University of Zurich[UZH]

Department of Informatics

Martin Glinz

Software Quality

Chapter 4

Debugging

4.1 Foundations

4.2 The Debugging Process

4.3 Reproducing Errors

4.4 Simplifying and Automating Test Cases

4.5 Techniques for Defect Localization

4.6 Defect Fixing

Terminology

Debugging – The process of finding and correcting a defect that causes an observed error

Defect (fault) – A faulty element in a program or other artifact

Error – A deviation of an observed result from the expected / correct result

- The term bug may denote a defect or an error
- An error may be caused by a combination of multiple defects
- The very same defect may manifest in more than one error
- „Program“ is meant in a comprehensive way: may be a single method or a component, or a complete system

Causes and Effects

- Typically, a defect
  - does not immediately lead to an error that can be observed,
  - but to faulty program states,
  - that propagate
  - and eventually manifest as observable errors
- The main task of debugging is identifying / reconstructing the cause-effect chain from a defect to an observable error

Where defects occur

- Classic: defect is a coding error, caused by a human mistake
- Alternatively:
  - Defects in other artifacts: requirements specification, system architecture, system design, user manual, ...
  - Defects in the data
  - Defects in processes
  - Human mistakes when using or operating a system
- Some defects are not local, but affect a complete system or sub-system

Example: A simple sorting problem [Zeller 2005]

Name: `sample`

Author: Andreas Zeller

Language: C

Call: `./sample` arg_1 arg_2 ... arg_n

Precondition: arg_1 arg_2 ... arg_n are integers, $n \in$ IN

Postcondition: The arguments appear in ascending order on the standard output device

Executing `sample` with test data:

```
$ ./sample -5 0 -9
Output: -9 -5 0   ✔
$ _
```

```
$ ./sample 11 14
Output: 0 11   ✘
$ _
```

Program `sample`: The code

```
/* sample.c -- Sample C program to be debugged */

#include <stdio.h>
#include <stdlib.h>

static void shell_sort(int a[], int size)
{
    int i, j;
    int h = 1;

do {
        h = h * 3 + 1;
    } while (h <= size);
    do {
        h /= 3;
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}
```

Program `sample`: The code – 2

```
int main(int argc, char *argv[])
{
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    printf("Output: ");
    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);

    return 0;
}
```

What now?

Observation:

There are input data, for which `sample` computes a wrong result

Question:

- How do we find the defect in the code that causes this error?
- Is there a way of systematically searching for a defect?

4.1 Foundations

4.2 The Debugging Process

4.3 Reproducing Errors

4.4 Simplifying and Automating Test Cases

4.5 Techniques for Defect Localization

4.6 Defect Fixing

The main steps of the debugging process

- Describe the problem precisely
  - Sometimes this alone reveals the source of the problem
- Is the problem a software error?
  If yes:
  - Perform classic debugging

  If no:
  - Search and fix the problem elsewhere, e.g.
    - Defects in user manuals
    - Faulty business processes
    - Training deficits
- Check the effectiveness of the fix

The classic software debugging process

- Reproduce the error
- Simplify and (if possible) automate the test case that produces the error
- Localize the defect that causes the error
  - Create and test hypotheses
  - Observe program states
  - Check the validity of assertions in the program
  - Isolate cause-effect chains
- Fix the identified defect(s)

Checking the effectiveness of the fix

- Make sure that the defect has been fixed:
  - Re-run the test case(s) that resulted in errors
  - Everything ok now?
- Make sure that the fix did not create any new defects
  - Run your regression test suite
  - No new problems found?

