



Reproducing Problems

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken





Pattern: Reproduce Problem

Problem A problem occurs only under specific circumstances, which may not be known or not be reproducible.

Solution There are three distinct patterns that help in reproducing problems:

1. **TRACK PROBLEMS** to collect all facts about how to reproduce the problem.
2. **AUTOMATE TEST** to reproduce the problem automatically. This is especially useful if the problem does not occur predictably.
3. **MANAGE VERSIONS** to reproduce specific versions of your program.



The lifetime of a problem—40 years ago —

1. The user-programmer experiences a problem.■
2. He isolates the cause and fixes it.■
3. He verifies that it is fixed. We're done.■





The lifetime of a problem—today _____

1. The *user* experiences a problem.■
2. He calls the help desk,■
3. He convinces the help desk that it *was* a problem.■
4. He gives all necessary information to the help desk.■
5. The programmer gets the problem report.■
6. She tries to *reproduce the problem*.■
7. Once the problem is reproduced, she becomes a user-programmer. Now, it's just like 40 years ago :-)



How do we know about problems? _____

From the user. By e-mail, phone, fax or ordinary mail.

From the program. Using an automatic *talkback* feature.





Pattern: Report Problem

Problem A user experiences a problem. How can a developer reproduce the problem such that he can fix it?

Solution Identify the *information* that developers need to reproduce problems. Typical information includes:

- The *product release*
- The *operating environment*
- The *problem history*
- A description of the *expected* behavior
- A description of the *experienced* behavior

This information is then written into *instructions* on how to write a *problem report*.



What is relevant?

Variant 1—not enough information:

From: user@inter.net

To: support@vendor.com

Your program crashed. Just wanted to let you know.

X.



What is relevant? (2)



7/55

Variant 2—too much information:

From: another_user@inter.net

To: support@vendor.com

I experienced the following problem: [...]

With the enclosed dump, you can reproduce it easily.

Sincerely,

Y.

[Attachment: ISO image of user's hard disk, 10 GB]



What is relevant? (3)



8/55

Variant 3—irrelevant information:

From: yet_another_user@inter.net

To: support@vendor.com

Your program doesn't work properly. It cannot connect to the internet. Could this be because I installed a new desktop background color pattern? Or do I have to set up this modem thing?

Sincerely,

Z.





What is relevant? (4)

Solution: You set up a set of *guidelines*—either for the users or for your helpdesk.

To enable us to fix a DDD bug, you *must* include the following information:

- * Your DDD configuration. Invoke DDD as

```
$ ddd --configuration
```

to get the configuration information. If this does not work, please include at least the DDD version, the type of machine you are using, and its operating system name and version number.

- * The debugger you are using and its version (e.g., ‘gdb-4.17’ or ‘dbx as shipped with Solaris 2.6’).
- * The compiler you used to compile DDD and its version (e.g., ‘gcc-2.8.1’). [...]





What has happened?

If the error occurs after a long series of events, it is often difficult for the user to retrace all the steps from the program invocation to the error.

Hmmm—I opened File, Open File—and then what? Wait. . .

A possible solution is to record all important events in a *log file*, which can later be examined and reproduced by the vendor.

Be aware of provacy issues, though!





Pattern: Manage Versions

Problem You must reproduce a specific configuration of the product as installed at the customer's site.

Solution Keep your code under *version control*. Whenever a new version is shipped, mark its source base with an appropriate tag. Use this tag to recreate the source base—and with it, the product itself.





Tracking Problems

Once a problem report has been filed, it must be stored somewhere.

Have a shared 'PROBLEMS' document. Easy to install, but does not scale.■

Have a *problem database* which stores all problem reports.




The Bugzilla problem database



Search for bugs - Mozilla (Build ID: 2002072204)

File Edit View Go Bookmarks Tools Window Help Debug QA

http://bugzilla.mozilla.org/query.cgi



Bugzilla Version 2.17

This is **Bugzilla**: the Mozilla bug system. For more information about what Bugzilla is and what it can do, see [mozilla.org's bug pages](#).

