Extracted from:

Your Code as a Crime Scene

Use Forensic Techniques to Arrest Defects, Bottlenecks, and Bad Design in Your Programs

The Pragmatic Bookshelf

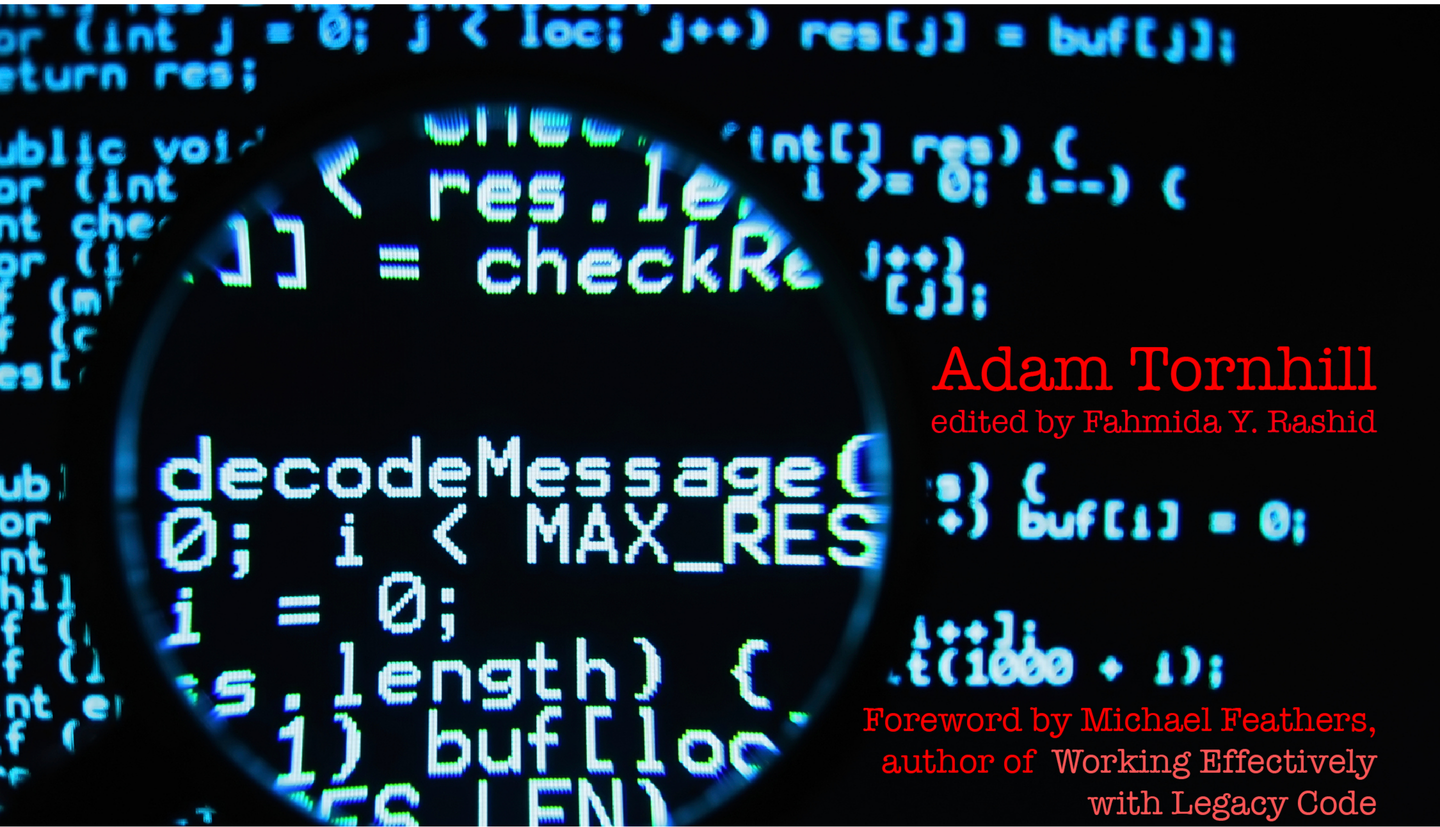
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Investigator:	
Date:	
Case #:	
Location:	

