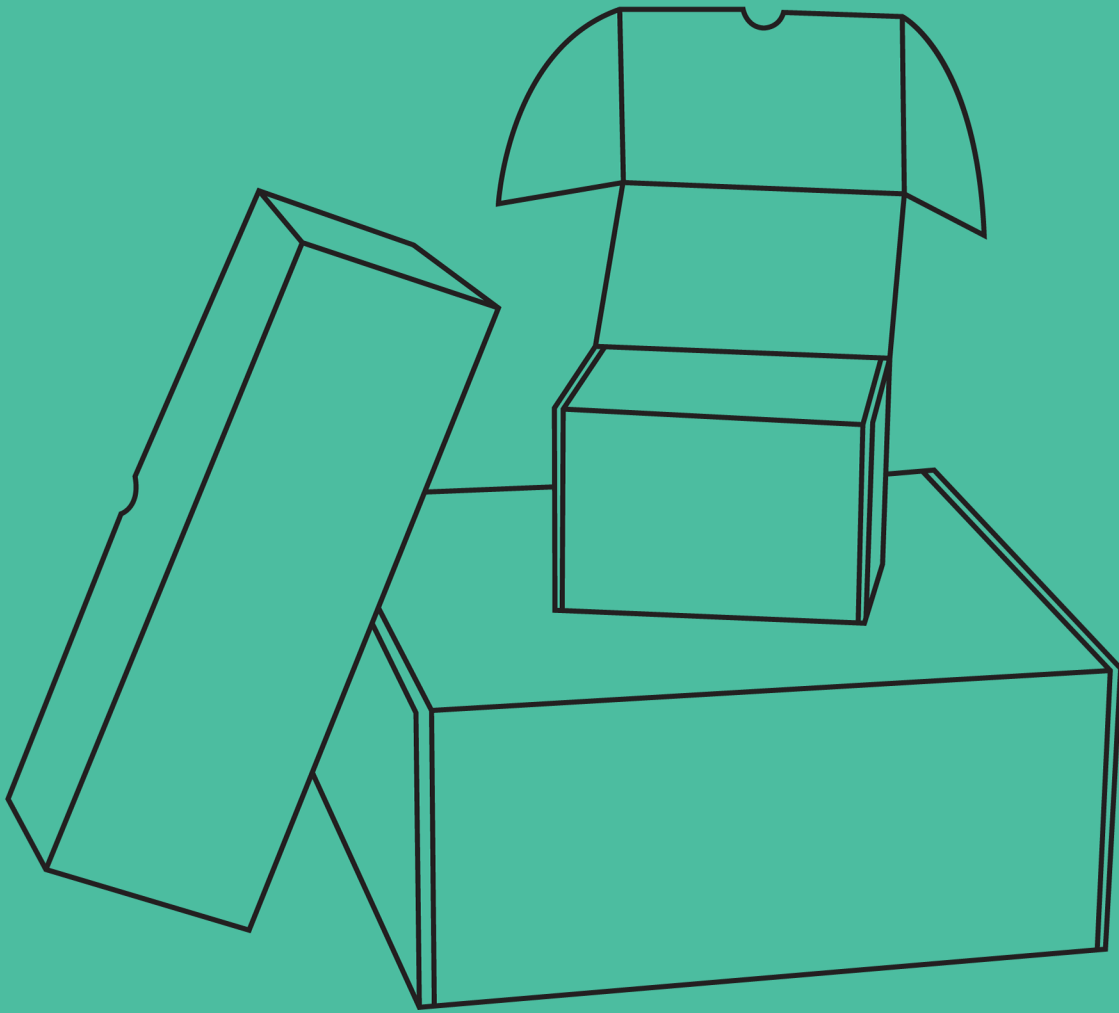


GRADUAL MODULARIZATION FOR RUBY AND RAILS

IMPROVE COLLABORATION,
SYSTEM DESIGN, AND FLEXIBILITY



STEPHAN HAGEMANN

Gradual Modularization for Ruby and Rails

Improve collaboration, system design, and flexibility

Stephan Hagemann

This book is for sale at <http://leanpub.com/package-based-rails-applications>

This version was published on 2023-02-06



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2020 - 2023 Stephan Hagemann

Contents

Acknowledgments	i
1. Introduction to Package-based Rails Applications	1
1.1 The inescapable challenges with components	1
1.2 Component-based Challenges	3
1.2.1 External dependency drift	3
1.2.2 Configuration drift	7
1.2.3 Global impact of local changes	9
2. Getting started with packaging in a Rails application	10
2.0.1 Sportsball	10
2.1 Let's get started	11
2.2 From one to many packages	13
2.2.1 Packwerk packages vs gems as components	14
2.2.2 A completely packagified application	16
2.2.3 Introducing app/packages	18
2.2.4 Getting things to work	21
2.2.5 Making packwerk packages	22
2.2.6 Visualizing dependencies	23
2.3 Accepting the intended dependencies - explicitly adding package dependencies	25
2.4 Removing the circular dependencies - creating rails_shims package	27
2.5 Analyzing and removing root dependencies (almost) - Aligning app and test code	32
2.6 Removing all dependencies from root	35
2.7 Recapping differences and similarities to components	38
3. Gradual Modularization: A New Model for Application Composition	40
3.1 Gradual Typing as a Role Model	40
3.2 Commonalities of previous modularization approaches	41
3.3 Gradularity - Modularization Game Changer	42
3.3.1 Enforce Visibility	42
3.3.2 Allow privacy circumvention	43
3.3.3 API versioning	44
3.3.4 Enforcing a package namespace	45
3.3.5 Enforcing a separate database	46

CONTENTS

3.3.6	Enforcing a typed API	46
3.3.7	Enforcing proper documentation	47
3.3.8	Configurable failure modes	47
3.3.9	Here is my code, run it in the cloud for me, I do not care how	48
3.3.10	Analyze to modularize the right stuff	49
3.3.11	Caveats	50
3.3.12	Managing the caveats	52
4.	Making Components work with Packages	54
4.1	Engines	54
4.2	Gems	57
4.2.1	Converting the gem fully into an engine	59
4.2.2	Converting the gem into an engine (as little as possible!)	60
5.	Dependency Violation Management Refactorings	65
5.1	Do absolutely nothing	66
5.2	Accept the dependency	68
5.3	Merge the two packages	69
5.4	Split the violated package	71
5.5	Move the code between packages	72
5.6	Duplicate the functionality	73
5.7	Abstract away the dependency	74
5.8	Dependency injection	75
5.8.1	Naive dependency injection	76
5.8.2	Dependency injection with typing	79
5.8.3	Dependency injection with typing and type abstraction	84
5.9	Dependency Location - The service locator pattern	95
5.10	Emit and listen to events	99
5.10.1	To event or not to event - and where?	100
5.10.2	Calculating backend responses with events	101
5.10.3	Determining where to send responses	103
5.10.4	Wiring backend and frontend together	106
5.10.5	The package effect of eventing	108
5.11	Beyond dependency refactorings	109
6.	Privacy Violation Management Refactorings	110
6.1	Packages without consumers should protect their public API	112
6.1.1	The root package	112
6.1.2	The UI and admin packages	113
6.1.3	packages/prediction_interface and interface classes	114
6.2	Expose existing service classes	116
6.3	Expose existing service classes	119
6.4	ActiveRecord handled naively	120
6.5	Hide ActiveRecord	122

CONTENTS

6.5.1	A new form of Team	123
6.5.2	A TeamRepository to manage teams	127
6.5.3	Migrating consumers to the new API	131
7.	Ruby at Scale	137
7.1	The giants who's shoulders we stand on	137
7.2	Gusto's Ruby at Scale work supporting Gradual Modularity	138
7.3	Stimpack	138
7.4	Code Teams	140
7.5	ParsePackwerk	142
7.6	Code ownership	143
7.7	PackageProtections	144
7.8	Danger::Packwerk	146
7.9	ModularizationStatistics	147
7.10	Honorable mentions: More of Gusto's Ruby at Scale	148
7.11	The Structure of Ruby at Scale	149
8.	Measuring Modularization Progress	150
8.1	Basic technical measures of modularization progress	151
8.1.1	How modularized is the app?	151
8.1.2	Are there an adequate amount of packages?	152
8.1.3	How much are packages owned by only one team?	154
8.1.4	How discerning is the package dependency graph?	155
8.1.5	Basic technical measures create a path	159
8.2	Refining technical measures of modularization progress	160
8.2.1	How many packages protect themselves from breaking their own stated dependencies?	160
8.2.2	How many packages protect themselves against privacy violations? . .	162
8.2.3	How much are packages operating in their own namespace?	163
8.2.4	How many packages protect themselves against untyped APIs?	163
8.3	Growing the list of technical measures your organization can support	164
8.3.1	Adding your own protections	165
8.4	Sociotechnical measures that go beyond modularization	167
8.4.1	High-level vs low-level metrics (and tests)	168
8.4.2	The DevOps Research and Assessment metrics	169
9.	Creating Modularization Progress	174
9.1	Just do it	174
9.2	Beer-athon move files	174
9.3	Hackathons	174
9.4	Kudos	174
9.5	Dedicated Project Team	174
9.6	Champions	174
9.7	Vision	174

Acknowledgments

<https://github.com/Shopify/packwerk/graphs/contributors> - For making and open sourcing packwerk

Ngan Phan - For many deep technical and strategic discussions and for helping me understand Rails much better

Alex Evanczuk - For challenging me on lots of ideas, for adding many of his own, for feedback on early versions of this book, for making me understand Sorbet a little bit better

James Shkolnik - For many great architecture discussions and early feedback

Sarah Tipton - For the time and for proof-reading near illegible drafts

Kent Beck - For hearing me out and giving space to even the most outlandish ideas

1. Introduction to Package-based Rails Applications

Shopify released a new tool to the Ruby open-source community in October of 2020. It is called *packwerk* (find it at <https://github.com/shopify/packwerk>). Packwerk “is a Ruby gem used to enforce boundaries and modularize Rails applications.”

Packwerk does a lot of digging in Ruby code to understand the structure of an application and it didn’t initially run on the codebase I was working on because of edge cases in parsing Ruby files (if there is such a thing in programming languages at all, it definitely is a thing in Ruby). After forking the gem and making some parsing more defensive, I got it to run in the codebase and have been exploring the tool and its implications ever since.

I have since discussed the relationship between component-based Rails and what a tool like packwerk offers with many folks. And I myself have done a lot of soul searching as to what this means for my past work.

To not bury the lead: this is huge. How huge? You can pretty much forget what I wrote in large parts of component-based Rails huge. And I am sooooo excited for the positive implications this tool and others like it will have on our big Rails apps today, Rails in the coming years, and the generation of coming software engineers who are taking their first professional steps with Ruby and Rails. As I said: I believe THIS IS HUGE. There. I yelled it.

1.1 The inescapable challenges with components

I started working on *component-based Rails applications* (CBRA) in 2010 and started calling them that in 2012 when I started to work on the book that has since been published by Pearson. I felt passionate about this work because I wanted to add a tool to the tool belts of Ruby and Rails engineers to manage the growth of their applications. Since then I have spoken at many conferences, at private workshops for companies, and have used the ideas in my work. Writing and publishing the book on leanpub took me over a year. When I got the chance to publish the book with Pearson rewriting and publishing the book took me almost another year and a half because I foolishly had agreed to update what I written from Rails 4 to Rails 5.

What makes me write this book today is literally embodied in that problem. It took me a year and a half (twice!) to publish a book on component-based Rails applications. Now, that could be because it is a technical topic and those are tough to write because of the amount of code samples. Those have to be exact so people can follow them. In my case, almost all the code samples built on each other. For a mistake I made in the beginning, which happened a couple of times, I not only had to go back and fix it, I also had to roll that change forward to all other code samples in most cases.

But code samples weren't the only reason it took so long to write Component-based Rails applications, especially the second time around. There are a couple of big problems with component-based Rails that also contributed.

For component-based Rails, we are exploiting the fact that with bundler and gems we have a mechanism to specify dependencies. When we use this to wrap a gem around a part of code we are saying that this code can be tested in isolation, that can be configured in isolation - this is where we get our improved method of working with large applications from.

But we are using gems and bundler. Or one could say, as I have said many times, we are *exploiting* gems and bundler. We are deliberately using the concept of a gem for only one of the two parts it was made for. Gems were made to package *and* distribute cohesive parts of code that could do well as generic libraries for many use cases. For component-based Rails we are dropping the use of the second (distribute) and only use the first (packaging). One could go even further and say that *packaging* is also not what we are actually after: We really want *modularization*; a way to substructure our applications to reason about parts more independently.

Despite wanting and using only one aspect of gems, we get all of them. Gemspecs ask us to add all sorts of metadata to the gem, which is used when the gem is published. We don't really need that for components. With gemspecs we specify the *internal* dependencies we are interested in, but also *external* dependencies because we have to. So, in addition to the stuff we don't need, there is a bunch of stuff that gets duplicated. The Gemfile in the root of the application now does double duty and specifies both internal and external dependencies, but for *all* of the app (making every component spec somewhat redundant).

There is some value in a component stating which external dependencies it relies on. It gives a developer a better understanding of how the functionality of the gem is achieved and what externally provided functionality they might tap into in the future. But there is also a limitation in here in that the hurdle to add a dependency to a component is not just as high as adding it to the app, it is in fact higher. That is because every time we introduce a dependency to a component we now need to also manage the dependency versions between the component and the main app.

So is the cost trade-off worth it? I am not sure that this can be said in general. One thing regarding the application structure and external dependencies that pushes on the scale towards "not worth it" is that 1/ we already can state dependencies on the main app level. And 2/ external dependencies don't suffer from the same problem we are trying to solve within our app with components: we can't *entangle* ourselves with our external dependencies.

Anyone who writes code within Rails apps regularly should be a bit skeptical about the above sentence. In the talk "Architecture the lost years" <https://www.youtube.com/watch?v=WpkDN78P884>) Robert Martin explains vividly why the fact that every Rails application looks first and foremost like a *Rails application* is not necessarily a good thing. The fact that these apps tend to not add their own structures and patterns on top of Rails is leading to so many of the problems big Rails apps face like slow test suites, difficult test setup and the like. The way I use *entangled* here is to express that *two pieces* of code can't function without each other. And in this sense Rails doesn't ever know or need to know that your app even exists.

1.2 Component-based Challenges

Components come at a cost. It is the cost of maintaining the additional structure that allows us to reason about how parts of the system are connected. This includes the gemspec, the gemfile, the gem entry point file, the spec helper, and in the case of gems that are also engines so much more. When I started with CBRA I believed (and I still do) that these costs are worth it.

However, these are not the only costs and this section discusses the more subtle, and as we'll see more dangerous, costs associated with gems as components.

1.2.1 External dependency drift

A gem specifies its production dependencies in the gemspec and its development dependencies in either the gemspec or the Gemfile or both.

1.2.1.1 How this plays out for 'ordinary' gems

The authors of ordinary gems typically have the intent of making their gem as widely usable as they can manage to make it. For this reason, dependency *versions* can be specified to varying degrees of specificity. We can for example say that we need nokogiri version 1.10.10 by adding a line to the gemspec reading `spec.add_dependency 'nokogiri', '1.10.10'`. With this an app developer wanting to use our gem is forced to install this exact version of nokogiri. If they have a different version currently, they will need to upgrade or downgrade to get bundler to successfully download all the right gem versions. If a different gem specifies a different exact version of nokogiri our gems can't be used together. So, gem developers tend to not specify exact dependency versions at all. They make versions more flexible.

To make a gem version more flexible, we can instead add the following line to our gemspec `spec.add_dependency 'nokogiri'`. Now we are saying that we don't have *any* specific requirements as to the version of nokogiri. Now, this is unlikely to ever be true in practice. For any gem that follows semantic versioning, for example, we know that if the major version changes, there are breaking changes to the API.



Semantic versioning specifies that the versioning of a piece of software should follow the following interpretation of major, minor, and patch versions (<https://semver.org/spec/v2.0.0.html>).

Given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make incompatible API changes,
- MINOR version when you add functionality in a backwards compatible manner, and
- PATCH version when you make backwards compatible bug fixes.

Oh, and we can't just assume that we can still use the above line because the gem we are depending on is currently pre 1.0 and we have tested it with all versions and it worked... so we must be fine. We can't do this because our gem now communicates that *this specific version* of our gem will always work with all versions of the other gem and, yes, that includes future versions of that gem. This is also the reason why `spec.add_dependency 'nokogiri', '> 1'` shouldn't be used - it has the same problem.

In practice, you can of course always release a new version of your gem and communicate to folks in a different way that they need to upgrade your gem in order to get it to behave correctly. You are effectively communicating that the previous version of your gem had a bug.

The only ways in which to solve this problem within the `gemspec` are to use one of the following two versions:

- `spec.add_dependency 'nokogiri', '>= 1', '< 1.4'`
- `spec.add_dependency 'nokogiri', '~ 1.1'`

The first version says that any version of `nokogiri` between 1 and 1.4 is usable with this version of the gem (1 is ok, 1.4 is not). The second version says that any version between 1.1 and 2 is ok to use (1.1 is included, 2 is not).

With this problem out of the way, we can address the second problem: verifying that our gem actually works with those various dependencies that we say we are ok with. Most modern CI systems give us some way to specify a *build matrix*, meaning we can run our tests against different configurations - different dependency versions being a special case. For the above example (the first case) we might specify that we want to run tests against `nokogiri` versions 1.0, 1.1, 1.2, and 1.3.

Note that anything we do here is not perfect - semantic versioning has never been perfect and it can't be (in Ruby at least). Imagine a subtle thing changes in the gem you depend between versions 1.1.1 and 1.1.4 that no one thought to test specifically. Semantic versioning says this shouldn't happen, but it is not enforceable and in my experience it happens all the time.



Other languages, like Elm lang, *can* to some extent: <https://elm-lang.org/>:

Elm can detect all API changes automatically thanks to its type system. We use that information to guarantee that every single Elm package follows semantic versioning precisely. No surprises in PATCH releases. (details)

Even with this kind of support, a change in implementation can make a library behave differently between versions and break your software.

Despite it not being perfect, every entry in our build matrix increases the confidence in our gem's versatility in dealing with a number of dependency versions.

1.2.1.2 Components are not 'ordinary' gems

I have never seen components be treated in the way I described above for 'ordinary' gems. And why would we? Setting up a build matrix costs effort, not to speak of the added cost to run the build matrix as part of CI - every new combination of tests to run adds as cost whatever it costs to run a single version.

And we don't have to do this with components because we know that a component is ultimately only going to run against the version of its dependencies that are specified by the main application. It is after all the app's Gemfile and its Gemfile only that gets used to infer what our app's runtime dependencies are.

In CBRA I proposed that we lock down every version of every runtime dependency to the exact version we want to use. This helps because bundler will only resolve and install our dependencies if for all dependencies one version works for every dependencies dependency constraints. I.e., if a component specifies a dependency version different from the main app, bundler crashes. If two components specify different versions of a dependency, bundler will crash.

I am saying "this helps" because this is not perfect: we don't specify dependencies of our direct dependencies that we don't care about in our gemspec and thus the above approach doesn't cover these gems at all. As such, they could still drift from what is running in production.

To completely solve this problem, one can compare the Gemfile.lock files of the app and all the components and ensure that all the versions of dependencies of components matches what is present in the main app.

A story of dependency drift: "Oops! We just lost two orders of magnitude"

Every time the API of one of your dependencies changes you are faced with the potential of that change affecting the proper functioning of your code. I recently came across an example of where a tiny change could have had major negative impacts.

Imagine code that fills out official forms, similar to the DE 9C form used in California shown below.

D. SOCIAL SECURITY NUMBER	E. EMPLOYEE NAME (FIRST NAME)	(M.I.)	(LAST NAME)
F. TOTAL SUBJECT WAGES	G. PIT WAGES	H. PIT WITHHELD	
D. SOCIAL SECURITY NUMBER	E. EMPLOYEE NAME (FIRST NAME)	(M.I.)	(LAST NAME)
F. TOTAL SUBJECT WAGES	G. PIT WAGES	H. PIT WITHHELD	
D. SOCIAL SECURITY NUMBER	E. EMPLOYEE NAME (FIRST NAME)	(M.I.)	(LAST NAME)
F. TOTAL SUBJECT WAGES	G. PIT WAGES	H. PIT WITHHELD	

Some fields from California's form DE 9C "Quarterly Contribution Return and Report of Wages"

Note how this form ever so slightly separates the dollar from the cent amounts, i.e., it expects all the amounts to be entered with two digits after the decimal point.

Let's say the amount we are looking at is more than thousands of dollars - then how to write into this field becomes interesting: do you start writing digits from the left or the right? To start writing from the left you count out how many digits (including two decimals) your number has, count from the right of the field to that number so you know where to start writing the amount. To start writing from the right you start writing into the rightmost box while reading your amount (again, including decimals) in reverse order of how you would normally read it.

Both of these approaches add quite a bit of overhead to what we commonly do: start with the highest value digit, read and write from the left.

The way our program was entering these numbers was as follows. We created a number representation which always had two decimals. That number was then displayed in an appropriate font size over the form right-aligned to the right side of the field. In many situations in life I would expect a random person to say that \$1234 is the same as \$1234.00. Visually, using our algorithm, the form would have shown these two numbers as \$12.34 and \$1234.00 respectively. Oops! We just lost two orders of magnitude!

If by now you are wondering what this has to do with dependency version drift, check out the behavior of `round` for a couple of numbers in two versions of Ruby.

```

1  # Ruby 2.4.6
2  1234.23.round(2)    #=> 1234.23
3  1234.00.round(2)    #=> 1234.0
4  1234.round(2)       #=> 1234.0
5
6  1234 == 1234.0      #=> true
7  1234 === 1234.0     #=> true
8
9  # Ruby 2.5.5
10 1234.23.round(2)    #=> 1234.23
11 1234.00.round(2)    #=> 1234.0
12 1234.round(2)       #=> 1234
13
14 1234 == 1234.0      #=> true
15 1234 === 1234.0     #=> true

```

Between Ruby 2.4 and Ruby 2.5 the rounding function for integer values changed in such a way that no longer would a call to `round` turn an integer into a float and no longer would a decimal be added to the number. Oops!