Search for bugs

Summary: [contains all of the words/strings] Search

Product:	Component:	Version:	Target:
Browser	Accessibility	1.01	---
Bugzilla	Accessibility APIs	1.1	Future
Calendar	Account Manager	1.2	3.0
CCK	Address Book	1.3	Jan
Chimera	Addressbook/LDAP (non-UI)	1.4	M1

A comment: [contains all of the words/strings]

The URL: [contains all of the words/strings]

Whiteboard: [contains all of the words/strings]

Keywords: [contains all of the keywords]

Status:	Resolution:	Severity:	Priority:	Hardware:	OS:
UNCONFIRMED	FIXED	blocker	--	All	All
NEW	INVALID	critical	P1	DEC	Windows 3.1
ASSIGNED	WONTFIX	major	P2	HP	Windows 95
REOPENED	LATER	normal	P3	Macintosh	Windows 98
RESOLVED	REMOVED	minor	P4	PC	Windows ME
VERIFIED	DUPLICATE	trivial	P5	SGI	Windows 2000
CLOSED	WORKSFORME	enhancement		Sun	Windows NT

Email and Numbering

Any of:

- bug owner
- reporter
- QA contact
- CC list member
- commenter

contains []

Any of:

- bug owner
- reporter
- QA contact
- CC list member
- commenter

contains []

Only include bugs numbered: [] (comma-separated list)

Only bugs with at least: [] votes

Bug Changes

Only bugs changed in the last [] days

Only bugs where any of the fields [Bug creation] alias assigned_to bug_file_loc were changed between [] and Now (YYYY-MM-DD) to this value: (optional) []

Sort results by: Bug Number Search

Document: Done (6.201 secs)





Problem Identification

Each problem has a unique *identifier* (also called *PR number* or *CR number*).

This developers can refer to it in

- e-mails
- change logs
- status reports





Problem Severity

Blocker Blocks development and/or testing work. This highest level of severity is also known as *Showstopper*.

Critical Crashes, loss of data, severe memory leak.

Major Major loss of function.

Normal This is the “standard” problem.

Minor Minor loss of function, or other problem where an easy workaround is present.

Trivial Cosmetic problem like misspelled words or misaligned text.

Enhancement Request for enhancement.





Problem Status and Resolution

FIXED This problem has been fixed and tested.

INVALID The problem described is not a problem.

WONTFIX The problem described is a problem which will never be fixed.

LATER The problem will be fixed in a later version.

REMIND Like LATER, but might still be fixed earlier.

DUPLICATE The problem is a duplicate of an existing problem.

WORKSFORME All attempts at reproducing this problem were futile. If more information appears later, the problem will be re-assigned.





Pattern: Track Problems

Problem What problems are in my current product? Which problems must be fixed before I can ship the next release?

Solution Use a *problem tracking system* to keep track of problem reports.

Using the problem tracking system, assign each problem report

- a unique *identifier* such that one can refer to the problem (frequently called *PR number*)
- a *status* such that one can find problems that do persist in the current version
- a *severity* such that the most serious problems get fixed first.





Managing Problem Reports

Who files problem reports? This can be support personnel only; in general, though, it is probably useful if any developer can add new entries.

Who assigns ownership, severity, etc? This depends on how your work is organized.

Who closes issues? This can be the individual tester, or some quality assurance instance that verifies fixes.





Drowning in Problems

In October, 2002, the Mozilla problem database listed roughly *28,000* open problems waiting to be resolved.

Reasons:

1. Lots of problems :-)
2. Lots of *duplicates*—problem reports that apply to the same defect





Pattern: Identify Duplicates

Problem A problem tracking system lists several problems that are so similar that they are likely caused by the same defect.

Solution Allow problems to be marked as *duplicates* of each other.

Whenever one submits a new problem report, one should first query whether a *similar problem* has already occurred. If so, the new problem should be marked as duplicate of the old problem.





Making Problem Reports Obsolete

Once in a while, a problem database should be cleaned up by searching for *obsolete* problems. A problem report could be declared obsolete if, for instance,

- The problem will never be fixed—for instance, because the program is no longer supported.
- The problem is old and has only occurred once, or
- The problem is old and has only occurred internally.





Reproducing Problems

Once a problem report is in the problem database, it will eventually be processed by some programmer in order to fix the problem.

The first task of the programmer is to *make the problem reproducible*. ⇒ Whenever someone wants the problem to occur, there is a well-defined and reliable way to do so.

If the problem is *not* reproducible, then you'll never know whether it has been successfully fixed :-)





Pattern: Automate Test

Problem Testing software requires user interaction and is thus expensive and error-prone.

Solution Automate the tests such that they can be executed automatically.

Applicability Choose a *level* for automating execution:

- If your program can be decomposed into individual *units*, use the interface of these units for testing them.
- If your program interacts with its environment, you can use *capture/replay* techniques to automate its execution.
- If your program offers an *automation interface*, you can use this interface to write appropriate test scripts.





