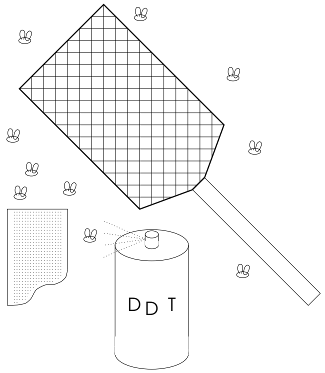
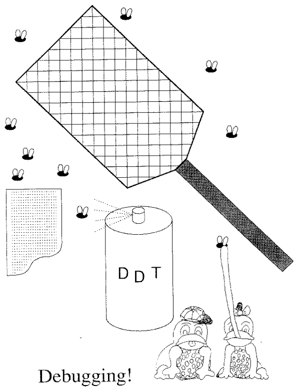
Debugging and Testing

Daniel M. Berry



Debugging!



Debugging!

Hebrew Word for Bug

Everyone in Israel calls a bug באג. The verb “to debug” is לדבג, causing the gerund “debugging” to be דיבוג; it works out very nicely!

The official word for bug is תקר which is also the official word for a puncture in a tire. No one uses תקר for either, using באג for “bug” and פנצ׳ר for “puncture”. Officially then, debugging is תיקון תקר and so is reparing a flat tire!

Mark Weiser

Mark Weiser from Xerox Parc has a entry on his web page:

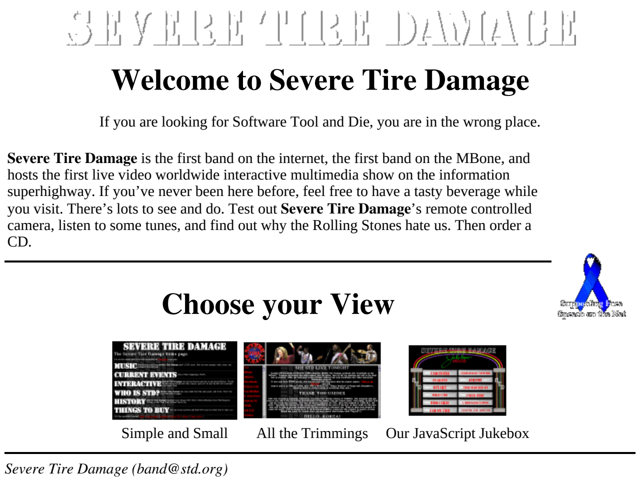
"I am the drummer for Severe Tire Damage , first live band on the internet."

and that band's own description is:

Severe Tire Damage is the first band on the internet, the first band on the MBone, and hosts the first live video worldwide interactive

multimedia show on the information superhighway. If you've never been here before, feel free to have a tasty beverage while you visit. There's lots to see and do. Test out Severe Tire Damage's remote controlled camera, listen to some tunes, and find out why the Rolling Stones hate us.

Also recall (or learn for the first time) that Mark's Ph.D. work was in debugging using program slices.



So I sent the following message to him

Dear Mark..

The name of your rock group, Severe Tire Damage, is very interesting from the computer science/software engineering point of view.. I am not sure that you are aware of it.

Perhaps you know that the Hebrew word for bug that everyone uses is "boog" and the gerund for debugging is "diboog" which makes it sound like the verb is in binyan pi'el

and thus the infinitive is l'dabeg. HOWEVER.. the OFFICIAL word promulgated by the official Hebrew Language Institute.. is "teker" and debugging is "tikun teker" (repairing the teker).. Well.. so what does this have to do with your rock band?????

Well the OTHER meaning of the word "teker" is a hole in a tire (a puncture), and of course repairing a puncture is "tikun teker" again..

Of course NO one uses teker for that either, the common word for a puncture being "panchure" :-)..

But one day, I saw a bunch of fellas sitting at a table in the same restaurant in which I was eating, and they were all wearing T-shirts saying "tikun teker". I decided that I just HAD to have that T-shirt to wear when I was lecturing about debugging here in the Technion or anywhere else in Israel. So I went an bought the shirt off the back of the biggest one there.

The morning I was giving my lecture on testing and debugging, I woke up, put the T-shirt on, and went down to drive to work and lo and behold.. I had a flat tire!! :-).. a definite case of severe tire damage. :-)

I reported to the class first that the official word for bug was "teker" and explained that I had a "boog" in my tire this morning and that I diboogged it. Also, someone had thrown an apple core away near my car, and there was a swarm of bugs of another kind near by... I had the class rolling on the floor.

