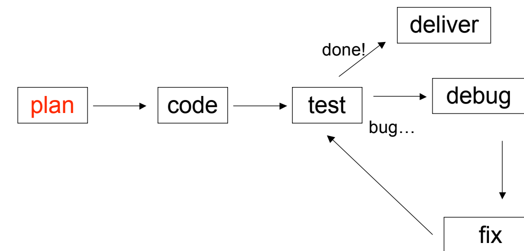
Testing and Debugging

COEN 11

Source: The Practice of Programming, Kernighan and Pike,
Addison Wesley, 1999

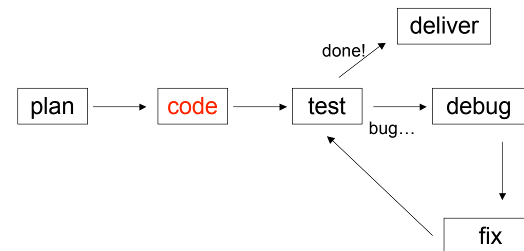
The Development Cycle



Planning

- Given a problem
  - Input
  - Output
- Develop a solution
  - Data Structures
  - Main variables
  - Functions -- task, input, and output

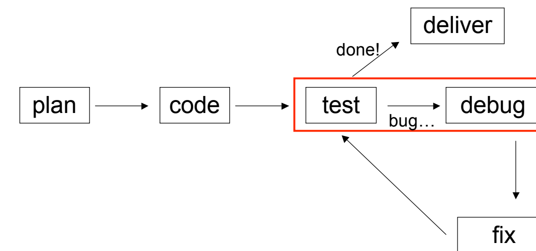
The Development Cycle



Coding

- ✱Writing the code
  - ➔Developing the functions in the plan, according to the plan

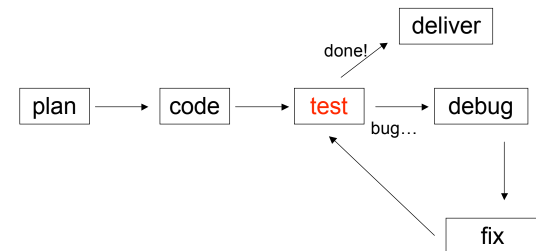
The Development Cycle



Testing and Debugging

- ✱Not the same thing!
  - ➔Testing is a determined, systematic attempt to break a program that you think is working
  - ➔Debugging is what you do when you know that a program is broken
- ✱Dijkstra: "Testing can demonstrate the presence of bugs, but not their absence."

The Development Cycle



Testing

* Testing is a determined, systematic attempt to break a program that you think is working

Testing

* We want to test to find errors rapidly, efficiently, and effectively
  * Think about potential problems as you code
  * Do systematic testing, from easy tests to elaborate ones
  * Automate to help eliminate manual processes and encourage extensive testing

Test as you write the code

* Think systematically about what you are writing as you write it
  * Verify simple properties of the program as it is being constructed

Test as you write the code

* Test code at its boundaries
  * Most bugs occur at boundaries
    * Non-existent or empty input
    * Single input item
    * Nearly full, exactly full, or over-full array
    * Etc...
  * Technique: Boundary condition testing
    * As each small piece of code is written, check right then that the condition branches the right way or that the loop goes through the proper number of times

Test as you write the code

- Test pre- and post-conditions
  - Verify that expected or necessary properties hold before (pre-condition) and after (post-condition) some piece of code executes
  - Making sure that input values are within range is a common example of testing pre-condition
  - Ignoring the problem can lead to catastrophic events

Test as you write the code

- Program defensively
  - Add code to handle "can't happen" cases
    - Situations where it is not logically possible for something to happen but it might anyway
  - This makes sure that a program protects itself against
    - incorrect use
    - illegal data

Test as you write the code

- Check error returns
  - Always check the error returns from library functions and system calls
  - If functions like open, read, or malloc return an error, computation cannot continue normally
  - Even output errors can be serious
    - For example, there may not be enough space in the disk

Test as you write the code

- The effort of testing as you go is minimal and really pays off
  - Thinking about testing as you write a program will lead to better code, because that's when you know best what the code should do