Required infrastructure

- Problem reporting infrastructure
  - Process for handling problem reports
  - Tool for problem report administration and tracking
    For example, Bugzilla
- Configuration management system for software artifacts

4.1 Foundations

4.2 The Debugging Process

4.3 Reproducing Errors

4.4 Simplifying and Automating Test Cases

4.5 Techniques for Defect Localization

4.6 Defect Fixing

A sample bug report

Example: Mozilla bug report no. 24735 from 1999

-> Start mozilla

-> Go to bugzilla.mozilla.org

-> Select search for bug

-> Print to file setting the bottom and right margins to .50
(I use the file /var/tmp/netscape.ps)

-> Once it's done printing do the exact same thing again on
the same file (/var/tmp/netscape.ps)

-> This causes the browser to crash with a segfault

[Zeller 2005, p. 55]

Goal: Create an as simple as possible test case that reproduces the reported problem

Typical problems

- Reproducing the environment in which the problem occurs
- Reproducing the history trail may be necessary
- For software errors: reproduce a program run that causes the error; this may include
  - Input data
  - Initial persistent data
  - User interaction, interaction with neighboring systems
  - Time
  - Communication with other processes
  - Process threads
  - Random data

Time-dependent errors: a case

In early 1992 a company installed a new barrier gate control system in a couple of parking garages. In the morning of September 12, 1992, the operators of all these garages called the support line and reported the same problem: the exit barriers didn't open anymore.

What caused this problem?

Hint: The date had been coded with two integers, one for the year and one for the day of the year.

4.1 Foundations

4.2 The Debugging Process

4.3 Reproducing Errors

4.4 Simplifying and Automating Test Cases

4.5 Techniques for Defect Localization

4.6 Defect Fixing

Simplifying

- Given: a test case which reliably causes a reported error
- Goal:
  - Remove all irrelevant parts of the test case
  - Automate the simplified test case
- In an optimally simplified test case, all constituents are relevant, i.e. removing anything from the case no longer produces the reported error
- How to simplify?
  - Simplify environment
  - Reduce history trail
  - Simplify inputs / interactions

Automating

- The error-provoking test case must be executed frequently in the debugging process:
  - for finding simplifications
  - for testing hypotheses when systematically locating a defect

➪ Automation pays off

- Test automation techniques: → Chapter 4 of this course

Simplify the environment

- Determine which states or conditions in the system's environment are relevant and which ones aren't
  - Hardware and operating system
  - State of persistent data
  - Time
  - State of neighboring systems
- Irrelevant states and conditions can be safely ignored
- Goal: minimize the effort for setting up the test environment in which the a test case produces the reported error
- Means: systematic trying

Simplify the error history

- Can we reduce the number of steps, required for provoking the error?
- Means: systematic trying
- Example: Mozilla bug report no. 24735 (see above) reports the following error-provoking sequence of steps:
  Start mozilla; Go to bugzilla.mozilla.org; Select search for bug; Print to file setting the bottom and right margins to .50; Once it's done printing do the exact same thing again on the same file.

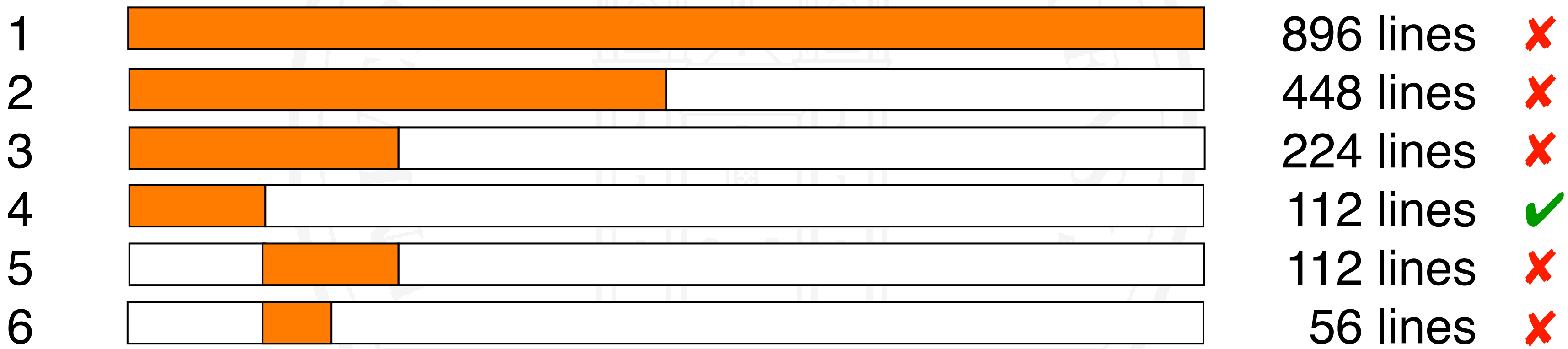
  Actually, the following steps suffice to provoke the error:
  Start mozilla; Go to bugzilla.mozilla.org; Select search for bug; Press Alt-P; Left-click on the Print button in the print dialog window.