And now then, I see your band.. hm.. I am going to the web page... maybe I will put a copy of the page as the first slide after the title.. to be shown while wearing that T-shirt.

Nu?

I will see you in Boston in two weeks..

Dan

His reply, one day later..

Thanks! I enjoyed your message.

-mark

Debugging

Two kinds of bugs:

- **inappropriate specifications implemented correctly**
- **failure to implement specifications correctly**

The first kind comes into program from the beginning.

The second kind is introduced to program during development.

Neither suddenly appears in an otherwise healthy program,
say as a result of contagion from other buggy programs.

Response

For first kind, redesign from new requirements.

For second kind, redevelop from point at which bug is introduced.

An old proverb —

An ounce of prevention is worth a pound of cure.

An newer proverb —

An decigram of prevention is worth a kilogram of cure.

Testing

More detail later...

But bottom line:

Testing is used for the purpose of exposing the bugs to be corrected.

Then the source of the bugs must be found and corrected.

Debugging

The process of testing, exposing bugs, locating their source and correcting them is called “debugging”.

Techniques -1

Adding additional output requests:

- at branch-in and -out points
- at specific assignments

Techniques -2

One can even leave these requests in permanently, but surround them by a conditional that executes them only if flag is true.

Flag can be run-time or compile-time.

Tracers

- **statement counts**
- **flow of control**
- **trace of assignments**
- **instrumented assertions**
- **snapshots**

Postmortems

- **statement counts**
- **flow of control up to bombout**
- **trace of statements up to bombout**
- **snapshots**

Interactive Debuggers -1

- **tracing particular statements**
- **tracing assignments to particular variables**
- **setting break points**
- **single stepping**
- **interrogating variables**
- **printing snapshots**

Interactive Debuggers -2

- **setting variables and**
- **continuing**
- **attaching to specific processes**

Testing -1

According to Boehm (circa 1975),

Programmers in large software projects typically spend their time as follows:

45-50%	**program checkout**
33%	**program design**
20%	**coding**

Testing -2

Yet, in spite of this checkout expense, delivered “verified” and “validated” code is still notoriously unreliable.

We have attempted to make the design and coding processes more systematic.

Now it is time to look at testing process.

Testing -3

We often hear that testing is confirming that program works.

Or...

Testing is demonstrating that errors are *not* present.

Testing -4

Nonsense! wrong! bubbe meises!

Already know that:

Program testing can be used to show the presence of errors but never their absence.

— E.W. Dijkstra

Testing -5

Therefore, the proper definition of testing testing is executing a program with the intention of finding errors

— G. Myers

Psychology of Testing -1

A program is its programmer's *baby*!

Thus trying to find errors in one's own program is like trying to find defects in one's own baby.

Therefore, it is best to have someone other than the programmer doing the testing.

Psychology of Testing -2

Tester must be highly skilled, experienced professional.

It helps if he or she possesses a diabolical mind.

"heh ... heh ... heh!"

— Count Dracula

Psychology of Testing -3

It is well known that what is achieved in any endeavor depends a lot on what are the goals.

Myers says:

If your goal is to show absence of errors, you will not discover many.

If your goal is to show presence of errors, you will discover large percentage of them.

Psychology of Testing -4

If you are trying to show the program correct, your subconscious will manufacture safe test cases.

Therefore, the tester should be someone other than the programmer, who just *loves* bugs.

Definitions -1

Testing — ...

Proof — against assertions

Verification — testing against simulated environment

Validation — testing against real environment

Definitions -2

Certification — controlled testing against predefined standard

Module testing — in isolation, from specifications

Integration testing — verification of interfaces

System testing — verification against specifications of whole system

Definitions -3

Acceptance testing — validation against specifications of user for whole system

Installation testing — validation of subsequent installations

Debugging **— what one has to do if testing is successful, i.e., if errors are found, debugging is correction of errors**

Module Testing -1

Continuum of methods to make test cases:

test against specs	test against code
ignore code	ignore specs

Neither extreme is completely satisfactory.

Combinatorics of either extreme is too large.

Module Testing -2

We need the intelligence offered by points in middle to help group cases into equivalence classes.

Two test cases are in the same equivalence class if they find the same error.

Try to find a set of test cases with exactly one element of each equivalence class.

Module Testing -3

As you’re building test suite, each new test case added to suite should expose at least one (and as many as possible) previously undetectable error.

It is hoped to get enough test cases to find *all* errors.

