Unit Testing

Steven Zeil

July 22, 2013

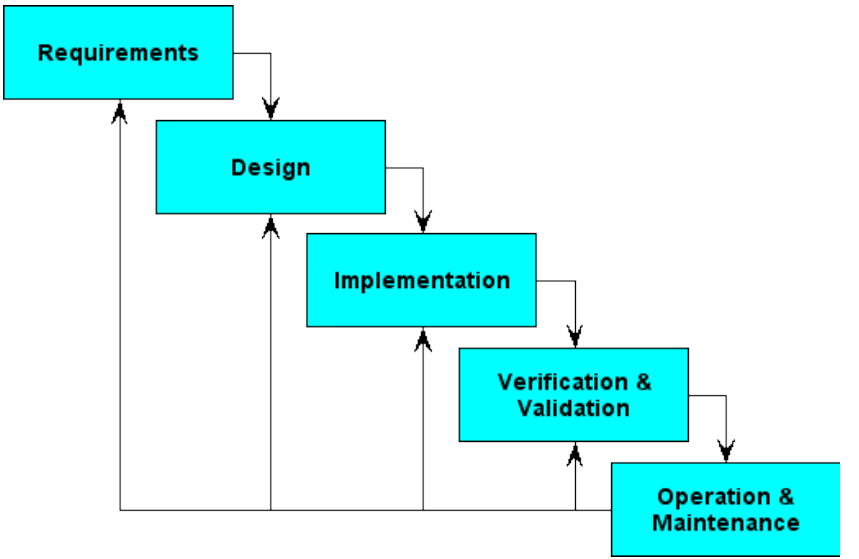
Contents

1 Types of Testing

Testing Stages

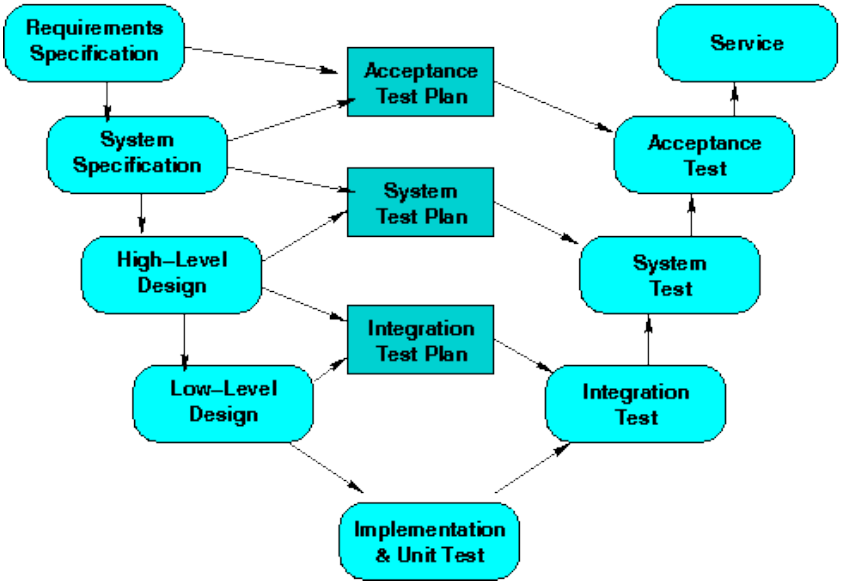
V&V is often portrayed as a single phase of water-fall
Actually it's a full-life-cycle activity

- Requirements must be validated
- Designs may be validated & verified
- Implementation is tested
- final system is tested
- maintenence changes are tested



Testing through the lifecycle

Testing is only one form of V&V, but even it is broken into stages



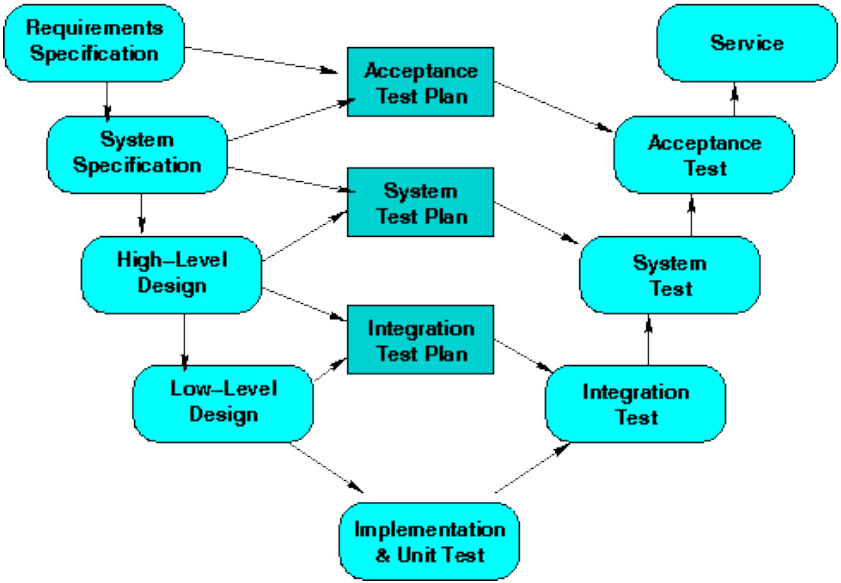
Testing by Stage

- Unit Test: Tests of individual subroutines and modules,
- Integration Test: Tests of "subtrees" of the total project hierarchy chart (groups of subroutines calling each other).
- System Test: Test of the entire system,
- Regression Test: Unit/Integration/System tests that are repeated after a change has been made to the code.
- Acceptance Test: A test conducted by the customers or their representatives to decide whether to purchase/accept a developed system.