If you ask Ruby as to whether there is a difference it will staunchly tell you, in both versions, that no- there is no difference between the numbers. This “oops!” is a doozy because of how difficult it is to test the visual difference between two forms and how there doesn't seem to be a direct way of verifying the differences in these numbers based on their Ruby representation. This problem has the potential of going unnoticed for a long time.

Dependency drift is a special case of a more general problem: configuration drift.

1.2.2 Configuration drift

As opposed to the problem of dependency version drift, the problem of configuration drift is not solvable for the general case.

The effect of configuration drift is the same as that of dependency version drift: the environment in which code runs in different from development and test in comparison to the environment in production.

Configuration drift occurs between a component and the main app when the tests run with a different configuration. Most obvious are gem-only (non-engine) components that run without a dependency on Rails. Rails' libraries monkey patch quite a few Ruby modules, which is what all the app's code will run against in production. In this gem however, these changes won't be present during tests, because Rails is never loaded. If the changes to Ruby modules are additive only (like adding methods) this will likely not cause problems. If however the implementation of *existing* methods is changed, it is more likely to cause problems.

Another source of this drift are the test helpers that set up the test environment. These helpers could load files differently from what gets exposed from the gem, or they could load files in a different order, which could lead to subtle differences in behavior.

For engine-based components the surface area for potential configuration drift is massively larger than that for gem-only-based components. This is due to the fact that we need a Rails app harness for testing of our component. When generating a new engine with `rails plugin new ...` a full Rails app gets written into `test/dummy` with all the files in config including the default initializers that come with Rails. Any of these files can make a relevant change to how Rails is configured and how our code operates. And unfortunately it is very hard to spot which differences will matter. Let alone spotting the differences in the first place.

A story of configuration drift: Rails initializers doing what?

The dummy applications that engines use for testing create a part of the test harness for the engine under test. A configuration change in one of the initializers will impact the code we write and thus what we can expect in test from the application.

It is unfortunate that the Rails maintainers chose the word "dummy" for the *dummy app*, because it is closely related to the word "dummy" used to describe a particular form of object mimicking real objects for testing purposes. As Martin Fowler describes in "Mocks Aren't Stubs" <https://martinfowler.com/articles/mocksArentStubs.html>: "Dummy objects are passed around but never actually used. Usually they are just used to fill parameter lists." Dummies are one option in a list of mimickers that also include fakes, stubs, spies, and mocks. Rails engine dummy apps are actually

none of these.

Rails engine dummy apps are the *real* Rails app that an engine runs inside of for the purposes of tests. And this has real implications...

I recently came across a Rails app with multiple engines that all had a test dummy application in support of the engines' tests. One of the initializers looked like this in the main application:

```
1 ActiveSupport.on_load(:action_controller) do
2   wrap_parameters format: []
3 end
```

And it looked like this in all of the engines:

```
1 ActiveSupport.on_load(:action_controller) do
2   wrap_parameters format: [:json]
3 end
```

What is the difference? The default comment on this piece of code is `Enable parameter wrapping for JSON`. You can disable this by setting `:format` to an empty array. As such, the main app in this case disables parameter wrapping while spec dummy apps do not.

The implication of this is that if a controller is set up to return json, let's say of a person with a name, then in the main app this would return:

```
1 {"name": "Aladdin"}
```

While in an engine it would return:

```
1 {"person": {"name": "Aladdin"}}
```

The dependency version drift example discussed previously had the potential of going unnoticed because it was such a subtle shift in behavior. The shift in behavior we are looking at here is not subtle. Any frontend that would expect the former would crash on the latter and vice versa. As such, this issue would blow up in a production environment as soon as the first query came in and would thus show up in monitoring and hopefully trigger alerting rules as soon as it went live.

There is really only one reason this kind of difference can be in a codebase for a significant amount of time: the feature is unused.

A feature that is unused being in the codebase is a much less problematic situation than a production crash in the short term, but causes issues in the long run as well: it adds noise to the code, noise that makes it harder to develop new features and debug issues because it is more difficult to understand what is going on.

The bottom line is always the same: we are no longer sure that our component's tests are verifying

their exact behavior in production.

In addition to making testing more cumbersome, components can also hide global impacts and make them look like local config.

1.2.3 Global impact of local changes

We already discussed how the usage or non-usage of gems in certain contexts can change the behavior of our code. The inverse is true too: a component we write can inadvertently change the behavior of the rest of the system.

This problem is most easily explained via initializers that components with Rails engines can define. Initializers when used well, set up needed config for the context (the component) in which they are defined. But initializers have no safeguards to enforce that that is all that happens.

Initializers can for example add custom inflections to the inflections list used by the application. The effect of this is not only that the inflections within the component changes - nope - they change for the entire application.

If this kind of change causes a breakage of existing code, one would expect to find this problem during the next CI run. New code that gets written in the main app will likely just conform to the inflections defined. While this doesn't create circular dependencies and it likely won't be causing issues in production, it does make the functionality of the app more difficult to understand.

Why do it if we don't need to? The answer is probably in most cases that it is done because during development it is not obvious that that is in fact what is happening.

2. Getting started with packaging in a Rails application

For the purpose of this book, we will use a sample app called Sportsball. We are going to follow the same progression as CBRA did, slowly developing the entire application into a set of packages.



Delete the app folder?

In CBRA, I felt like I needed to make a statement to get people's attention. Feedback I have gotten on this move has shown me that that was a good decision. We could pull the same stunt here and do the same thing with packages, but there is less reason, as will hopefully become clear in this chapter. Most importantly, what took 3 chapters in CBRA will only take one chapter here. So, hopefully, no need for stunts.

2.0.1 Sportsball

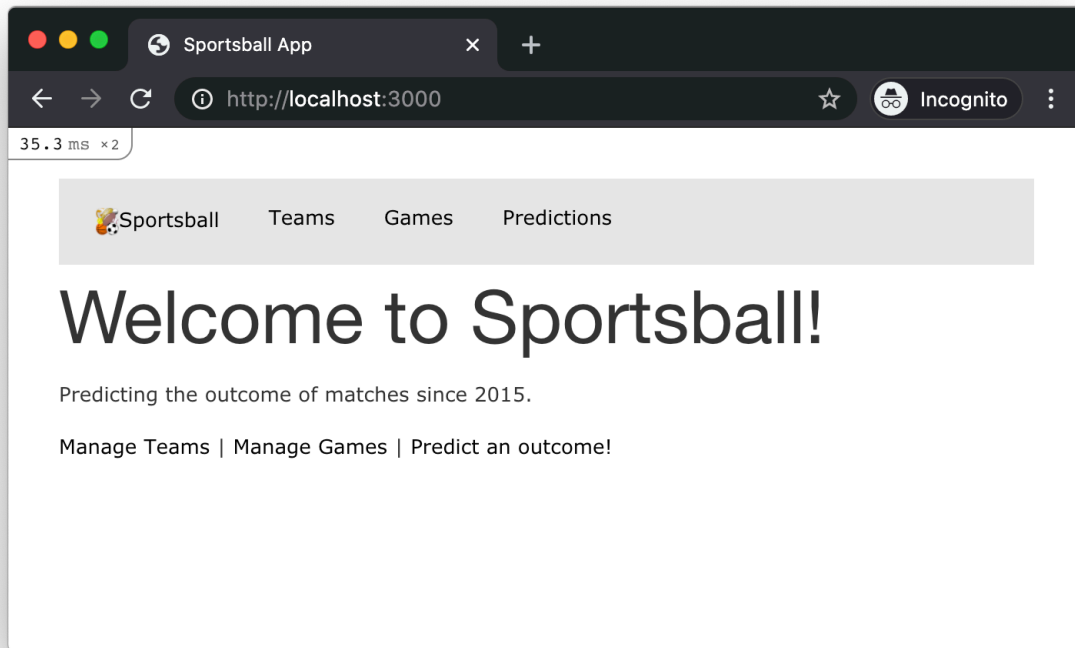
The source code for Sportsball in all of its iterations is available at <https://github.com/shageman/package-based-rails-applications-book> (this repo includes source code folders per chapter and section and the code used to generate sample code) and <https://github.com/shageman/package-based-rails-applications> (this repo includes the Sportsball application within branches for each chapter and section). We won't go through all of its embarrassingly few features here as that is not relevant for a modularization discussion and it also turns out to all be standard Rails stuff.

Throughout this chapter we refer back to the development and structure of the CBRA quite a lot.

The process of extraction that was discussed there is *not* necessary to know for following this. However, it does make sense to compare the code structures of the two versions side by side. Find the code from CBRA at <https://github.com/shageman/component-based-rails-applications-book>.

Sportsball's features are quickly enumerated: it allows us to

- manage sports teams,
- the games they play,
- and, given existing game information, it can predict for a given matchup which team will win.



Sportsball homepage

2.1 Let's get started



The code discussed in this section is available at <https://github.com/shageman/package-based-rails-applications-book/tree/main/c2s01/sportsball>. If you have checked out <https://github.com/shageman/package-based-rails-applications>: `git checkout c2s01`.

Start by making a fresh Rails app:

```
1 $ rails new sportsball
```

With or without the features of Sportsball, to get started with packages add a gem dependency on the packwerk gem by appending this line to the app's Gemfile: `gem 'packwerk'`.

Now we bundle to install packwerk.

Then we run `bundle exec packwerk init`. This creates two files in the root of our app: `packwerk.yml` and `package.yml`. Alternatively, you can also install a packwerk binstub via `bundle binstub packwerk` and then run `bin/packwerk init`.

`packwerk.yml` is the main config file for packwerk. Initially every config option is commented out because defaults apply. Among other things, it lets us define included and excluded files and where to look for packages.

Initial `packwerk.yml`

```
1  # See: Setting up the configuration file
2  # https://github.com/Shopify/packwerk/blob/main/USAGE.md#setting-up-the-configuratio\
3  n-file
4
5  # List of patterns for folder paths to include
6  # include:
7  # - "**/*.{rb,rake,erb}"
8
9  # List of patterns for folder paths to exclude
10 # exclude:
11 # - "{bin,node_modules,script,tmp,vendor}/**/*.*"
12
13 # Patterns to find package configuration files
14 # package_paths: "**/"
15
16 # List of custom associations, if any
17 # custom_associations:
18 # - "cache_belongs_to"
```

With `package.yml`, packwerk already created a first package for us. This package is called the root package: it encompasses the entire application. Without comments the initial version is this:

Initial `package.yml`

```
1  enforce_dependencies: true
2
3  enforce_privacy: false
4
5  # public_path: app/public/
6
7  # dependencies:
8  # - "packages/billing"
```

When asked about why I am so excited about packages - this is the reason. The text up to now can barely be called a chapter, yet if you were to hold CBRA and this book next to each other, it would become obvious that we are now as far as we were at the end of the second chapter. It took a lot more effort to get to this state with components. The comparison isn't even fair.

It gets worse for components when you consider that I don't need to write the equivalent of Chapter 3 from CBRA at all - that chapter discusses how to manage dependencies, test, assets, do continuous integration, and deployments. That chapter is necessary for CBRA applications because we are doing a bunch of things in a non-standard way. Now, we have a standard Rails application. All the standard literature applies because we haven't changed anything about the Rails mechanics and I have nothing to add there. The comparison isn't even fair.

I am not joking. It is not fair. But it will be fun, I promise.



Comparing to CBRA

Check out Chapter 2 Section 3's CBRA code at <https://github.com/shageman/component-based-rails-applications-book/tree/master/c2s03/sportsball> for the closest equivalent of the current state of the app

Jumping ahead... With Chapter 3 not necessary, let's skip to the equivalent of Chapter 4 in CBRA: From one package to many.



I promise I will stop with the CBRA analogies at some point. In part that is because after this section we are leaving behind the analogies to components and will pursue the questions that come up when we try to morph package dependencies over time to manage the complexity of our application.

2.2 From one to many packages



The code discussed in this section is available at <https://github.com/shageman/package-based-rails-applications-book/tree/main/c2s02/sportsball>. If you have checked out <https://github.com/shageman/package-based-rails-applications>: `git checkout c2s02`.

There was so little to do thus far - so let's put more meat on the bone of this chapter.

As we haven't added any code yet, let's do that real quick, so we are truly up to par with the CBRA Sportsball application.

TODO: how much of the added code to show

At this point, in a component-based application, we need to think critically about which extraction we would like to perform first, because the dependency tree we create will be easier one way vs another and extraction order matters. We basically are forced to extract the lowest level code first, in order to ensure all the other code still works. There is, of course, a way around this, and we'll cover this further down among the refactoring patterns: we can really extract anything if we "just" employ one or more of the dependency-breaking techniques to ensure that what gets extracted has

access to everything it needs to function in and be tested in isolation. Why “just?” Because this, while possible, is generally very hard, takes time, and lots of effort.

For a package-based application, we need to take a little detour and explore how packages are actually defined using packwerk.

2.2.1 Packwerk packages vs gems as components

For all modularization work, we are generally looking for three properties: a named *container*, its *content*, and *explicit dependencies* on other containers. With gems the name of the gem is the container name. The files within the gem are the content. Finally, the dependencies specified in the `gemspec` are the dependencies.

Packwerk packages also specify these three properties. Packages are aligned with folder structures. The creation of a package is very straightforward: Place a `package.yml` into a folder and all the files in this folder and all subfolders are now considered part of the package. This defines our content. The name of a package is its relative path from the root of the application. Dependencies can be specified by adding a `dependencies` entry to the `package.yml` file. This entry takes an array of package names.

Package dependencies specified in `package.yml` are not allowed to create cyclic package dependencies. From a user’s perspective, this constraint makes package dependencies and gem dependencies behave in pretty much the same way.

Example package structure

The following folder structure plus `package.yml` files is a valid structure for package dependencies.

```
1 packages
2 |— a
3 |   └─ package.yml
4 |— b
5   └─ package.yml
```

Then the following two package files create a package dependency from package a to package b.

`packages/a/package.yml`

```
1 enforce_dependencies: true
2 enforce_privacy: false
3 dependencies:
4   - packages/b
```

`packages/b/package.yml`

```
1 enforce_dependencies: true
2 enforce_privacy: false
```

Think back to the root package that packwerk automatically created for us in the first steps of this chapter and you might notice something that we didn't see with components. Packages can be defined in subfolders of other packages. If you keep the root package (which I recommend) than every package defined in addition is naturally in a subfolder of the root package. It is important to note that every file can only be in one package. The package boundary is then the only way I can think of making sense: a package defined in a subfolder “ends” the reach of the parent package into this folder. I.e., every file is in the root package if that file is not a descendant of a non-root package.

Looking once more at the three properties for modularization: *container*, *content*, and *explicit dependencies* - I have thought this to be the right list for components for a long time. Today, I believe I missed one more important property: a *clear public API*.

Gems tend to address this via their README. Take <https://github.com/sferik/twitter> for example: after some housekeeping, <https://github.com/sferik/twitter#configuration> explains how to set up the gem, then <https://github.com/sferik/twitter#configuration> explains how to use it.

In addition of this explicit communication of how a gem *should be used*, Ruby allows us to explicitly communicate how it shouldn't be used. By defining methods or constants as `private` we could hide them from easy access from any consumer, thus adding to the overall messaging of what is intended for this gem.

Packages create a new avenue for expressing what the public API of a package is through the configuration key `enforce_privacy`. For this chapter we will leave this setting off - very much in the spirit of CBRA where we never made a big deal of the public API question. We will return to it and its contribution to a well-modularized system later.

Suffice it to say for now that `enforce_privacy` can be turned on in two ways: It can be set to `true` and it can be set to an array of constant names. The latter way allows us to list explicitly which constants packwerk should consider private. Setting privacy to `true` will make packwerk consider all constants private unless they are placed into the folder `app/public` underneath the package's root directory. Any privacy violations will be reported by packwerk, the same way that privacy violations are in the `deprecated_references.yml`.

Packwerk's privacy enforcement is a simple way to start thinking about the “exposed surface area” of modularized parts of the application. I believe there is a winning combination here: put a README in the root of the package to explain what it does for anyone who is new to it and programmatically check on the adherence to the API with `enforce_privacy`.

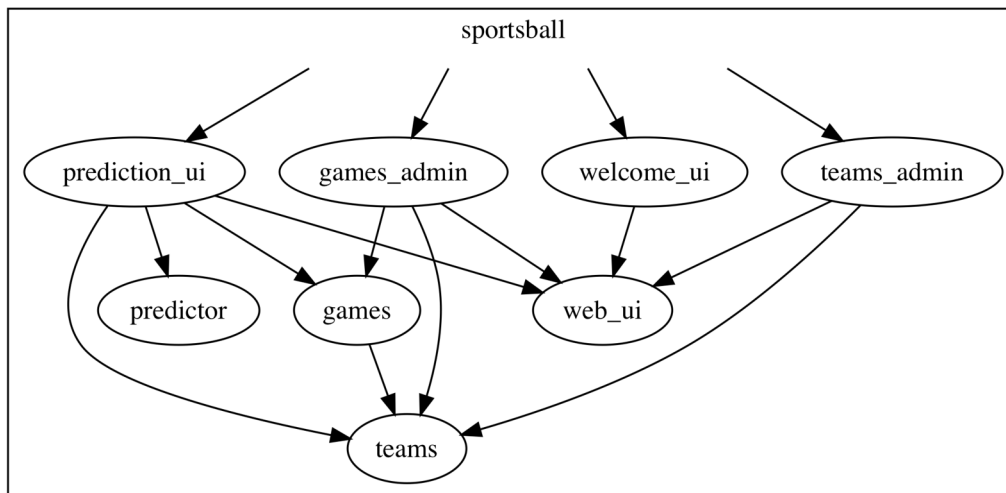
2.2.2 A completely packagified application

In CBRA it was interesting to extract every model and controller into their own component because every move that needed to be made introduced new challenges that needed to be solved during the refactor. To continue the “it’s not even fair” theme, let’s do the same with packages.

In CBRA we ended with the following components:

- games - ActiveRecord model representing a game
- games_admin - the admin UI to manage games
- prediction_ui - UI to trigger the prediction of a game
- predictor - implementation of the prediction algorithm
- teams - ActiveRecord model representing a team
- teams_admin - the admin UI to manage teams
- web_ui - the base UI files, including javascript, stylesheets, and layout
- welcome_ui - the controller and view of the Sportsball’s entry page

At the conclusion of the main refactorings the component structure of Sportsball was as follows.



Sportsball’s CBRA structure

Taking stock of our app structure as it stands currently, we can see that we will need to move some files so we are able to create the above components as packages.

Folder structure before packaged folders

```
1 sportsball/app
2 |─ assets
3 | |─ config
4 | | |─ manifest.js
5 | |─ images
6 | |─ stylesheets
7 | | |─ application.css
8 | | |─ games.scss
9 | | |─ scaffolds.scss
10 | | |─ teams.scss
11 | | |─ welcome.scss
12 |─ channels
13 | |─ application_cable
14 | | |─ channel.rb
15 | | |─ connection.rb
16 |─ controllers
17 | |─ application_controller.rb
18 | |─ concerns
19 | |─ games_controller.rb
20 | |─ predictions_controller.rb
21 | |─ teams_controller.rb
22 | |─ welcome_controller.rb
23 |─ helpers
24 | |─ application_helper.rb
25 | |─ predictions_helper.rb
26 |─ javascript
27 | |─ channels
28 | | |─ consumer.js
29 | | |─ index.js
30 | |─ packs
31 | | |─ application.js
32 |─ jobs
33 | |─ application_job.rb
34 |─ mailers
35 | |─ application_mailer.rb
36 |─ models
37 | |─ application_record.rb
38 | |─ concerns
39 | |─ game.rb
40 | |─ prediction.rb
41 | |─ predictor.rb
42 | |─ team.rb
```

```
43 └─ views
44   └─ games
45     └─ _form.html.slim
46     └─ _game.json.jbuilder
47     └─ edit.html.slim
48     └─ index.html.slim
49     └─ index.json.jbuilder
50     └─ new.html.slim
51     └─ show.html.slim
52     └─ show.json.jbuilder
53   └─ layouts
54     └─ application.html.slim
55     └─ mailer.html.erb
56     └─ mailer.text.erb
57   └─ predictions
58     └─ create.html.slim
59     └─ new.html.slim
60   └─ teams
61     └─ _form.html.slim
62     └─ _team.json.jbuilder
63     └─ edit.html.slim
64     └─ index.html.slim
65     └─ index.json.jbuilder
66     └─ new.html.slim
67     └─ show.html.slim
68     └─ show.json.jbuilder
69 └─ welcome
70   └─ index.html.slim
```

All controllers being in the same folder means that we can, at best, create one package for them and this package would include *all* controllers. The same is true for the models. We also have no way of creating a package for, let's say *games admin*, because the controller and its views are not in the same folder, isolated from other folders.

2.2.3 Introducing app/packages

There are numerous ways to reorganize the files of our app to allow for the packages we intend to create. I am going to go with a new folder that is a subfolder to app: `app/packages`. I like this because we communicate through this folder that on entering we are in a different organizational structure than we are commonly used to from Rails.