Simplify inputs

- Example: Mozilla bug report no. 24735 (see above)
  - The erroneous printing function uses the currently displayed web page as input
  - This page consists of 896 lines of html code
- Which parts of this data cause the error and which ones are irrelevant?
- Means: binary search [Kernighan and Pike 1999]
  - Partition the set of input data into two halves
  - Test both halves individually
  - Recursively continue with that half which provokes the error

Simplify inputs – 2: An example [Zeller 2005]

- Example: Mozilla bug report no. 24735 (see above)
- Binary search yields a single fault-provoking line of html code in twelve steps:



...

12 <SELECT NAME="priority" MULTIPLE SIZE=7> 1 line ✗

Simplify inputs – 3

- What to do if both halves don't provoke the error while the whole does?

  <SELECT NAME="priority" MULTIPLE SIZE=7> ✘
  <SELECT NAME="priority" MULTIPLE SIZE=7> ✔
  <SELECT NAME="priority" MULTIPLE SIZE=7> ✔

- Instead of halves use smaller portions, e.g., quarters

  <SELECT NAME="priority" MULTIPLE SIZE=7> ✔
  <SELECT NAME="priority" MULTIPLE SIZE=7> ✘
  ☞ <SELECT NAME="priority" MULTIPLE SIZE=7> ✘
  <SELECT NAME="priority" MULTIPLE SIZE=7> ✔

