

Extracted from:

Release It!

Design and Deploy Production-Ready Software

This PDF file contains pages extracted from Release It!, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragmaticprogrammer.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 © 2009 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
Programmers

Release It!

Design and Deploy
Production-Ready Software



Michael T. Nygard



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 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://www.pragmaticprogrammer.com>

Copyright © 2007 Michael T. Nygard.

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-10: 0-9787392-1-3

ISBN-13: 978-0-9787392-1-8

Printed on acid-free paper.

P3.0 printing, February 2009

Version: 2009-3-27

4.3 Cascading Failures



The standard system architecture for enterprise systems, including websites and web services, comprises a collection of functionally distinct farms or clusters that are interconnected through some form of load balancing. We usually refer to the individual farms as *layers*—for example, as in Figure 4.6, on the following page—even though they might not really be a single stack.

In a service-oriented architecture, these look even less like traditional layers and more like a directed, acyclic graph.

System failures start with a crack. That crack comes from some fundamental problem. Various mechanisms can retard or stop the crack, which are the topics of the next chapter. Absent those mechanisms, the crack can progress and even be amplified by some structural problems. A cascading failure occurs when a crack in one layer triggers a crack in a calling layer.

An obvious example is a database failure. If an entire database cluster goes dark, then any application that calls the database is going to experience problems of some kind. If it handles the problems badly, then the application layer will start to fail. One system I saw would tear down any JDBC connection that ever threw a `SQLException`. Each page request would attempt to create a new connection, get a `SQLException`, try to tear down the connection, get another `SQLException`, and then vomit a stack trace all over the user.

A cascading failure occurs when problems in one layer cause problems in callers.

Cascading failures require some mechanism to transmit the failure from one layer to another. The failure “jumps the gap” when bad behavior in the calling layer gets triggered by the failure condition in the called layer.

Cascading failures often result from resource pools that get drained because of a failure in a lower layer. Integration Points without Timeouts is a surefire way to create Cascading Failures.

Just as integration points are the number-one source of cracks, cascading failures are the number-one crack accelerator. Preventing cascading failures is the very key to resilience. The most effective patterns to combat cascading failures are Circuit Breaker and Timeouts.

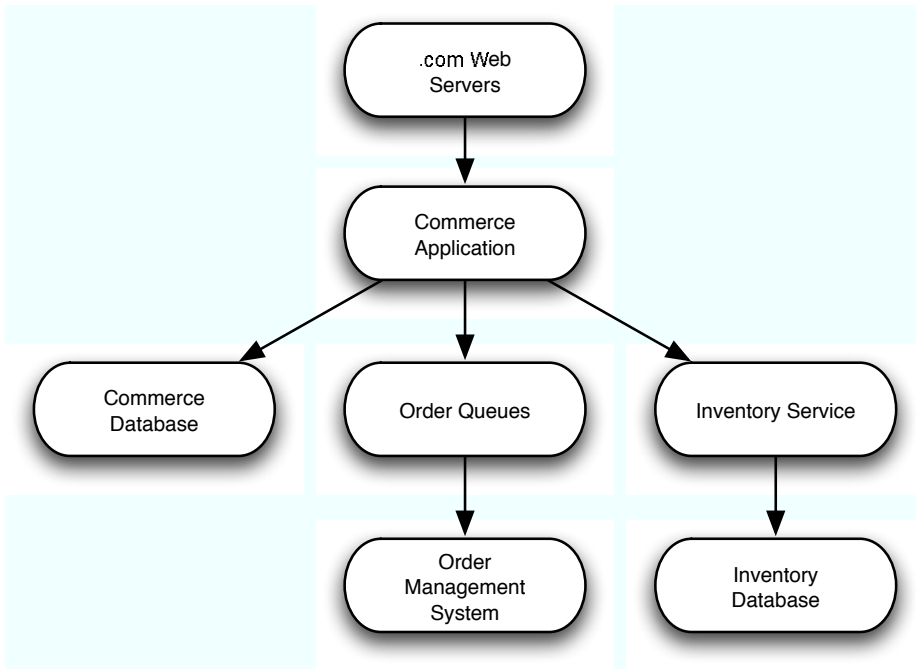


Figure 4.6: Layers Often Found in Commerce Systems



Remember This

Stop cracks from jumping the gap

A cascading failure occurs when cracks jump from one system or layer to another, usually because of insufficiently paranoid integration points. A cascading failure can also happen after a chain reaction in a lower layer. Your system surely calls out to other enterprise systems; make sure you can stay up when they go down.