Goals of Testing

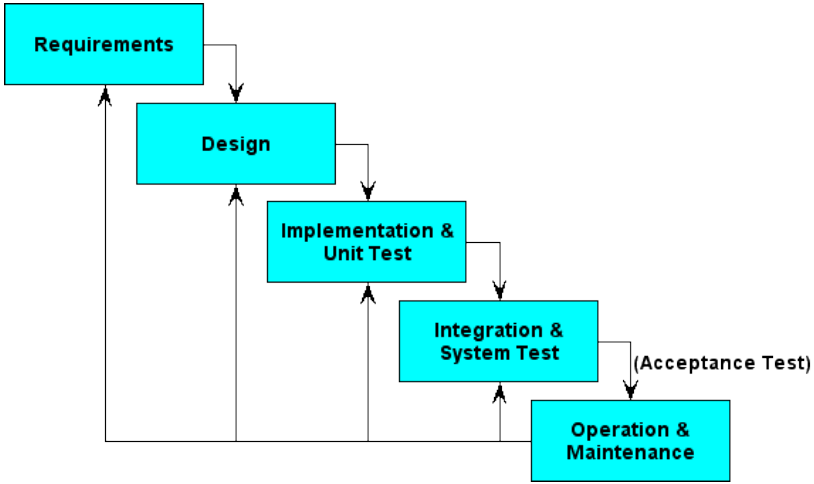
Different stages of testing have different overall goals:

- Unit Test: does a piece of the program work?
- Integration Test: does a large chunk of the program work?
- System Test: does the program work?
- Regression Test: has the program changed?
- Acceptance Test: should we pay for it?



A Testing-Aware Waterfall

We can modify the waterfall model to incorporate this broader view of testing:



2 Unit Testing

Unit testing: Testing Individual Modules

By testing modules in isolation from the rest of the system

- Easier to design and run extensive tests
- Much easier to debug any failures
- Errors caught much earlier

Main challenge is *how* to test in isolation.

2.1 Scaffolding

Scaffolding

To do Unit tests, we have to provide replacements for parts of the program that we will omit from the test.

- *Scaffolding* is any code that we write, not as part of the application, but simply to support the process of Unit and Integration testing.
- Scaffolding comes in two forms
  - Drivers
  - Stubs

. . . . . .

2.1.1 Drivers

Drivers

A *driver* is test scaffolding that calls the module being tested.

- Often just a simple main program that reads values, uses them to construct ADT values, apply ADT operations and print the results

. . . . . .

Example: A Test Driver for Time

- The Time class

```
#ifndef TIMES_H
#define TIMES_H

#include <iostream>
```


```
/**
 * Times in this program are represented by three integers: H, M, & S, representing
 * the hours, minutes, and seconds, respecitvely.
 */


struct Time {
  // Create time objects
  Time(); // midnight
  Time (int h, int m, int s);

  // Access to attributes
  int getHours() const;
  int getMinutes() const;
  int getSeconds() const;

  // Calculations with time
  void add (Time delta);
  Time difference (Time fromTime);

  /**
   * Read a time (in the format HH:MM:SS) after skipping any
   * prior whitepace.
   */
  void read (std::istream& in);


  /**
   * Print a time in the format HH:MM:SS (two digits each)
   */
```



```
  void print (std::ostream& out) const;

  /**
   *  Compare two times. Return true iff time1 is earlier than or equal to
   *  time2
   */
  bool noLaterThan(const Time& time2);


  /**
   *  Compare two times. Return true iff time1 is equal to
   *  time2
   *
   */
  bool equalTo(const Time& time2);

  // From here on is hidden
  int secondsSinceMidnight;
};

#endif // TIMES_H
```