In this `app/packages` folder we can now create, as subfolders, all the folders we listed above. After then moving all the files related to the respective packages into their folder we get this:

Folder structure after packaging folders

```
1 .
2 ├── assets
3 |   ├── config
4 |   |   └── manifest.js
5 |   ├── images
6 |   └── stylesheets
7 |       ├── application.css
8 |       ├── games.scss
9 |       ├── scaffolds.scss
10 |       ├── teams.scss
11 |       └── welcome.scss
12 ├── channels
13 |   └── application_cable
14 |       ├── channel.rb
15 |       └── connection.rb
16 ├── controllers
17 |   ├── application_controller.rb
18 |   └── concerns
19 ├── helpers
20 |   └── application_helper.rb
21 ├── javascript
22 |   ├── channels
23 |   |   ├── consumer.js
24 |   |   └── index.js
25 |   └── packs
26 |       └── application.js
27 ├── jobs
28 |   └── application_job.rb
29 ├── mailers
30 |   └── application_mailer.rb
31 ├── models
32 |   ├── application_record.rb
33 |   └── concerns
34 ├── packages
35 |   ├── games
36 |   |   └── models
37 |   |       └── game.rb
38 |   ├── games_admin
39 |   |   ├── controllers
40 |   |   |   └── games_controller.rb
41 |   |   └── views
42 |   |       └── games
```

```
43 | | | | | _form.html.slim
44 | | | | | _game.json.jbuilder
45 | | | | | edit.html.slim
46 | | | | | index.html.slim
47 | | | | | index.json.jbuilder
48 | | | | | new.html.slim
49 | | | | | show.html.slim
50 | | | | | show.json.jbuilder
51 | | | | | prediction_ui
52 | | | | | | controllers
53 | | | | | | | predictions_controller.rb
54 | | | | | | helpers
55 | | | | | | | predictions_helper.rb
56 | | | | | | views
57 | | | | | | | predictions
58 | | | | | | | | create.html.slim
59 | | | | | | | | new.html.slim
60 | | | | | predictor
61 | | | | | | models
62 | | | | | | | prediction.rb
63 | | | | | | | predictor.rb
64 | | | | | teams
65 | | | | | | models
66 | | | | | | | team.rb
67 | | | | | teams_admin
68 | | | | | | controllers
69 | | | | | | | teams_controller.rb
70 | | | | | | views
71 | | | | | | | teams
72 | | | | | | | | _form.html.slim
73 | | | | | | | | _team.json.jbuilder
74 | | | | | | | | edit.html.slim
75 | | | | | | | | index.html.slim
76 | | | | | | | | index.json.jbuilder
77 | | | | | | | | new.html.slim
78 | | | | | | | | show.html.slim
79 | | | | | | | | show.json.jbuilder
80 | | | | | welcome_ui
81 | | | | | | controllers
82 | | | | | | | welcome_controller.rb
83 | | | | | | views
84 | | | | | | | welcome
85 | | | | | | | | index.html.slim
```

```
86 └─ views
87   └─ layouts
88       └─ application.html.slim
89       └─ mailer.html.erb
90       └─ mailer.text.erb
```

Check out the code to this section to explore all the changes, which also includes analogous moves for the tests of the respective packages.

With this change we *have* obviously moved away from Rails' standard structure and we will need to fix a few things to make Rails happy again

2.2.4 Getting things to work

If we try to fire up the Rails server right now, it will crash because Rails can't find any of our app code.

The fix is, fortunately, not very cumbersome. Add this line to `application.rb`:

```
1 config.paths.add 'app/packages', glob: '*/{*,*/concerns}', eager_load: true
```

This line adds two different globs (`app/packages/**` and `app/packages/**/*.concerns`) to the Rails load paths. In both globs, with the first star we grab all package folder directly in the `app/packages` folder. The second star allows us to replicate the models, views, controllers, folders that we commonly find inside `app`. While the first glob points directly at these MVC folders, the second glob adds any `concerns` folders we might have.

Of course, one could make this path addition more specific and only add the packages that we actually have and only add the MVC folders that each package actually has. However, as we'll see later, flexibility in which files go where is one of the strengths of this approach and as such we'll continue to strive to make any config changes necessary as generic as possible.

This particular glob is not the only way to structure this new packages folder. A viable alternative would be this one:

```
1 config.paths.add 'app/packages', glob: '{*,*/concerns}', eager_load: true
```

This glob just removes one layer of star matchers. Given that we want to keep the packages folders, this means that we won't go with MVC folders to subdivide the contents of each package. I personally love the fact that, with this approach, we have controllers sitting next to models and views - all things that belong together right there next to each other.



Given the flexibility of globbing, it should be clear that other structures are possible too. With a bit of tinkering, even structures that differ from one package to another depending on what best fits the particular content of a package.

Fire up the Rails server and voilà: the server starts up but all our views are not being found. So, on to views!

The problem we're up against now is that Rails can't find the views for our controllers because it expects them in their normal location of `app/views`. Again, the fix is pretty straightforward. In this case, it is `ApplicationController` that has the power to add additional search paths to the list that the Rails server uses to find views corresponding to a given controller action.

Once we change `app/controllers/application_controller.rb` to the following, the app should start up fine and our views should be back. Adjust this code appropriately if you are going with a different packaging structure.

`app/controllers/application_controller.rb`

```

1 class ApplicationController < ActionController::Base
2   append_view_path(Dir.glob(Rails.root.join('app/packages/*/views')))
3 end

```

2.2.5 Making packwerk packages

Now that we have moved all the files, we can packageify all of these folders by adding the most basic `package.yml` to the root of all our package folders. I.e., we'll add this file as `app/packages/games/package.yml`, `app/packages/games_admin/package.yml`, etc.

```

1 enforce_dependencies: true
2 enforce_boundaries: false

```

We are now ready for the true magic of packwerk's static analysis of our codebase - ready to blow everything we could do with components out of the water.

Run this: `bundle exec packwerk update-deprecations`

You will see a bunch of `deprecated_references.yml` files getting created in the various package folders. In fact, you will find these files in all packages, except for `predictor`. Here is the one for `prediction_ui`:

```

1 ---
2 ".":
3   "::ApplicationController":
4     violations:
5       - dependency
6     files:
7       - app/packages/prediction_ui/controllers/predictions_controller.rb
8 app/packages/games:
9   "::Game":

```

```
10     violations:
11     - dependency
12     files:
13     - app/packages/prediction_ui/controllers/predictions_controller.rb
14 app/packages/predictor:
15     "::Predictor":
16     violations:
17     - dependency
18     files:
19     - app/packages/prediction_ui/controllers/predictions_controller.rb
20 app/packages/teams:
21     "::Team":
22     violations:
23     - dependency
24     files:
25     - app/packages/prediction_ui/controllers/predictions_controller.rb
```

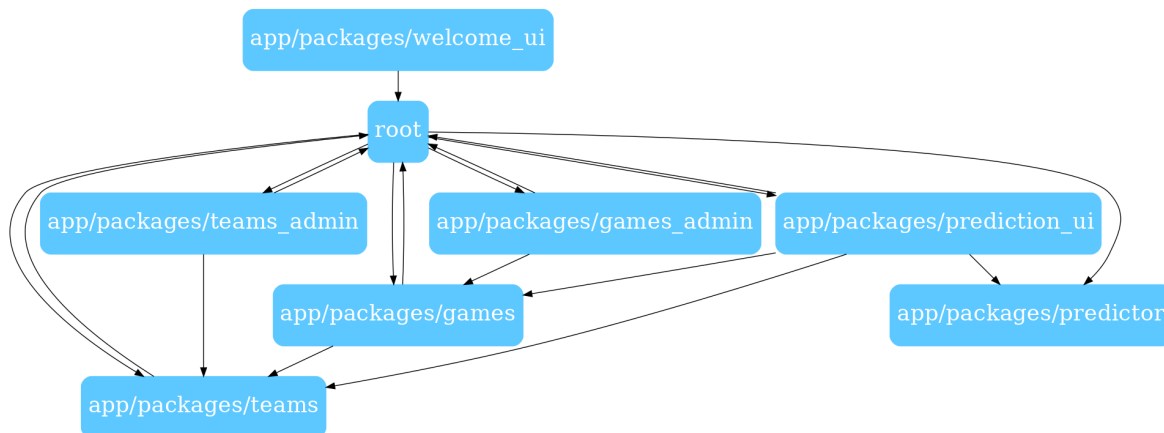
This yaml file is the next step to us creating a well-structured, package-based application. What we are seeing here is that the `app/packages/prediction_ui` package (and since all packages are in `app/packages` we can say `prediction_ui` package for short) depends on four other packages: `games`, `predictor`, `teams`, and the root package. It is telling us *of what type* our violations are. We only activated dependency enforcement, so that's all we are seeing. Additionally, we are seeing the file in which the violation is occurring - for us it is `predictions_controller.rb` in every case.

2.2.6 Visualizing dependencies

Together, `deprecated_references.yml` and `package.yml` give us a full dependency graph, which using tools like `graphwerk` (<https://github.com/tricycle/graphwerk>) or `pocky` (<https://github.com/mquan/pocky>) we can represent as a dependency diagram similar to what we showed above for the component-based approach.

The following is the diagram that `pocky` gives us for this structure when we run it's rake task from the root of the application.

```
1 bin/rake pocky:generate[root]
```



Package dependency diagram created by pocky

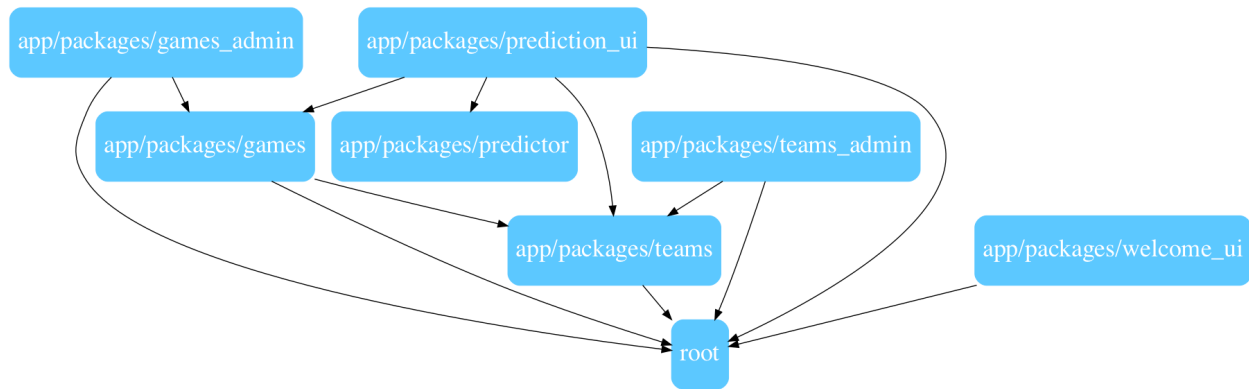
The big difference between our CBRA-based dependency diagram and this one is stark. This is not (yet) an *acyclic* dependency graph. The root package depends on all other packages and most other packages depend on the root package. If we remove the root package's dependencies, we can see that the remaining subgraph is very similar to the CBRA-based one already.

We can do this with pocky by calling it as follows, this time not via the rake task, from irb

```

1 Pocky::Packwerk.generate(
2   default_package: 'root',
3   package_path: [
4     'app/packages/games',
5     'app/packages/games_admin',
6     'app/packages/prediction_ui',
7     'app/packages/predictor',
8     'app/packages/teams',
9     'app/packages/teams_admin',
10    'app/packages/welcome_ui',
11  ]
12 )

```



Package dependency diagram without root package

As I said before: It wasn't even fair.



Comparing to CBRA

Check out Chapter 3 Section 7's CBRA code at <https://github.com/shageman/component-based-rails-applications-book/tree/master/c3s07/sportsball> for the closest equivalent of the current state of the app

There are a couple of differences between the CBRA version of the app and the package-based state we have now. For one, we have this bundle of circular dependencies revolving around the root package. As we discussed above, removing circular dependencies was one of the main drivers behind tackling the large-scale application structure in the first place, so we definitely want to get rid of this issue.

We also haven't looked at tests at all. With components we can isolate not only production code but also test code. Co-locating app and test code helps us reason about parts of the application in isolation, which is another big part of the benefits of modularization. And it is actually all the test code that is currently in the root package that creates the dependencies between the root and all the other packages.

Removing the circular dependencies and align test and app code is thus interesting from a practical perspective and it is what we will turn towards for the rest of this chapter.

2.3 Accepting the intended dependencies - explicitly adding package dependencies



The code discussed in this section is available at <https://github.com/shageman/package-based-rails-applications-book/tree/main/c2s03/sportsball>. If you have checked out <https://github.com/shageman/package-based-rails-applications>: `git checkout c2s03`.

In the previous section we discovered that a good chunk of the dependencies thus far created in the app were the dependencies we were trying to recreate from the original CBRA Sportsball application. Packwerk gives us a mechanism to explicitly state that we *accept* a particular dependency for a particular package.

To accept these dependencies in packwerk, we add a new key to the `package.yml` of a package. Dependencies are listed as an array containing the names (i.e., relative path from the application root) of the packages whose dependency we want to accept.

Here are two examples.

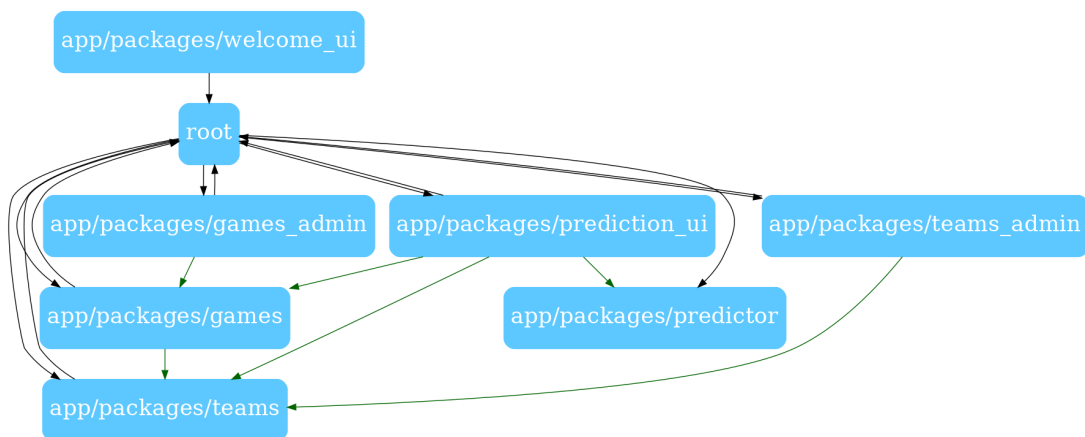
`package.yml` for games and games admin packages

```

1 # app/packages/games/package.yml
2 enforce_dependencies: true
3 enforce_privacy: false
4 dependencies:
5 - app/packages/teams
6
7 # app/packages/games_admin/package.yml
8 enforce_dependencies: true
9 enforce_privacy: false
10 dependencies:
11 - app/packages/games

```

When we do this for all packages and recreate the dependency diagram, we get this



Package dependency diagram created by pocky

The visual change is small. Most visibly, for some reason GraphViz decided it should place the vertices in different spots, but that is secondary. More importantly, pocky also made a visual change. The accepted dependencies are now colored in green and thus differentiated from the not accepted dependencies, which are still colored in black.

One important factor to note here is that we accepted dependencies that create a *non-cyclic* part of the graph. This is in fact a requirement by packwerk - the tool will not allow us to create *cyclic accepted* dependencies.

If, for example, we were to try and have games depend on games_admin, i.e., change the above package.yml files to the following.

package.yml for games and games admin packages that will not work

```

1 # app/packages/games/package.yml
2 enforce_dependencies: true
3 enforce_privacy: false
4 dependencies:
5 - app/packages/teams
6 - app/packages/games_admin
7
8 # app/packages/games_admin/package.yml
9 enforce_dependencies: true
10 enforce_privacy: false
11 dependencies:
12 - app/packages/games

```

When we now try to validate the consistency of our packwerk configuration we will be called out for the new dependency:

```

1 $ bundle exec packwerk validate
2 Packwerk is running validation...
3
4 Validation failed !
5 Expected the package dependency graph to be acyclic, but it contains the following c\
6 ycles:
7
8     - app/packages/games -> app/packages/games_admin -> app/packages/games

```

This is the mechanism by which we get the first level of dependency protection from packwerk that is very similar to what we get from components. We will get into how the situation here is slightly more nuanced in a bit.

2.4 Removing the circular dependencies - creating rails_shims package



The code discussed in this section is available at <https://github.com/shageman/package-based-rails-applications-book/tree/main/c2s04/sportsball>. If you have checked out <https://github.com/shageman/package-based-rails-applications>: `git checkout c2s04`.

Let's continue the cleanup of our app's dependencies by turning to the central cyclic dependency knot. From the dependency diagram we can not tell the reason for the relationships that create these cycles. If we recall that all the arrows are drawn from both the accepted dependencies in `package.yml` files and incidental dependencies in `deprecated_references.yml` files.

Continuing the spirit of building a dependency graph as similar as possible to what CBRA allowed for, and looking back to [Image XYZ](#), we know that we don't want any dependencies on the root/main application. In fact, we want the main application to only be the glue that binds together the various parts that contribute the actual functionality of our application.

Knowing we did not accept any dependency onto the root package and that the `deprecated_references.yml` of a package defines the dependencies of *it* on *other packages*, we can look through the `deprecated_references.yml` files of all the packages dependent on root to understand why they depend on the root package. Here is that file for the games package:

Games package `deprecated_references.yml`

```

1  ---
2  ".":
3    "::ApplicationRecord":
4      violations:
5        - dependency
6      files:
7        - app/packages/games/models/game.rb

```

We can glean from this file that there is only one file, `game.rb`, which violates the dependency constraint to the root or “.” package by referring to the constant `ApplicationRecord`. This is not surprising as with `Game` we have an ActiveRecord model that in its class definition inherits from `ApplicationRecord`.

Looking through all packages for why they depend on the root package manually will become tedious quickly. In understanding that, for the moment, we are looking for dependencies onto the root package only, we can write a little script to get all of these dependencies at once.

Script to summarize dependencies on the root package

```

1  #!/usr/bin/env ruby
2
3  require 'open3'
4  require 'yaml'
5  require 'deep_merge'
6  require 'pp'
7
8  file_names, _stderr, _status = Open3.capture3("find . -iname 'deprecated_references.\
9  yml'")
10 dependencies = file_names.split("\n").map do |filename|

```

```

11   YAML.load_file(File.expand_path(File.join(Dir.pwd, filename)))['.']
12 end.compact
13
14 dependent_things = dependencies.reduce(Hash.new) do |memo, dep|
15   memo = memo.deep_merge!(dep)
16 end
17
18 pp dependent_things

```

This script uses the bash `find` command to collect the location of all `deprecated_references.yml`. It then loops over this list, opens the respective files as YAML and extracts the entry for the root package, via `['.']`. To prevent issues from files where this key is not present, the resulting list is compacted, thus removing all nil entries. The script then loops over the resulting hashes once more and joins them together with `deep_merge!`, the eponymous method from a gem that is used here. The following result is output by this code.

All violations to the root package

```

1  {":ApplicationController"=>
2    {"violations"=>["dependency"],
3     "files"=>
4       ["app/packages/prediction_ui/controllers/predictions_controller.rb",
5        "app/packages/welcome_ui/controllers/welcome_controller.rb",
6        "app/packages/teams_admin/controllers/teams_controller.rb",
7        "app/packages/games_admin/controllers/games_controller.rb"]},
8    ":ApplicationRecord"=>
9      {"violations"=>["dependency"],
10       "files"=>
11         ["app/packages/teams/models/team.rb",
12          "app/packages/games/models/game.rb"]}]}

```

We are faced with two constants, `ApplicationController` and `ApplicationRecord`, as the basis of the violations. Similar to the ActiveRecord classes we already had a look at, all the controllers in the app subclass `ApplicationController` to become proper controllers in the application.

These constants are just two of all the ones that we commonly have in Rails applications. The following tree of the files under `app/` (without the `packages` folder) shows the others: `ApplicationCable`, `ApplicationHelper`, `ApplicationJob`, and `ApplicationMailer`:

Non-packaged files in the app folder

```
1 $ tree app/
2 app/
3 ├── assets
4 │   ├── config
5 │   │   └── manifest.js
6 │   ├── images
7 │   │   └── logo.png
8 │   └── stylesheets
9 │       ├── application.css
10 │       ├── games.scss
11 │       ├── scaffolds.scss
12 │       ├── teams.scss
13 │       └── welcome.scss
14 ├── channels
15 │   └── application_cable
16 │       ├── channel.rb
17 │       └── connection.rb
18 ├── controllers
19 │   ├── application_controller.rb
20 │   └── concerns
21 ├── helpers
22 │   └── application_helper.rb
23 ├── javascript
24 │   ├── channels
25 │   │   ├── consumer.js
26 │   │   └── index.js
27 │   └── packs
28 │       └── application.js
29 ├── jobs
30 │   └── application_job.rb
31 ├── mailers
32 │   └── application_mailer.rb
33 ├── models
34 │   ├── application_record.rb
35 │   └── concerns
36 └── views
37     ├── layouts
38     │   ├── application.html.slim
39     │   ├── mailer.html.erb
40     └── mailer.text.erb
```

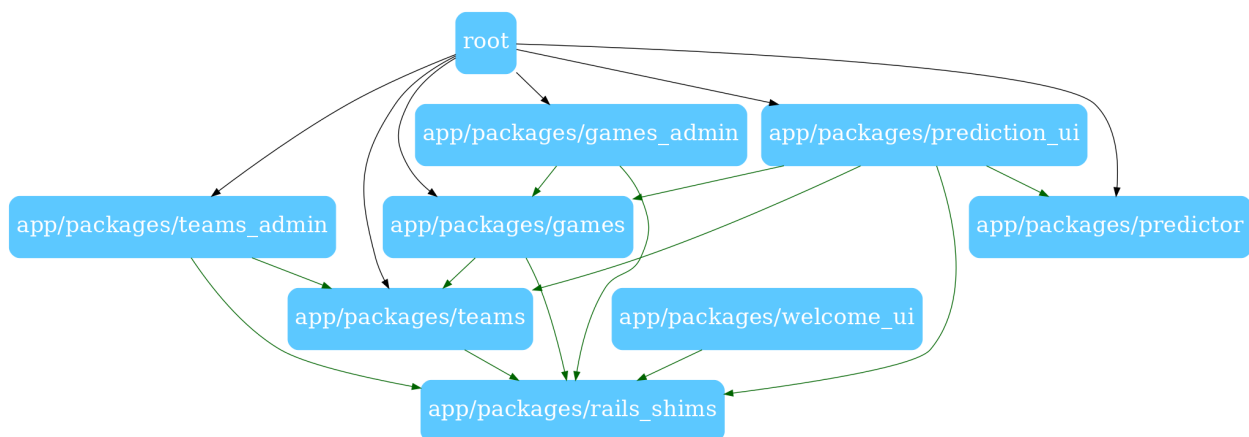
All of these `Application***` classes, highlighted in the above list, provide the configurable glue between features of the Rails framework and the code implementing the application. In the CBRA book, I proposed to isolate applications from 3rd party functionality via what I called *3rd party access gems* to create a “wrapper” that we control around APIs that are not ours, but that we want to use. This wrapper allows us to convert, map, or drop any parts of the 3rd party code that we intend to use (or not) but want to change to better reflect our intent with it inside of the application.

