123

Chapter 4

Software Testing

124

Software Life Cycle

(I. Sommerville: Software Engineering, Add.Wes.Pub. 4th ed. 1992)

Development

In most cases, 100% of development efforts are distributed as follows:

Specifications / Design	30% - 40%
Implementation	15% - 30%
Testing	25% - 50%

Maintenance (corrections, modifications, enhancements)

Usually, 2x - 4x of development expenses

Maintenance consumes 65% - 80% of total expenses

There was a case of 130x

Copyright 1995 - 2003
C. Gotsman & Y.M. Kimchi
Computer Science Dept. Technion

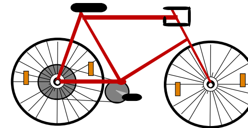
For personal use only

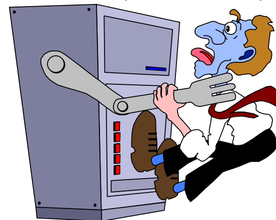
125

Software Requirements

A (software) product should be

- Correct
  - Meets requirements.
- Useful
  - Meets customer expectations
- Robust
  - Resistant to user/environmental errors (fault tolerant)
  - Easily modified/enhanced
- Friendly
  - "Easy" to learn and use
  - Human engineering
- Efficient



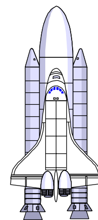


126

Testing

Software testing will check all the requirements

including the "easily modified" **one**



Copyright 1995 - 2003
C. Gotsman & Y.M. Kimchi
Computer Science Dept. Technion

For personal use only

127

Testing Categories

- Black box white box
- Static dynamic
- Top-down bottom-up

128

Black Box Testing
(Behavioral testing)

- Testing *Input-Output* relationships only (including elapsed time)
  - Pros
    - » This is what the product is about.
    - » Implementation independent.
    - » Superficially: Easy to decide what to check.
  - Cons
    - » For complicated products it is hard to identify erroneous output.
    - » Hard to test the wide spectrum of the product functionality.
    - » Practically: Choosing input with high probability of ***error detection*** is very difficult.

Copyright 1995 - 2003
C. Gotsman & Y.M. Kimchi
Computer Science Dept. Technion

For personal use only

Black Box Testing Example

In file: "sort_test.c"

```
#include <stdio.h>
#include "sort.h"   /* For sort() and PrintVec() */
#define SIZE 100000 /* Max size of array to sort */
#define MAX_L 80
int main(void)
{ char line[MAX_L];
  int vec[SIZE], size = 0;

  printf("Enter up to %d int's (one in a line)\n",SIZE);
  while (size < SIZE && fgets(line,MAX_L,stdin) != NULL)
    vec[size++] = atoi(line);    /* Read and set vec[] */
  sort(vec,size);  /* 'size' is now actual array size */
  printf("\n  Sorted vec[] is:\n");
  PrintVec(vec,size);
  sort(vec,size); PrintVec(vec,size); /* Idempotent */
  return 0;
}
```


White Box Testing
(Operational Testing)

Testing *how* **input** *becomes* **output** (including algorithms)

- **Pros**
  - » Easier to detect the weaknesses of the product.
  - » The only way to check all **execution paths** (unreachable code, erroneous branches and loops, etc.)
- **Cons**
  - » Implementation weaknesses are not necessarily those of the product.
  - » Too many trees may hide the forest.

- **Gray Box:** Both styles of testing

If you invest enough in test planning you may arrive at a correct solution

Copyright 1995 - 2003
C. Gotsman & Y.M. Kimchi
Computer Science Dept. Technion

For personal use only

131

White Box Testing Example
In file: "sort.c"

```
void            /* A recursive array sorting function */
sort(int* arr, int size)
{ int *low = arr,      lsz = size/2,    i = 0;
  int  *up = arr+lsz, usz = size-lsz,  j = 0;
  int *tmp, k = 0;

  if (size < 2) return;   /* Boundary conditions */

        /* Print given array before sorting (for testing)*/
      printf("\nlow is: ");    PrintVec(low,lsz);
      printf("\nup  is: ");    PrintVec(up,usz);

  sort(low,lsz);
  sort(up,usz);
```