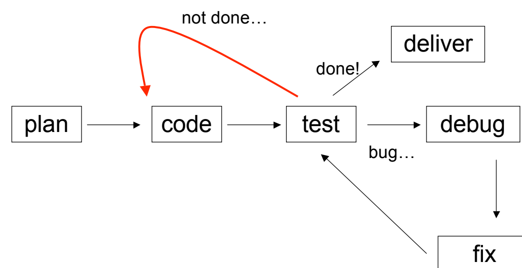
Systematic Testing

- It is important to test your program systematically
  - You need to know at each step what you are testing and what results you expect
  - You need to be orderly so you don't overlook anything
  - You must keep records so you know how much you have done

Systematic Testing

- Test incrementally
  - Testing should go hand-in-hand with program construction
  - Write part of the program, test it, write some more, test it, and some more, test it, and so on
  - If you have two packages that have been written and tested independently, test that they work together when you finally connect them

The Development Cycle



Systematic Testing

- Test simple parts first
  - Tests should focus first on the simplest and most commonly executed features of a program
  - Only when those are working properly should the testing move on
  - Then, at each stage, expose more to testing and build confidence that basic mechanisms are working correctly

Systematic Testing

- Test simple parts first
  - Example: binary search in an array
    - Search an array with no elements
    - Search an array with one element and a trivial value that is
      - <=, =, >=
    - Search an array with two elements and trivial values
      - Check all five possible positions
    - Check behavior with duplicate elements in the array and trivial values
      - <=, =, >=
    - Search an array with three elements as with two elements
    - Search an array with four elements as with two and three

Systematic Testing

- Verify conservation properties
  - Inside a program
    - Counting the elements in a data structure provides a trivial consistency check
    - A linked-list should have the property that every element inserted into it can be retrieved
      - This condition is easy to check with a function that dumps the contents of the list into a file or an array
      - At any time, the number of insertions into a data structure minus the number of deletions must equal the number of elements contained

Systematic Testing

- Measure test coverage
  - Complete coverage is often quite difficult to achieve
    - Need to make sure that every statement of a program has been executed sometime during the sequence of tests
    - Testing cannot be considered completed unless every line of program has been exercised by at least one test
  - It is hard to use normal inputs to force a program to go through particular statements
    - Need to come up with unconventional inputs

Test Automation

- Testing by hand is tedious and unreliable
  - Proper testing involves
    - lots of tests
    - lots of inputs
    - lots of comparisons of outputs
- Testing should therefore be done by programs!
  - Programs don't get tired or careless!

Test Automation

- It is worth taking the time to write a script or trivial program that encapsulates all the tests
  - ➔ A complete test suite can be run by "pushing a single button"
  - ➔ The easier a test suite is to run,
    - the more often you will run it and
    - the less likely you will skip it when time is short

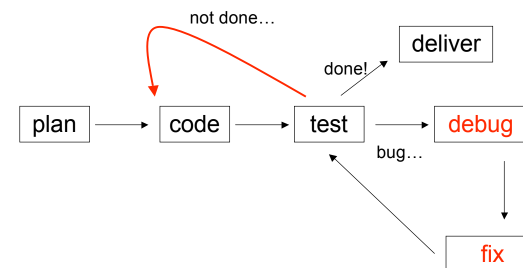
Stress Tests

- Use
  - ➔ High-volume of input
  - ➔ Random inputs
  - ➔ Malicious input
  - ➔ Illegal inputs
- Force errors

Testing Reality

- As SW systems are getting larger and more complex and development schedules are getting shorter
  - ➔ the pressure to ship without adequate testing is increasing

The Development Cycle



Debugging

- Programs usually do not work the first time
  - Dealing with bugs is part of programming

Debugging

- Reality
  - There will always be errors
  - Need to find and fix them
  - Good programmers know that they spend as much time debugging as writing
    - Need to learn from mistakes
    - Every bug you find can teach you how to prevent a similar bug from happening or to recognize it if it does

