Extracted from:

New Programmer's Survival Manual

Navigate Your Workplace, Cube Farm, or Startup

This PDF file contains pages extracted from *New Programmer's Survival Manual*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com .

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2010 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

New Programmer's Survival Manual

Navigate Your Workplace, Cube Farm, or Startup

Josh Carter

Edited by Susannah Davidson Pfalzer

New Programmer's Survival Manual

Navigate Your Workplace, Cube Farm, or Startup

Josh Carter

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Susannah Pfalzer (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Copyright © 2011 Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.
ISBN-13: 978-1-934356-81-4
Printed on acid-free paper.
Book version: P1.0—November 2011

For Daria and Genevieve.

[*White Belt*] Writing code that fails well is just as important as writing code that works well.

What happens when code fails? It's going to. Even if you wrote *your* part perfectly, there are all kinds of conditions that could cause the overall system to fail:

- A rogue mail daemon on computer, busy sending offers of great wealth from some foreign country, consumes all the RAM and swap. Your next call to `malloc()` returns `ETOOMUCHSPAM`.
- Java Update 134,001 fills up the system's hard drive. You call `write()`, and the system returns `ESWITCHTODECAF`.
- You try to pull data off a tape, but the tape robot is on a ship at sea, rolling waves cause the robot to drop the tape, and the driver returns `EROBOTDIZZY`.
- Cosmic rays flip a bit in memory, causing a memory access to return `0x10000001` instead of `0x1`, and you discover that this makes for a very bad parameter to pass into `memcpy()` after it returns `EMEMTRASHED`.

You may think, "Yeah, right" but all these cases actually happened. (Yes, I had to fix a tape robot controller because it would drop tapes when on a Navy ship.) Your code cannot naively assume that the world around it is sane—the world will take every opportunity to prove it wrong.

How your code fails is just as important as how it works. You may not be able to *fix* the failure, but if nothing else, your code should strive to fail gracefully.

Order of Operations

In many textbook programs, their environment is a clean slate, and the program runs to completion. In many messy, nontextbook programs, the environment is a rugby match of threads and resources, all seemingly trying to beat each other into submission.

Consider the following example: you're creating a list of customer names and addresses that will be fed to a label printer. Your code gets passed a customer ID and a database

connection, so you need to query the database for what you need. You create a linked list whose add() method looks like this:

Download ListUpdate.rb

```
def add(customer_id) # BAD BAD BAD, see text
  begin
    @mutex.lock
    old_head = @head
    @head = Customer.new
    @head.name =
      @database.query(customer_id, :name)
    @head.address =
      @database.query(customer_id, :address)
    @head.next = old_head
  ensure
    @mutex.unlock
  end
end
```

(Yes, I know this example is contrived. Bear with me.)

This code works in the happy path: the new element is put at the head of the list, it gets filled in, and everything is happy. But what if one of those queries to the database raises an exception? Take a look at the code again.[9]

This code doesn't fail gracefully. In fact, it does collateral damage by allowing a database failure to destroy the customer list. The culprit is the order of operations:

- The list @head and @head.next are absolutely vital to the list's integrity. These shouldn't be monkeyed with until everything else is ready.
- The new object should be fully constructed before inserting into the list.
- The lock should not be held during operations that could block. (Assume there's other threads wanting to read the list.)

9. Answer: First, the head of the list has already been set to the new element, so the head will have at least one blank field. Second, the rest of the list will vanish because head.next gets updated only after the database queries. And bonus badness: the list stays locked for the duration of the database queries—operations that could take an indeterminate amount of time to complete.

Transactions

In the previous section, the example had only one essential bit of state that needed to stay consistent. What about cases where there's more than one? Consider the classic example of moving money between two bank accounts:

Download Transaction.rb

```
savings.deduct(100)
checking.deposit(100)
```

What happens if the database croaks right after the money has been deducted and the deposit into checking fails? Where did the money go? Perhaps you try to solve that case by putting it back into the savings account:

Download Transaction.rb

```
savings.deduct(100)      # Happily works

begin
  checking.deposit(100)  # Fails: database went down!
rescue
  begin
    # Put money back
    savings.deposit(100) # Fails: database still dead
  rescue
    # Now what???
  end
end
```

Nice try, but that doesn't help if the second deposit() fails, too.