132

White Box Testing Example (Cont.)
In file: "sort.c"

```
        /* Print given array  after sorting (for testing)*/
      printf("\nsorted low is: "); PrintVec(low,lsz);
      printf("\nsorted up  is: "); PrintVec(up,usz);

       /* Merge the sorted two halves */
  tmp = CHECK((int *) malloc(lsz * sizeof(int)));
  while (i<lsz) tmp[k++] = low[i++]; /* Save low[] */
  for (i=k=0; i<lsz && j<usz;)       /* Merge */
    arr[k++] = (tmp[i] < up[j]) ? tmp[i++] : up[j++];

      /* if i==lsz, then up[] is set o.k. in vec[]*/
  while (i<lsz) arr[k++] = tmp[i++];
  free(tmp);
  return;
}
```

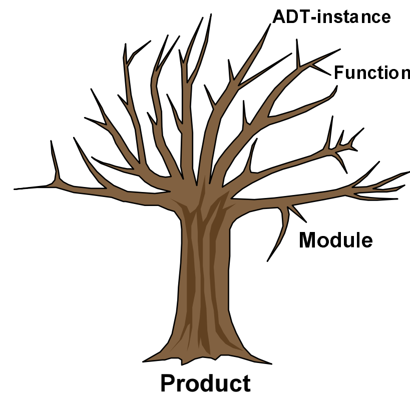
Copyright 1995 - 2003
C. Gotsman & Y.M. Kimchi
Computer Science Dept. Technion

For personal use only

133

Hierarchical Testing

- Low-level
  - Functionality testing
    - » Basic functions
    - » Low-level ADTs
- Intermediate-level
  - Testing functionality and interactions
    - » Functions and ADTs of a certain level
    - » Modules
- High-level
  - Interactions among modules and their overall functionality
    - » Product interface
    - » Visible capabilities



- **Acceptance testing**
  - **Behavioural testing**

134

Top-Down Testing

- **From upper-level modules downward**
  - » **Stubs** are used for lower level functionalities.
    Example:

```
Set SetIntersection(Set a, Set b)
{ return FIXED_SET;  /* some constant value */ }
```

- Pros
  - » **Concurrent** coding and testing
  - » **Early** detection of **specification** / **design** errors
  - » **Early** prototype

Copyright 1995 - 2003
C. Gotsman & Y.M. Kimchi
Computer Science Dept. Technion

For personal use only

135

Top-Down Testing (Cont.)

– **Cons**

» **Stubs** cannot simulate some types of functionality
E.g: If we **really** need the intersection of two sets (cf. prev. example), then FIXED_SET is not useful. We may have to wait until intersection() is ready. However, we may write a simpler implementation of the intended functionality.

» **Output** is required from levels that are not meant to supply it

```
void  interface_control()
{  ...
  while (comm = read_command_line())
    switch(comm) {
      case READ:  ...
          ...
      case EXIT:  ...
      default: error(comm);
    }
  return;
}
```

136

Bottom-Up Testing

– **From a single basic function upward**

» **Test-drivers** are used to "simulate" higher level functionalities, e.g.: **"set_app.c"** is a substitute for **"graph.c"**.

– Pros and Cons are switched from "top down".

A careful combination of "top down" and "bottom up" strategies will benefit from the advantages of both

Copyright 1995 - 2003
C. Gotsman & Y.M. Kimchi
Computer Science Dept. Technion

For personal use only

137

Static and Dynamic Testing

- **Static testing** (white box only)
  - Code analyzers (e.g., tools like lint)
  - Inspection (code review)
  - Proofs (by tools, or by mathematical arguments)

- **Dynamic testing** (black or white box)
  - Predefined tests
    - » The only way for regression testing (comparing an old version against a new one)
    - » Testing the product under extreme conditions
  - Random tests
    - » When quantity supersedes quality.
  - "real life" tests
    - » The only way for MMI testing.

138

Static Testing

- **Code analysis**
  - Unreachable code
  - Undeclared objects
  - Objects declared and never used
  - Parameters type/number mismatch
  - Variable used before initialization
  - Variable is assigned to twice, without using the first value
    - » Identifying such cases is very hard (e.g, when pointers are used) .
  - Function results not used
  - Possible array bound violations
  - Misuse of pointers

- **Code inspection** — A forgotten art
  - **Self** - The default choice. Subtle errors and micro-flaws may be overlooked.
  - **Code review** - Very efficient at the beginning and toward the end of coding

Copyright 1995 - 2003
C. Gotsman & Y.M. Kimchi
Computer Science Dept. Technion

For personal use only

139

Code Analysis
Using "lint"

[15] cat dumb.c

```
#include <stdio.h>
main()
{   int a = sqrt(4), b = sqrt(4.);
    printf("sqrt(4) = %d , sqrt(4.) = %d \n", a, b);
}
```