Debugging

- Debugging is hard
  - Can take long and unpredictable amounts of time
  - Goal is to avoid to do much of it
- Main techniques to help reduce debugging time
  - Good design
  - Good style
  - Boundary condition tests
  - Assertions and sanity checks
  - Defensive programming
  - Well-designed interfaces
  - Limited global data
  - Checking tools

Debugging

- Role of the language
  - A major force in the evolution of programming languages
    - the attempt to prevent bugs through language features
  - Some features make classes of errors less likely
    - Range checking on subscripts
    - Restricted pointers
    - No pointers at all
    - Garbage collection
    - String data types
    - Typed I/O
    - Strong type-checking

Debugging

- Role of the language
  - Some features are prone to error
    - Goto statements
    - Global variables
    - Unrestricted pointers
    - Automatic type conversions
  - Programmers should know the potentially risky bits of their language and take extra care when using them
  - Programmers should enable all compiler checks and pay attention to the warnings

Debugging

- Role of the language
  - No language prevents you from making mistakes!

Debuggers

- Compilers for major languages usually come with sophisticated debuggers
  - Often packaged as part of a development environment that integrates
    - Creation and editing of source code
    - Compilation
    - Debugging

Debuggers

- Debuggers provide several facilities
  - For stepping through a program one statement or function at a time
  - For stopping at particular lines or when specific condition occurs
  - For formatting and displaying the values of variables

Debuggers

- In the right environment and in the hands of an experienced user
  - A good debugger can make debugging effective and efficient, if not painless
- With such powerful tools
  - Why would anyone debug without them?
  - Why even talk about debugging?

Debuggers

- Reasons
  - Good debuggers are not always available
  - Debuggers are system-dependent
    - You may not have access to the familiar debugger from one system when you work on another
  - Some programs are not handled well by debuggers
    - Multi-process or multi-threaded programs
    - Operating systems
    - Distributed systems
  - In this cases
    - You are on your own, with print statements, experience, and ability to reason about code

Debuggers

- Tendency
  - Use debuggers to get a stack trace
  - Stepping through a program not productive
  - Better to
    - think harder and
    - add output statements and self-checking code at critical places
  - Debugging statements stay with the program
    - Debugger sessions are transient

Good Clues, Easy Bugs

- Something goes wrong!
  - Program crashed, or printed nonsense, or seems to be running forever
  - Realistically
    - It is the programmer's fault
  - Fortunately, most bugs are simple and can be found with simple techniques
    - Examine the evidence in the erroneous output and try to infer how it could have been produced
    - If possible, get a stack trace from a debugger
    - Pause to reflect
    - Reason back from the state of the crashed program to determine what could have caused this

Good Clues, Easy Bugs

- Debugging involves <u>backwards reasoning</u>
- Once we understand what happened, or how it happened
  - We will know what to fix and, along the way, likely discover a few other things we hadn't expected

Good Clues, Easy Bugs

- Look for familiar patterns
  - Common bugs have distinctive signatures
    - Using n instead of &n when referring to the address of the variable
    - Mismatching types and conversions in printf and scanf
    - Failing to initialize
      - a local variable
      - memory returned by allocators

Good Clues, Easy Bugs

- Examine the most recent change
  - If you are changing one thing at a time
    - The bug will most likely be either in the new code or has been exposed by it
    - Specially if the bug appear in the new version, but not in the old one
      - You should preserve at least the previous version, which you believe to be correct so that you can compare behaviors
      - You should keep records of changes made and bugs fixed so that you don't have to rediscover this information while trying to fix a bug
  - Source code control systems and history mechanisms are helpful here

Good Clues, Easy Bugs

- Don't make the same mistake twice
  - After you fix a bug, ask whether you may have made the same mistake somewhere else
  - Easy code can have bugs if its familiarity causes us to let down the guard

Good Clues, Easy Bugs

- ✱Debug it now, not later
  - ➔Being in too much of a hurry can hurt in other situations as well
  - ➔Don't ignore a crash when it happens
    - ➢Track it down right away
    - ➢Since it may not happen again until it is too late

Good Clues, Easy Bugs

- ✱Get a stack trace
  - ➔Although debuggers can probe running programs
    - ➢Great to examine the state of a program after death
      - The source line number of the failure, often part of a stack trace
      - Improbable values of arguments
        - » zero pointers
        - » integers that are huge when they should be small
        - » negative numbers that should be positive
        - » character strings that aren't alphabetic

No Clues, Hard Bugs

- ✱If you really have no clue, life gets tougher!