Automating Unit Tests

If your program can be decomposed into individual *units* that offer interfaces for manipulating their resources, then these interfaces can frequently be used for automating the test of these units.

This is called *unit testing*—a great way to both automate testing and simplify debugging.





What Unit Tests do

When a single unit test is executed, it does three things:

It *sets up* an environment for embedding the unit.

Frequently, a unit will require services of other units or the operating environment; this part sets up the stage.

It *tests the unit*. Each possible behavior of the unit is covered by a test case, which first performs the operation(s) and then verifies whether the outcome is as expected.

It *tears down the environment again*. This means to bring everything back in the state encountered initially.





Example: Testing Rational numbers _____

Let us assume you manage a Java class for rational numbers, called `Rational`.

The constructor takes a numerator and a denominator:

`Rational(1, 3)` creates the rational number $\frac{1}{3}$.



Desired Properties of Rational numbers _

Identity. Is $\frac{1}{3} = \frac{1}{3}$?

Different representations. Is $\frac{2}{6} = \frac{1}{3}$?

Integers. Is $\frac{3}{3} = 1$?

Non-Equality. Is $\frac{1}{3} \neq \frac{2}{3}$?



RationalTest.java

```
public class RationalTest extends TestCase {
    private Rational a_third;

    // Create new test
    public RationalTest(String name) {
        super(name);
    }

    // Setup environment
    // will be called before any testXXX() method
    protected void setUp() {
        a_third = new Rational(1, 3);
    }

    // Release environment
    protected void tearDown() {
        a_third = null;
    }
}
```



RationalTest.java (2)

```
// Test for equality
public void testEquality() {
    assertTrue(new Rational(1, 3).equals(a_third));
    assertTrue(new Rational(2, 6).equals(a_third));
    assertTrue(new Rational(3, 3).equals(
        new Rational(1, 1)));
}

// Test for non-equality
public void testNonEquality() {
    assertTrue(!(new Rational(2, 3).equals(a_third)));
}
```



RationalTest.java (3)

```
// Set up a suite of tests
public static Test suite() {
    TestSuite suite =
        new TestSuite(RationalTest.class);
    return suite;
}
```



RationalTest.java (4)



```
// Assign a name to this test case
public String toString() {
    return getName();
}

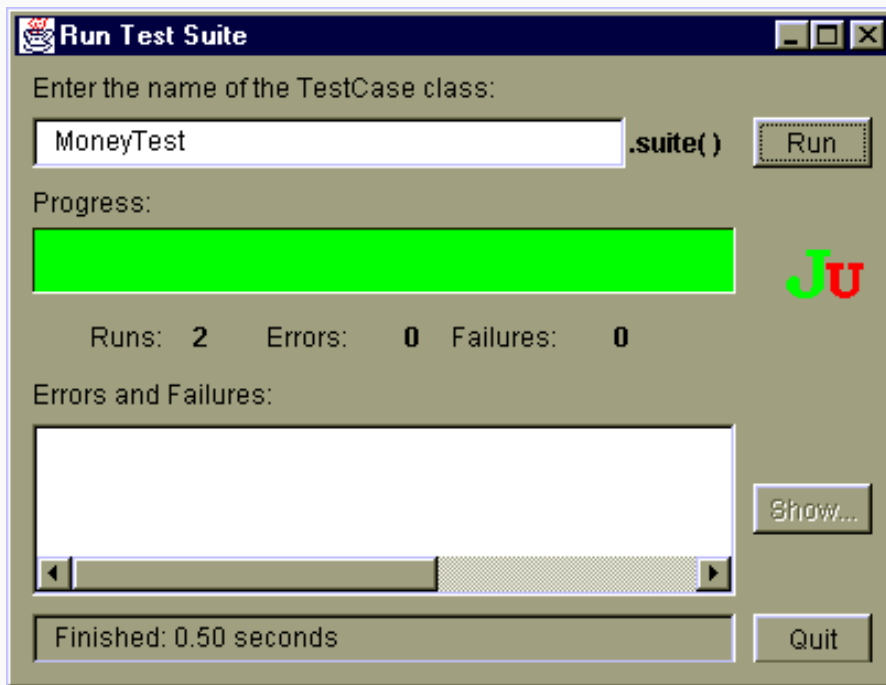
// Main method: Invokes GUI
public static void main(String args[]) {
    String[] testCaseName =
        { RationalTest.class.getName(); }

    // Run using a textual user interface
    // junit.textui.TestRunner.main(testCaseName);

    // Run using a graphical user interface
    junit.swingui.TestRunner.main(testCaseName);
}
}
```