Scrutinize resource pools

A cascading failure often results from a resource pool, such as a connection pool, that gets exhausted when none of its calls return. The threads that get the connections block forever; all other threads get blocked waiting for connections. Safe resource pools always limit the time a thread can wait to check out a resource.

Hammer Time

The layer-jumping mechanism often takes the form of blocked threads, but I've also seen the reverse—an overly aggressive thread. In one case, the calling layer would get a quick error, but, because of a historical precedent, it would assume that the error was just an irreproducible, transient error in the lower layer. At some point, the lower layer was suffering from a race condition that would make it kick out an error once in a while for no good reason. The upstream developer decided to retry the call when that happened. Unfortunately, the lower layer didn't provide enough detail to distinguish between the transient error and a more serious one. As a result, once the lower layer started to have some real problems (losing packets from the database because of a failed switch), the caller started to pound it more and more. The more the lower layer whined and cried, the more the upper layer yelled, "I'll give you something to cry about!" and hammered it even harder. Ultimately, the calling layer was using 100% of its CPU making calls to the lower layer and logging failures in calls to the lower layer. A circuit breaker would really have helped here.

Defend with Timeouts and Circuit Breaker

A cascading failure happens *after* something else has already gone wrong. Circuit Breaker protects your system by avoiding calls out to the troubled integration point. Using Timeouts ensures that you can come back from a call out to the troubled one.

4.4 Users



Users are a terrible thing.⁸ Systems would be infinitely more stable without them. The human users of a system have this knack for creative destruction. When your system is teetering on the brink of disaster like a car on a cliff in a movie, some user will be the seagull landing on the hood. Down she goes! Human users have a gift for doing exactly the worst possible thing at the worst possible time.

Users are a terrible thing. Worse yet, other systems that call ours march remorselessly forward like an army of Terminators, utterly unsympathetic about how close we are to crashing.

Traffic

Every user consumes some system resources. Unless you are building a peer-to-peer system such as BitTorrent, your system's capacity is limited. It scales with the amount of hardware and bandwidth you've bought, not the number of users you've attracted.

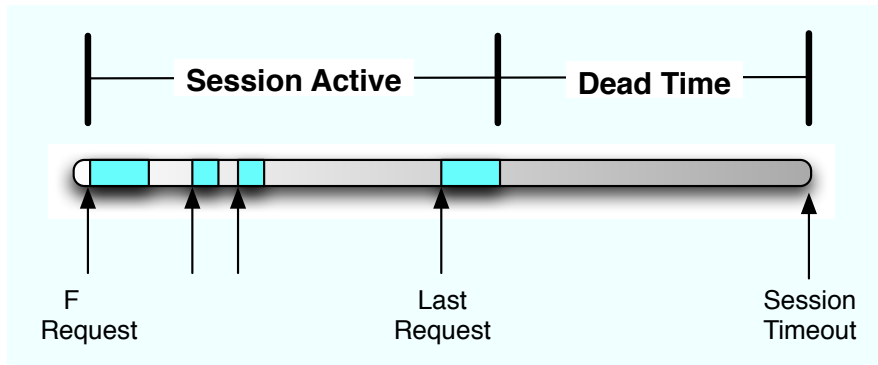
As traffic grows, it will eventually surpass your capacity.⁹ Then comes the biggest question: How does your system react to excessive demand?

In Section 8.1, *Defining Capacity*, on page 163, we will see the definition of capacity: when transactions take too long to execute, it means that the demand on your system has exceeded its capacity. Internally to your system, however, there are some harder limits. Passing those limits makes cracks in the system, and cracks always propagate faster under stress.

One such hard limit is memory available, particularly in Java or J2EE systems. Excess traffic can stress the memory system in several ways. First and foremost, in web systems, every user has a session. The session stays resident in memory for a certain length of time after the last request from that user. Every additional user means more memory.

8. Obviously, I'm being somewhat tongue-in-cheek. Although users do present numerous risks to stability, they are also the reason our systems exist.

9. If traffic isn't growing, then you have other problems to worry about!



During that dead time, the session still occupies valuable memory. A session is not a magic “Bag of Holding.”¹⁰ Every object you put into the session sits there in memory, tying up precious bytes that could be serving some other user.