With packages, we can actually try to do this for Rails itself for the first time. Given the changes we have already made to the application to support our packages, we don’t have to make any additional changes, we just have to come up with a name for a package that would fit all of these files. Since all of these files typically contain only the code that adapts or configures Rails capabilities, I propose to call the package `rails_shims` to indicate the connection to Rails and the adaptation function of its contents. In this role, this package is different from all the other packages we have created so far in that it does not contain domain-specific logic, but from a structural and a packwerk packaging perspective that is perfectly fine to do.

To add the `rails_shims` package we make the following changes to our app:

- Move all the `application_` files and folders from `app` to `app/packages/rails_shims`
- Create an initial `app/packages/rails_shims/package.yml`
- Add `app/packages/rails_shims` to all other packages’ `package.yml` (except for `Predictor`, which doesn’t depend on Rails functionality)

With all these changes in place, we can create a new version of the app’s dependency diagram, which now shows all circular dependencies removed.



Package dependency diagram created by pocky

We can see `root` depending on all packages except `app/packages/rails_shims` and `app/packages/welcome_ui`. Let’s analyze and tackle `root` depending on these packages next.

2.5 Analyzing and removing root dependencies (almost) - Aligning app and test code



The code discussed in this section is available at <https://github.com/shageman/package-based-rails-applications-book/tree/main/c2s05/sportsball>. If you have checked out <https://github.com/shageman/package-based-rails-applications>: `git checkout c2s05`.

In CBRA it was interesting to map the dependencies of the main app onto components because the main app didn't *have to depend* on all components. For example, if there is no app code in main application directly, the main application would still depend on all the components that it would need to configure in its initializers. It would also need to directly depend on any component that it would mount into the routes. This glueing function of the main app is a prominent feature of CBRA.

This is different for packages: we have configured the application such that all packages are autoloaded. This is done without explicitly stating any of the constant from the packages so that is invisible to packwerk. Packages do *not* show up as dependencies of the main application – it is invisible glue – because of this. Hence, there is no informational value to show these kinds of relationships.

With the above being the case, why is the main app still dependent on so many packages? And since it is and we know that the glue is invisible, doesn't that mean that we're getting information here on the boundaries we have drawn thus far? Information that tells us that our boundaries could be better? Let's find out.

Packwerk reports the following deprecated references.

Deprecated references of the root package

```

1  app/packages/games:
2    "::

```

```
16     - dependency
17     files:
18     - spec/packages/games_admin/routing/games_routing_spec.rb
19 app/packages/prediction_ui:
20   "::PredictionsHelper":
21     violations:
22     - dependency
23     files:
24     - spec/packages/prediction_ui/helpers/predictions_helper_spec.rb
25 app/packages/predictor:
26   "::Prediction":
27     violations:
28     - dependency
29     files:
30     - spec/packages/predictor/models/predictor_spec.rb
31   "::Predictor":
32     violations:
33     - dependency
34     files:
35     - spec/packages/predictor/models/predictor_spec.rb
36 app/packages/teams:
37   "::Team":
38     violations:
39     - dependency
40     files:
41     - spec/packages/teams/models/team_spec.rb
42     - spec/packages/teams_admin/views/edit.html.slim_spec.rb
43     - spec/packages/teams_admin/views/index.html.slim_spec.rb
44     - spec/packages/teams_admin/views/new.html.slim_spec.rb
45     - spec/packages/teams_admin/views/show.html.slim_spec.rb
46     - spec/support/object_creation_methods.rb
47 app/packages/teams_admin:
48   "::TeamsController":
49     violations:
50     - dependency
51     files:
52     - spec/packages/teams_admin/routing/teams_routing_spec.rb
```

This listing points out something we have completely neglected thus far: tests. That is because all of the references listed here are coming from the spec folder.

If we look one folder level deeper, we see that there are two violations stemming from spec/support, both from spec/support/object_creation_methods.rb in fact. All other violations, and there are

quite a few, are located in `spec/packages`. Let's tackle these latter violations first.

The immediate question is: Should we accept these violations and add the packages to the dependencies of the root package or handle this a different way?

Comparing this state and the current question once again to the CBRA approach: With components, we aligned tests and app code from the start as that is a built in feature for all types of gems. That was not only supported but it also created the main vehicle for component isolation: if the tests can run independently, the app code must be internally cohesive and independently "correct" as well.

As such, to achieve a similar level of analysis and isolation, I believe we should work to align app and test code.

When initially we debated moving the files that should go into a package together into the `app/packages` folder, the argument was that we couldn't have created *one* package around files being in disparate folders under `app` without finding a new root for them. This looks very similar: all our packages' app code is in `app/packages` all the tests are in `spec/packages`.

In the previous situation, we flipped a hierarchical relationship: In a standard Rails app we first see the functional group (model, view, controller, etc) and only further down in the hierarchy the domain group (team, game, etc). When we made the first packages, we flipped this within `app/packages`.

To allow for app and test code to be aligned, we need to do another flip. For each `PACKAGE`, if we make the following change, we can make packages that contain both app and spec code.

- `app/packages/PACKAGE` -> `packages/PACKAGE/app`
- `spec/packages/PACKAGE` -> `packages/PACKAGE/spec`

This is certainly not the only way we can perform this code alignment but it feels like a very natural way. Incidentally, it also mimics CBRA's proposal of moving all app gems into `/components`, which makes the code structure look almost identical in both approaches.



Alternative structures

Instead of aligning app and spec code, another straightforward way would be to make packages out of the all the folders in `spec/packages`. With this solution `package.yml` files for the spec packages would depend on their respective app counterparts and potentially other packages if they need additional code to implement their functionality.

This has some interesting side effects. Most notably, it would become very obvious if application code depended on test code! It would also separate the dependencies between the two.

If we want these effects but still align app and spec code as discussed here, we could also choose to add two packages per `PACKAGE` folder: `packages/PACKAGE/app/package.yml` and `packages/PACKAGE/spec/package.yml`

Since we are changing the location of packages, we need to update our initial changes that made a separate package folder possible. The full list of changes needed for this step is as follows:

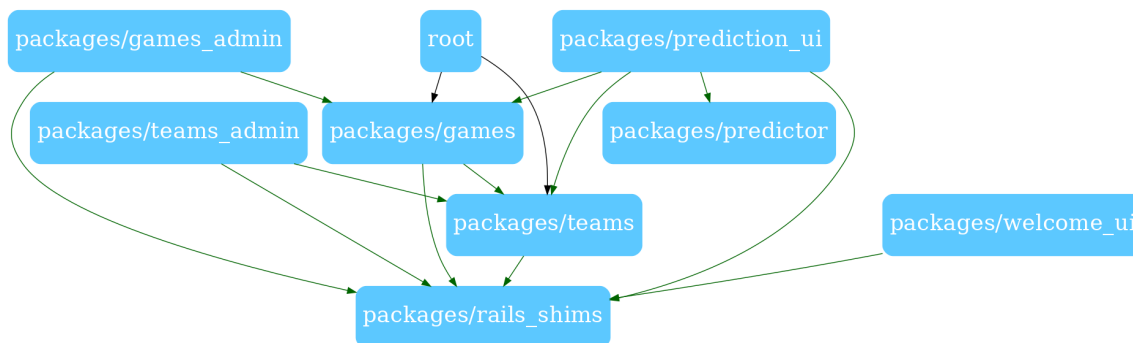
- Create `packages/PACKAGE/app` and `packages/PACKAGE/spec` for all packages in the application and move their respective files into these new folders
- Replace the previous addition of `config.paths` in `config/application.rb` which was pointing at `app/packages` to now read:

```
1 config.paths.add 'packages', glob: '*\app\{*,*\concerns}', eager_load: true
```

- Change `append_view_path` to add `'packages/*/app/views'` in `packages/rails_shims/app/controllers/application_controller.rb`
- Update calls to `define_derived_metadata` to `/packages/./spec...` in `spec/spec_helper.rb`
- Update the glob for the `view.lookup_context.view_paths` to `Rails.root + ('packages/*/app/views')` in `spec/spec_helper.rb`
- Update the paths to all packages in the `package.yml` files of all packages

While this is quite the list of changes, they are all very mechanical, simply adjusting all of the references to the old paths with the respective references to the new paths.

With this all done, you guessed it, we can generate and take another look at the package dependency diagram, which now looks like this.



Package dependency diagram created by pocky

TODO: rspec won't run any tests with just rspec The fix is to update `--pattern` in `.rspec`

```
1 --require spec_helper
2 --default-path packages
3 -I spec
```

2.6 Removing all dependencies from root



The code discussed in this section is available at <https://github.com/shageman/package-based-rails-applications-book/tree/main/c2s06/sportsball>. If you have checked out <https://github.com/shageman/package-based-rails-applications>: `git checkout c2s06`.

With the previous section creating one more big change to our package organization and the dependency diagram, there is only one more, smaller thing to tackle. There are two last deprecated dependencies from root onto games and teams. Let's take one more look at the `deprecated_references.yml` for the root package:

Remaining deprecated references of the root package

```

1  ---
2  packages/games:
3    "::Game":
4      violations:
5        - dependency
6      files:
7        - spec/support/object_creation_methods.rb
8  packages/teams:
9    "::Team":
10     violations:
11       - dependency
12     files:
13       - spec/support/object_creation_methods.rb

```

The file that causes these last two violations is `object_creation_methods.rb`. This file is a helper for setting up object state for tests. I have long used this pattern to avoid adding a gem like `factory_bot`, which I have found to make it too easy to not notice when test setup goes from “easy” to “well supported” to “totally reliant on the gem because conjuring up 20+ classes to support one test would hardly be possible manually.” It turns out that writing object creation methods for *simple* objects is simple. For example, check out the two methods Sportsball has for creating teams for tests:

```

1  module ObjectCreationMethods
2    def new_team(overrides = {})
3      defaults = {
4        name: "Some name #{counter}"
5      }
6      Team.new { |team| apply(team, defaults, overrides) }
7    end
8
9    def create_team(overrides = {})
10     new_team(overrides).tap(&:save!)
11   end

```

The issue, with regard to packwerk, comes from the line that states `Team.new` because of the explicit reference to the constant. Maybe some would perceive this as a benefit of `factory_bot`, because due to all its metaprogramming we would never explicitly state the constant we are creating and as such we would never see the dependency. I don't think it is an advantage.

Once again we have a couple of options as to handle this.

We could ignore the dependencies. We could accept the dependencies. We could exclude the dependencies. We could try to remove the dependencies.

The last option would be the most involved and because of that I want to discuss it briefly. The idea would once again be to align test code with the package it belongs to. With that thought we could try to make space in the `teams` and `games` packages to add the respective object creation methods. Once again that idea would map nicely to the work we did in the CBRA approach where gems can define and export their own spec helpers to use themselves and to allow collaborators to use to set up their tests.

Unfortunately, this approach won't work for packages because how differently code gets isolated and loaded. First, "exporting spec helpers" isn't really possible because the test setup isn't isolated as with components. Rather, all packages use the same test setup and thus if we add an object creation method for one, we add them for all. Second, currently object creation methods get made available to specs from within `spec_helper.rb`, like so:

```
1 RSpec.configure do |config|
2   config.include ObjectCreationMethods
3 end
```

`spec_helper.rb` certainly doesn't fit into one package over another and as such, even if we moved the module definition into packages the above code would remain in the root package thus retaining the dependencies between this group of packages.

More changes would be needed to make this approach fruitful, but for this exploration we will leave this idea here.

Accepting the dependencies between the root package and `games` and `teams` doesn't strike me as a good choice because of the difference in the analyses that we just went through between packages specs and object creation methods: In the first case, seeing the dependency was helpful and led us to refactor the app quite significantly. In the second case, we're not sure what to do (and with `factory bot` we wouldn't have seen anything).

Let's step back: The entire application depends on all packages. For all packages, all tests are set up by the application. Seeing the first dependencies was not very enlightening and we removed them. Seeing the second dependencies is maybe just as unenlightening? There are once again a couple of viable positions to take. For the following, we are going to go with the simple solution of excluding the `spec/support` folder from `packwerk`'s analysis. We can do this as follows:

Spec support excluded from packwerk analyses

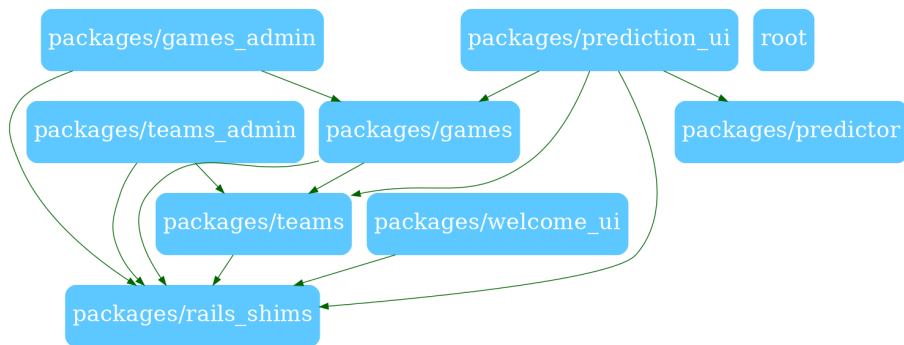
```

1 exclude:
2   - "{bin,node_modules,script,tmp,vendor}/**/*"
3   - "vendor/bundle/**/*"
4   - "**/lib/tasks/**/*rake"
5   - "spec/support/**/*"

```

With this change we are saying that we do not care about dependencies within spec support. At the same time we continue to care about and get notified about any dependencies from root to other packages that might crop up in other places of the app.

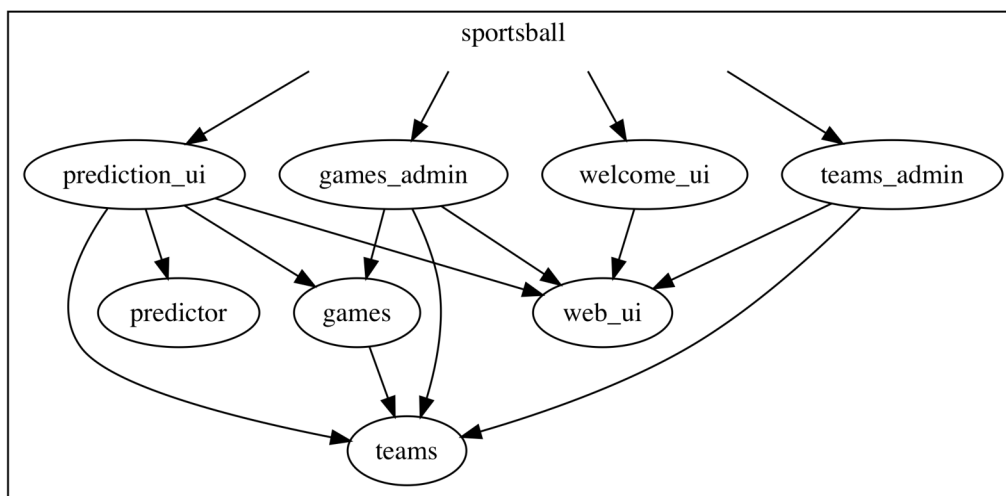
With this change our dependency diagram looks like this:



Package dependency diagram created by pocky

2.7 Recapping differences and similarities to components

Taking another look at the CBRA dependency diagram we can see that our high-level structure is not the same, but very similar.



Sportsball's CBRA structure

The two diagram generation gems are not the same: Pocky is not cbra-deps. As such the diagrams take some squinting to overlay them, but they are structurally very similar.

Of the three big differences I see only one is visible in this diagram:

1. rails-shims is a new concept that CBRA has not been able to expose. This is the one difference visible in the graph.
2. Thinking back to Chapter 1 - with packages we have one test setup for all packages whereas with CBRA we had one test setup per component.
3. With CBRA, we loaded the parts (components) of our applications via gem dependencies, exploiting the fact that path references to gems are possible in Gemfile configs. With packages, we made a few changes to the default load paths of a Rails application.

Finally, the biggest difference is not in the application structure at all, but in the effort that it took to get there. With CBRA, at this point, I was *exhausted*. There was so much to explain. Here, we got to every intermediate step much more simply because the changes to the common way of doing things are so much smaller.

And with that we have a bunch of space to explore all the things we can now build on top of this new foundation.

So let's go!

3. Gradual Modularization: A New Model for Application Composition

The ease of the package modularization work in Chapter 2 is hopefully appreciated by anyone who has worked on structuring any large application in any language and with any framework. Certainly if you have worked with components, you should have been able to follow along with the differences and the dramatic simplification of most of the work.

I believe this to be the case, not only because the tooling we get from packwerk simplifies things, but because we are looking at an innovation that goes far beyond the tool, language, and framework in question here. I believe that the way packwerk allows us to *think* about code modularity is far superior to any approaches we have seen in software engineering up to now. And it will, or at least I hope it will, change the way we work to manage the complexity of large applications going forward.

Why do I believe this? Because the *Gradual Modularization* that we experience with packwerk changes the decisions made by engineers in the process of modularization from “expensive, because time consuming to implement and reverse” to “cheap, no regrets learning through experimentation.”

We can, for the first time, decide to start modularizing a portion of the code in an aspirational way. We can denote that we want a certain part of the code to be a module and not have it done yet. We represent a desired *future state* without getting in the way of needed work in the *current state*. This allows us to create a gradually expanding support system towards a better application structure.

3.1 Gradual Typing as a Role Model

I use the term *Gradual Modularization* leaning on *gradual typing*, a term coined by Jeremy Siek and Walid Taha in 2006 to describe a novel way to type computer software.¹ They describe “languages that literally provide static and dynamic typing in the same program, with the programmer controlling the degree of static checking by annotating function parameters with types, or not. We use the term gradual typing for type systems that provide this capability.”

Why did the authors believe this was such an interesting area to study? “Static and dynamic typing have different strengths, making them better suited for different tasks. Static typing provides early error detection, more efficient program execution, and better documentation, whereas dynamic typing enables rapid development and fast adaptation to changing requirements.” (ibid.)

By combining both approaches within a single language, software engineers can benefit from both of these properties and use whichever is more appropriate in a given context. And they don’t do this only once but can revisit this decision whenever they make a change to the system.

¹Siek, Jeremy; Taha, Walid (September 2006). Gradual Typing for Functional Languages <http://scheme2006.cs.uchicago.edu/13-siek.pdf>. Scheme and Functional Programming 2006. University of Chicago https://en.wikipedia.org/wiki/University_of_Chicago. pp. 81–92.

Since their initial analysis, this way of typing has caught on with languages like Typescript, whose tag line is literally “Typed JavaScript at Any Scale” <https://www.typescriptlang.org/>). “Scale” is not only a happy accident, but a conspicuous similarity in concern for both gradual typing and Gradual Modularization.

For Ruby, we have had Sorbet <https://sorbet.org/>) for a while, which is a form of gradual typing. And, of course, at the end of 2020, with version 3.0 Ruby got its native gradual static typing addition <https://www.ruby-lang.org/en/news/2020/12/25/ruby-3-0-0-released/>) in the form of the new tools RBS <https://github.com/ruby/rbs>) and typeprof <https://github.com/ruby/typeprof>).

In my experience, with dynamically typed languages we tend to write more tests to cover cases that in statically typed languages the compiler would catch. On the contrary, in statically typed languages (and especially strongly typed languages), we tend to spend more time building the type hierarchy (duh). In gradually typed languages we can choose based on a given problem in a specific part of the code, which of the two investments we want to make. Moreover, we have that decision as an option with every change we make.

3.2 Commonalities of previous modularization approaches

When it comes to modularization all existing approaches force us to make an analogous tradeoff. If we choose not to modularize, we get a pliable system where lots of parts can interact without much overhead (similar to the rapid development feature of dynamic typing). If we choose to modularize, we get a structured application that has parts that we can analyze in isolation but that may have “clunkier” interaction patterns because we need to build ways around the isolation of the parts where they are to work together (similar to aspects of static typing).

- Java modules <https://www.oracle.com/corporate/features/understanding-java-9-modules.html>)
- Node modules <https://nodejs.org/api/modules.html>)
- Go packages <https://golang.org/pkg/>)
- Python modules <https://docs.python.org/3/tutorial/modules.html>)
- Clojure libs <https://clojure.org/reference/libs>)

All of them behave in this way. And, as I have repeated too many times already, Rails components do, as well. All of these modules, packages, libs, and components also share the property that they make their external dependencies on other modules, packages, libs, and components explicit.

When initially giving talks on component-based Rails I often got the question: “which component should I extract first?” And my answer would inevitably include: “Consider that when you want to extract one thing. You are probably going to at least extract two: the thing you want and the common stuff that you need access to from the common code you are extracting out of.” While we

can get around this by flipping dependency directions and injecting needed dependencies, there is still a kernel of truth here.

One module tends to beget more modules. The modularized way of thinking about the structure of our application will have us answer questions like: Should this module really depend on that other module? Could it depend on a part of it? Should the dependency direction be the other way around?

In effect, to start with “old-style modularization” has a lot of overhead - both at the start and when working on restructurings.

3.3 Gradularity - Modularization Game Changer

Working with Gradual Modularization is a game changer because, with it as a tool, we can *aspirationally express* the desired state of modularization for a system and we can create the supports to get there, step by step, in a deliberate way.

What makes this possible is that we have moved away from the old-style modularization. Structure is no longer a constituent part of the language, the framework, or library tooling that is enforced and, if violated, breaks the application. Instead, structure becomes a *property* of the system that can be analyzed and interacted with.