- And a test driver for it:

```
/**
 * Test driver for Time ADT
 */

#include "time.h"

#include <iostream>
```



```
#include <string>

using namespace std;

/**
   Reads multiple lines of input, each with two times.
   Prints these on one line, then applies each of the
   time operations on the pair, printing the results, one
   line per operation.
**/

int main (int argc, char** argv)
{
  // Test default constructor
  Time t1; // test default constructor
  cout << "Default: ";
  t1.print (cout);
  cout << endl;

  while (cin.good())
    {
      t1.read (cin);
      if (!cin)
    break;
      t1.print(cout);
      cout << " ";
      int h, m, s;
      char c;
      cin >> h >> c >> m >> c >> s;
      if (!cin)
    break;
```


```
  Time t2 (h, m, s); // test other constructor
  t2.print (cout);
  cout << endl;

  cout << "Gets: " << t1.getHours() << " "
   << t1.getMinutes() << " "
   << t1.getSeconds() << " "
   << t2.getHours() << " "
   << t2.getMinutes() << " "
   << t2.getSeconds() << endl;

  {
cout << "add: ";
Time t3 = t1;
t3.add(t2);
t3.print (cout);
cout << endl;
  }

  {
cout << "difference: ";
t1.difference(t2).print (cout);
cout << endl;
  }

  {
cout << "noLaterThan: " << t1.noLaterThan(t2)
     << " equalTo " << t1.equalTo(t2) << endl;
  }
}
```


```
  return 0;
}
```

.....

Running the Test Driver

We can run this manually or we could record a set of test inputs in a file:

```
01:00:10 14:00:00
14:00:00 01:00:10
00:00:00 23:59:59
00:00:01 23:59:59
```

and a set of *expected outputs*:

```
Default: 00:00:00
01:00:10 14:00:00
Gets: 1 0 10 14 0 0
add: 15:00:10
difference: 00:00:00
noLaterThan: 1 equalTo 0
14:00:00 01:00:10
Gets: 14 0 0 1 0 10
add: 15:00:10
difference: 00:00:00
noLaterThan: 0 equalTo 0
00:00:00 23:59:59
Gets: 0 0 0 23 59 59
add: 23:59:59
difference: 00:00:00
noLaterThan: 1 equalTo 0
00:00:01 23:59:59
```


```
Gets: 0 0 1 23 59 59
add: 24:00:00
difference: 00:00:00
noLaterThan: 1 equalTo 0
```

and then comparing our output to the expected:

```
./testTime < testTime0.dat > testTime0.out
diff testTime0.expected testTime0.out
```

.....

2.1.2 Stubs

Stubs

A *stub* is test scaffolding written to replace types and functions used *by* the module under test.

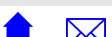
.....

Stub Example

- Suppose we wanted to test *BidCollection*

```
#ifndef BIDCOLLECTION_H
#define BIDCOLLECTION_H

#include "bids.h"
#include <iostream>

class BidCollection {

  int MaxSize;
  int size;
  Bid* elements; // array of bids
```


```
public:

  /**
   * Create a collection capable of holding the indicated number of bids
   */
  BidCollection (int MaxBids = 1000);

  ~BidCollection ();



  // Access to attributes
  int getMaxSize() const {return MaxSize;}

  int getSize() const {return size;}



  // Access to individual elements

  const Bid& get(int index) const {return elements[index];}



  // Collection operations


  void addInTimeOrder (const Bid& value);
  //  Adds this bid into a position such that
  //   all bids are ordered by the time the bid was placed
```

```
  //Pre: getSize() < getMaxSize()



  void remove (int index);
  // Remove the bid at the indicated position
  //Pre: 0 <= index < getSize()


  /**
   * Read all bids from the indicated file
   */
  void readBids (std::string fileName);

  // Print the collection
  void print (std::ostream& out) const;
};


#endif
```

but had not yet implemented `Bid`

- Could try to write a simple stub for `Bid`:

```
typedef std::string Bid;
```

  - Unfortunately, this would not compile because of the calls to `Bid` member functions in addInTimeOrder.

A Bid Stub

We would need something a little elaborate

```
#ifndef BIDS_H
#define BIDS_H

#include <iostream>
#include <string>

#include "time.h"

//
//  Stub for Bids class
//


class Bid {

  std::string data;

public:

  Bid (std::string bidder, double amt,
       std::string item, Time placedAt)
    : data (bidder + " " + item)
  {}

  Bid() : data("default") {}

  // Access to attribute
  std::string getBidder() const {return data;}
```


```
  double getAmount() const {return 1.0;}
  std::string getItem() const {return data;}
  Time  getTimePlacedAt() const {return data;}

  // Print a bid
  void print (std::ostream& out) const
  {out << data;}
};




#endif
```

- That would be enough to let us test that BidCollection worked,
  - in particular that it kept elements in sorted order, provided access to them, and could be printed.

. . . . . .

3 Integration Testing

Integration Testing

Integration testing is testing that combines several modules, but still falls short of exercising the entire program all at once. Integration testing usually combines a small number of modules that call upon one another.

Integration testing can be conducted

- bottom-up (start by unit-testing the modules that dont'call anything else, then add the modules that call those starting modules and thest the combination, then add the modules that call those, and so on until you are ready to test `main()`.)
- or *top-down* (start by unit-testing `main()` with stubs for everything it calls, then replace those stubs by the real code, but leaving in stubs for anything called from the replacement code, then replacng those stubs, and so on, until you have assembled and tested the entire program).

- If conducted *bottom-up*, relieves the need for stubs
- Drivers are still needed

. . . . . .