When memory gets short, a large number of very surprising things can happen. Probably the least offensive is throwing an `OutOfMemoryError` exception at the user. If things are really bad, the logging system might not even be able to log the error. For example, `Log4j` and `java.util.logging` both create objects to represent a log event. If no memory is available to create the log event, then nothing gets logged. (This, by the way, is a great argument for external monitoring in addition to log file scraping.) A supposedly recoverable low-memory situation will rapidly turn into a serious stability problem. In fact, if you are making any native calls, then a low-memory condition will cause “`malloc`” to fail in the native code, for example, inside a Type 2 JDBC driver. It seems that few programmers of native code do good error checking, because I’ve seen JVM crashes result from native calls during a memory crisis.

Every user consumes more memory.

Your best bet is to keep as little in the session as possible. For example, it’s a bad idea to keep an entire set of search results in the session for pagination. It’s better if you requery the search engine for each new page of results. For every object you put in the session, consider that it might never be used again. It could spend the next thirty minutes uselessly taking up memory and putting your system at risk.

¹⁰ In case you didn’t play *Dungeons & Dragons*, a Bag of Holding was much bigger on the inside than on the outside. Things you put into it were available but weighed almost nothing. It was a convenient explanation for characters that could keep two broadswords, a mace, full-plate armor, and half a million gold pieces with them all the time.

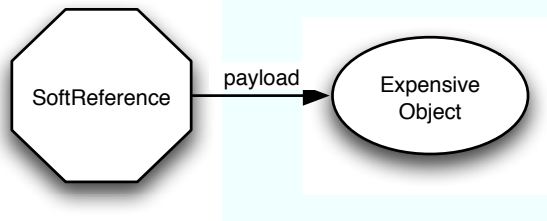


Figure 4.7: `SoftReference` and Its Payload

It would be wonderful if there was a way to keep things in the session (therefore in memory) when memory is plentiful but automatically be more frugal when memory is tight. Good news! There is a way to do exactly that. `java.lang.ref.SoftReference` objects hold a reference to some other payload object.

You construct a `SoftReference` with the large or expensive object as an argument. The `SoftReference` object actually is a Bag of Holding. It keeps the payload for later use.

```
MagicBean hugeExpensiveResult = ...; SoftReference ref = new
SoftReference(hugeExpensiveResult);
```

```
session.setAttribute(EXPENSIVE_BEAN HOLDER, ref);
```

This is not a transparent change. Any JSPs or servlets that access this object will know that they are going through a layer of indirection. If memory gets low, the garbage collector is allowed to reclaim the payload of a `SoftReference`, so long as there is no hard reference to that payload.

```
Reference reference = (Reference)session.getAttribute(EXPENSIVE_BEAN HOLDER);
MagicBean bean = (MagicBean) reference.get();
```

What is the point of adding this level of indirection? When memory gets low, the garbage collector is allowed to reclaim any “softly reachable” objects. An object is softly reachable if the only references to it are held by `SoftReference` objects. The expensive object in Figure 4.7 is softly reachable. The expensive object in Figure 4.8, on the next page, on the other hand, *not* softly reachable. It is strongly reachable because of the hard reference from the servlet.

The actual decision about when to reclaim softly reachable objects, how many of them to reclaim, and how many to spare is totally up to the garbage collector. The only guarantee is this: all softly reachable objects will be reclaimed before an `OutOfMemoryError` is thrown.

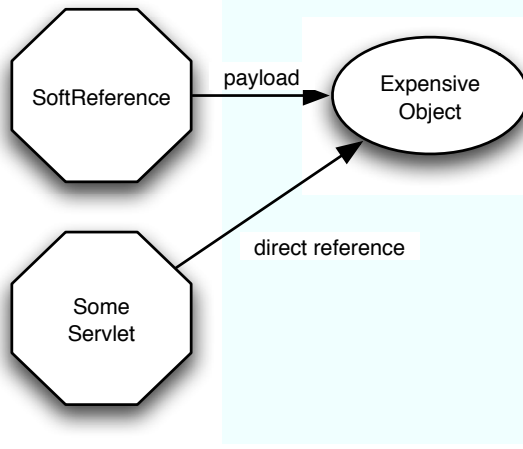


Figure 4.8: Strongly Reachable Payload Object

In other words, the garbage collector will take advantage of all the help you give it before it gives up. Be careful to note that it is the payload object that gets garbage collected, not the `SoftReference` itself. After the payload gets garbage collected, any calls to `SoftReference.get()` will return null. Any code that uses the payload object must be prepared to deal with a null payload, as shown in Figure 4.9, on the following page. It can choose to recompute the expensive result, redirect the user to some other activity, or take any other protective action.

`SoftReference` is a useful way to respond to changing memory conditions, but it does add complexity. Generally, it's best to just keep things out of the session. Use the `SoftReference` approach when you cannot keep large or expensive objects out of the session. `SoftReferences` let you serve more users with the same amount of memory.