No Clues, Hard Bugs

- ✱Make the bug reproducible
  - ➔Make sure you can make the bug appear on demand
    - ➢Construct input and parameter settings that reliably cause the problem

No Clues, Hard Bugs

- Make the bug reproducible
  - If the bug can't be reproduced every time
    - Try to understand why not
    - Does some set of conditions make it happen more often than others
    - Even if you can't make it happen every time
      - If you can decrease the time spent waiting for it, you will find it faster

No Clues, Hard Bugs

- Display output to localize your search
  - If you don't understand what the program is doing
    - Adding statements to display more information can be the easiest, most cost effective way to find out
    - Put them in to verify your understanding or refine your ideas of what's wrong

No Clues, Hard Bugs

- Display output to localize your search
  - Display specific messages at specific places in the code to check where the execution is going
  - If you are displaying the value of a variable
    - Format it the same way each time
    - For example, pointers in hexadecimal

No Clues, Hard Bugs

- Write self-checking code
  - If more information is needed, you can write your own *check* function to
    - Test a condition
    - Dump relevant variables
    - Abort the program – call abort()
  - Also, add calls to *check* wherever they may be useful in your code
  - After the bug is fixed, don't throw *check* away
    - Leave it in the source, commented out or controlled by a debugging option
    - It can be turned on again when the next difficult problem appears

No Clues, Hard Bugs

- Write self-checking code
  - For harder problems, *check* may evolve to do verification and display of data structures
  - This approach can be generalized to routines that perform ongoing consistency checks of data structures and other information
  - In a program with intricate data structures, it is a good idea to write these *checks* before problems happen
    - They can be turned on when trouble starts

No Clues, Hard Bugs

- Write self-checking code
  - Don't use *check* functions only when debugging
    - Leave them installed during all stage of program development
    - If they are not expensive, it may be wise to leave them always enabled

Last Resorts

- Some frustrating situations
  - Wasting time
    - Chasing bugs that aren't there
      - the test program is the one that is wrong
    - Testing the wrong version of the program
    - Failing to update or recompile before testing

Last Resorts

- Once in a long while, the problem really is somebody else's fault!
  - The compiler
  - A library
  - The operating system
  - The hardware
- But you should never consider that at first
  - Generally, the programmer is the one to blame

Last Resorts

- Advice
  - Careful when calling library functions or system calls
    - They may be macros!

Last Resorts

- Source of erratic behavior
  - Memory leaks
    - The failure to reclaim memory that is no longer in use
  - Forgetting to close files
    - Until the table of open files is full and the program cannot open any more

Non-Reproducible Bugs

- Bugs that won't stand still are the most difficult to deal with
  - Possibility: Variables that were not initialized
- Behavior that changes when debugging code is added
  - Possibility: Memory allocation error – writing outside of allocated memory

Non-Reproducible Bugs

- Crash site that is far away from anything that could be wrong
  - Possibility: overwriting memory by storing into a memory location that isn't used until much later
  - Sometimes this is a dangling pointer problem
    - A pointer to a local variable may be inadvertently returned from a function and used
    - A dynamically allocated value that has been freed is still pointed to and used

Non-Reproducible Bugs

- Crash site that is far away from anything that could be wrong
  - Freeing an allocated item twice may cause trouble later
    - a subsequent call may slip on the mess made earlier

Fixing

- Important
  - Really understand what is going on before changing anything!
  - When fixing bugs, be extra careful not to "break" anything
  - Keep a very detailed log of what you have changed
  - Keep versions
  - Have a test set to run after every change

Software Reality

- Nowadays
  - most programmers do not have the fun of developing a brand new system from the ground up
- Instead
  - they spend much of their time using, maintaining, modifying, and debugging code written by other people