Test-Driven Development

Robert Keller

1 September 2004

What is TDD?

- A way of programming
- Driving the development process through tests
- Also called “Test-First Design”
- Often associated with XP (“Extreme Programming”) or “Agile Development”

References

- Kent Beck, Test Driven Development, Addison Wesley, 2003.
- David Astels, Test Driven Development, Prentice-Hall, 2003.

The Basic Idea

- Write tests first
- Then code so as to satisfy the tests
- Do the above iteratively

The Extreme Case

- Start with a program that does nothing (so it will fail almost any test).
- Write a simple test according to the spec. (even if the spec is only mental).
- Code enough of the program to satisfy the test.

Stubbing

- The object of TDD is to keep the development moving and end up with tested code in the process.
- It is ok to use stubs and other devices to satisfy the test.
- When later, more stringent, tests are introduced, these stubs will cause the tests to fail. So the stubs will need to be replaced with more robust code.

Next Step

- Write another test that the program should satisfy, but that your program won't.
- Code to satisfy that test.
- etc.

Refactoring

- For the preceding process to work most effectively, it is necessary to refactor the code whenever possible, to

  eliminate **any** duplication of code or function.

- Refactoring is done when all tests presented up to this iteration are working.

- Thus, if refactoring is responsible for breaking a test, you find out immediately and fix it.

Worked Example

- Develop a **PriorityQueue** class in Java.
- The elements in the priority queue implement the interface **Comparable**.
- A priority queue has the methods:
  - void **insert**(Comparable): insert in the queue
  - Comparable **remove**(): remove the smallest from the queue
  - boolean **isEmpty**(): tell whether the queue is empty or not

Step 0

- Define two classes:
  - PriorityQueue
  - PriorityQueueTest
- The latter class will have a callable main().
- Get the program to compile and run (it does nothing but immediately terminate).

Step 0 Result

```
class PriorityQueue
{
}

class PriorityQueueTester
{
public static void main(String arg[])
   {
   }
}
```

Step 1

- Write a test that should succeed in the long run, but will fail now.

```
public static boolean Test01()
  {
  PriorityQueue pqueue = new PriorityQueue();
  try
    {
    pqueue.insert(new Integer(50));
    }
  catch(Exception e)
    {
    return false;
    }
  return true;
  }

// Fails to compile, because insert not coded.
```

Sample Test Suite

```
public static String allTests()
  {
  StringBuffer failures = new StringBuffer();

  if( !Test01() ) failures.append("01 ");
  if( !Test02() ) failures.append("02 ");
  if( !Test03() ) failures.append("03 ");

  return failures.toString();
  }
```

Sample Main

```
public static void main(String arg[])
  {
  String failures = allTests();

  if( failures.equals("") )
    {
    System.out.println("succeeded");
    }
  else
    {
    System.out.println("failed: " + failures);
    }
  }
```

Comments on Test Suites

- The programming of a useful test suite is not completely trivial.
- A suite must be constructed so that:
  - Everything is automated.
  - It is very obvious whether or not any test in the suite failed.
  - That is, the work of making a determination needs to be done in the suite itself, not by the user.

More on Test Suites

- Each test in a suite is independent of the others.
- Ideally, each test tests only one (new) idea.
- One test should not rely on another having been executed (e.g. dependent on state changes)
- The output of the suite should not be simply **binary**; it should inform which individual tests failed, if any, but in a concise readable fashion.

Step 2: Repair the previous program

- Here we need to add method **insert**(Comparable):

```
class PriorityQueue
{
PriorityQueue()
   {
   }

void insert(Comparable in)
   {
   }
}
```

Step 3: Add a new test that fails

```
public static boolean Test02()
  {
  PriorityQueue pqueue = new PriorityQueue();

  try
    {
    Integer valueIn = new Integer(50);
    pqueue.insert(valueIn);

    Comparable valueOut = pqueue.remove();

    return valueIn.equals(valueOut);
    }
  catch(Exception e)
    {
    return false;
    }
```

Step 3 (continued)

```
class PriorityQueue
{
PriorityQueue()
   {
   }

void insert(Comparable in)
   {
   }

Comparable remove()
   {
   return null;
   }
}
```

Step 4: Get the new test to work

```
class PriorityQueue
{
Comparable value;

PriorityQueue()
  {
  }

void insert(Comparable in)
  {
  value = in;
  }

Comparable remove()
  {
  return value;
  }
}
```

Step 5: Create another test

```
public static boolean Test03()
  {
  PriorityQueue pqueue = new PriorityQueue();

  try
    {
    Integer valueIn[] = new Integer[100];
    Comparable valueOut[] = new Comparable[100];

    valueIn[0] = new Integer(50);
    valueIn[1] = new Integer(60);

    pqueue.insert(valueIn[0]);
    pqueue.insert(valueIn[1]);

    valueOut[0] = pqueue.remove();
    valueOut[1] = pqueue.remove();

    return valueIn[0].equals(valueOut[0]
        && valueIn[1].equals(valueOut[1]);
    }
```

Step 6: Make that test work

```
class PriorityQueue
{
int capacity = 10;

int count = 0;

Comparable value[] = new Comparable[capacity];

PriorityQueue()
  {
  }
```

Step 6 continued

```
void insert(Comparable in)
  {
  for( int i = 0; i < count; i++ )
    {
    if( in.compareTo(value[i]) <= 0 )
      {
      for( int j = count-1; j >= i; j-- )
        {
        value[j+1] = value[j];
        }
      value[i] = in;
      count++;
      return;
      }
    }

  value[count++] = in;
  }
```

Step 6 continued

```
Comparable remove()
  {
  Comparable result = value[0];
  int count1 = count-1;
  for( int i = 0; i < count1; i++ )
    {
    value[i] = value[i+1]; // slide values up
    }
  count = count1;
  diagnose();
  return result;
  }
```

Achieving Closure

- At this point, we have a chance at having the methods working, as long as the queue doesn't overflow.
- We should then add tests that address this issue.
- Also, we need to code and test the isEmpty() method.
- When we think we are done, we should still add more stringent tests to make sure, including ones that interleave insertion and removal.

Critique of TDD?

Some Issues

- Most programs are not just 1-level. Accordingly, test suites should be organized in a hierarchy along the lines of package structure.
- Some say that tests should not be given a sequential numbering (test01, test02, …). Why or why not?
- How much diagnostic information should be passed along in the case of failing tests?