Your Code As a Crime Scene

Use Forensic Techniques to Arrest Defects, Bottlenecks, and Bad Design in Your Programs



Adam Tornhill

edited by Fahmida Y. Rashid

Foreword by Michael Feathers, author of *Working Effectively with Legacy Code*

Your Code as a Crime Scene

Use Forensic Techniques to Arrest Defects,
Bottlenecks, and Bad Design in Your Programs

Adam Tornhill

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Fahmida Y. Rashid (editor)
Potomac Indexing, LLC (indexer)
Cathleen Small (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

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

Printed in the United States of America.
ISBN-13: 978-1-68050-038-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2015

CHAPTER 8

Detect Architectural Decay

In the previous chapter, you learned how temporal coupling detects hidden dependencies in your system. Now it's time to learn how to perform an analysis of temporal coupling on your code.

In this chapter, we'll analyze two systems of different sizes. The smaller project shows how temporal coupling can still give us fresh insights into the design even when we're very familiar with the code. The larger project shows how to detect architectural decay so that we can make improvements early in the process. You'll also see that the structures you're working with aren't always aligned with the official architecture.

Let's see how information-rich the change patterns in a system can be for our analysis.

Support Your Redesigns with Data

I once worked on a project with severe problems in its database access. Changes were awkward, they took longer than they should, and bugs swarmed like mosquitoes at a Swedish barbecue.

Learning from mistakes is important, so I decided to redesign the worst parts of the database layer. Something interesting happened. Even though the database layer was in better shape, developers still complained about how fragile and unstable it was. Changes still broke unrelated features. What went wrong? Did I mess up?

While the database improved, it turned out that wasn't where the true problems were. The database was just the messenger subtly warning us about temporal coupling (and we shot the messenger).

Other parts of the system unexpectedly depended on the data storage. The true problem was in automatic system tests. A minor change to the data format triggered a cascade of changes to the test scripts. This wasn't obvious because the scripts didn't explicitly call the database code.

After reading the previous chapter, you now can see how a temporal coupling analysis could've helped us find this problem earlier. Redesigns are about minimizing risk and prioritizing areas of code that have the largest impact on the work we're doing now. Get it wrong like we did, and you will miss an opportunity to make genuine improvements to your code. Let's see how we can use temporal coupling to avoid these mistakes.

Analyze Temporal Coupling

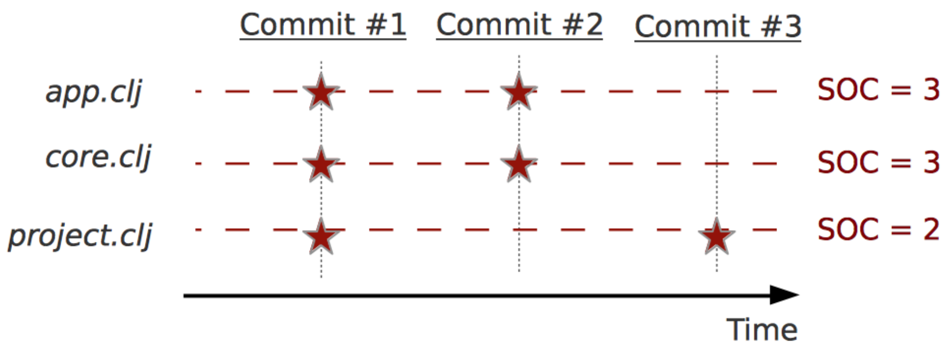
In Chapter 7, *Treat Your Code As a Cooperative Witness*, on page ?, we said that temporal coupling can be an interview tool for your codebase. The first step in an interview is to know who you should talk to.

Let's use *sum of coupling* analysis to find our first code witness.

Use Sum of Coupling to Identify the Modules to Inspect

You've already seen that there are different reasons for modules to be coupled. Some couples, such as a unit and its unit test, are valid. So modules with the highest degree of coupling may not be the most interesting to us. Instead, we want modules that are architecturally significant. A sum of coupling analysis finds those modules.