Expensive to Serve

Some users are way more demanding than others. Ironically, these are usually the ones you want more of. For example, in a retail system, users who browse a couple of pages, maybe do a search, and then go away are both the bulk of users and the easiest to serve. Their content can usually be cached (however, see Pattern 10.2, *Use Caching Carefully*, on page 210 for important cautions about caching). Serving their pages usually does not involve external integration points. You will likely do some personalization, maybe some clickstream tracking, and that's about it.

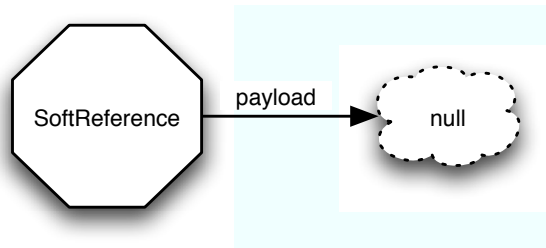


Figure 4.9: SoftReference After Payload Is Garbage Collected

But then there's that user who actually wants to buy something. Unless you've licensed the one-click checkout patent, checkout probably takes four or five pages. That's already as many pages as a typical user's entire session. On top of that, checking out can involve several of those troublesome integration points: credit card authorization, sales tax calculation, address standardization, inventory lookups, and shipping. In fact, more buyers don't just increase the stability risk for the front-end system, they can place back-end or downstream systems at risk too. (See Antipattern 4.8, *Unbalanced Capacities*, on page 98.) Increasing the conversion rate might be good for the profit-and-loss statement, but it is definitely hard on the systems.

There is no effective defense against expensive users. They are not a direct stability risk, but the increased stress they produce increases the likelihood of triggering cracks elsewhere in the system. Still, I don't recommend measures to keep them off the system, since they are usually the ones who generate revenue. So, what should you do?

The best thing you can do about expensive users is test aggressively. Identify whatever your most expensive transactions are, and double or triple the proportion of those transactions. If your retail system expects a 2% conversion rate (which is about standard for retailers), then your load tests should test for a 4%, 6%, or 10% conversion rate.

Conversion rate: the percentage of site visitors who actually buy something.

Unwanted Users

We would all sleep easier if the only users to worry about were the ones handing us their credit card numbers. In keeping with the general theme of "weird, bad things happen in the real world," there are definitely weird, bad users out there.

Total Conversion

If a little is good, then a lot must be better, right? In other words, why not test for a 100% conversion rate? As a stability test, that's not a bad idea. I wouldn't use the results to plan capacity for regular production traffic, though. By definition, these are the most expensive transactions. Therefore, the average stress on the system is guaranteed to be less than what this test produces. Build the system to handle nothing but the most expensive transactions, and you will spend ten times too much on hardware.

Some of them don't mean to be bad. For example, I've seen badly configured proxy servers start re-requesting a user's last URL over and over again. I was able to identify the user's session by its cookie and then trace the session back to the registered customer. Logs showed that the user was legitimate. For some reason, fifteen minutes after the user's last request, the request started reappearing in the logs. At first, these requests were coming in every thirty seconds. They kept accelerating, though. Ten minutes later, we were getting four or five requests *every second*. These requests had the user's identifying cookie but not his session cookie. So, each request was creating a new session. It strongly resembled a DDoS attack except that it came from one particular proxy server on one Navy base.

Once again, we see that sessions are the Achilles heel of web applications. Want to bring down nearly any dynamic web application? Pick a deep link from the site, and start requesting it, without sending cookies. Don't even wait for the response; just drop the socket connection as soon as you've sent the request. Web servers never tell the application servers that the end user stopped listening for an answer. The application server just keeps on processing the request. It sends the response back to the web server, which funnels it into the bit bucket. In the meantime, the 100 bytes of the HTTP request causes the application server to create a session (which may consume several kilobytes of memory in the application server). Even a desktop machine on a broadband connection can generate hundreds of thousands of sessions on the application servers.

DDoS: distributed denial-of-service attack. Many computers ganging up on a site with the purpose of saturating the bandwidth, CPU, or memory of the site's servers. Think Gulliver and the Lilliputians.

In extreme cases, such as the flood of sessions originating from the Navy base, you can run into problems worse than just heavy memory consumption. In our case, the business users wanted to know how often their most loyal customers came back. The developers wrote a little interceptor that would update the “last login” time whenever a user’s profile got loaded into memory from the database. During these session floods, though, the request presented a user ID cookie but no session cookie. That meant each request was treated like a new login, loading the profile from the database and attempting to update the “last login” time.

