is
- impact on code modification