[16] cc dumb.c -lm
[17] gcc dumb.c -lm
dumb.c: In function `main':
dumb.c:6: warning: type mismatch in implicit declaration for built-in function `sqrt'
[18] lint dumb.c
dumb.c(5): warning: main() returns random value to invocation environment
sqrt, arg. 1 used inconsistently dumb.c(3) :: dumb.c(3)
sqrt used(dumb.c(3)), but not defined
printf returns value which is always ignored
[19]

140

Proving Correctness

– **Lack of formal semantics** for most languages prevents complete proofs

– **Automatic provers** are still too weak for today's software complexity

– **Chances for errors increase with length of text**
  » *Write* **short code** (e.g, divide into more functions).
  » *Provide* **short proofs** for correctness (even if they are **informal**).

– **Partial proofs**
  » The code is **logically consistent** with the specifications.
  » There are no **infinite loops** (not trivial at all).

Copyright 1995 - 2003
C. Gotsman & Y.M. Kimchi
Computer Science Dept. Technion

For personal use only

141

Proving Correctness (Cont.)

– Proof techniques

» *Prove* pre-conditions and post-conditions for functions.

» *Divide* the code into n **segments** at points A_0, A_1, ..., A_n.
Define properties P_i, that are expected to be TRUE at A_i, respectively.
Prove that (P_i => P_{i+1}) while executing the code-segment [A_i, A_{i+1}].

» *Define* and *prove* invariants.
E.g: properties of the data that are preserved during the execution of a loop,
properties of the data that are preserved during its lifetime.

142

Proving Correctness

Example: in "graph.c"

```
static Result
depth_first_traversal(Graph g, Vertex v, Set component)
{ Result  r = SetAdd(component, v);
  Set neigh = GraphNeighbors(g, LABEL(v));
  if (neigh == NULL)  return FAILURE;
  if (r != FAILURE) {
    Vertex vert;
    SET_FOREACH(vert, neigh) {
      if (!SetIsIn(component, vert))
        if ((r = depth_first_traversal(g, vert, component))
               == FAILURE)   break;
    }
  }
  SetDestroy(neigh);
  return r;
}
```

Copyright 1995 - 2003
C. Gotsman & Y.M. Kimchi
Computer Science Dept. Technion

For personal use only

143

Proving Correctness (Cont.)

Example: **in “graph.c”**

- **Assumptions**
  - The MACRO `SET_FOREACH()` is *correct*
  - The functions `SetAdd()`, `GraphNeighbours()`, `SetIsIn()` are *correct.*
    - » **Pay special attention to cases like** “`v` *is not a vertex of the graph* `g`”
  - `component` is *empty* before first call to `depth_first_traversal()`
  - The graph `g` is *valid*
- *FALSE* **Claim**
  - `depth_first_traversal()` is called once for each vertex in `g`
- **Correct Claim**
  - `depth_first_traversal()` is called exactly once for each vertex in the connected component of `v` in `g`.
  - **Loop invariant** example: `component` contains only vertices connected to `v`.
  - When returning to original caller, `component` contains a vertex iff it is in the connected component of `v`.

144

Proving Correctness (Cont.)

Example: **in “graph.c”**

- **Preconditions**
  - The graph `g` is *valid*
  - `v` is a vertex in `g`
  - `component` is a set of vertices
- **Post-Conditions**
  - `component` contains all vertices in `g` that are connected to `v`
- **A weaker Post-Condition**
  - `component` contains all vertices in `g` that are neighbours of `v`
- **A stronger Post-Condition**
  - For every `v` in `component`, `component` contains every vertex `u` such that `u` is connected to `v`
- **oops… Post-Conditions is erroneous:**
  - Vertices that are connected to `v` via elements in `component` are not added

Copyright 1995 - 2003
C. Gotsman & Y.M. Kimchi
Computer Science Dept. Technion

For personal use only

145

Proving Correctness (Cont.)

Example: **in "graph.c"**

- Post-Conditions
  - All vertices in `g` that are connected to `v` via vertices not in `component` were added to `component`
- **A weaker Post-Condition**
  - All neighbours of `v` that are not in `component` were added to `component`
- **A stronger Post-Condition**
  - For every **new** `v` in `component`, `component` contains a **new** vertex `u` **iff** `u` is connected to `v` **via vertices that are not in old** `component`
- **An Initial Condition**
  - The set `component` is empty
- **The Final Post-Condition**
  - The set `component` contains **all** and **only** vertices that are connected to the initial `v`

146

Dynamic Testing

- **Predefined Testing** (black-box or white-box)
  Output is compared to *expected* output
  - **Preprocessor controlled code** (white-box only)
    The only way for *digging* **into the** *heart* **of the code**
    » Testing code is controlled by **MACRO** defined **identifiers**.
    » Code usually **outputs** the **status** of some **objects**.
    » Requires **modification** whenever the code is modified.
    » Requires **separate compilation** of the part to be tested
  - **Test drivers** (black-box or white-box)
    The only way for *implementation-independent* **predefined testing**
    » Testing file is at the same hierarchical level of the code that it tests.
    » When white-box testing is used, test-driver is modified when the code is.

