

Extracted from:

Software Design X-Rays

Fix Technical Debt with Behavioral Code Analysis

This PDF file contains pages extracted from *Software Design X-Rays*, 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 printed versions; the content is otherwise identical.

Copyright © 2018 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

Raleigh, North Carolina

The
Pragmatic
Programmers

Software Design X-Rays

Fix Technical Debt with
Behavioral Code Analysis



Adam Tornhill

edited by Adaobi Obi Tulton

Software Design X-Rays

Fix Technical Debt with Behavioral Code Analysis

Adam Tornhill

The Pragmatic Bookshelf

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 books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at <https://pragprog.com>.

The team that produced this book includes:

Publisher: Andy Hunt

VP of Operations: Janet Furlow

Managing Editor: Brian MacDonald

Supervising Editor: Jacquelyn Carter

Development Editor: Adaobi Obi Tulton

Copy Editor: Candace Cunningham

Indexing: Potomac Indexing, LLC

Layout: Gilson Graphics

For sales, volume licensing, and support, please contact support@pragprog.com.

For international rights, please contact rights@pragprog.com.

Copyright © 2018 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.

Printed in the United States of America.

ISBN-13: 978-1-68050-272-5

Encoded using the finest acid-free high-entropy binary digits.

Book version: P1.0—March 2018

The Dirty Secret of Copy-Paste

While visualizations are important to get the overall picture, the numbers from an X-Ray analysis often provide more details that help uncover design issues. The next figure shows the detailed results from the X-Rays of `LinkTagHelperTest.cs` and `ScriptTagHelperTest.cs`.

Copy-paste detection

↕ Coupled Functions	Coupling ↕ (%)	↕ Commits	Similarity ↓ (%)
LinkTagHelperTest.cs/RunsWhenRequiredAttributesArePresent	44	41	98
ScriptTagHelperTest.cs/RunsWhenRequiredAttributesArePresent			
LinkTagHelperTest.cs/MakeTagHelperOutput	32	41	87
ScriptTagHelperTest.cs/MakeTagHelperOutput			
LinkTagHelperTest.cs/DoesNotRunWhenARequiredAttributeIsMissing	32	41	87
ScriptTagHelperTest.cs/DoesNotRunWhenARequiredAttributeIsMissing			
	--	--	--

The table in the preceding figure presents an interesting finding. We see that several methods have a high degree of *code similarity*. That is, the implementation of several methods is very similar, which is an indication of copied-and-pasted code. For example, the highlighted row shows that there's a code similarity of 98 percent between two methods in different files. The [figure on page 6](#) shows part of the code, and you see that there's a shared test abstraction wanting to get out.

Since these methods are changed together in almost half the commits that touch those files, this is copy-paste that actually matters for your productivity. Let me clarify by revealing a dirty secret about copy-paste.

LinkTagHelperTest.cs	ScriptTagHelperTest.cs
<pre> public void RunsWhenRequiredAttributesArePresent(TagHelperAttributeList attributes, Action<LinkTagHelper> setProperties) { // Arrange var context = MakeTagHelperContext(attributes); var output = MakeTagHelperOutput("link"); var hostingEnvironment = MakeHostingEnvironment(); var viewContext = MakeViewContext(); var globbingUrlBuilder = new Mock<GlobbingUrlBuilder>(new TestFileProvider(), Mock.Of<IMemoryCache>(), PathString.Empty); globbingUrlBuilder.Setup(g => g.BuildUrlList(It.IsAny<string>(), It.IsAny<string>(), It.IsAny<string>())) .Returns(new[] { "/common.css" }); var helper = new LinkTagHelper(hostingEnvironment, MakeCache(), new HtmlTestEncoder(), new JavaScriptTestEncoder(), MakeUrlHelperFactory()) { ViewContext = viewContext, GlobbingUrlBuilder = globbingUrlBuilder.Object }; setProperties(helper); </pre>	<pre> public void RunsWhenRequiredAttributesArePresent(TagHelperAttributeList attributes, Action<ScriptTagHelper> setProperties) { // Arrange var context = MakeTagHelperContext(attributes); var output = MakeTagHelperOutput("script"); var hostingEnvironment = MakeHostingEnvironment(); var viewContext = MakeViewContext(); var globbingUrlBuilder = new Mock<GlobbingUrlBuilder>(new TestFileProvider(), Mock.Of<IMemoryCache>(), PathString.Empty); globbingUrlBuilder.Setup(g => g.BuildUrlList(It.IsAny<string>(), It.IsAny<string>(), It.IsAny<string>())) .Returns(new[] { "/common.js" }); var helper = new ScriptTagHelper(hostingEnvironment, MakeCache(), new HtmlTestEncoder(), new JavaScriptTestEncoder(), MakeUrlHelperFactory()) { ViewContext = viewContext, GlobbingUrlBuilder = globbingUrlBuilder.Object }; setProperties(helper); </pre>

Only minor variations
in setup

Clone Detection 101

Copy-paste detectors are underused in our industry despite the obvious risks and costs associated with software clones. The pioneering work in this field was done by Brenda Baker in her seminal paper [On Finding Duplication and Near-Duplication in Large Software Systems \[Bak95\]](#). There are several clone-detection algorithms to choose from, all with different trade-offs. The simplest algorithms look for common text patterns in the code. More elaborate clone detectors compare the *abstract syntax trees* to detect structural similarities and yield better precision.⁵



These algorithms are implemented by several open and commercial clone detectors. For example, I use *Clone Digger* for Java and Python,⁶ and *Simian* for .NET code.⁷ It's also an interesting learning experience to implement a simple clone detector yourself. The *Rabin-Karp algorithm* is a good starting point (see [Efficient randomized pattern-matching algorithms \[KR87\]](#)).

In the previous chapter we saw that low-quality code isn't necessarily a problem. Now we'll challenge another wide-spread belief by asserting that copy-paste code isn't always bad.

5. https://en.wikipedia.org/wiki/Abstract_syntax_tree
6. <http://clonedigger.sourceforge.net/>
7. <http://www.harukizaemon.com/simian/>

Like everything else, the relative merits of a coding strategy depend on context. Copy-paste isn't a problem in itself; copying and pasting may well be the right thing to do if the two chunks of code evolve in different directions. If they don't—that is, if we keep making the same changes to different parts of the program—that's when we get a problem.

This is important since research on the topic estimates that in your typical codebase, 5–20 percent of all code is duplicated to some degree. (See [On Finding Duplication and Near-Duplication in Large Software Systems \[Bak95\]](#) and [Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics \[MLM96\]](#) for studies of commercial software systems.) That's a lot of code. We can't inspect and improve all of it, nor should we. Just as with hotspots, we need to prioritize the software clones we want to get rid of. The change coupling analysis combined with a code-similarity metric is a simple and reliable way to identify the software clones that really matter for your productivity and code quality. Again, note that this is information you cannot get from the code alone; we need a temporal perspective to prioritize the severity of software clones.

Once we've identified the software clones that matter, we want to refactor them. We typically approach that refactoring by extracting the repeated pattern into a new method and parameterizing it with the concept that varies. This makes the code a little bit cheaper to maintain as our temporal dependency disappears. We also get less code, and that's good because all code carries a cost. It's a *liability*.⁸ The more code we can remove while still getting the job done, the better. Killing software clones is a good starting point here.

8. <https://blogs.msdn.microsoft.com/eleee/2009/03/11/source-code-is-a-liability-not-an-asset/>