Module Testing -4

A certain amount of skill and deviousness is needed to generate test cases.

We show first how to find *things* to test.

In the following, remember that the tester is *not* the programmer.

Finding Things to Test -1

Each such thing must appear in at least, and preferably at most one, test case.

A test case may and should test more than one thing

Generate things from:

Specifications

- **each**
  - **input condition & option**
  - **file & std input**
  - **command line**
- **boundaries of input**
- **boundaries of output**
- **each invalid input**

Code -1

- **each conditional branch in each direction**
- **each path at least once if reasonable:**
  - **for loops, do**
    - **0,**
    - **1,**
    - **representative number, &**
    - **max number (if exists)**

    of times

Code -2

- sensitivities to particular values, e.g.,
  - division by 0,
  - overflow,
  - underflow,
  - subscript boundaries,
  - nil pointers.

Finding Things to Test -2

In any of the above, when you have a large range, use boundary, typical, and sensitive values.

Once you have found things to test, generate test cases, each being the data for one run of the program.

Finding Things to Test -3

As you generate test cases, also generate expected results for each one, from the specifications.

Try to put as many things as possible in one test case, but obviously do not put any cases after a test of handling of erroneous input, especially if the specified response is to halt the program!

Specifics (Hints for Goodies)

Should have:

- documentation
- input options
- command line options
- error testing

Documentation

For each test case:

- **describe what is tested**
- **give input**
- **give output (assumed to be expected)**

Input Options

Thoroughly test all input options described in the specifications, in all combinations,

including that of same input as standard input and as file input, if that is an option.

Command Line Options -1

Test all command line options in all combinations.

Test that saying nothing is equivalent to explicit choice of default settings.

Command Line Options -2

In the above, “in all combinations” must be taken with a grain of salt, as it could lead to combinatorial explosion.

If an analysis of the code shows that some options are truly independent, then the number of combinations can be reduced.

Command Line Options -3

But that's dangerous, as a mistake could be made in determining independence.

Also the bug could be the non-independence!

So verify the independence assumption with some test cases and then assume it.

Error Testing

Test all specified error conditions.

Test all algorithm implied errors.

Try to crash the program.

Regression Testing

When you are doing an enhancement,

you should run *all* old tests,

even ones that are invalid for original purpose; just provide new expected results.

Tools -1

There are tools that help,

especially generating test cases from code,

- **each branch in each direction**
- **each path once**

Tools -2

Also the tools help in managing the test case, i.e., running program against test case and comparing output with expected output & reporting only when there is not a match.

This can be done with shell scripts as well.

Hopelessness of Thoroughness

How many of you have observed this phenomemon?

You’ve designed good thorough test cases and the reviewers agree with you.

You run the exhaustive tests and they fail (to find bug).

You turn in the program.

Hopelessness, cont’d

You’re relaxing over a cup of coffee and bingo, you think of a bug in the program or another test case.

You run to do the test case before the deadline.

How many of you have observed this?

This is the feeling of hopelessness of getting thorough test cases.

Earthquake Testing

Remember the 1989 San Francisco earthquake that was seen alive on TV at 5:01 pm by everyone tuned in for the beginning of the first game of the 1989 World Series baseball championship?

The epicenter was under Scotts Valley, CA just underneath Borland’s world headquarters.

One of the two buildings collapsed.

No one was hurt because everyone had gone home to watch the World Series.

In the collapsed building, most of the computers were destroyed, but ...

the software in the building still worked, when the diskettes were put into other computers!

Surprise!!!

So Borland issued T-shirts that said in front,

“Borland, the epicenter of software development!”

and in back,

“The only software that has been tested to be earthquake proof up to 7.2 on the Richter scale!”

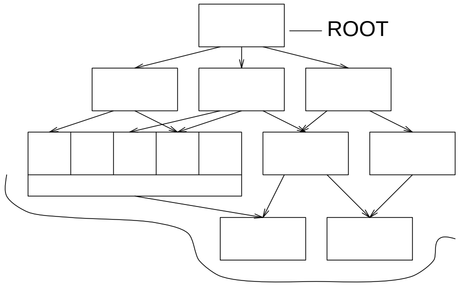
How many of *you* have tested that your software is earthquake proof?

Nu?

Integration Testing

Integration testing is testing how a system consisting of more than one module works together, i.e., testing the interfaces.