- Continue with eighths, etc.
- Result: <SELECT> ✘

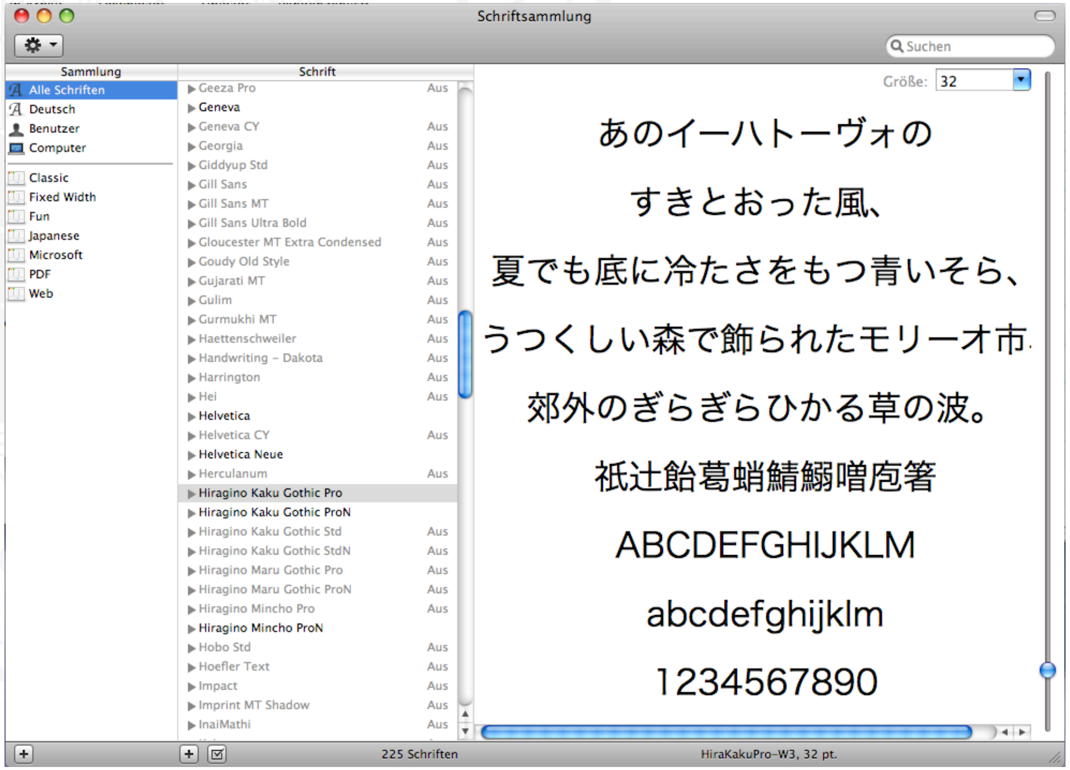
Automating the simplification

- Simplification can be automated partially
  - In particular, the technique of binary searching
  - Applicable for simplification of input data or interaction sequences
- Example: Zeller's ddmin delta debugging algorithm [Zeller 2005, Chapter 5.4-5.5]

Another example

Microsoft PowerPoint 2004 Version 11.0 on MacBook Pro with Mac OS 10.5.6 crashed during startup if the font *Hiragino Kaku Gothic Pro* was disabled in the font collection.

Using interval bisection on the set of all fonts we can find a minimal set of deactivated fonts that causes the error. This set only contains the font Hiragino Kaku Gothic Pro.



4.1 Foundations

4.2 The Debugging Process

4.3 Reproducing Errors

4.4 Simplifying and Automating Test Cases

4.5 Techniques for Defect Localization

4.6 Defect Fixing

Overview

- Create and test hypotheses
- Static and dynamic program analysis
  - Control flow
  - Data flow
- Analyze program states
- Observe program execution (stepping, breakpointing)
- Dynamically check program assertions
- Determine and isolate cause-effect chains
- Debugging by “gut feeling”

Creating and testing hypotheses

- The basis of systematic debugging
- Principle: Get insight through theory and experimentation
  1. Create a hypothesis
  2. Derive predictions from hypotheses
  3. Verify predictions experimentally
  4. If predictions and experiment results match
     - Correctness of hypothesis becomes more probable
     - Try to further confirm hypothesis

     } → Theory

     Otherwise:
     - Reject hypothesis
     - Create new or modified hypothesis; continue with step 2
- Important: record the track of all tested hypotheses

Finding hypotheses

Possible ways:

- Analysis of problem description
- Static analysis of the code
- Analysis of a erroneous execution run
- Comparison of correct and erroneous execution runs
- Building new hypotheses on the basis of previous ones:
  - Must be compatible with previously accepted
  - Must not use assumptions that stem from previously rejected hypotheses

Derive and check predictions

- Techniques
  - Static or dynamic analysis of the code
  - Observation of system states
  - Dynamic checking of assertions
- Deductive approach: draw logical conclusions from
  - existing knowledge
  - the source code
  - test cases and test results
- Experimental approach: observe
  - program execution
  - program state

Example: Program `sample` (cf. 4.1)

- First hypothesis

  Program runs correctly

  Prediction: Entering `11 14` yields `11 14` as result

  Experiment: `$ ./sample 11 14`

  `Output: 0 11` ✘

  ➥ Hypothesis is rejected

Example: Program `sample` (cf. 4.1)

- Second hypothesis

  Program prints wrong variables

  Prediction: `a[0]==11, a[1]==14`, but result is

  `Output: 0 11`

  Experiment: Replace code for input and sorting by

  ```
  a[0] = 11; a[1] = 14; argc = 3;
  ```