JUNIT user interface





Benefits of automated testing

- ✓ It can be used as *verifier*.■
- ✓ Automated testing allows *faster changes*.■
- ✓ Automated testing also allows for *faster debugging*.■
- ✓ Also, an automated test case can be used as *debuggee* instead of the original program.■
- ✓ Automated test cases can serve as *specifications*.■
- ✓ Automated testing is much more *reliable* than say, written user instructions.■
- ✓ After the fix, an automated test can be used as *prevention* against regression.■
- ✓ Test cases can *drive and measure the development process*.

See also *Extreme Programming!*





Pattern: Test Unit

Problem Some unit of the program must be tested.

Solution Create a *test case* that tests the unit via its interface.

Applicability The unit must operate *without user interaction*. If other interaction with the environment is required, the test case must take care to set up and tear down the appropriate resources.





Pattern: Test Early, Test Often

Problem A test that used to pass suddenly fails. Can we relate the failing test to a particular change?

Solution Test early and often.

This means that you should test *as soon as the code is written*, not later (“test early”).

Furthermore, you should test *after each small change to the code* (“test often”). This way, if a test suddenly fails, you can relate it to the most frequent change.



Capture and Replay

Problems with unit testing:

- Program may not be decomposable into units
- Interactive interfaces cannot be tested

In such cases, a *capture/replay tool* comes in handy.





Pattern: Capture and Replay

Problem A program interacts with its environment—especially with the user. To fully automate the program execution, this interaction must faithfully be reproduced as well.

Solution Use a *capture and replay* tool. As the name says, such tools come in two modes:

***Capturing* interaction.** The program is normally executed, interacting with its environment. However, the tool *records* all input from the environment to a *script* before forwarding it to the program.

***Replaying* interaction.** The program is executed under control of the tool. The tool redirects the program input such that it no longer gets its input from the environment, but rather from the previously recorded script.





Example: Getting Core Dump Contents _____

```
$ sample-with-efence -11 14  
Electric Fence 2.1 Copyright (C) 1987-1998 Bruce Perens.  
Segmentation fault (core dumped)  
$ -
```

Issue: Create a tool that automates GDB such that it reports where a core dump occurred.





Example: Getting Core Dump Contents (2)

```
$ gdb sample-with-efence
GNU gdb 5.1.1
Copyright 2002 Free Software Foundation, Inc. [...]
(gdb) core-file core
Core was generated by './sample-with-efence -11 14'.
Program terminated with signal 11, Segmentation fault.
#0  0x0804865f in shell_sort (a=0x40160ff8, size=3)
    at sample.c:17
17          int v = a[i];
(gdb) where
#0  0x0804865f in shell_sort (a=0x40160ff8, size=3)
    at sample.c:17
#1  0x08048777 in main (argc=3, argv=0xbffffa94)
    at sample.c:35
#2  0x400446cf in __libc_start_main ()
    from /lib/libc.so.6
(gdb) quit
$ -
```





Using EXPECT

The EXPECT utility automates the execution of interactive command-line tools.

EXPECT commands include:

Spawn. Start a program under EXPECT control. In our example, this would be GDB.

Send. Send a string of characters to the spawned program—for instance, GDB commands.

Expect. Wait until the program has issued a specific output—typically, a *prompt* such as "(gdb) ".



An EXPECT script

```
# Get the location for the current core file
```

```
spawn gdb sample-with-efence
expect "(gdb) "
send "core-file core\n"
expect "(gdb) "
send "where\n"
expect "(gdb) "
send "quit\n"
```





Executing the EXPECT script

```
$ expect gdb.exp
spawn gdb sample-with-efence
GNU gdb 5.1.1
Copyright 2002 Free Software Foundation, Inc. [...]
(gdb) core-file core
Core was generated by './sample-with-efence -11 14'.
Program terminated with signal 11, Segmentation fault.
#0  0x0804865f in shell_sort (a=0x40160ff8, size=3)
    at sample.c:17
17          int v = a[i];
(gdb) where
#0  0x0804865f in shell_sort (a=0x40160ff8, size=3)
    at sample.c:17
#1  0x08048777 in main (argc=3, argv=0xbffffa94)
    at sample.c:35
#2  0x400446cf in __libc_start_main ()
    from /lib/libc.so.6
(gdb) quit
$ _
```