Sum of coupling looks at how many times each module has been coupled to another one in a commit and sums it up. For example, in the following figure, you see that module app.clj changed with both core.clj and project.clj in Commit #1, but just with core.clj in Commit #2. Its sum of coupling is three.



The module that changes most frequently together with others must be important and is a good starting point for an investigation. Let's try it out on

Code Maat by reusing the the logfile we mined in Chapter 3, *Creating an Offender Profile*, on page ?.

Move into the top-level directory in your Code Maat repository and type the following command:

```
prompt> maat -l maat_evo.log -c git -a soc
entity,soc
src/code_maat/app/app.clj,105
test/code_maat/end_to_end/scenario_tests.clj,97
src/code_maat/core.clj,93
project.clj,74
...
```

You can see that this command uses the same format we saw in the earlier hotspot analysis. The only difference is that we're requesting -a soc (sum of coupling) instead.

We see that app.clj changes the most with other modules. Let's keep an eye on app.clj as we dive deeper.

Measure Temporal Coupling

At this point you know that app.clj is the module with the most temporal coupling. The next step is to find out which modules it's coupled to. We use Code Maat for this analysis:

```
prompt> maat -l maat_evo.log -c git -a coupling
entity,coupled,degree,average-revs
src/code_maat/parsers/git.clj,test/code_maat/parsers/git_test.clj,83,12
src/code_maat/analysis/entities.clj,test/code_maat/analysis/entities_test.clj,76,7
src/code_maat/analysis/authors.clj,test/code_maat/analysis/authors_test.clj,72,11
...
```

The command line is identical to the one you just used, with the exception that we're requesting -a coupling instead. The resulting .CSV output contains plenty of information:

1. *entity*: This is the name of one of the involved modules. Code Maat always calculates pairs.
2. *coupled*: This is the coupled counterpart to the *entity*.
3. *degree*: The degree specifies the percent of shared commits. The higher the number, the stronger the coupling. For example, git.clj and git_test.clj change together in 83 percent of all commits.
4. *average-revs*: Finally, we get a weighted number of total revisions for the involved modules. The idea here is that we can filter out modules with too few revisions to avoid bias.

You see a typical pattern in the output: each unit changes together with its unit test (e.g. git.clj and git_test.clj, entities.clj and entities_test.clj).

This kind of temporal coupling is expected and not a problem. Code Maat was developed with test-driven development, so I'd say that getting any other result would've been a problem. Just plain old physical coupling—nothing too exciting here.

Things get interesting a bit farther down:

```
prompt> maat -l maat_evo.log -c git -a coupling
...
src/code_maat/app/app.clj,src/code_maat/core.clj,60,23
src/code_maat/app/app.clj,test/code_maat/end_to_end/scenario_tests.clj,57,23
...
```

We see that app.clj changed with core.clj 60 percent of the time and with scenario_tests.clj 57 percent of the time. There's no way to tell why just from the names alone, but 60 percent is a high degree of coupling. We are talking about every second (or so), change in app.clj triggering a change in two other modules. That can't be good. Let's investigate why.

Check Out the Evolution Radar

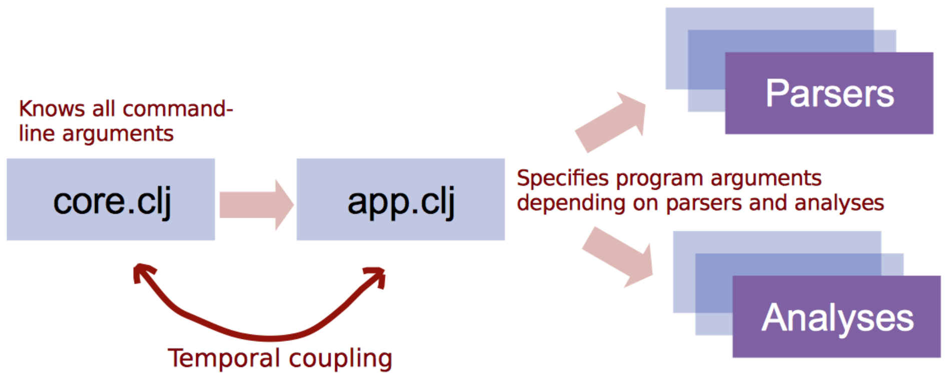
In a large codebase, a temporal coupling analysis sparks an explosion of data. Code Maat resolves that by allowing us to specify optional thresholds. The research tool *Evolution Radar*[1] takes a different approach and lets us zoom in and out to the level of detail we're interested in. So check out the tool and take inspiration.