With packwerk, we have a tool that currently allows us to enforce *privacy* and *dependency* of parts of application modularization. However, given that this structure is a property that we can analyze, it should be clear that we can also think about other properties that might be useful in a modularization framework.

Here are a couple of ideas in no particular order:

3.3.1 Enforce Visibility

It should be straightforward to add *visibility* as a concept to packages. Visibility would denote that a package will accept usage only for a given set of consumers. This is the inverse of dependency in that the given package states its allowed consumers, not its allowed dependencies.

If the software development org is targeting a package to be owned by one team, then visibility specification gives power to the team owning a package to declare any limitations to outside use. One use case for this is that a package is an implementation detail of another package owned by the same team. This team could use visibility to indicate that the package is effectively *private* to their domain.

For this we could add a setting `visible_to: []` to the configuration. This would add errors to the deprecated references whenever some package uses the given package without being in the list of packages to which the given package says it is visible.

Visibility could, alternatively, be derived from folder hierarchies. I.e, given the following packages

- /package/X
- /package/Y
- /package/Y/subpackages/A

One could derive from this nesting that the only access allowed from a visibility perspective are $X \rightarrow Y$, $Y \rightarrow X$, $Y \rightarrow A$, meaning a package has visibility into all of its peers and into its “subpackages.”

The first implementation idea can express more scenarios. Whereas with folder-based subpackages could only ever have one non-peer package they could be visible to, with the config there is no such limitation. In addition, changes in visibility cause less of a change when it is a configuration setting versus a folder structure. That said, if, for example, one were to expect a huge number of packages as subpackages to one package, say because there is one package doing integrations and there is one integration for every S3-like service, then it might be nice to still have the ability to use subfolders like this. In that way, the subpackages would not clobber the “main” packages folder.

In my experience, nesting components didn’t work well, for the same reasons discussed here. Whichever way it is implemented though, adding visibility would increase the expressive power of the language to describe our desired modularization state.

3.3.2 Allow privacy circumvention

There are cases where it can be interesting for a package to get special access to the non-public parts of another package.

Take the following scenario: Package ABC contains both app and spec code. Test helpers to make objects for ABC are implemented using the object creation method approach discussed in Chapter 2 and that these are in a package named `spec_helpers`. This situation is not resolvable with accepted package dependencies because:

- The test code in ABC depends on `spec_helper` to create objects in tests
- `spec_helper` depends on ABC to refer to the constants of the objects it creates

One solution to this would be to break up the ABC package into `ABC_app` and `ABC_spec`. Now the dependency cycle is broken and the accepted dependencies can be `ABC_spec → spec_helper → ABC_app`.

This solution creates a new potential problem though. If ABC was enforcing its privacy, then any test in `ABC_spec` that tests internals of the app code is now violating the privacy of `ABC_app`.

This can be turned into a virtue: One way to remove the privacy violations is to write tests only against the public API. This tests the package’s functionality like any consuming package would experience it in the application. This, however, does not give us a way to do package-internal unit testing.

This is where another functionality expansion could come in: With a setting like `allow_private_access_from: []`, a package could denote that certain other packages are not considered when privacy violations are being calculated.

We could then even go as far as creating an `ABC_spec_external` and `ABC_spec_internal` as a way to write both privacy-respecting tests as well as internal tests - all while not introducing deprecated references. Incidentally, anyone interested in using package ABC would find examples of how to do so in “`ABC_spec_external`”.



External and internal tests in Go

Incidentally, the same analysis is why Go handles packages with names ending in `_test` in a special way (<https://medium.com/@matryer/5-simple-tips-and-tricks-for-writing-unit-tests-in-golang-619653f90742>). Tests for a package `abc` that are in `abc_test` are forced to be external tests, while tests written directly in `abc` have access to all the internals of a package.

3.3.3 API versioning

When privacy enforcement is turned on for a package, designing a good public API becomes as interesting a task as designing one for a class or library. And as with all software engineering, the first pass is likely not going to be right forever. *Additions* to an existing API tend to be easy: All existing functionality still works. *Changes* to an existing part of the API, however, or *removal* of functionality have the potential of breaking something.

For packages that have a lot of consumers, could packages support such API changes? The problems with lots of consumers is that an API upgrade will cause a lot of forced upstream changes to make the consuming code conform to the new API. Any support to make this kind of upgrade more iterable or allow teams to adopt changes at their own pace, would be helpful here.

The default way packwerk supports the specification of a package’s public API is via the `app/public` folder. Anything inside this folder is public, everything outside is private.

The expansion of packwerk’s capabilities regarding this topic could be a support for *API versioning*.

The change could, for example, be that `app/public` contains the current API and any folder `app/public_*` contains previous, but deprecated, versions of the same API. Using the old versions would be possible in parallel to using the current version, but would add an `api_version_violation` for every use to a violating package.

Changing the semantics of the public API is not without challenges. Most importantly, we have to prevent issues from possibly overriding contents in classes or modules because we don’t separate them between API versions. One way to do this that prevents problems, but is quite verbose, is to wrap the API contents into different API modules, like so. Let’s say the public API is in a module named `Api`.

```
/packages/x/app/public/x/api_v2/api.rb
```

```
1 module X
2   module ApiV2
3     module Api
4       def usage
5       end
6     end
7   end
8 end
```

```
/packages/x/app/public_previous/x/api_v1/api.rb
```

```
1 module X
2   module ApiV1
3     module Api
4       def deprecated_usage
5       end
6     end
7   end
8 end
```

Here `X::ApiV2::Api` is the current public API while `X::ApiV1::Api` contains the deprecated API. Both work, but on one we are able to report a usage violation.

In addition to these inter-package rules, there are a number of rules that are packaging and modularization related that make sense to be enforced on a per-package basis.

3.3.4 Enforcing a package namespace

When working with gems and engines, most code lives inside the namespace derived from the gem name. For a gem `foobar` for example, most code will typically be in `module Foobar`. This is not something we have looked into with packages and with the refactorings from Chapter 2 yet. Separation of namespaces greatly reduces the chances for name collisions of classes and adds context to every class, which makes its fully qualified name more likely to be unique within a large application.

Refactoring towards namespace components or packages is something that I have typically seen happen after the initial moves towards greater modularization, i.e., after the initial extraction refactorings.

A config named `enforce_namespace` could be false by default so it doesn't get in the way of initial refactorings. When set to true, it would add a deprecation violation for any constant defined outside the namespace derived from the package name. When set to a string, it would instead expect every

constant to be defined in the namespace of that string. This could be used if it makes sense for the namespace to diverge from the package name, like, for example, in situations with nested packages where the nested packages express a hierarchy that is to be captured.

In addition to adding the violations, one could suggest during `package check` that a package can have `isolate_namespace` activated when there are no more violations.

3.3.5 Enforcing a separate database

Going one step further than just isolating namespaces would be to enforce the use of an isolated database connection with something like `enforce_separate_database`. The meaning of this would be that all the models in the packages have to use the database connection of the package and no models outside can. Since Rails 6, *multiple databases* are natively supported in ActiveRecord (https://guides.rubyonrails.org/active_record_multiple_databases.html) - the capabilities are right there in the framework!

The implications of the constraint for application development are massive. First, “Applications cannot join across databases” (https://guides.rubyonrails.org/v6.1.4/active_record_multiple_databases.html). For practical purposes this removes the entanglement of models with `belongs_to` or `has_many`. Second, a separate database is a necessary prerequisite for extracting a package into its own application. Should that be the desired destination for the refactoring, turning this enforcement is one of the fastest ways to learn about all the changes needed across the entire stack. E.g., if a model is not yet in the separate database, we won’t be able to move it there while there are still joins with models outside of the package (and inside the package if we are moving models one-by-one). This means that we have to first create the public API that allows collaborators to move off of direction usage (and joins) of the models. The encompassed refactorings will spread to all the collaborators of this package and when successfully completed the possibility of extracting a package as a separate application will be *a lot* closer.

3.3.6 Enforcing a typed API

Piggy-backing on the relationship between Gradual Modularization and gradual typing, let’s turn to typing. With Ruby 3 or Sorbet typing methods both being gradual we don’t have to type everything to type anything. How might we best exploit this when it comes to packages?

I expect the *public API* of a package to contain some of the most interesting code to receive strong typing. This code is the code that will be most often used between packages, which will sometimes or often align with team boundaries, functional boundaries - said another way: Boundaries that likely make sense to explain better and ensure correct usage more. Typing can play an important role in that.

As such, a config `enforce_typed_api` when set to true would add deprecation violations for every method in the `app/public` folder that does not type its params and return values.

Typing the public API will expose when constants from other packages are handed to a given package as parameters. This is a very nice side-effect of doing this, because it will actually add dependency

violations where private constants are used or where constants from packages the given package does not depend on are used.

If the public API of a package is fully typed and there are no violations, there might be some further properties we can make use of when running the system in which these packages are - more on that below.

3.3.7 Enforcing proper documentation

Enforcing documentation is a less technical, but potentially equally important approach to ensuring that good packages evolve over time.

There are a multitude of ways one could try to address this, one simple way of doing this could be `enforce_readme`, which simply verifies that a `readme` doc is present in the root of a package. A more detailed way would be to verify that every method in the public API has an RDoc (<https://github.com/ruby/rdoc>) entry.

Let's turn from *things we can enforce* to *how to enforce things* next.

3.3.8 Configurable failure modes

Modularization strategies are particularly interesting for large applications. Large applications tend to be developed and maintained by a lot of people grouped into many teams. As we just discussed, there are specific issues that stem from many teams working on a domain together. It may be the only way to tackle broad and deep domains, but it creates issues nonetheless.

Teams have separate priorities, their own backlogs, and different timelines for feature development. It is, therefore, prudent to expect that timelines for modularization progress are different too. And once again, we can easily sketch an expansion of current `packwerk` capabilities which allows us to support such a situation even better.

By adding a new setting, `failure_mode:`, which can be set to 'warn', or 'fail', a package owner could be able to specify whether violations in the given package should affect the return code (and thus effectively act as a pass/fail decision) of a `packwerk check` run.

But why stop there? Instead of only allowing the setting of failure mode on the level of a package, it could be configurable on a *per violation* basis (e.g., warn on privacy violations, fail on dependency violations). Or they could be configurable on a *per consumer* or a *per violation per consumer* basis. Or a combination of all of the above.

There is quite a bit of complexity one could bring into the system here and there is certainly a balance of tooling and configuration complexity that should be weighed against the benefits of this and the other proposals. This really holds for all of the ideas and proposals in this chapter.

It is important to not view these ideas as criticisms of the current feature set of `packwerk`. In fact, the correct way to view them is as a massive endorsement of `packwerk` and the novel way of thinking it allows and all of the additional opportunities it unlocks.

But wait, there's more! Beyond reasonably expanding the expressive power of packwerk's core functionality, there are various features we might want to build on top of it.

3.3.9 Here is my code, run it in the cloud for me, I do not care how

```
here is my source code
run it on the cloud for me
i do not care how
```

This haiku was written by Onsi Fakhouri (<https://twitter.com/vmwaretanzu/status/854129165511143425>) while at Pivotal Software. He came up with this to describe how Cloud Foundry (<https://www.cloudfoundry.org/>) was changing a developers relationship to their deployment and how they didn't have to concern themselves with the details of virtual machines (VM) or containers anymore. The abstraction that Cloud Foundry provides was modelled after heroku's deployment offering: Using a *buildpack*, the runtime for an application gets automatically chosen during the deployment process. A developer has to only think about their application and how it connects to external services, for which it gets configuration via environment variables. The needed conventions are summarized as "12 factor applications" (<https://12factor.net/>). While 12 factor apps greatly reduce the need to specify application details for deployment in comparison to a traditional virtual machine or container, we are still defining *before* the deploy that a certain piece of code (an application) *is to be deployed as a certain service*.

The same holds true for "serverless." With this abstraction, developers don't think about the runtime anymore, knowing that the function code will be invoked when events trigger the need for the function to execute. The developer does still specifically denote that a certain piece of functionality *is to be a function and executed as such*.

I believe that packages make a contribution here, too. Once again, we can exploit the fact that the modularization of a system is an analyzable property of an application. A property that can change over time and that can lead to various characteristics to be true for parts of an application at certain points in time.

To be more concrete, imagine a package that enforces its dependencies and its privacy and for which neither have any violations in the system. In fact, let's imagine as a special case that there are no dependencies and that the public API is empty. And, again, no violations by or onto this package. How would such a package be useful in an application? There are ways such a package can contribute value (and functionality!). It just isn't through an API of in-process message invocation, but rather through remote procedure calls or asynchronous message patterns. We'll have more to discuss on this in upcoming chapters.

Given this special case of a package, let's say it only consumes and produces messages on a message broker API. For example, if a message { "company_id": 1, "request": "tax_calculation" } gets sent on a certain channel, this package will calculate a super complicated tax formula (in its own time - it is complicated tax stuff after all). When it is done, it sends a message on a different channel, something like { "company_id": 1, "tax": 42 }.

No other package needs to know the implementation of the tax calculation and because of the way it is implemented, the API can indeed be empty. In fact, it is due to these properties that we could also have deployed this function (the way I have described it thus far, this is a *function* in the mathematical sense) as a serverless / lambda function or a standalone Cloud Foundry service.

Given all of this, we can finally take a look at what a deployment service built on top of something like packwerk could do.

Instead of defining that a certain package should be a service or a lambda function, a packwerk based deployment system could *determine* that a certain package *could* run as a service or lambda function. If a service (just as described above) has

- an empty API,
- no privacy violations,
- no dependency violations (inbound or outbound for the package or its children)

then the deployment system *knows* that this package doesn't need the code from any other packages (it could actually delete it) and that it *could* run this package on its own as a lambda function.

If, during the next deploy, it is determined that the package no longer meets all of the above properties - maybe someone used a piece of code and thus violating the privacy boundary - then the deployment system would see that too and *definitely not* deploy the package as a standalone lambda function.

In an important way "here is my source code, run it in the cloud for me, *I do not care how*" feels even more true with this capability than with what Cloud Foundry and buildpacks can offer.

3.3.10 Analyze to modularize the right stuff

We already saw in Chapter 2 the power of visualizations when it comes to representing the structure of an application as supported by package dependencies and their deprecations.

Graphwek and pocky already implement a bunch of ideas that might come to mind:

- Structure for accepted dependencies
- Structure of deprecated references
- Structure of subsets of packages
- Colors differentiating accepted vs deprecated dependencies
- Thickness of arrow representing the amount of constants forming a dependency connection

Here are some ideas that could help analyzing different aspects of modularization progress:

- Size of components
- Components in/not in cyclic deprecated dependency structures
- Separate representation of dependency vs privacy violations (and similarly all other kinds of violations that might get implemented in the future)

- Separate color or unused accepted dependencies (i.e., an allowed dependency of which no constants are used)
- Indication of the size of the public API for a package
- Indication of the fraction of a public API used by a dependent package

The haphazardness of this list feels weird and points to another opportunity: Visualizations are just one way to represent these kinds of aspects of the dependency structure of an application. For some of the properties discussed above, a non-visual analysis might be much better suited. If the various properties underlying the visualizations described above are themselves treated as data, then we can use them in many other ways.

Monitoring dashboards could be backed by these properties to highlight trends over time.

The data could be aggregated to create a smaller set of modularization quality metrics.

The data could be handed to data scientists who could run who knows what kind of analyses on this data. For example:

- Clustering algorithms to find more strongly connected subgraphs in the overall structure and potentially merge packages to simplify the graph.
- Optimization algorithms to determine the best change to make to impact any metric of the graph. I.e., if I move this constant from this package to that package, the number of violations decreases by X%. (This likely requires more granular dependency information to actually do in practice)

Or we could use it to write code that works on the package graph. For example, a code manipulation bot that moves constants into the public API of a package to reduce privacy violations.

3.3.11 Caveats

Some of the ideas above suffer from the current property of packwerk that it only surfaces details on *deprecated dependencies* and not details about *accepted dependencies*. It also doesn't report on the relationships between all constants. If one still wanted to do the analyses, a brute force way to get around these shortcomings would be to move every file into its own package and remove all accepted dependencies. I expect that having packwerk optionally output this kind of information is not a big change to the internals of its operation, but rather an adjustment of how it presents its results.

There are, however, some deeper caveats to some of the high-flying ideas in this chapter that sit deeply in the how Ruby works and those will be harder to overcome.

It shouldn't come as a surprise that a tool like packwerk was built in a language like Ruby, with dynamic typing, one global namespace, privacy as a second class concept, and isolation of anything being notoriously hard.

It is also no surprise that a tool like packwerk comes out of a company like Shopify. The same features that made it so easy and fast to get started with Ruby and Rails made it tougher to manage growth when the number of developers reached the hundreds and the thousands.

Especially *dynamic typing* and the *global namespace* have interesting consequences for the practical applicability of many of the ideas presented above.

3.3.11.1 One global namespace

While I believe that Ruby's global namespace is an important reason for the creation of packwerk, packwerk does not solve all issues stemming from it. When I flippantly say that a package is "isolated" I am ignoring these issues.

If two packages both add constants directly to the root of the global namespace, they could define the same constant twice. At runtime, this would mean that the module would get reopened and have the union of the functionalities as defined the packages. That is, unless the packages define the same methods twice and have different implementations. Then we have... well nothing good. Just chaos.

We can, of course, enforce that each package defines its constants in the namespace indicated by the package name only. This is, in fact, what most gems and engines do in practice. And this is also what the authors of packwerk recommend <https://github.com/Shopify/packwerk/blob/main/USAGE.md#defining-packages>):

Note: It is helpful to define a namespace that corresponds to the package name and contains at least all the public constants of the package. This makes it more obvious which package a constant is defined in.

Naming collisions (which effectively violate the "closed for modification" principle) are not the only problem. Monkey patching could be used in one package to change the behavior of a module in a different package even if both packages were namespacing all their contents... how strongly can I say "just don't do this?"

Maybe the most subtle way to break the isolation between packages is to change the behavior of some common dependency. Take a Rails application in which one package needs and configures the ActiveSupport cache for the first time. Nothing bad is going to happen at this point: the cache wasn't configured before so it stands to reason that it was unused before. When some time later a second package also starts using the cache, still nothing bad is going to happen, the second package just uses the already configured cache available in the application. It is only when even later the first package gets deleted for some reason that suddenly the tests for the second package fail. The two packages were not completely isolated.

It is a good point to make that "this is a bad example! A package shouldn't modify anything in the global namespace!" - I think that is right, but in practice really difficult to detect and without removing Ruby's dynamism impossible for all edge cases. In fact, a package adding a namespace to the global namespace is, even if small, just that: a change of the global namespace.

With a language like Ruby, isolation may never be perfect. I believe that is not the goal we should be shooting for. Packwerk shows us the way to make impactful, practical steps in better modularization for Ruby and Rails. I believe these practical benefits will make this approach spread to other languages and frameworks, in their own variations, in due time.

To manage these caveats in practice, we can hold ourselves to standards that will make sure that we can use packages for a lot of what we have already seen and the presented ideas for the future.

3.3.12 Managing the caveats

A package with no dependency violations and no privacy violations is a good start.

Ensuring that code within a package defines modules only in the package’s namespace is a very impactful next step.

Another big step is *statically typing the public interface*, which we will have to turn to in detail in the next chapters. As a sneak peak of the problem, picture this method as part of the public API of a package:

```
1  def compute_year_end_taxes_for(company)
2    # ... some, probably simple, calculation
3  end
```

What is that `company`? Is that maybe one of the main ActiveRecord objects of this application that has `has_one` and `has_many` relationships to all sorts of other models and that accesses through the many methods defined on the object even more other classes in the application? Oops. There goes our isolation.

Or is `company` a value object with a bunch of fields? Our isolation is probably fine.

If we had typing as in the example below, which uses Sorbet <https://sorbet.org/>), we (and packwerk!) can easily tell.

```
1  sig {params(company: Company).returns(Tax)}
2  def compute_year_end_taxes_for(company)
3    # ... not as simple as we thought after all
4  end
```

I am standing on the shoulder of giants when adopting the word “gradual” from research into gradual typing. Is what we are looking at here really the modularization analog of that transformative idea in how we type applications?

The original paper on gradual typing is analyzing its topic in a degree of formality that I can not replicate here <http://scheme2006.cs.uchicago.edu/13-siek.pdf>). Intuitively though, “old-style modularization” shares its rigidity and certainty properties with static typing. Package-based

modularization (very unscientifically) shares some similarity with gradual typing through the notion that we can define and lock down certain aspects of modularization at build time, but whatever happens at runtime is fine as long as it works. Anyone up for researching this further?

For those not super hot on academic research, maybe some tinkering with Ruby is a better next step? What if Ruby allow a module to specify that it isn't open for modification? This starts to alleviate the problem of naming. What if a module could state that it only can depend on modules that are closed for modification? What if this property could be managed through a list of violations so it becomes as iterable as Gradual Modularization?

4. Making Components work with Packages

For those that went with components and have them in their applications today, the degree to which we can integrate components and packages is likely to be the one question standing in the way of adopting YAMS (yet another modularization strategy). Remember, components is the term I use for using unbuilt gems, gems we leave in the application codebase, to be used as a vehicle for creating large scale code isolation.

The good news is that the barrier to entry is extremely low for components that are already engines and manageable for gems that are not. We'll tackle both situations in turn.

4.1 Engines



The code discussed in this section is available at <https://github.com/shageman/package-based-rails-applications-book/tree/main/c4s01/sportsball>. If you have checked out <https://github.com/shageman/package-based-rails-applications>: `git checkout c4s01`.