> ***Modifying white-box tests when code is modified requires strict discipline and managerial priority***

Copyright 1995 - 2003
C. Gotsman & Y.M. Kimchi
Computer Science Dept. Technion

For personal use only

147

Preprocessor Controlled Dynamic Test

Example

```
  ...   /* Defining DBG bit flags */
#define  DBG_GRAPH  (0x4)
  ...
#define  DBG_LIST   (0x20)
#define  DBG_MODE  (DBG_GRAPH | DBG_LIST) /* 0 is NO_DBG */

#if DBG_MODE
void
Print_dbg_data(void* obj, int type)
{   ...
}
#endif
   ...
#if DBG_MODE & DBG_LIST
   Print_dbg_data(my_list, LIST);
#endif
```

148

Test-driver Controlled Dynamic Test

Example: **file "graph_test.c"**

```
#include "graph.h"

int main(void)
{   ...
  test_graph = GraphCreate();
  neighbr = GraphNeighbours(test_graph, "testV1");
  SetPrint(neighbr);
  GraphAddVertex(test_graph, "testV1");
  neighbr = GraphNeighbours(test_graph, "testV1");
  SetPrint(neighbr);
    ...
}
```

Copyright 1995 - 2003
C. Gotsman & Y.M. Kimchi
Computer Science Dept. Technion

For personal use only

149

Dynamic Testing (Cont.)

- **Random Testing**
  - When too **many different** possibilities, with equal importance, exist.
  - **Test-drivers** control the randomly generated input.
  - **Biased** random generators are always better, when available.

It is very tempting to use random test-generators where they are least effective

- **Field Testing** **("real life" testing)**
  - **Hook** a user of the product (a person, or a higher level product), and use him as a guinea pig.

Field testing tests only main scenarios, unless specifically prepared otherwise

150

Random Testing

Example: **file "set_rndtest.c"**

```
#include "set.h"

Set Create_rnd_set(void);

int main(void)
{   ...
  set0 = Create_rnd_set();
  set1 = Create_rnd_set();
  set2 = SetIntersect(set0, set1); /* usually != EMPTY */
  set3 = SetUnion(set0, set1);
  SetPrint(set0); SetPrint(set1);
  SetPrint(set2); SetPrint(set3);
    ...
}
```

Copyright 1995 - 2003
C. Gotsman & Y.M. Kimchi
Computer Science Dept. Technion

For personal use only

151

Special Testing Methods

- **Thread Testing**

  When a product has **too many** possible **paths** of execution (e.g, a *real-time* product that depends on external signals and the *order* they arrive).
  The test-driver should concentrate on special execution paths:
  Those that are more **frequent**, or more **sensitive** / **dangerous**.

- **Stress Testing**

  A product that will work under heavy load (e.g, on-line banking system) should be tested under increasing load - much heavier than expected.

- **Error Implantation**

  For measuring the effectiveness of a test set, errors may be introduced into the software, in order to check which of them are detected by testing.
  Though many assumptions are required, if quantitative measurements are expected, it still may serve as an "educated guess" about testing quality.

152

Software Quality

- **Software Quality Assurance** (SQA)