Investigate Temporal Couples

Once we make such a finding, we need to drill down into the code. Because all changes are recorded in our version-control system, we can perform a diff on the modules. I'd recommend focusing on the shared commits and look for recurring modification patterns within those commits.

Code Maat is written in Clojure. Although an exciting language, it's far outside the scope of this book. So let's stay with temporal coupling, and allow me to walk you through the design to spot the problems.

1. http://www.inf.usi.ch/phd/dambros/tools/evoradar.php


Knows all command-line arguments
core.clj
app.clj
Temporal coupling
Specifies program arguments depending on parsers and analyses
Parsers
Analyses

I'm a bit ashamed to admit that core.clj is the command-line interface of Code Maat. (I changed it later to a better name.) It parses the arguments you give it, converts them to a Clojure representation, and forwards them to app.clj.

app.clj glues the program together by mapping the given arguments to the correct invocations of parsers, analyses, and output formats. As you can see, the program arguments cause the coupling; every time a new argument is added, two distinct modules have to evolve to know about it.

So, your first takeaway is actually a reminder about the power of names that you learned about in Chapter 5, *Judge Hotspots with the Power of Names*, on page ?. With proper naming, we'd have a better entry point for our manual code inspection. Second, we failed to encapsulate a concept that varies. If we extract the knowledge of all command-line arguments from app.clj, we break the coupling and make the code easier to evolve and maintain.

Use Temporal Coupling for Design Insights

The analysis on Code Maat illustrates how we can use temporal coupling analysis on small projects. Code Maat (which I wrote to learn Clojure during my daily commute) is a single-developer project with less than 2,000 lines of code.

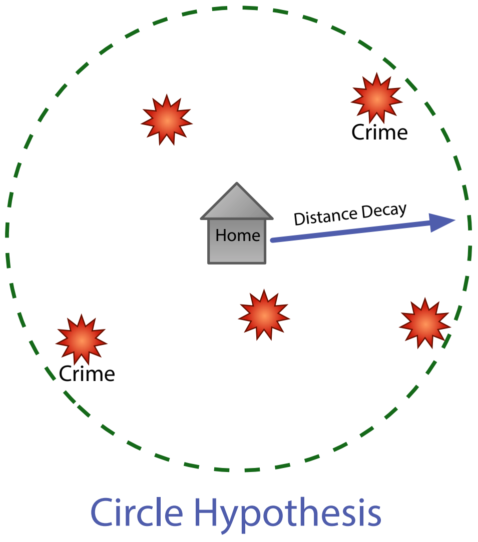
Such small projects don't need a hotspot analysis. We already know which modules are hard to change. Temporal coupling is different because it provides insights into our design. We get active feedback on our work so that we can spot improvements we hadn't even thought of.

Keep Your Temporal Coupling Algorithms Simple

The algorithm we've used so far isn't the only kid in town. Temporal coupling means that some entities change together over time. But there isn't any formal definition of what *change together* means. In research papers, you'll find several alternative measures.

One typical alternative adds the notion of time to the algorithm; the degree of coupling is weighted by the age of the commits. The idea is to prioritize recent changes over changes in the more distant past. A relationship thus gets weaker with the passage of time. However, as you'll see soon when we discuss software defects, a time parameter doesn't necessarily improve the metric.

The algorithm that Code Maat implements, the percent of shared commits, is chosen because when faced with several alternatives that seem equally good, simplicity tends to win. The Code Maat measure is straightforward to implement and, more importantly, intuitive to reason about and verify.