Let's say a component `testengine` is registered in the `Gemfile` as follows:

Gemfile entry adding `testengine` to the app

```
1 gem 'testengine', path: 'testengine'
```

If you want to follow along, but don't have a component in your application, you can easily create one with the following command. Execute it from the root of the application:

Generating an engine

```
1 $ rails plugin new testengine \
2   --full \
3   --mountable
```

We can check what packwerk thinks of this by running its validations:

Output of `packwerk validate` for an engine

```
1 $ bundle exec packwerk validate
2 Packwerk is running validation...
3
4 Validation successful !
```

This output does not, however, tell us whether the engine's constants are being considered in packwerk's analysis. In order to check that, we need to open the lid on how packwerk finds out which folders to work on. In older versions of packwerk (prior to commit <https://github.com/Shopify/packwerk/commit/702d2dce1b9b47bc3b72173eae1e1a872bc57efc>) there was an explicit `load_paths` configuration in `packwerk.yml`. Since that commit, packwerk utilizes the Rails app's autoload paths to determine which folders to include. Within a running application, these folders are stored in `ActiveSupport::Dependencies.autoload_paths`. We can use the following one-liner to check which autoload paths are in the app at this point.

Autoload paths after adding an engine

```
1 $ bundle exec rails r 'puts ActiveSupport::Dependencies.autoload_paths'
2 packages/games/app/models
3 packages/games_admin/app/controllers
4 packages/games_admin/app/views
5 packages/prediction_ui/app/controllers
6 packages/prediction_ui/app/helpers
7 packages/prediction_ui/app/views
8 packages/predictor/app/models
9 packages/rails_shims/app/channels
10 packages/rails_shims/app/controllers
11 packages/rails_shims/app/controllers/concerns
12 packages/rails_shims/app/helpers
13 packages/rails_shims/app/jobs
14 packages/rails_shims/app/mailers
15 packages/rails_shims/app/models
16 packages/rails_shims/app/models/concerns
17 packages/teams/app/models
18 packages/teams_admin/app/controllers
19 packages/teams_admin/app/views
20 packages/welcome_ui/app/controllers
21 packages/welcome_ui/app/views
22 testengine/app/controllers
23 testengine/app/controllers/concerns
24 testengine/app/helpers
25 testengine/app/jobs
26 testengine/app/mailers
```

```

27 testengine/app/models
28 testengine/app/models/concerns
29 ~/.rvm/.../gems/actionmailbox-7.0.1/app/controllers
30 ~/.rvm/.../gems/actionmailbox-7.0.1/app/jobs
31 ~/.rvm/.../gems/actionmailbox-7.0.1/app/models
32 ~/.rvm/.../gems/activestorage-7.0.1/app/controllers
33 ~/.rvm/.../gems/activestorage-7.0.1/app/controllers/concerns
34 ~/.rvm/.../gems/activestorage-7.0.1/app/jobs
35 ~/.rvm/.../gems/activestorage-7.0.1/app/models
36 spec/mailers/previews

```

This output is fantastic news for anyone hoping to add testengine into their package-based modularization regime. The fact that folders underneath testengine/app are autoloaded means that packwerk will immediately work with them.

The reason for this are the following lines from packwerk's source code: https://github.com/Shopify/packwerk/blob/4033012974e46a5c02013fcd3f2e479ff55477/lib/packwerk/application_load_paths.rb

Packwerk's extract_application_autoload_paths implementation

```

1 def extract_application_autoload_paths
2   Rails.application.railties
3     .select { |railtie| railtie.is_a?(Rails::Engine) }
4     .push(Rails.application)
5     .flat_map do |engine|
6       paths = (engine.config.autoload_paths + engine.config.eager_load_paths + engine\
7 e.config.autoload_once_paths)
8       paths.map(&:to_s).uniq
9     end
10 end

```

Two lines of note here tell us that 1/ only Rails::Engine's are being processed and 2/ that for every engine all of its autoload, eagerload, and autoload_once paths are returned.

In other words: To include engines as packages in our applications, all we have to do is add a package.yml to the root of the engine's folder.

testengine/package.yml

```

1 enforce_dependencies: true
2 enforce_privacy: false

```

And because Rails engines, by default, happily autoload all paths under ./app you can move the engine's public code into app/public and activate enforce_privacy. When doing this, be sure to maintain any subfolder structures to ensure that any modules or nested modules are implementing Rails' autoloading folder conventions.

4.2 Gems



The code discussed in this section is available at <https://github.com/shageman/package-based-rails-applications-book/tree/main/c4s02/sportsball>. If you have checked out <https://github.com/shageman/package-based-rails-applications>: `git checkout c4s02`.

It will turn out that adding components that are gems but not engines to packwerk is more difficult to do and comes with a rather uncomfortable tradeoff. This, for the record, is probably the only time that anything is really *easier* with engines.

Let's say a component `testgem` is registered in the Gemfile as follows:

Gemfile entry adding `testgem` to the app

```
1 gem 'testgem', path: 'testgem'
```

To generate such a gem, execute this command from the root of the application:

Generating an engine

```
1 $ bundle gem testgem --no-coc --no-ext --no-mit --no-rubocop --test=rspec --ci=github
2
3 $ rm -rf testgem/.git
4 $ rm -rf testgem/.gitignore
```

Different versions of bundler have slightly different parameters, so adjust the first command as necessary. Removing git's files and folders is a reminder for all of those that actually want to use components that without doing this, the code within the gem won't actually make it into the repository.

Once again, we can check what packwerk thinks of this by running its validations. However, before this command will successfully run, all "TODO" entries in the `gemspec` file for the gem need to be fixed.

Output of `packwerk validate` for a gem

```
1 $ bundle exec packwerk validate
2 Packwerk is running validation...
3
4 Validation successful !
```

So far so good. Let's again check the application's autoload paths.

Autoload paths after adding a gem

```
1 $ bundle exec rails r 'puts ActiveSupport::Dependencies.autoload_paths'
2 packages/games/app/models
3 packages/games_admin/app/controllers
4 packages/games_admin/app/views
5 packages/prediction_ui/app/controllers
6 packages/prediction_ui/app/helpers
7 packages/prediction_ui/app/views
8 packages/predictor/app/models
9 packages/rails_shims/app/channels
10 packages/rails_shims/app/controllers
11 packages/rails_shims/app/controllers/concerns
12 packages/rails_shims/app/helpers
13 packages/rails_shims/app/jobs
14 packages/rails_shims/app/mailers
15 packages/rails_shims/app/models
16 packages/rails_shims/app/models/concerns
17 packages/teams/app/models
18 packages/teams_admin/app/controllers
19 packages/teams_admin/app/views
20 packages/welcome_ui/app/controllers
21 packages/welcome_ui/app/views
22 ~/.rvm/.../gems/actionmailbox-7.0.1/app/controllers
23 ~/.rvm/.../gems/actionmailbox-7.0.1/app/jobs
24 ~/.rvm/.../gems/actionmailbox-7.0.1/app/models
25 ~/.rvm/.../gems/activestorage-7.0.1/app/controllers
26 ~/.rvm/.../gems/activestorage-7.0.1/app/controllers/concerns
27 ~/.rvm/.../gems/activestorage-7.0.1/app/jobs
28 ~/.rvm/.../gems/activestorage-7.0.1/app/models
29 spec/mailers/previews
```

While packwerk is happy with this run, we, decidedly, are not. As we can see from the above output, the gem's folders are not listed because they are not autoloaded. This means that the gem's folders are not being recognized by packwerk.

Given that we saw this work for an engine nad see that it is not working now, we know that we are facing at least two problems:

- This gem is no engine and
- there are no autoload paths.

For the latter, remember that the `lib` folder (even in Rails applications) does not get autoloaded by default. Also, gems, by default, have no `app` folder:

Folder structure of a generated gem

```
1 $ tree .
2 .
3 └─ Gemfile
4 └─ README.md
5 └─ Rakefile
6 └─ bin
7   └─ console
8   └─ setup
9 └─ lib
10  └─ testgem
11    └─ version.rb
12    └─ testgem.rb
13 └─ spec
14   └─ spec_helper.rb
15   └─ testgem_spec.rb
16 └─ testgem.gemspec
17
18 4 directories, 10 files
```

Here, we are at a crossroads. There are two ways to get a gem to be picked up by packwerk.

4.2.1 Converting the gem fully into an engine

The first option is to convert the gem into an engine with all its trimmings. I believe the most practical way to do this to generate a new engine with the command from the beginning of this chapter¹ and then copy the non-boilerplate files from the gem into the new engine. Make sure to also, copy any external gem dependencies into the new gemspec.

If this feels like a dull approach with a lot of downsides, that's because it is: All of the default folders and especially the overhead for testing, most notably the spec/dummy folder, come into play with this approach.

Converting a simple gem into an engine in this way also changes the dependencies - at the very least it makes the gem dependent on Rails.

The big upside of this approach is that, afterwards, the gem is in fact an engine. Wait what? Doesn't that create a lot of overhead? To dig into why this might be an advantage, let's check out the second possible approach.

¹rails plugin new testengine --full --mountable

4.2.2 Converting the gem into an engine (as little as possible!)



The code discussed in this section is available at <https://github.com/shageman/package-based-rails-applications-book/tree/main/c4s03/sportsball>. If you have checked out <https://github.com/shageman/package-based-rails-applications>: `git checkout c4s03`.

If the full engine conversion feels too heavy handed, like a sledge hammer to the problem perhaps, there is another option. We can try to employ something smaller - a little finishing hammer maybe. But enough butchering analogies. What was the problem again?

The reason for packwerk not picking up our gem is that it is not an engine and it doesn't have autoload paths.

A thing about packwerk that we haven't covered yet is that packwerk only works in Rails applications that use zeitwerk (<https://github.com/fxn/zeitwerk>). Zeitwerk is "enabled by default in Rails 6 applications running on CRuby" (https://guides.rubyonrails.org/autoloading_and_reloading_constants.html#enabling-zeitwerk-mode).

Zeitwerk's role in Rails is to tie filenames to constants and vice versa. I.e., it is zeitwerk that defines and enforces that a file `./app/models/domain/super_secret/main_line.rb` defines a constant `Domain::SuperSecret::MainLine` (whatever the concept of that might be).

Knowing this, we can sketch out the alternative approach to "enginifying" and autoloading the gem's code. If we activate zeitwerk within the gem, we can then work to make the gem's code autoloading compatible in the way that an engine would do it. Having done that, we know that the code would behave the same if loaded by an engine. The gem's isolated tests don't have to change for this as we are only changing *how* the code gets loaded. This solves one of the two problems, namely that the gem is autoloading-compatible.

To solve the other problem, we need to actually create an engine for the gem. In contrast to our first approach, when converting "minimally" we can do so without the gem depending on Rails. We can do this by loading just an engine if the gem finds itself in a context that has Rails already loaded.

Let's do this step by step.

First, add zeitwerk as a dependency to the gem's gemspec. We can make this a development-only dependency because, as we just discussed, we will rely on an engine's loading mechanisms (which are also zeitwerk under the hood) whenever the gem is in a Rails context. And when will the gem be in a Rails context? Whenever it is actually used - i.e., in *production*. To make sure our autoloading is set up properly, we add zeitwerk for development.

Adding zeitwerk to testgem/testgem.gemspec

```
1 Gem::Specification.new do |spec|
2
3   ...
4
5   spec.add_development_dependency 'zeitwerk'
6 end
```

A minimal engine file for testgem looks as follows. This code itself is straightforward, but as the usage of `::Rails::Engine` makes clear: this code needs Rails to be loaded to work.

A minimal engine file: testgem/lib/testgem/engine.rb

```
1 echo "module Testgem
2   class Engine < ::Rails::Engine
3     isolate_namespace Testgem
4   end
5 end
```

Ensuring that engine.rb only gets used when Rails is active, we can use Ruby's `defined?(Rails)` call as seen in the following implementation of the gem's entry point file testgem/lib/testgem.rb.

A version of testgem/lib/testgem.rb loads zeitwerk or the engine

```
1 # frozen_string_literal: true
2
3 if defined?(Rails)
4   require 'testgem/engine'
5 else
6   require 'zeitwerk'
7   loader = Zeitwerk::Loader.new
8   loader.tag = File.basename(__FILE__, '.rb')
9   loader.inflector = Zeitwerk::GemInflector.new(__FILE__)
10  app_paths = Dir.glob(File.expand_path(File.join(__dir__, '../app', '/*')))
11  app_paths.each { |k| loader.push_dir(k) }
12  loader.setup
13 end
14
15 require_relative 'testgem/version'
16
17 module Testgem
18   class Error < StandardError; end
19   # Your code goes here...
20 end
```

The bottom part (after the `require_relative`) is the original file. Obviously, the biggest complexity comes from the lines above, specifically the ones loading zeitwerk. First, note that we don't *always* load zeitwerk but rather have a switch between engine and zeitwerk. The reason we can't always call the zeitwerk code is that, if we did, the zeitwerk config loaded here directly would conflict with the one coming from the internals of the engine.

The code loading and configuring zeitwerk is fairly standard for the gem and the various lines can be analyzed by referencing the gem's documentation (<https://github.com/fxn/zeitwerk#synopsis>).

For our purposes the most relevant lines are the ones collecting and then “pushing” the `app_paths`. First, we load all folders under the `./app` folder into an array and then use zeitwerk's `push_dir` method to register all of these folders with zeitwerk. These lines ensure that zeitwerk, in development mode, autoloading the same files that the engine will in production mode.

The big, massive, cringe worthy downside is that we just created a switch in our code that has us run one code path in tests and another in production ... and we can't test that they are the same without going back to the first approach to making the gem packwerk compatible. The best chance we have at verifying overall system behavior is to add at least one integration test in the main application that verifies some feature of the gem and by extension ensures that our autoloading is set up properly. And you are writing this kind of test anyways, right?

If we now create a folder like `./app/services` inside the gem and run packwerk validations, we'll see that the folder gets picked up:

```
1 $ bundle exec rails r 'puts ActiveSupport::Dependencies.autoload_paths'
2 packages/games/app/models
3 packages/games_admin/app/controllers
4 packages/games_admin/app/views
5 packages/prediction_ui/app/controllers
6 packages/prediction_ui/app/helpers
7 packages/prediction_ui/app/views
8 packages/predictor/app/models
9 packages/rails_shims/app/channels
10 packages/rails_shims/app/controllers
11 packages/rails_shims/app/controllers/concerns
12 packages/rails_shims/app/helpers
13 packages/rails_shims/app/jobs
14 packages/rails_shims/app/mailers
15 packages/rails_shims/app/models
16 packages/rails_shims/app/models/concerns
17 packages/teams/app/models
18 packages/teams_admin/app/controllers
19 packages/teams_admin/app/views
20 packages/welcome_ui/app/controllers
21 packages/welcome_ui/app/views
```

```

22 testgem/app/services
23 ~/.rvm/.../gems/actionmailbox-7.0.1/app/controllers
24 ~/.rvm/.../gems/actionmailbox-7.0.1/app/jobs
25 ~/.rvm/.../gems/actionmailbox-7.0.1/app/models
26 ~/.rvm/.../gems/activestorage-7.0.1/app/controllers
27 ~/.rvm/.../gems/activestorage-7.0.1/app/controllers/concerns
28 ~/.rvm/.../gems/activestorage-7.0.1/app/jobs
29 ~/.rvm/.../gems/activestorage-7.0.1/app/models
30 spec/mailers/previews

```

We can also now add a class into this folder and add a test in the spec folder. When running rake spec we can verify that this works without any additional manual require or require_relative statements.

A super useful autoloaded class `testgem/app/services/testgem/sample.rb`

```

1 echo "module Testgem
2   class Sample
3     def test
4       3
5     end
6   end
7 end

```

A test as rivetting as the implementation is useful `testgem/spec/services/testgem/sample_spec.rb`

```

1 # frozen_string_literal: true
2
3 RSpec.describe Testgem::Sample do
4   it 'returns 3 when tested' do
5     expect(subject.test).to eq(3)
6   end
7 end

```

In general, if you use this approach - in either variant, really - the largest part of your work will be to appropriately move the domain-implementing files from the `lib` folder (where they will typically be for a gem) into the `app` folder. In that process, any inconsistencies between constant naming and file/folder naming will be surfaced by zeitwerk and the tests you should have for the gem.

So, should one make gems packwerk compatible?

Given the overhead of making gems “compatible” with packwerk, I recommend considering this for situations where gems will likely stick around for a while and where they contain quite a bit of code representing a significant part of you app’s domain. It is in these cases that the additional

protections we gain from adding packwerk have a chance of outweighing the downsides of the conversion process.

For engines, the tradeoff is simpler: just do it.

5. Dependency Violation Management Refactorings

Chapter 2 ended with a version of Sportsball that had no more package violations. Most of them were removed by accepting them as package dependencies. Remember that we also started the app with a version of Sportsball that had no violations: At the beginning there was only one package and that *also* meant that there were no violations. So we *started* without violations and we *ended* without violations. There are at least two interesting questions here:

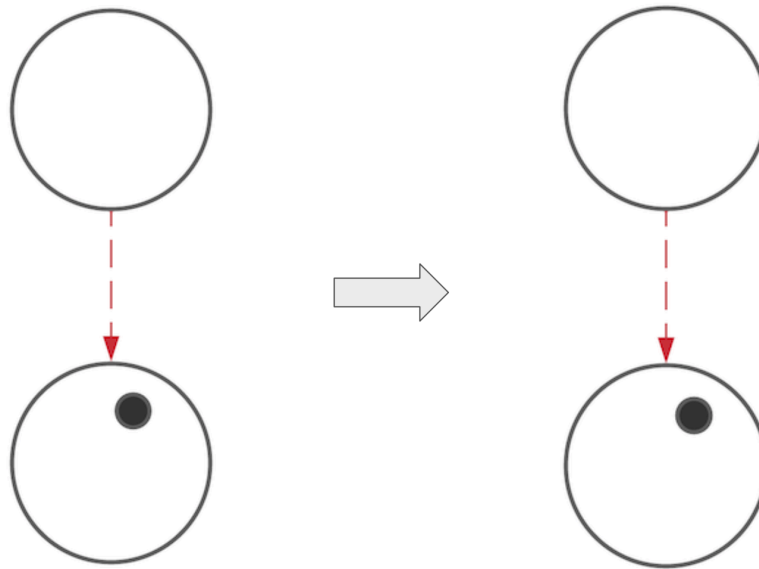
- We only *removed* violations by accepting dependencies. Do we have *other* means of removing violations? If so, what are they?
- How can we compare the various versions of Sportsball throughout the refactorings in terms of *quality*? I.e., how do we know which direction of the refactoring we want to go? Could we, for example, consider a refactoring that works the other way, from the back to the Chapter to the front. Would that be *good*?

Let's tackle the first question in this Chapter and the second one later. This may seem like the wrong order. After all, how should we know which kind of refactorings we want to do if we don't know where we are going. An analogy might help explain the order: let's first figure out what our tools are and later try turn to what stuff we can build with them.

This chapter contains all the violation removal refactorings I have seen in practice. Writing that feels very unsatisfactory, because I am saying that I am not sure whether there are other ways of removing violations out there. The saving grace is that the refactorings listed here have thus far removed every packaging barrier I have come across. As such, this list is at least a good starting point.

I have decided to present the refactorings in the rough order from smallest to largest change - by approximation anyways.

5.1 Do absolutely nothing



The package-effect of doing nothing

Let's take a deep breath and celebrate the fact that there is a very powerful *fix* that we have with packages that takes *no work at all*: do nothing about the violation. Accept a violation (not as a dependency!) as the violation that it is.

As we already saw in [Chapter 2](#), we are able to create an app that functions perfectly well and has packwerk violations in it all over the place. This in itself is a stark departure from previous modularization techniques and approaches. If we had built a Ruby gem or a Java gradle build or a go package which needed some dependency to pass its tests (but didn't declare it), the tests would not pass. And, of these three, only in Ruby might you get lucky and, despite the tests not passing, all the missing stuff was unexpectedly added by the rest of the app to the global namespace. Java and Go programs exhibiting this problem would simply not compile.

I have tried various analogies for the magic that is this property of packwerk packages and Gradual Modularization. A weird version of Schroedinger's cat comes to mind: the boundaries are only there if you look for them. The Ruby interpreter doesn't look, so it doesn't have any problems running the app. We *can* look (and we can let our continuous integration system look) and enforce these boundaries as much as we like to and no more.

Maybe a less high-flying analogy is easier. Have you ever had a friend tell you that traffic signs are just suggestions? Well, while that is not true for most traffic signs, it is true for packages. We *can* but do not *have to* heed them.

So, when might we want to look and care vs when might we not? A reason to care is the reason we turn to any modularization approaches in the first place. We are trying to find ways to manage the complexity of the systems we are building. *Microservices* is "looking," *SOA* is "looking," *components*

is “looking.”

When might one not want to look? Consider this analogy: When would you ever ignore street signs? An ambulance does to a certain extent (with a lot of noise) to get to places faster.

Imagine the following situation: You have a well modularized application with a couple of modules that have only a few dependencies drawn between them. And let’s say that a new feature is promising to close the huge deal with an important customer, but only if it can be part of the offering in the next couple of weeks. What if this new feature would require a completely novel combination of all the modules in the system? In this case, we might decide that we want to ignore the boundaries we defined for ourselves in the system and instead connect them in the most expedient way to get the feature out. We will accept violations of the package structure because we put a higher priority on a short-term business need. Later, we can see how we want to (if at all) redefine the boundaries and dependencies to get back to a system that is better modularized.

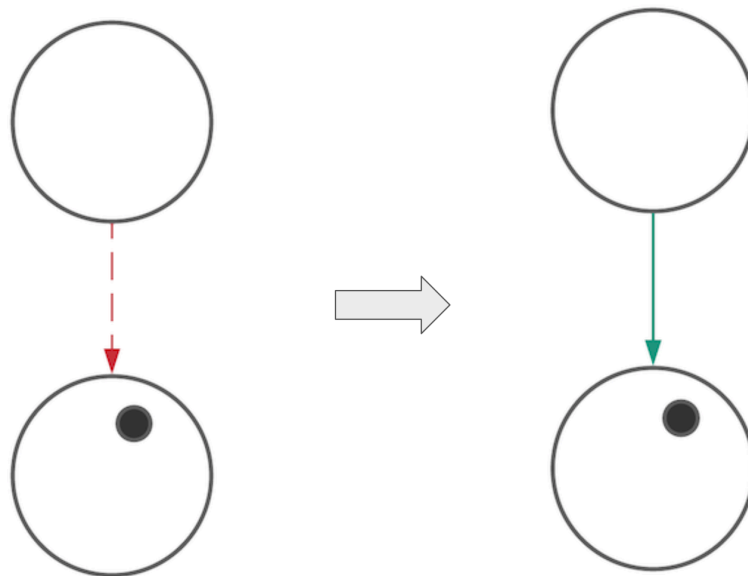
The above example shows a partial answer to the second question posed at the top of this chapter: the direction to go in next depends on which kind of improvement (technical or business) we currently want to prioritize.