  Usually done by a separate, quality dedicated, team

- **Aspects of Software Quality Control**

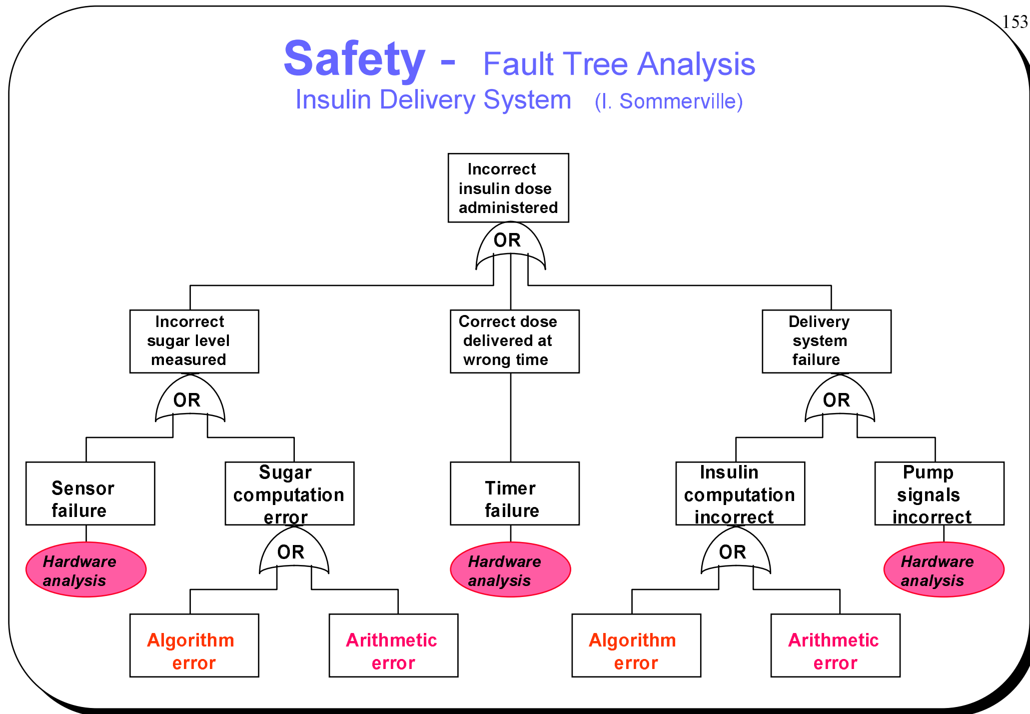
  Safety: The probability of failure in the system (i.e, software + hardware + user)

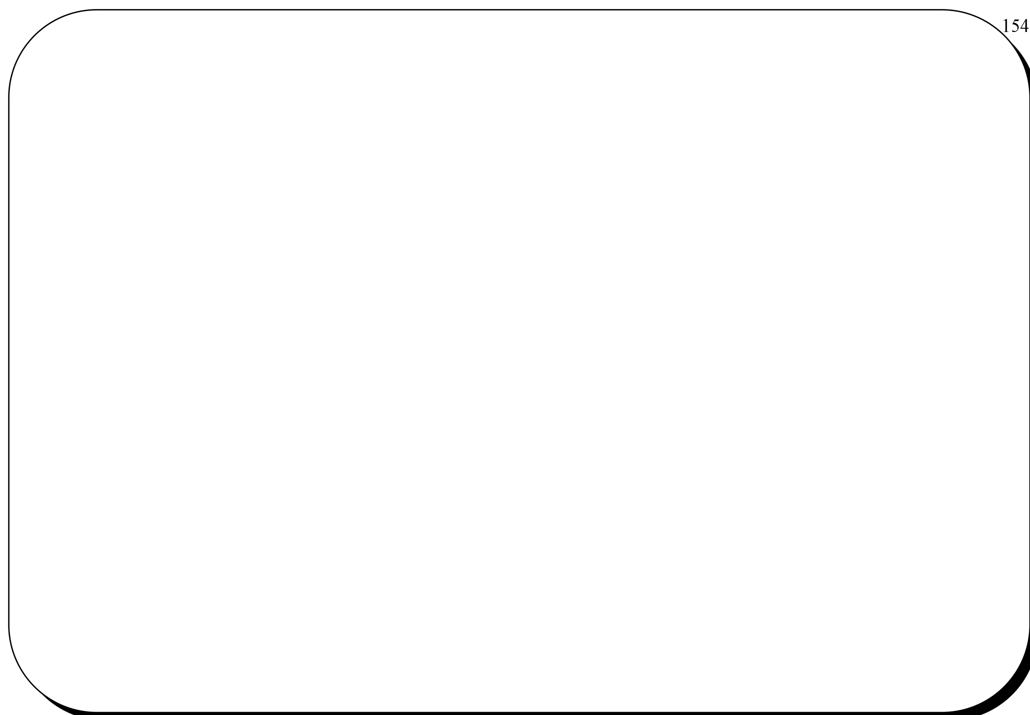
  Reliability: The probability of software error during a given period of time

  Usability: Is it easy to use ? Does it meet user expectations ?

Copyright 1995 - 2003
C. Gotsman & Y.M. Kimchi
Computer Science Dept. Technion

For personal use only





Copyright 1995 - 2003
C. Gotsman & Y.M. Kimchi
Computer Science Dept. Technion

For personal use only