Interestingly enough, simplicity may win in criminal investigations, too. In a fascinating study, researchers trained people on two simple heuristics for predicting the home location of criminals:

- *Distance decay*: Criminals do not travel far from their homes to offend. Thus, crimes are more likely closer to an offender's home and less likely farther away.
- *Circle hypothesis*: Many serial offenders live within a circle defined by the criminals' two farthest crime locations.

Using these simple principles allowed the participants to predict the likely home location of serial killers with the same accuracy as a sophisticated geographical profiling system. (See *Applications of Geographical Offender Profiling [CY08]*.) We build the techniques in this book on the same kind of simplicity.

Know the Limitations of Temporal Coupling

Our simple definition of temporal coupling as modules that change in the same commit works well. Often, that definition takes us far enough to identify unexpected relationships in our system. But in larger organizations, our measure is too narrow. When multiple teams are responsible for different parts of the system, the temporal period of interest is probably counted in days or even weeks. We'll address this problem in Chapter 12, *Discover Organizational Metrics in Your Codebase*, on page ?, where you'll learn to group multiple commits into a logical change set based on a custom timespan.

Another problem with the measure is that we're limited to the information contained in commits. We may miss important coupling relationships that occur *between* commits. The solution to this problem requires hooks into our text editors and our IDE to record precise information on our code interactions. Tools like that are under active research.

Yet another bias is moving and renaming modules. While version-control systems track renames, Code Maat does not. (If I ever turn Code Maat into a commercial product, that's a feature I'd add.) It sounds more limiting than it actually is: problematic modules tend to remain where they are. The good thing is that because we lose some of the supporting information, the results we get are more likely to point to true problems. Consider renaming the module as a reset switch triggered by refactoring.

Catch Architectural Decay

Temporal coupling has a lot of potential in software development. We can spot unexpected dependencies and suggest areas for refactoring.

Temporal coupling is also related to software defects. There are multiple reasons for that. For example, a developer may forget to update one of the (implicitly) coupled modules. Another explanation is that when you have multiple modules whose evolutionary lifelines are intimately tied, you run the risk of unexpected feature interactions. You'll also soon see that temporal coupling often indicates architectural decay. Given these reasons, it's not surprising that a high degree of temporal coupling goes with high defect rates.

Temporal Coupling and Software Defects

Researchers found that different measures of temporal coupling outperformed traditional complexity metrics when it came to identifying the most defect-prone modules (see *On the Relationship Between Change Coupling and Software Defects [DLR09]*). What's surprising is that temporal coupling seems to be particularly good at spotting more severe bugs (major/high-priority bugs).

The researchers made another interesting finding when they compared the bug-detection rate of different coupling measures.

Temporal Coupling and Software Defects

Some measures included time awareness, effectively down-prioritizing older commits and giving more weight to recent changes. The results were counterintuitive: the simpler sum of coupling algorithm that you learned about in this chapter performed better than the more sophisticated time-based algorithms.

My guess is that the time-based algorithms performed worse because they're based on an assumption that isn't always valid. They assume code gets better over time by refactorings and focused improvements. In large systems with multiple developers, those refactorings may never happen, and the code keeps on accumulating responsibilities and coupling. Using the techniques in this chapter, we have a way to detect and avoid that trap. And now we know how good the techniques are in practice.

Enable Continuing Change

Back in Chapter 6, *Calculate Complexity Trends from Your Code's Shape*, on page ?, we learned about Lehman's law of increasing complexity. His law states that we must continuously work to prevent a "deteriorating structure" of our programs as they evolve. This is vital because every successful software product will accumulate more features.

Lehman has another law, the *law of continuing change*, which states a program that is used "undergoes continual change or becomes progressively less useful" (see *On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle [Leh80]*).

There's tension between these two laws. On one hand, we need to evolve our systems to make them better and keep them relevant to our users. At the same time, we don't want to increase the complexity of the system.

One risk with increased complexity is features interacting unexpectedly. We make a change to one feature, and an unrelated one breaks. Such bugs are notoriously hard to track down. Worse, without an extensive regression test suite, we may not even notice the problem until later, when it's much more expensive to fix.