  Result: `Output: 11 14` ✔

  ➥ Hypothesis is rejected

Static and dynamic analysis

- Analyzing the control flow and the data flow of a program (see Chapter 3 on data flow testing and Chapter 11 on static analysis of my Software Engineering course)
- Static Analysis
  - Yields the potentially possible control and data flows
  - No program execution required
  - Independent of any concrete test cases
- Dynamic Analysis
  - Analyzes a concrete program run (based on a test case)
  - Yields actual control and data flows for this run

Example: static vs. dynamic program slicing

```
int main() {
int a, b, sum, mul;
sum = 0;
mul = 1;
a = read ();
b = read ();
while (a<=b) {
sum = sum + a;
mul = mul * a;
a = a +1;
}
write (sum);
write (mul);
}
```

Sample program

```
int main() {
int a, b, sum, mul;
sum = 0;
mul = 1;
a = read ();
b = read ();
while (a<=b) {
sum = sum + a;
mul = mul * a;
a = a +1;
}
write (sum);
write (mul);
}
```

Static slice of mul in line 13

```
int main() {
int a, b, sum, mul;
sum = 0;
mul = 1;
a = read ();
b = read ();
while (a<=b) {
sum = sum + a;
mul = mul * a;
a = a +1;
}
write (sum);
write (mul);
}
```

Dynamic slice of mul in line 13 with a=5, b=2

Analysis of program states

- The problem: a defect typically
  - leads to a sequence of erroneous states
  - that eventually manifest in observable errors
- Check suspicious program states
  - Instrumentation of the code:
    - Record variable values
    - Print or log variable values, maybe using a logging framework such as LOG4J [Logging Services]
  - Using a debugger
    - Compile program in debug mode
    - Halt execution at critical points (by setting breakpoints)
    - Inspect current variable values

Example: Program `sample` (cf. 4.1)

- Third hypothesis

  Sorting procedure called with wrong parameters

  Prediction: Values in array `a` and/or value of `argc` wrong

  Experiment: Prior to the call of shell_sort we instrument the source code with

```
printf("Parameters of shell_sort: ");
for (i = 0; i < argc; i++)
  printf("%d ", a[i]);
printf ("%d ", argc);
printf("\n");
```

  Result: `Parameters of shell_sort: 11 14 0 3` ✘

  ➥ Hypothesis is confirmed

- Alternatively, we could have used a debugger

Example: Program `sample` (continued)

- Theory: The input vector passed to shell_sort contains a non-allocated variable at the end, which is zero
- Prediction 1: Zero will always appear in the result
- Prediction 2: Any input vector containing only negative numbers and a zero will produce correct results
- Experiments:

```
$ ./sample 11 5 7
Output: 0 5 7
```
✘

```
$ ./sample 11 5 1
Output: 0 1 5
```
✘

```
$ ./sample -5 0 -9
Output: -9 -5 0
```
✔

```
$ ./sample 0 -21 -9
Output: -21 -9 0
```
✔

Example: Program `sample` (continued)

- All experiments confirm the theory

➥ Evidence that the passing of parameters to shell_sort is defective

Observe program execution

Using a debugger, we can

- Stepwise execute a program or halt it at breakpoints
  - Compare expected and actual control flow
  - Inspect parts of system state where appropriate
- Observe variable definition, modification and use

Checking assertions

- Specifying contracts for classes and methods with assertions:
  - Preconditions
  - Postconditions
  - Invariants

  Formally specified contracts can be checked dynamically by a suitable runtime system

- When an assertion is violated, analyze the program state

Causes and effects

- An observation:
  - In the decade of 1950 to1960 the decline of the population of storks in Europe is strongly correlated with the increasing number of tarmac roads
- Question:
  - Is the increasing number of tarmac roads the / a cause for the disappearance of storks?
- Testing for causality: a is a cause for b iff
  - b occurs if a has occurred previously
  - b does not occur if a has not occurred previously
  - All other variables are kept constant