A more abstract *EXPECT* script

```
# Get the location for the current core file
```

```
proc gdb_send {command} {  
    expect "(gdb) "  
    send "$command\n"  
}
```

```
spawn gdb sample-with-efence  
gdb_send "core-file core"  
gdb_send "where"  
gdb_send "quit"
```



Recording scripts with AUTOEXPECT



```
$ autoexpect gdb sample-with-efence
autoexpect gdb sample-with-efence
autoexpect started, file is script.exp
GNU gdb 5.1.1
Copyright 2002 Free Software Foundation, Inc. [...]
(gdb) core-file core
...
(gdb) where
...
(gdb) quit
autoexpect done, file is script.exp
$ expect script.exp
GNU gdb 5.1.1
Copyright 2002 Free Software Foundation, Inc. [...]
(gdb) core-file core ...
```





Exploring AUTOEXPECT scripts

```
send "where\n"  
expect "where\n"  
#0  0x0804865f in shell_sort (a=0x40160ff8, size=3)  
    at sample.c:17  
#1  0x08048777 in main (argc=3, argv=0xbffffa94)  
    at sample.c:35  
#2  0x400446cf in __libc_start_main ()  
    from /lib/libc.so.6  
(gdb) "
```

Problem: *verbosity*—AUTOEXPECT cannot distinguish output from a prompt

Script is good for short-term usage only (debugging); must be edited otherwise



Capture and Replay of GUIs

In principle, work just like EXPECT and AUTOEXPECT, *but*

- we must know which part of the graphical user interface we should send our input to, and
- which part to read the expected output from.

This makes creating scripts *much harder* than for command-line tools.





Example: ANDROID

ANDROID is an EXPECT variant for X window system clients.

Script excerpt:

```
send_xevents wait 0 @298,198 btndn 1
send_xevents wait 164 @298,198 btnup 1
send_xevents wait 1574 @202,350 btndn 1
```





Example: Commercial Tools

Script excerpt:

```
SelectOption "Favorites/Search/Google"
FocusOn "Search Term"
Type "Debugging"
LeftMouseClicked "Google Search"
WaitFor "Searched the web for Debugging."
```





Automation vs. Usability

Automation and usability are *conflicting goals*:

- An interface that is easy to automate is hard to use directly by end users;
- a user-friendly interface is hard to automate.

Therefore, it is best to have at least *two* interfaces: one for users, and one for automation.

This requires *separating* functionality and presentation.



Creating Automation Interfaces

Idea: Introduce an interface such that your program can be automated.

Application programming interfaces (APIs). Make your program available as a unit.

Network and Web interfaces. Turn your program into a *component* whose services are available on the network (.NET, JAVA bean, CORBA)

Built-in programming languages. Include an interpreter for Python, Perl, TCL, or Visual Basic.





Pattern: Automate Execution

Problem An application does not provide an appropriate interface for automation.

Solution Create an *automation interface*. This can be an application programming interface, a web interface, or a built-in programming language.





Does Automation Pay Off?

The initial investment in separating presentation from functionality pays off as soon as

- ✓ another *alternate interface* is required, since it can rely on the common functionality,
- ✓ substantial *testing* is required, since it can use the automation interface,
- ✓ systematic *problem tracking* is required, since problems can frequently be described in terms of automated test cases.

And: *automated testing is required for automated debugging!*





Concepts

- ⇒ Reports about problems encountered in the field are stored in a *problem database* and classified with respect to status and severity.
- ⇒ Whenever you encounter a failure, write a test case that *reproduces* it—if possible, automatically.
- ⇒ The *benefits* of automated tests include: verification, faster changes, faster debugging, potential use as specifications, reliability, prevention against regression, ability to drive and measure the development process.





Concepts (2)

- ⇒ *Unit tests* automate the execution of program units via programming interfaces.
- ⇒ *Capture and replay* tools simulate the *program environment*—especially the user's input to a user interface.
- ⇒ *Scripts* as generated by capture tools must frequently be edited and abstracted to be useful in the long term.
- ⇒ In general, capture/replay scripts are likely to *change* with the program behavior.





Concepts (3)

- ⇒ Automation and usability are *conflicting goals*. Programs should provide different interfaces for each purpose. This requires *separating* functionality and presentation.
- ⇒ *Interfaces for automation* include application programmer interfaces (APIs), web service interfaces and built-in programming languages.
- ⇒ Test as *soon* as you can and as *often* as you can.