To prevent horrors like that from happening in our system, let's see how we can use temporal coupling to track architectural problems and stop them from spreading in our code.

Identify Architecturally Significant Modules

In the following example, we're going to analyze a new codebase. Craft.Net[2] is a set of Minecraft-related .NET libraries. We're analyzing this project because it's a fairly new and cool project of suitable size with multiple active developers.

To get a local copy of Craft.Net, clone its repository:

```
prompt> git clone https://github.com/SirCmpwn/Craft.Net.git
```

Let's perform the trend analysis step by step so that we can understand what's happening. Each step is nearly identical; the time period is the only thing that changes. We can automate this with a script later. Let's find the first module to focus on.

Move into the Craft.Net directory and perform a sum of coupling analysis:

```
prompt> git log --pretty=format:'[%h] %an %ad %s' --date=short --numstat \
 --before=2014-08-08 > craft_evo_complete.log
prompt> maat -l craft_evo_complete.log -c git -a soc
entity,soc
Craft.Net.Server/Craft.Net.Server.csproj,685
Craft.Net.Server/MinecraftServer.cs,635
Craft.Net.Data/Craft.Net.Data.csproj,521
Craft.Net.Server/MinecraftClient.cs,464
...
```

Notice how we first generate a Git log and then feed that to Code Maat. Sure, there's a bit of Git magic here, but nothing you haven't seen in earlier chapters. You can always refer back to Chapter 3, *Creating an Offender Profile*, on page ?, if you need a refresher on the details.

When you look for modules of architecural significance in the results, ignore the C# project files (.csproj). The first real code module is MinecraftServer.cs. As you see, that class has the most cases of temporal coupling to other modules. Looks like a hit.

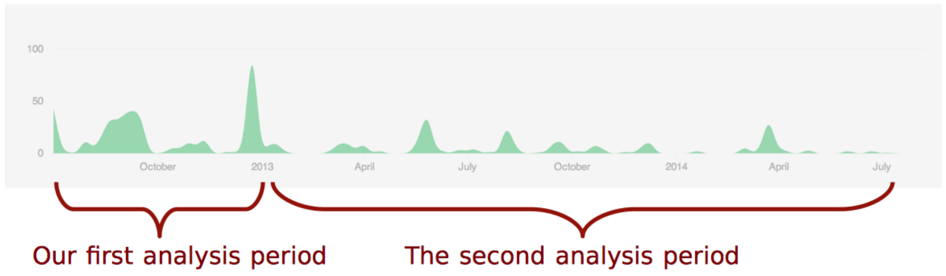
The name of our code witness, MinecraftServer, is also an indication that we've found the right module; a MinecraftServer sounds like a central architectural part of any, well, Minecraft server. We want to ensure that the module stays on track over time. Here's how we do that.

2. https://github.com/SirCmpwn/Craft.Net

Perform Trend Analyses of Temporal Coupling

To track the architectural evolution of the MinecraftServer, we're going to perform a trend analysis. The first step is to identify the periods of time that we want to compare.

The development history of Craft.Net goes back to 2012. There was a burst of activity that year. Let's consider that our first development period.



To perform the coupling analysis, let's start with a version-control log for the initial period:

```
prompt> git log --pretty=format:'[%h] %an %ad %s' --date=short --numstat \
  --before=2013-01-01 > craft_evo_130101.log
```

We now have the evolutionary data in craft_evo_130101.log. We use the file for coupling analysis, just as we did earlier in this chapter:

```
prompt> maat -l craft_evo_130101.log -c git -a coupling > craft_coupling_130101.csv
```

The result is stored in craft_coupling_130101.csv. That's all we need for our first analysis period. We'll look at it in a moment. But to spot trends we need more sample points.

In this example, we'll define the second analysis period as the development activity in 2013 until 2014. Of course, we could use multiple, shorter periods, but the GitHub activity shows that period contains roughly the same amount of activity. So for brevity, let's limit the trend analysis to just two sample points.