Causes and effects – 2

- Experimental proof of (or evidence for) causality
  - Generally rather difficult: Problem of controlled experiments
  - For debugging, it is doable:
    - Controlled environment
    - Test case reproducible
- In debugging, a cause for an error f can be viewed as the difference between
  - a test case exhibiting the error f (1)
  - a test case that runs correctly (2)
- Again, we look for a minimal cause

➥ Search a minimal difference between (1) and (2)

Example: Program `sample` (cf. 4.1)

- Fourth hypothesis

  `shell_sort` *should be called with* `argc-1` *(instead of* `argc`*)*

  Prediction: Result is correct

  Experiment: Execute with modified source code (or modify state of running program with a debugger)

  Result: `Output: 11 14` ✔

  ➥ Hypothesis is confirmed

- From the first hypothesis we know that calling shell_sort with `argc` leads to an error
- The difference in the code is „`-1`“ in line 36
- This is a minimal cause of the error

Identifying and isolating cause-effect chains

- The immediate cause of an error normally is not a defect, but an erroneous program state, eventually caused by a defect
  - Identify cause-effect chains
  - and isolate them from the irrelevant rest of the program
- Time-consuming: Requires creation and test of many hypotheses
- Systematic procedure needed
- Automatable: Zeller's Delta Debugging algorithm [Zeller 2002]

Isolating causes with Delta Debugging

- Difference between isolation and simplification:
  - Simplification: Find a minimal error-provoking test case
  - Isolation: Find an error-provoking and an error-free test case with a minimal difference
- Example: Isolation of minimal error cause in this input:

Minimal difference = minimal cause

Error-provoking:	`5;6;2:1;`	✘	Reduce erroneous case ↓
	`5;6;2:1;`	✔	
	`5;6;2:1;`	✔	
	`5;6;2:1;`	✔	Extend error-free case ↑
Error-free:	∅ (empty input)	✔	

Debugging by gut feeling

- To some extent, experienced software engineers develop an ability to "smell" the cause of an error
- In many cases, debugging by intuition is faster than any systematic debugging procedure
- Problem:
  - We need to stop intuitive debugging at the right time when it does not succeed...
  - ...and then switch to systematic debugging
- Suggested procedure
  - For a strictly limited time, debug by intuition
  - If success: Eureka! else: stop and start systematic debugging

Fixing a localized defect

If a defect has been located

- Estimate severity of defect
- Determine what and how much has to be fixed
- Estimate impact on other parts of the system
- Make the required modifications to the code and/or the documentation carefully and systematically
- Avoid quick-and-dirty patching of code

Check effectiveness of problem resolution

- Make sure that the reported problem no longer exists
  In case of software errors:
  - Inspect the modified code and documentation
  - Test the modified units
    - using the error-provoking test case(s)
    - by writing more unit test cases
- Check for unexpected side effects
  - Adapt the regression test suite to the modified code
  - Perform a regression test
- Create a new configuration / release

Learning from the fixed defect

Defects are typically due to mistakes by humans

- Try to determine / guess the reasons why somebody made the mistake(s) that led to the defect
- Investigate if there are any similar defects in the source code that stem from the same kind of mistake
- Are there any constructive means to avoid such defects in the future, e.g., by
  - changing a process
  - training people

References

S.C. McConnell (1993). *Code Complete: A Practical Handbook of Software Construction*. Redmond: Microsoft Press.

B.W. Kernighan, R. Pike (1999). *The Practice of Programming*. Reading, Mass.: Addison-Wesley.

M. Weiser (1992). Programmers Use Slices When Debugging. *Communications of the ACM* **25**(7):446–452.

A. Zeller (2002). Isolating Cause-Effect Chains from Computer Programs. *Proceedings of the 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Charleston, South Carolina.1–10.

A. Zeller (2005). *Why Programs Fail: A Guide to Systematic Debugging*. Amsterdam: Morgan Kaufmann and Heidelberg: dpunkt.

Bugzilla. http://www.bugzilla.org

Logging Services. http://logging.apache.org