Assumed System Structure



Two Major Kinds of Testing

- **big-bang!!!!!!**
- **piece-meal**

(several orders possible)

Preliminary Definitions -1

Before can discuss any method, must define two terms:

For a given module *m*,

A *driver* for *m* is a module that repeatedly calls *m* with test data & maybe even checks results for each test case.

Preliminary Definitions -2

A *stub* for replacing *m* is a skeletal module with interface identical to that of *m*; its body either

- **does nothing**
- **prints out name of *m* maybe with parameter values.**

A stub may return pre-cooked results for pre-cooked inputs for planned test cases, maybe even testing that actual input is same as planned or some such.

Big-Bang Testing

Each module is tested in isolation under a driver and with a stub for each module it invokes.

And then one day ... cross your fingers ... do a big-bang test of the whole bloody system.

Probably over your bloody, dead body!

Advantage

- **Every module thoroughly tested in isolation.**

Disadvantages

- **Driver and stub must be written for each module (except no driver for root and no stubs for leaves).**
- ***No* error isolation — if error occurs, then in which module or interface is it?**
- **Interface testing must wait until all modules are programmed ∴ critical interface & major design problems are found very late into project, after all code has been committed!**

Piece-Meal Testing

There are several specific orders of adding modules one-by-one to an ever growing system until the whole system is obtained.

A module is tested as it is added in the context of whatever of its callers and callees are already in the system and drivers and stubs for callers and callees that are not yet in the system.

Orders

- **bottom-up**
- **top-down**
- **sandwich**
- **modified sandwich**

General Advantages

- **Error isolation — assuming that all previously added modules were thoroughly tested, any errors that crop up should be in the module being added or in its interface.**
- **Avoid making *both* driver and stub for each module; at most one is needed for each module.**

General Disadvantage

- **Some modules (which ones depends on the order of adding modules) are not tested thoroughly in isolation, because do not have driver and stub for every module.**

Bottom-Up

Test leaf modules in isolation with drivers.

Test any non-leaf module *m* in the context of all of its callees (previously tested!) with a driver for *m*.

Advantages

- **Thorough testing of each leaf module.**
- **Abstract types tend to be tested early.**

Disadvantages

- **Must make drivers for every module except root.**
- **Major design flaws and serious interface problems involving highest modules are not caught until late; could cause a major rewrite of everything that has been tested before.**

Top-Down

Only the root is tested in isolation with stubs for its callees.

Test any non-root module *m* by having it replace its stub in its callers and with stubs for its callees.

Advantages

- **Can test during top-down programming.**
- **Major flaws and major interface problems are found early.**
- **No drivers needed, as modules are tested in context of actual callers.**

Disadvantages

- **Stubs can be difficult to write for planned test cases.**
- **Lower modules are not tested thoroughly; tested only in ways used by upper modules and not as fully as possible against specs.**

Sandwich

Root and leaf modules are tested in isolation as in T-D and B-U methods.

Work from root to middle T-D, and work from leaves to middle B-U.

Advantages

- **Root and leaf modules tested thoroughly.**
- **Major design flaws and major interface problems found early.**
- **Most abstract data types tested early.**
- **No stubs and no drivers needed for middle modules.**

Disadvantages

- **Stubs needed for upper modules and drivers needed for lower modules.**
- **Upper-middle modules may not be thoroughly tested.**

Modified Sandwich

Do sandwich.

Also test upper-middle modules in isolation using drivers and actual callees.

Advantages

Those of sandwich +

- **Removal of its second disadvantage.**

Disadvantages

First of sandwich +

- **Extra drivers to write.**

Concluding Remarks

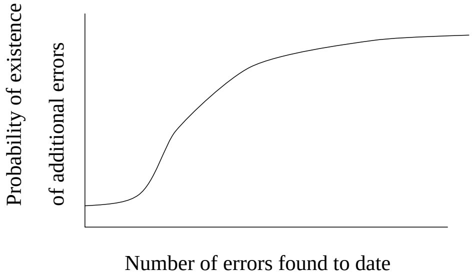
When testing exhibits a bug:

Remember, the program did not suddenly contract the bug.

The bug was required, designed, or programmed in.

Probability of More Errors -1

Myers observes that as the number of detected errors increases in a piece of software, so does the probability of the existence of more, undetected errors.



Probability of More Errors -2

Thus testing, and alas, debugging, is no substitute for programming it right in the first place.

Decay of Corrected Programs -1

Also Belady and Lehman have shown that assuming a non-zero probability that an attempted correction introduces new bugs, any system which is continually fixed eventually decays beyond useability.

Decay of Corrected Programs -2