The steps for the second analysis are identical to the first. We just have to change the filenames and exclude commit activity before 2013. We can do both in one sweep:

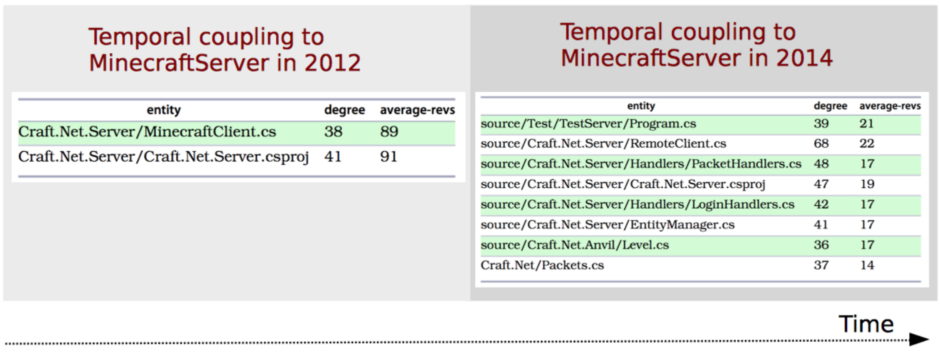
```
prompt> git log --pretty=format:'[%h] %an %ad %s' --date=short --numstat \
  --after=2013-01-01 --before=2014-08-08 > craft_evo_140808.log
prompt> maat -l craft_evo_140808.log -c git -a coupling > craft_coupling_140808.csv
```

We now have two sampling points at different stages in the development history. Let's investigate them.

Investigate the Trends

When we perform an analysis of our codebase, we want to track the evolution of all interesting modules. To keep this example short, we'll focus on one main suspect as identified in the sum of coupling analysis: the MinecraftServer module. So let's filter the data to inspect its trend.

I opened the result files, craft_coupling_130101.csv and craft_coupling_140808.csv, in a spreadsheet application and removed everything but the modules coupled to MinecraftServer to get the filtered analysis results.



Temporal coupling to MinecraftServer in 2012

entity	degree	average-revs
Craft.Net.Server/MinecraftClient.cs	38	89
Craft.Net.Server/Craft.Net.Server.csproj	41	91

Temporal coupling to MinecraftServer in 2014

entity	degree	average-revs
source/Test/TestServer/Program.cs	39	21
source/Craft.Net.Server/RemoteClient.cs	68	22
source/Craft.Net.Server/Handlers/PacketHandlers.cs	48	17
source/Craft.Net.Server/Craft.Net.Server.csproj	47	19
source/Craft.Net.Server/Handlers/LoginHandlers.cs	42	17
source/Craft.Net.Server/EntityManager.cs	41	17
source/Craft.Net.Anvil/Level.cs	36	17
Craft.Net/Packets.cs	37	14

There's one interesting finding in 2012: the MinecraftServer.cs is coupled to MinecraftClient.cs. This seems to be a classic case of temporal coupling between a producer and a consumer of information, just as we discussed in *Understand the Reasons Behind Temporal Dependencies*, on page ?. When we notice a case like that, we want to track it.

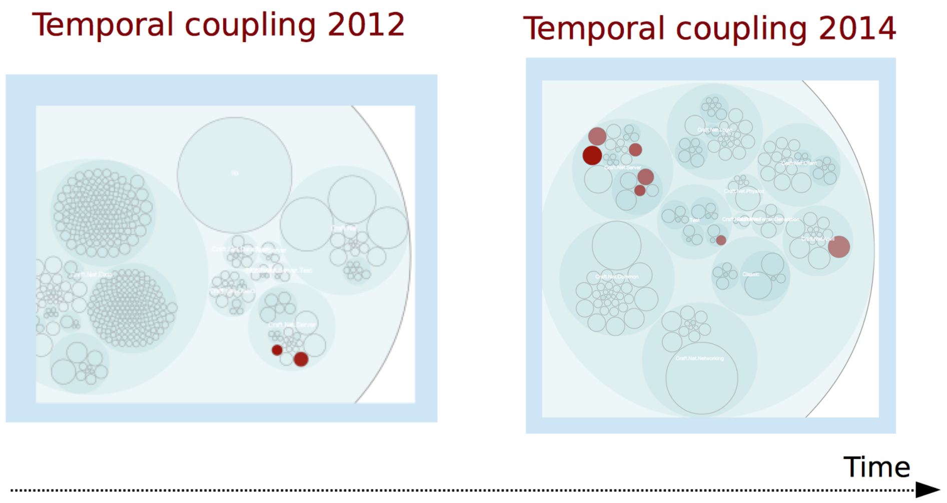
Forward to 2014. The coupling between server and client isn't present a year and a half later, but we have other problems. As you can see, the MinecraftServer has accumulated several heavy temporal dependencies compared to its cleaner start in the initial development period.