Consider how in a microservices architecture we might handle the business case described above. One way I have seen it done is to build a new service that reaches directly into the databases of the services we want to connect in a novel way. Unfortunately, with that approach, you potentially lose out on a part of the business logic present in the services. This is also, to many folks, a big big no no, because we are breaking an abstraction that was carefully crafted around the datastores. And, likely, in order to keep the new services functioning well you might all but stop development on the original services or be faced with having to manage a synchronized deploy of all the services if the data structures need to change. Comparing that again to packages, all we did was change the internals of one monolith. Nothing outrageous happened (because we can sometimes afford to “not look”).

Just as there are reasons for and against caring about the boundaries, there are costs associated with caring or not caring about the boundaries. Once again, a good modularization promises massive benefits for software maintenance in a multitude of ways. The modularization itself comes with costs, too. Similarly, *not* modularizing comes with increased costs of the maintenance of the system. Lastly, not doing anything about violations in a system in which we state that we want certain modules to be present comes with the very visible overhead of constantly being reminded of the intended structure (at a minimum via the folder structure) and the reality of violations stemming from actual undesired dependencies within the system.

It is as they say: nothing in life is free.

5.2 Accept the dependency



The package-effect of accepting the dependency

The simplest action we can take to remove a dependency violation is to *accept it*. This is the strategy we extensively used in [Chapter 2](#). Every use of this strategy “costs” us a one-line addition to the `package.yml`’s section of dependencies.

When introducing packages into an existing application, one that maybe feels a bit like that *ball-of-mud* you’ve heard about, you might quickly run into the issue that packwerk rejects the addition of a dependency. That is because packwerk doesn’t allow introducing cyclic package relationships. And given that *ball-of-mud* specifically refers to loads of stuff being entangled within the app there are basically only two outcomes: you get cyclic dependencies or you should still be splitting things up into more packages.

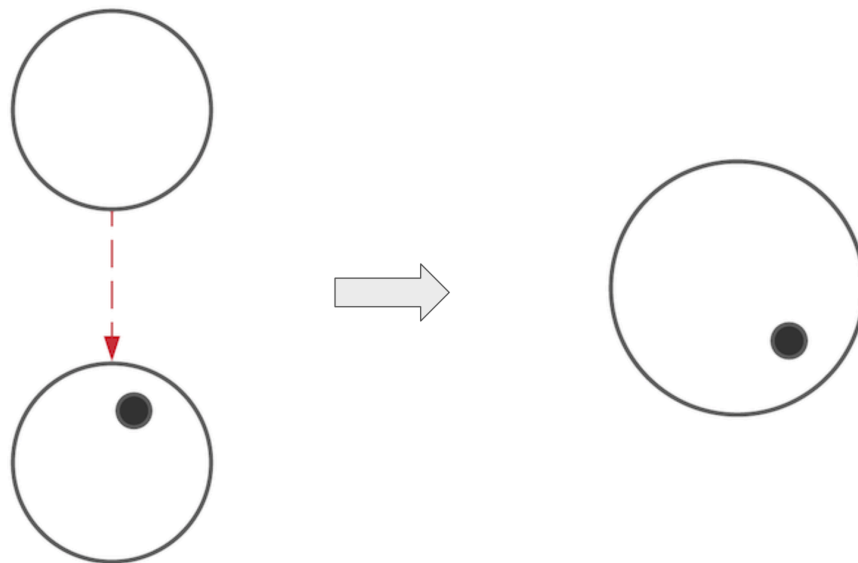
If references are going both ways between packages, in which direction should they be accepted? One could go with an optimization approach to this: count the number of references in both directions and add the dependency that removes more deprecated references. This may feel good and improve quality statistics, but makes it feel like the idiosyncrasies of the structure existing today are dictating the future structure. Packwerk gives us the opportunity to *design* the desired structure for our applications, which makes this rationale for choice of acceptance direction feel lacking.

As an alternative: read the names of the two packages out loud and start a discussion among the teams involved in this part of the application to answer which package feels more foundational, “deeper in the domain,” producing or offering something that the other package needs. If this is not a difficult discussion but rather “totally obvious” to everyone involved, then add the dependency from the less-foundational to the more-foundational package. However, if the discussion doesn’t reveal an obvious direction, consider not adding either dependency. Without a clear *direction* there

isn't a clear *dependency* - just entanglement. Split off more functionality into smaller packages and look into the other violation removal techniques discussed in this chapter to reduce the overall entanglement. Then, regularly return to the question of which dependency direction feels right.

Activating privacy enforcement adds another wrinkle: Privacy violations are unaffected by the addition of accepted dependencies. We'll cover how to handle this later on in this chapter.

5.3 Merge the two packages



The package-effect of merging packages

Ever so slightly more involved than a dependency addition is *merging packages*. When merging packages we have the option of two slightly different paths, namely to

- Create a new package with the desired folder name and move all the contents of the two existing packages into the new package. The only files needing special treatment are `package.yml` (which we create fresh for the new package), `deprecated_references.yml` (which we delete and regenerate), and other non-code files like a `README` which we merge or update.
- Move the contents of one of the packages into the other, treating the same files specifically once again.

Merging packages always removes all dependency and privacy violations that exist between them. Combine this amazing outcome with the simplicity of performing this change, and we have the most efficient way to ... well, what are doing here? Modularizing? Really, it is the opposite. We are *de-modularizing*. Keep merging packages and you end up with no boundaries enforced at all (back to the ball-of-mud).

So, when should we merge packages? More broadly we will tackle this question in the next chapter - a short analysis will do here.

We discussed previously how accepting dependencies likely leads to cyclic references. This problem gives a first indication of what may ultimately lead to merging packages: That is when packages are not good or useful splits of functionality in the application. Specifically, there are a couple of indicators that, when observed together or more strongly, indicate that two packages may be better as one:

- The more the two given packages are dependent on each other - especially in relation to their size (i.e., this may mean less for large packages and a lot more for small ones)
- The more other packages consume the two given packages' services together
- The more the two given packages consume other packages' services together
- The frequency with which changes in one of the given packages happen in conjunction with changes in the other

These indicators are close relatives to the principles of *package cohesion* compiled by Robert Martin (<https://wiki.c2.com/?ReuseReleaseEquivalencePrinciple>):

- *Common Closure Principle* (<https://wiki.c2.com/?CommonClosurePrinciple>)

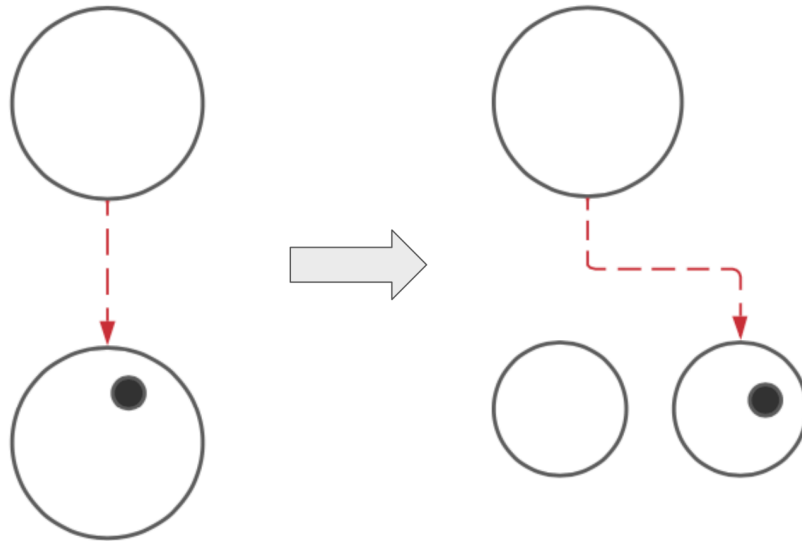
Classes within a released component should share common closure. That is, if one needs to be changed, they all are likely to need to be changed. What affects one, affects all.

- *Common Reuse Principle* (<https://wiki.c2.com/?CommonReusePrinciple>)

The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.

The third principle in this list, *Reuse Release Equivalence Principle*, doesn't fit our discussion as it is about the explicit release of packages, something we are not tackling for the moment.

5.4 Split the violated package



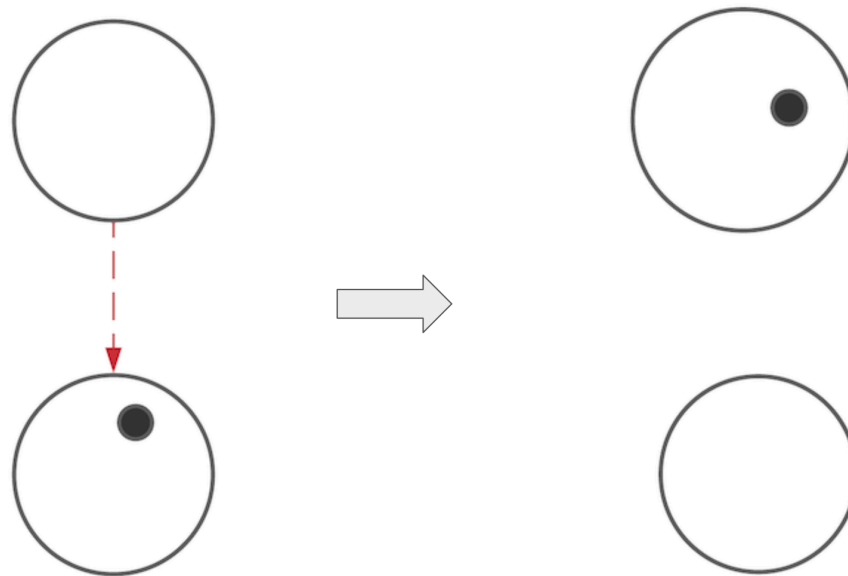
The package-effect of splitting packages

The inverse to merging packages is splitting packages. Splitting packages is unique in this list of refactorings in that it will never reduce the overall number of violations and will likely increase them. Nonetheless, one might call it the most essential package refactoring as it is the simplest way to create more packages.

In the best case, you get the dependency violation pattern depicted in Figure [The package-effect of splitting packages](#). The original package, which had the dependency violation on it, no longer does. But there is now a new package which basically has “taken over” the violations of the original package.

To split two packages, we follow the first variant of the merge package package refactoring with the only different that we only move parts of the existing package into the newly created package. By running `packwerk update-deprecation` we can confirm that we have moved all of the dependency violations we intended to move.

5.5 Move the code between packages



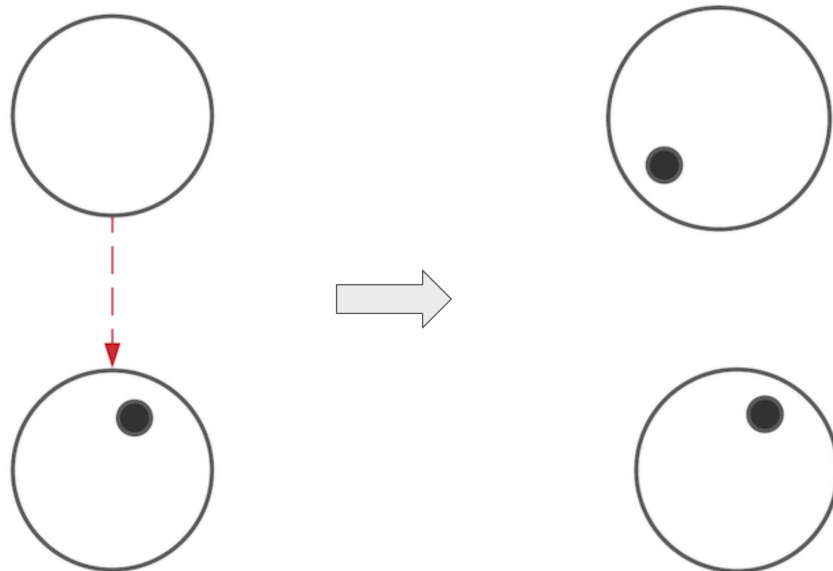
The package-effect of moving functionality

The most egregious violation of the Common Reuse Principle may be when some functionality is used in one package but not in the package that it is defined in. In this case, simply moving the code from the original package to the consuming package will remove all privacy and dependency violations from the latter.

This refactoring is not hard to do, but it is potentially difficult to see when it is appropriate. Packwerk makes it very easy to see that a package needs some code from another package, but it is not as easy to see whether that package needs that code too. That is, because we don't get usage information within packages from packwerk we need to rely on our knowledge of the system, sleuthing, and likely tests to understand whether code could be moved. I venture to guess that this refactoring opportunity is rare in most systems and given that, I would suggest not searching for this but rather being content when you come across and recognize it.

It should also be noted that if multiple packages depend on the functionality but the original one does not, we're likely looking at the case of a package that acts as a *utility package*. Moving the functionality out in this case does not make the system better, especially when the two consumers do not currently have a dependency relationship. To make it very obvious, imagine a package called `string_algorithms` that implements a `l33t` conversion and two consumers, `admin_interface` and `end_user_interface` don't depend on each other. Neither of the consumers makes a very convincing package to contain `l33t` themselves and `string_algorithms` feels like a pretty solid choice.

5.6 Duplicate the functionality



The package-effect of duplicating functionality

As an industry, we seem to produce examples for the rationale of reimplementing even common functionality ourselves every couple of years. Early in 2021 the yanking of version 0.3.6 of the *mimemagic* gem (<https://github.com/rails/rails/issues/41750>) caused quite a bit of scrambling across the Rails-user community. And if this doesn't seem like the perfect example (say because the contribution of this gem is "meaningful"), there is always the 2016 incident of the yanking of *leftpad*: a published javascript library whose sole community contribution is the addition of padding to the left of a string (https://www.theregister.com/2016/03/23/npm_left_pad_chaos/).

Reimplementing functionality can mean simply copy-paste, but it can also be more subtle. When the functionality consumed from another package is relatively small, and in some way "more than we need," consider *reimplementing the functionality* instead of consuming it from a different package. For example, imagine you use one function from another package that implements "find median element." Let's say that function can do this for all collections of objects. You might not want to reimplement that, because it seems too complex to maintain. Let us additionally assume that in your use-case you always deal with a *full, sorted array of integers*. On closer analysis of this fact you realize that this means that the median element will always element in the center of the array; A much simpler problem that the more generic problem solved by the other package. This sharp reduction in complexity may very well have you reconsider (re)implementing the functionality to remove the dependency.

If the two packages involved are owned or maintained by different teams, you should, all other things equal, consider this strategy more because there is always a non-zero cost to depending on something over which you don't have full control. It may feel like that cost is zero at first, but once the other team discovers an issue with their implementation and would like to change their

implementation or deprecate the functionality you depend on, you realize that there is a real cost after all. If you implement conditional builds for packages (i.e., running only the tests for packages that have changed or depend on changed packages), the cost (in the form of CI server costs) is more tangible every day because your package will have to be tested when the package you depend on changes.

Whether simply copy-pasting code or implementing a local, more tailored solution, be sure to copy and adapt the tests for this code as well.

Reimplementing functionality and removing the package dependency fixes dependency and privacy violations.

5.7 Abstract away the dependency

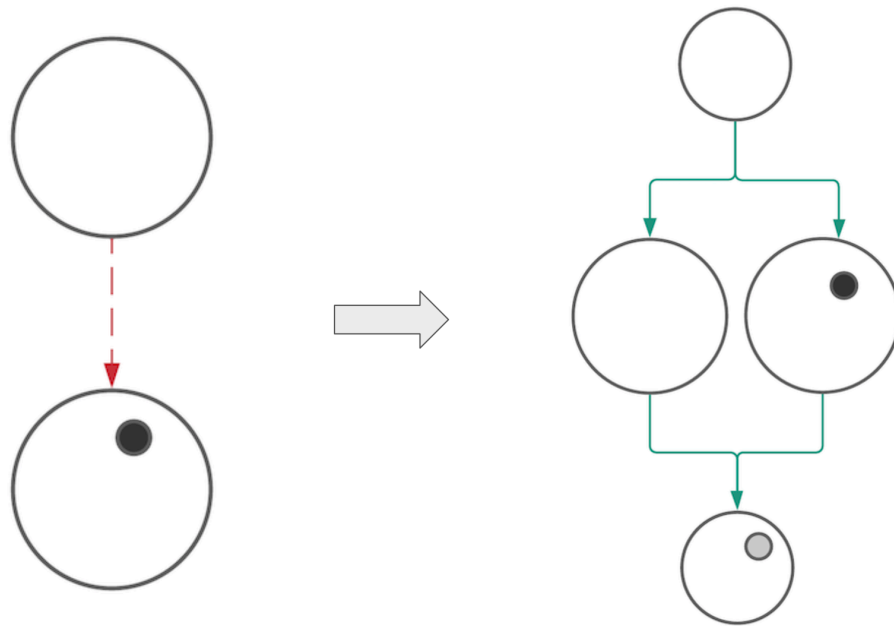
While having discussed five violation removal strategies, we haven't looked at any code yet. That's because things have been largely mechanical thus far: Observe the violation, write minimal package config or move or copy a whole bunch of code - problem solved. Having these refactorings available to us will be important throughout the journey to a better packaged application. One similarity of all of these refactorings is that they primarily focus on the package and its boundaries and not the actual implementation of the code.

So, if up to now we have focused on changing the *shape of the packages* to remove violations, we are now going to turn to refactorings of the *shape of the code* to achieve the same.

Because we are changing how parts of the system communicate and interact, these changes have a higher cost to them. For this chapter, this means that we'll have to discuss code changes. For your work, this means that you need to carefully weigh whether the given refactoring, its upside and its downsides, are appropriate for your situation.

There are two techniques in this category. The first one, *dependency injection*, replaces a direct and explicit dependency with a explicit parameterization of that dependency (and a new set of smaller explicit dependencies). The second one, *event-based interaction*, replaces the direct call to the functionality of the dependency with the sending and listening for events to trigger requests and responses in the application.

5.8 Dependency injection



The package-effect of dependency injection

Instead of knowing about and explicitly referencing a dependency, with this technique we hand (inject) the dependency into our code as a parameter (https://en.wikipedia.org/wiki/Dependency_injection). Dependency injection is very similar to the strategy pattern (https://en.wikipedia.org/wiki/Strategy_pattern) - the main difference really being that the strategy pattern intends to create the ability to swap out the dependency at different times, which isn't the primary purpose for us.

For the code sample of this section, we will be refactoring the usage of `Predictor` in the `PredictionUi`. The `PredictionUi` uses `Predictor` as a way to make the actual prediction after a user has selected two teams to match up and then estimates the likely outcome of their matchup. It is a simple one line, one time reference that makes this work.

Existing `PredictionsController` in `packages/prediction_ui/app/controllers/predictions_controller.rb`

```

1 class PredictionsController < ApplicationController
2   def new
3     @teams = Team.all
4   end
5
6   def create
7     predictor = Predictor.new
8     predictor.learn(Team.all, Game.all)
9     @prediction = predictor.predict(
10      Team.find(params["first_team"]["id"]),

```

```

11     Team.find(params["second_team"] ["id"]))
12   end
13 end

```

5.8.1 Naive dependency injection



The code discussed in this section is available at <https://github.com/shageman/package-based-rails-applications-book/tree/main/c5s07-1/sportsball>

In order to allow for any kind of dependency injection, we need a place to inject the dependency. Since the reference to `Predictor` is within a controller method, which has a signature that is dictated to us by the Rails framework, we can't inject it there. Instead, we will want to allow `Predictor` to be set on something we can refer back to from the controller when a request comes in. One way to do this would be to add a `@@predictor` class variable on `PredictionsController` and set that property from an initializer. If we want to emphasize the idea that we are replacing the package dependency from `PredictionUi` to `Predictor`, we should consider creating a new configuration option that communicates this package-level thinking. I tend to think that the latter approach is generally preferable, as it better communicates what kind of configuration is being created and hides more of the package internals (i.e., it will still work even without outside changes even if we rename `PredictionsController` or use `predictions` in other places).

The following implementation of a new `PredictionUi` module uses this idea to create a getter and setter for a new `@predictor` setting. Note that the instance variable is set and retrieved on methods defined on `self`, thus we are in fact dealing with class variables. This implementation raises considerations of thread-safety, which is why within the `configure` method we freeze the module once `predictor` is set (<https://blog.arkency.com/3-ways-to-make-your-ruby-object-thread-safe/>).

TODO: verify that freeze actually creates thread safety here

Configuration interface for `PredictionUi` in `packages/prediction_ui/app/services/prediction_ui.rb`

```

1  module PredictionUi
2    def self.configure(predictor)
3      @predictor = predictor
4      freeze
5    end
6
7    def self.predictor
8      @predictor
9    end
10 end

```

Next, we need to actually inject the dependency. Since, in this case, our dependency is static in that the predictor doesn't change based on some other input from the system, we can inject it once at the start of the application. To do this we add a new initializer.

New initializer in `config/initializers/configure_prediction_ui.rb`

```
1 Rails.application.config.to_prepare do
2   PredictionUi.configure(Predictor.new)
3 end
```

The actual configuration is wrapped in a `Rails.application.config.to_prepare` block, making the code inside the block play nicely with Rails autoloading. You can find more information on this at https://guides.rubyonrails.org/autoloading_and_reloading_constants.html#autoloading-when-the-application-boots. With Rails 7, this will change from “playing nicely” to not playing at all as Rails 7 won't allow autoloaded constants without appropriate wrapping in prepared blocks: <https://weblog.rubyonrails.org/2021/9/3/autoloading-in-rails-7-get-ready/>.