Imagine 100,000 transactions all trying to update the same row of the same table in the same database. Somebody is bound to get dead-locked. Once a single transaction with a lock on the user’s profile got hung (because of the need for a connection from a different resource pool), all the other database transactions on that row got blocked. Pretty soon, every single request-handling thread got used up with these bogus logins. As soon as that happens, the site is down.

So, one kind of bad user just blunders around leaving disaster in his wake. There are more crafty sorts, however, who deliberately do abnormal things that just happen to have undesirable effects. The first group isn’t deliberately malicious; they just do damage inadvertently. This next group belongs in its own category.

There is an entire parasitic industry that exists by consuming resources from other companies’ websites. Collectively known as *competitive intelligence* companies, these outfits leech data out of your system one web page at a time.

These companies will argue that their service is no different from a grocery store sending someone into a competing store with a list and a clipboard. There is a big difference, though. Given the rate that they can request pages, it’s more like sending a battalion of people into the store with clipboards. They would crowd out the aisles so legitimate shoppers could not get in.

Worse yet, these rapid-fire screen scrapers do not honor session cookies, so if you are not using URL rewriting to track sessions, each new page request will create a new session. Like a flash mob, pretty soon the capacity problem will turn into a stability problem. The battalion of price checkers could actually knock down the store.

Pragmatic Methodology

Welcome to the Pragmatic Community. We hope you've enjoyed this title.

Do you need to get software out the door? Then you want to see how to *Ship It!* with less fuss and more features.

And if you want to improve your approach to programming, take a look at the pragmatic, effective, *Practices of an Agile Developer*.

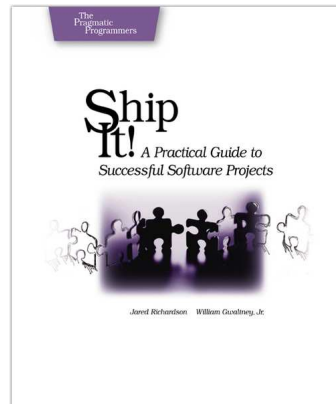
Ship It!

Page after page of solid advice, all tried and tested in the real world. This book offers a collection of tips that show you what tools a successful team has to use, and how to use them well. You'll get quick, easy-to-follow advice on modern techniques and when they should be applied. **You need this book if:**

- you're frustrated at lack of progress on your project.
- you want to make yourself and your team more valuable.
- you've looked at methodologies such as Extreme Programming (XP) and felt they were too, well, extreme.
- you've looked at the Rational Unified Process (RUP) or CMM/I methods and cringed at the learning curve and costs.
- **you need to get software out the door without excuses.**

Ship It! A Practical Guide to Successful Software Projects

Jared Richardson and Will Gwaltney
(200 pages) ISBN: 0-9745140-4-7. \$29.95
<http://pragmaticprogrammer.com/titles/prj>



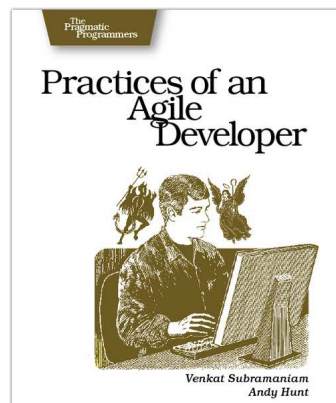
Practices of an Agile Developer

Agility is all about using feedback to respond to change. Learn how to

- apply the principles of agility throughout the software development process
- establish and maintain an agile working environment
- deliver what users really want
- use personal agile techniques for better coding and debugging
- use effective collaborative techniques for better teamwork
- move to an agile approach

Practices of an Agile Developer: Working in the Real World

Venkat Subramaniam and Andy Hunt
(189 pages) ISBN: 0-9745140-8-X. \$29.95
<http://pragmaticprogrammer.com/titles/pad>



The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Release It! Home Page

<http://pragmaticprogrammer.com/titles/mnee>

Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates

<http://pragmaticprogrammer.com/updates>

Be notified when updates and new books become available.

Join the Community

<http://pragmaticprogrammer.com/community>

Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy

<http://pragmaticprogrammer.com/news>

Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragmaticprogrammer.com/titles/mnee.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragmaticprogrammer.com/catalog
Customer Service:	orders@pragmaticprogrammer.com
Non-English Versions:	translations@pragmaticprogrammer.com
Pragmatic Teaching:	academic@pragmaticprogrammer.com
Author Proposals:	proposals@pragmaticprogrammer.com