When that happens, we want to understand why and look for places to improve. Let's see how.

React to Structural Trends

The following figure presents a visual view of the architectural decay we just spotted. It's the same enclosure diagrams we used back in Chapter 4, *Analyze*

Hotspots in Large-Scale Systems, on page ?, but now they're illustrating the modules coupled to MinecraftServer at two different points in time.



The obvious increase in temporal coupling says there are more modules that have to change with the MinecraftServer in 2014 than earlier in the development history. Note that the number of coupled modules isn't a problem in itself. To classify a temporal coupling, you need to look at the architectural boundaries of the coupled modules.

When the coupled modules are located in entirely different parts of the system, that's structural decay. Our data in the trend table on page 15 shows one obvious case in 2014: Craft.Net.Anvil/Level.cs.

That coupling, together with the growing trend, suggests that our MinecraftServer has been accumulating responsibilities.

Remember how we initially discussed code changes that seem to break unrelated features? The risk with the trend we see here is that it leaves the system vulnerable to such unexpected feature interactions.

If allowed to grow, increased temporal coupling leads to fragile systems. As you saw earlier, temporal coupling has a high correlation with defects. That's why we want to integrate the analysis into a team's workflow. Let's see how.

Use a Storyboard to Track Evolution

The trend analysis we just performed is reactive. It's an after-the-fact analysis. The results are useful because they help us improve, but we can do even better.

With more activity, you want more sample points. So why not make it a habit to perform regular analyses on the projects you work on?

If you work iteratively, perform the analyses in each iteration. This approach has several advantages:

- You spot structural decay immediately.
- You see the structural impact of each feature as you work with it.
- You make your evolving architecture visible to everyone on the team.

I recommend that you visualize the result of each analysis, perhaps as in Figure , , on page 16, print them all out, and put them on a storyboard for each iteration.

Think back to our initial example on automated tests with nasty implicit couplings to a database. With an evolutionary storyboard, we'd spot the decay as soon as we noticed the pattern—a few iterations at most, and that's it.

An iterative trend analysis of temporal coupling is a low-tech approach that helps us improve. It also has the notable advantage of putting focus on the right parts of the system. As such, an evolutionary storyboard is invaluable to complement and stimulate design discussions with peers.

If you find as much promise in this approach as I do, check out the article *Animated Visualization of Software History using Evolution Storyboards [BH06]*. The authors are the pioneers of the storyboard idea, and their paper shows some cool animations of growing systems.

Scale to System Architectures

This chapter started with a sum of coupling analysis. With that analysis, we identified the architecturally significant modules. We also noted that those modules aren't necessarily the ones we'd expect from our formal specification or design.

After that, we saw how a temporal coupling analysis gives us information we cannot extract from the code alone. It's information that gives us design insights and refactoring directions. When used as a refactoring guide, we can assume that modules that have changed together in the past are likely to continue to change together. We looked at that in our second analysis of Craft.Net.

You then learned to spot architectural decay by applying trend analyses to the coupling. Finally, you learned how to track potential decay with an evolutionary storyboard.

With temporal coupling behind us, we've completed our initial set of analysis methods. Before we move on to discuss teams and social dynamics, we're going to build on what we've learned so far. Until now, we have limited the analyses to individual files. But now you'll see how temporal coupling scales to system architecture, too. That will be exciting!