The tool you need here is a *transaction*. Its purpose is to allow several operations, potentially to several objects, to be either fulfilled completely or rolled back.

Transactions (here in a made-up system) would allow our previous example to look like this:

Download Transaction.rb

```
t = Transaction.new(savings, checking)
t.start

# Inject failure
checking.expects(:deposit).with(100).raises

begin
  savings.deduct(100)
  checking.deposit(100)
  t.commit
rescue
```

```
  t.rollback
end
```

You'll usually find transactions in databases, because our example scenario is exceedingly common in that field. You may find variations on this theme anywhere systems require an all-or-nothing interlock.

Failure Injection

So far, we've talked about how your code responds to *likely* failures. For purposes of testing, how do you ensure your code responds well when an essential resource dies, passes on, is no more, ceases to be, pushes up daisies, and becomes an ex-resource?

The solution is to inject failures using an automated test harness. This is easiest with a mock object framework, because you can tell the mock to return good data several times and then return something bogus or throw an exception. Likewise, in the code under test, you assert that the appropriate exception is raised.

Revisiting our list update problem, here's some test code that simulates a valid database response for key 1 and a failure on the query for key 2:

Download ListUpdate2.rb

```
require 'rubygems'
require 'test/unit'
require 'mocha'

class ListUpdateTest < Test::Unit::TestCase
  def test_database_failure
    database = mock()
    database.expects(:query).with(1, :name).
      returns('Anand')
    database.expects(:query).with(1, :address).
      returns('')
    database.expects(:query).with(2, :name).
①     raises

    q = ShippingQueue.new(database)
    q.add(1)

    assert_raise(RuntimeError) do
②     q.add(2)
    end
```

```
    # List is still okay
③   assert_equal 'Anand', q.head.name
    assert_equal nil, q.head.next
  end
end
```

① Injection of RuntimeError exception.

② Call will raise; the assert_raise is expecting it (and will trap the exception).

③ Verify that the list is still intact, as if q.add(2) were never called.

Failure injection of this sort allows you to think through—and verify—each potential scenario of doom. Test in this manner just as often as you test the happy path.

Test Monkeys

You can think through scenarios all day long and build tremendously robust code. Yet most fool-proof programs can be foiled by a sufficiently talented fool. If you don't have such a fool handy, the next best thing is a *test monkey*.

In my first job working on handheld computers, we had a program called Monkey that would inject random taps and drags into the UI layer, as if they had come from the touch-screen. There was nothing fancier than that. We'd run Monkey until the system crashed.

Monkey may not have been a talented fool, but a whole bunch of monkeys tapping like mad, 24 hours a day, makes up for lack of talent. Alas, no Shakespeare (but perhaps some E. E. Cummings) and a whole bunch of crashes. The crashes were things we couldn't have envisioned—that was the point.

In the same way, can you create a test harness that beats the snot out of your program with random (but valid) data? Let it run thousands or millions of cycles; you never know what might turn up. I used this technique on a recent project and discovered that once in a blue moon, a vendor API function would return "unknown" for the state of a virtual machine. What do they mean, they *don't know* the state? I had no idea the function could return that. My program crashed when it happened. Lesson learned…again.

• Click HERE to purchase this book now. discuss

Actions

Revisit the previous code with the customer list. How would you fix it? Here's a shell to work with:

Download ListUpdate2.rb

```
require 'thread'

class Customer
  attr_accessor :name, :address, :next

  def initialize
    @name = nil
    @address = nil
    @next = nil
  end
end

class ShippingQueue
  attr_reader :head

  def initialize(database)
    @database = database
    @head = nil
    @mutex = Mutex.new
  end

  def add(customer_id)
    # Fill in this part
  end
end
```

Use the test code from *Failure Injection,* on page 8 to see whether you got it right.