All the pieces are now in place for us to remove the dependency from `PredictionUi` onto `Predictor`. We can update the `PredictionsController` by using the newly configured `PredictionUi` module.

New `PredictionsController` in `packages/prediction_ui/app/controllers/predictions_controller.rb`

```
1 class PredictionsController < ApplicationController
2   def new
3     @teams = Team.all
4   end
5
6   def create
7     predictor = PredictionUi.predictor
8     predictor.learn(Team.all, Game.all)
9     @prediction = predictor.predict(
10       Team.find(params["first_team"]["id"]),
11       Team.find(params["second_team"]["id"]))
12   end
13 end
```

Remove dependency onto packages/predictor from packages/prediction_ui/package.yml

```

1 enforce_dependencies: true
2 enforce_privacy: false
3 dependencies:
4   - packages/rails_shims
5   - packages/games
6   - packages/teams
7   - packages/predictor

```

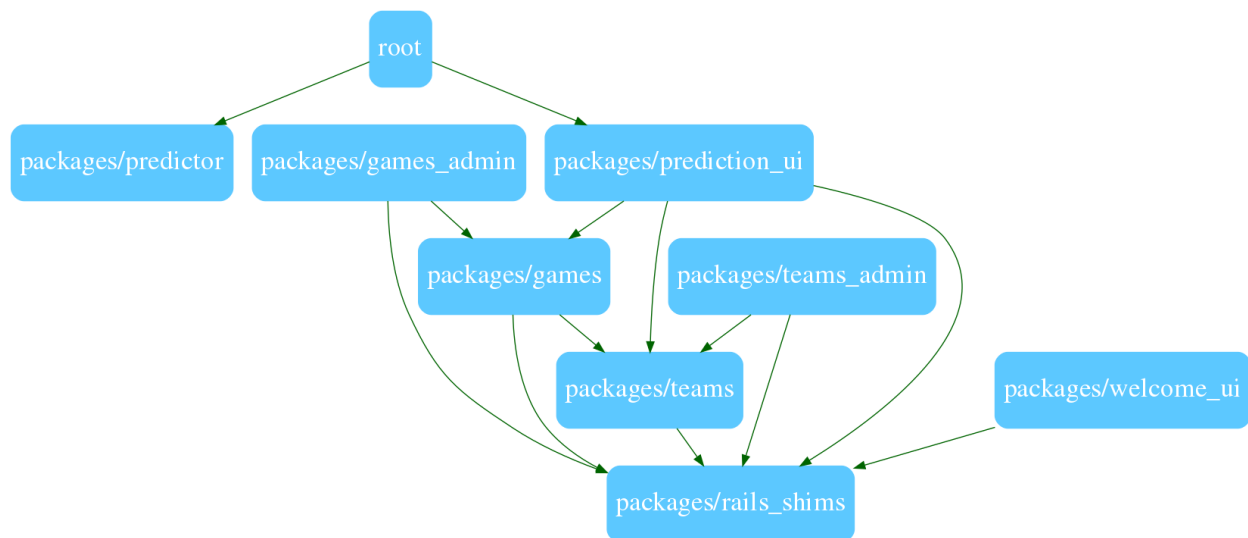
We can confirm with `packwerk check` that packwerk is still happy:

```

1 $ packwerk check
2 Packwerk is inspecting 66 files
3 .....
4 Finished in 0.84 seconds
5
6 No offenses detected
7 No stale violations detected

```

The diagram of the package structure confirms that we have successfully removed the dependency.



The package-effect of dependency injection

There are a couple things to note at this point.

This portion of the app was being covered by tests in `PredictionUi` and `Predictor`. Now, there is new code in an initializer. This code should be tested! Is it? It turns out it is, implicitly, through the system spec in `packages/prediction_ui/spec/system/predictions_spec.rb`. That test navigates to the prediction page and verifies that a (correct!) prediction is made, which wouldn't work if

the initializer were removed or broken. There may be edge-cases not covered by such a high-level integration test, but by testing at least the happy path, we ensure that the collaboration between packages at least works in some way.

The last code snippet showed that we replaced `Predictor.new` with `PredictionUi.predictor.new`. How valuable is that change really? We know that that code still resolves to `Predictor.new`. If we throw a debugger statement at the beginning of the `create` method and run the above-mentioned test, we get this output:

```

1  $ bundle exec rspec packages/prediction_ui/spec/system/predictions_spec.rb
2
3  [3, 12] in packages/prediction_ui/app/controllers/predictions_controller.rb
4      3:      @teams = Team.all
5      4:  end
6      5:
7      6:  def create
8      7:      debugger
9  => 8:      predictor = PredictionUi.predictor.new(Team.all)
10     9:      predictor.learn(Game.all)
11    10:      @prediction = predictor.predict(
12    11:          Team.find(params["first_team"]["id"]),
13    12:          Team.find(params["second_team"]["id"]))
14  (byebug) PredictionUi.predictor
15  Predictor

```

This observation is a practical example of what we discussed in [Chapter 3.3: Dynamic parameters](#) can hide dependencies on complex objects and the constants through which they are defined. This, in turn, hides the dependencies on the packages that these constants are defined in.

To make this problem explicitly visible we have to add some typing to the application.

5.8.2 Dependency injection with typing



The code discussed in this section is available at <https://github.com/shageman/package-based-rails-applications-book/tree/main/c5s07-2/sportsball>

We'll be using Sorbet (<https://sorbet.org/>) for our analysis of the impact of typing on our package dependencies. In order for us to have access to its capabilities, we need to first install and initialize Sorbet. Check out <https://sorbet.org/docs/adopting> for more detailed instruction how to get up and running with Sorbet and how to use it.

Adding Sorbet to Gemfile

```
1 gem 'sorbet-runtime'
2 gem 'sorbet', :group => :development
```

After running bundle we can initialize Sorbet with the following command:

```
1 $ SRB_YES=1 bundle exec srb init
2
3 :wave: Hey there!
4
5 This script will get this project ready to use with sorbet by creating a
6 sorbet/ folder for your project. It will contain:
7
8 - a config file
9 - a bunch of 'RBI files'
10
11 ... lots more output omitted here
```

At this point, type checking the app should result in no errors. Run `srb tc` to confirm:

```
1 $ srb tc
2 No errors! Great job.
```

We are now ready to add type signatures to the configuration methods of `PredictionUi` as follows.

Typed `PredictionUi` module in `packages/prediction_ui/app/services/prediction_ui.rb`

```
1 # typed: strict
2
3 module PredictionUi
4   extend T::Sig
5
6   sig {params(predictor: T.class_of(Predictor)).void}
7   def self.configure(predictor)
8     @predictor = T.let(predictor, T.nillable(T.class_of(Predictor)))
9     freeze
10  end
11
12  sig {returns(T.nillable(T.class_of(Predictor)))}
13  def self.predictor
14    @predictor
15  end
16 end
```

As a first iteration of this, we have added `Predictor` as the single parameter and return value of the setter and getter for the configuration. That is what the calls to the `sig` methods do in lines 6 and 12. Within the body of the `configure` method (line 8), we inform Sorbet that the type of `@predictor` is a nilable `Predictor`. The extension of `T::Sig` ensure that all of the above gets picked up by Sorbet.

Since `Predictor` already exists, this shouldn't create any problems from a typing perspective. As such, another run of `srb tc` can confirm that we indeed still have a valid type setup for Sorbet:

```
1 $ srb tc
2 No errors! Great job.
```

We can also confirm that Sorbet is indeed protecting the type of the parameter with a manual check on the rails console:

```
1 $ rails c
2 Running via Spring preloader in process 95731
3 Loading development environment (Rails 6.1.4)
4
5 2.7.3 :001 > PredictionUi.configure("some string")
6 Traceback (most recent call last):
7   1: from (irb):1
8  TypeError (Parameter 'predictor': Expected type T.class_of(Predictor), got "some str\
9  ing")
10 Caller: (irb):1
11 Definition: packages/prediction_ui/app/services/prediction_ui.rb:7````
```

Ok, so the Sorbet side of things is working and pretty uneventful so far. It turns out that packwerk doesn't think so:

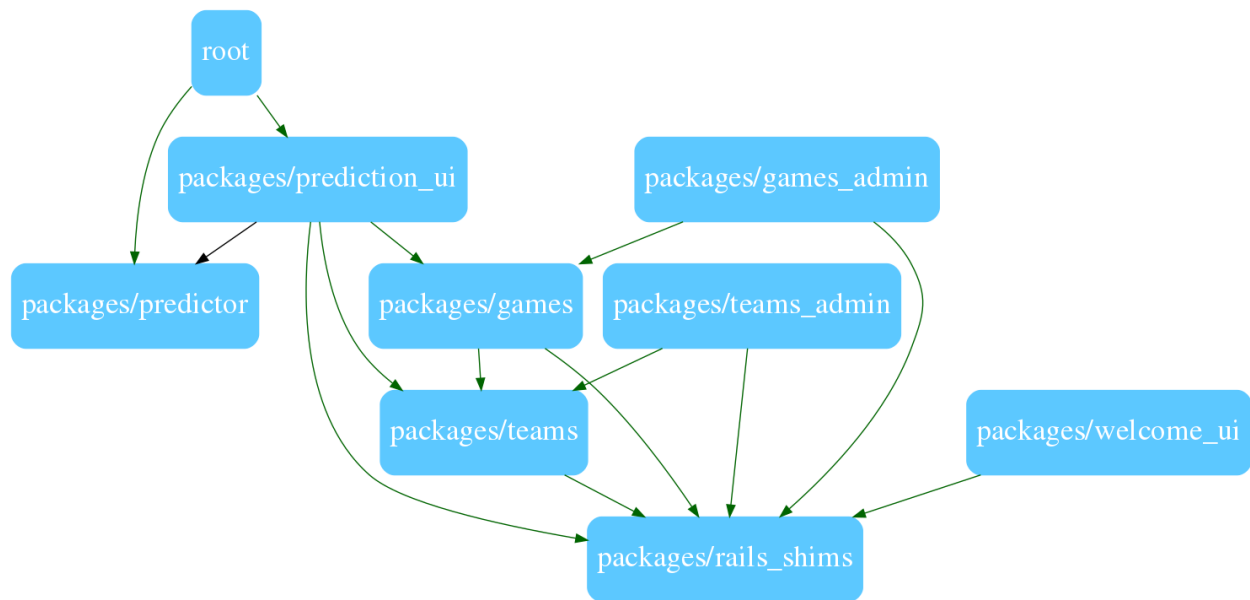
```
1 $ packwerk check
2 Packwerk is inspecting 66 files
3 .....E.....
4 Finished in 0.77 seconds
5
6 packages/prediction_ui/app/services/prediction_ui.rb:6:36
7 Dependency violation: ::Predictor belongs to 'packages/predictor', but 'packages/pre\
8 diction_ui' does not specify a dependency on 'packages/predictor'.
9 Are we missing an abstraction?
10 Is the code making the reference, and the referenced constant, in the right packages?
11
12 Inference details: this is a reference to ::Predictor which seems to be defined in p\
13 ackages/predictor/app/models/predictor.rb.
14 To receive help interpreting or resolving this error message, see: https://github.co\
```

```

15 m/Shopify/packwerk/blob/main/TROUBLESHOOT.md#Troubleshooting-violations
16
17
18 packages/prediction_ui/app/services/prediction_ui.rb:8:55
19 Dependency violation: ::Predictor belongs to 'packages/predictor', but 'packages/pre\
20 diction_ui' does not specify a dependency on 'packages/predictor'.
21 Are we missing an abstraction?
22 Is the code making the reference, and the referenced constant, in the right packages?
23
24 Inference details: this is a reference to ::Predictor which seems to be defined in p\
25 ackages/predictor/app/models/predictor.rb.
26 To receive help interpreting or resolving this error message, see: https://github.co\
27 m/Shopify/packwerk/blob/main/TROUBLESHOOT.md#Troubleshooting-violations
28
29
30 packages/prediction_ui/app/services/prediction_ui.rb:12:36
31 Dependency violation: ::Predictor belongs to 'packages/predictor', but 'packages/pre\
32 diction_ui' does not specify a dependency on 'packages/predictor'.
33 Are we missing an abstraction?
34 Is the code making the reference, and the referenced constant, in the right packages?
35
36 Inference details: this is a reference to ::Predictor which seems to be defined in p\
37 ackages/predictor/app/models/predictor.rb.
38 To receive help interpreting or resolving this error message, see: https://github.co\
39 m/Shopify/packwerk/blob/main/TROUBLESHOOT.md#Troubleshooting-violations
40
41
42 3 offenses detected
43
44 No stale violations detected

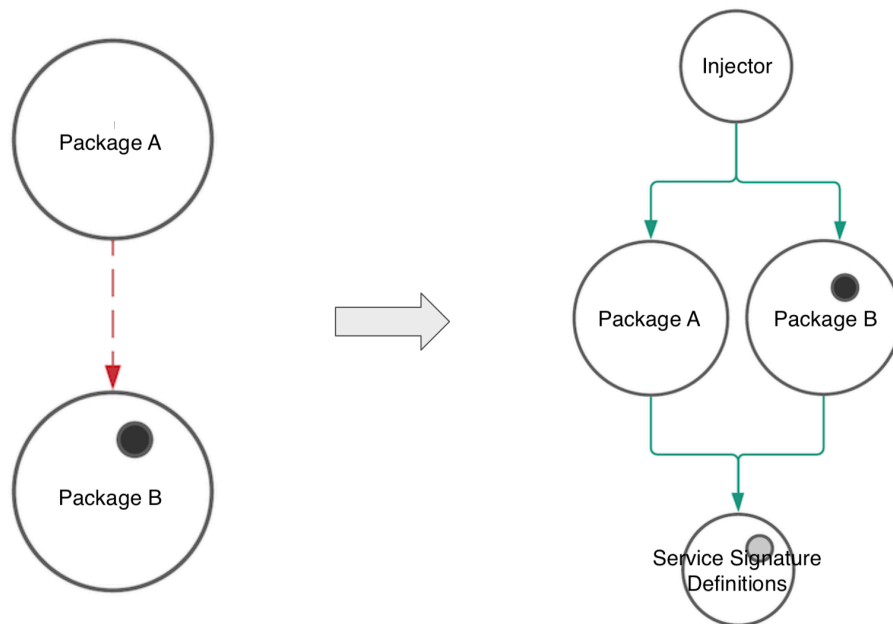
```

Exactly as we had hoped, packwerk once again can detect the previously hidden dependency. This works because Sorbet's type annotations are (also) simply valid Ruby code. This means that a line like `sig {params(predictor: T.class_of(Predictor)).void}` is actually referencing the constant, `Predictor`, which is what packwerk can pick up on to infer the existence of the dependency. In the image below we can see the single black line connecting `packages/prediction_ui` to `packages/predictor`.



Typing exposes the hidden dependency

At this point it makes sense to look back at the diagram from the beginning of this section describing the impact of service injection on the application structure. And when looking at it, it becomes clear that we haven't actually achieved the full structure yet. First, we separated `PredictionUi` (package A in the annotated version of the diagram below) from `Predictor` (package B) by injecting from an initializer (the injector). We need to create the fourth package, the *service signature definitions* package, go the next step towards separation.



The package-effect of service injection revisited

5.8.3 Dependency injection with typing and type abstraction



The code discussed in this section is available at <https://github.com/shageman/package-based-rails-applications-book/tree/main/c5s07-3/sportsball>

There is a principle in object-oriented design called the *dependency inversion principle*. It states that a high-level *module* should not depend on a low-level module, instead they should both depend on abstractions (https://en.wikipedia.org/wiki/Dependency_inversion_principle). I want to think about packages as the containers for these modules. An immediate problem arises in that we can't map modules to packages, one-to-one. If we were to do that, the principle would state that packages can't depend on packages, which would make everything we have been building up obsolete. Instead, we have to separate packages into two categories: those that contain *modules* and those that contain *abstractions of modules*. To make the terms “module” and “abstraction” more concrete, think of them this way:

- *modules* contain the domain, functionality, the code that does stuff - they are concrete
- *abstractions* are interfaces to the public facing parts of modules

Dependency inversion is related to what we have been discussing so far in that it is dependent on some way of getting the *concrete* to the *abstract*. And dependency injection is one in which we can do that. As we have seen, if we only use dependency injection without the idea of dependency inversion, we can (on a superficial level) remove the dependency between two packages. To break through the superficiality, however, we have to add dependency inversion to make explicit the connection that the two packages still have via the abstractions they both depend on. It is important to recognize that this doesn't *reduce* the level of separation between the packages in comparison to our code from the [naive dependency injection](#) examples, but rather it merely points out that there was a certain dependency all along. We used the step of [dependency injection with typing](#) to point out the missing explicitness of the dependency and also the need for dependency inversion.

To dive right in, let's imagine what `PredictionUi` would look like if we were to depend on an abstraction, the interface to, `Predictor` instead of the class itself.

PredictionUi module depending on a PredictorInterface

```

1  # typed: strict
2
3  module PredictionUi
4    extend T::Sig
5
6    sig {params(predictor: PredictorInterface).void}
7    def self.configure(predictor)
8      @predictor = T.let(predictor, T.nitable(PredictorInterface))
9      # freeze

```

```

10   end
11
12   sig {returns(T.nilable(PredictorInterface))}
13   def self.predictor
14     @predictor
15   end
16 end

```

Wherever we had `Predictor` before, we now have `PredictorInterface`. This makes our challenge that of creating this interface to `Predictor`. Let's take a look at the public methods of `Predictor`:

Predictor's public methods

```

1  # typed: true
2  class Predictor
3    def initialize(); end
4
5    def learn(teams, games); end
6
7    def predict(first_team, second_team); end
8  end

```

The `learn` method takes in two enumerables: all *teams* to be considered and the *games* of those teams to be used for the calculation of relative team strengths. `predict` takes two teams to be compared in the prediction. If we, naively, turn these facts into `PredictorInterface` we can use `Team`, `Game`, and Sorbet's `T::Enumerable` to create a first version:

Naive PredictorInterface with dependencies on Team and Game

```

1  # typed: strict
2
3  module PredictorInterface
4    extend T::Sig
5    extend T::Helpers
6    interface!
7
8    sig { abstract.params(teams: T::Enumerable[Team], games: T::Enumerable[Game]).void\
9    }
10   def learn(teams, games); end
11
12   sig { abstract.params(first_team: Team, second_team: Team).returns(Prediction) }
13   def predict(first_team, second_team); end
14 end

```

This will work; And it has two consequences we likely do not want: The package that includes `PredictorInterface` would have to depend on `packages/teams` and `packages/games`. That also means that the abstraction (the interface) depends on the concrete module (the models), which goes against the dependency inversion principle.

From these undesired consequences, we can deduce that the creation of a `PredictorInterface` forces the creation of interfaces for `Team` and `Game` also. Creating interfaces for these classes comes with the added complexity that these classes are subclasses of `ActiveRecord` and, from that, get a lot of functionality not explicitly specified in the class itself. Sorbet has ways to handle this and generate parts of the interface for a class. You can also use tools like `sorbet-rails` (<https://github.com/chanzuckerberg/sorbet-rails>) to generate interface files for models and a wide variety of other functionality autogenerated by Rails.

I see these kinds of tools as being helpful at the beginning of the conversion of a codebase to gradual typing. At that stage, the metaprogramming that comes with so much of the functionality of Rails makes it cumbersome to turn on high levels of protection for application code because calls to Rails functionality seem undefined and thus incorrect to Sorbet. However, when it comes to the creation of interfaces for “our code” we should be asking “what interface do we *need* (in a given context)” not “what interface do we *have*.” What we need in a given context is often a much smaller set than what we have, especially in the context of `ActiveRecord` objects. More concretely, a team model has 412 methods (what we have - just run `Team.new.methods.count` in a Rails console). And what do we need in the context of `Predictor`? Simply scan through the code of `Predictor` to find out:

What does `Predictor` need from a `Team`?

```

1 require "saulabs/trueskill"
2
3 class Predictor
4   def learn(teams, games)
5     @teams_lookup = teams.inject({}) do |memo, team|
6       memo[team.id] = {
7 markua-start-end
8         team: team,
9         rating: [Saulabs::TrueSkill::Rating.new(1500.0, 1000.0, 1.0)]
10      }
11      memo
12    end
13
14    games.each do |game|
15      first_team_rating = @teams_lookup[game.first_team_id][:rating]
16      second_team_rating = @teams_lookup[game.second_team_id][:rating]
17      game_result = game.winning_team == 1 ?
18        [first_team_rating, second_team_rating] :
19        [second_team_rating, first_team_rating]
20      Saulabs::TrueSkill::FactorGraph.new(game_result, [1, 2]).update_skills

```

```

21     end
22 end
23
24 def predict(first_team, second_team)
25   team1 = @teams_lookup[first_team.id][:team]
26   team2 = @teams_lookup[second_team.id][:team]
27   winner = higher_mean_team(first_team, second_team) ? team1 : team2
28   Prediction.new(team1, team2, winner)
29 end
30
31 private
32
33 def higher_mean_team(first_team, second_team)
34   @teams_lookup[first_team.id][:rating].first.mean >
35     @teams_lookup[second_team.id][:rating].first.mean
36 end
37 end

```

From the code, we can see that all we need is the method `id`. At least in the context of `Predictor` we can get away with an interface defining one method, not 412.

A interface to `Team` focussed on the use within `Predictor`

```

1  # typed: strict
2
3  module TeamInterface
4    extend T::Sig
5    extend T::Helpers
6    interface!
7
8    sig { abstract.returns(Integer) }
9    def id; end
10 end

```

With Sorbet, we define interfaces by adding the `interface!` statement to the class body which becomes available when the class extends `T::Helpers`. The interface to the `id` method is simple enough: It is a method that returns an integer value.

The analogous analysis for `Game` makes it clear that we need three methods for the interface for `Game`: `first_team_id`, `second_team_id`, and `winning_team` - all of which return integer values.

A interface to `Game` focused on the use within `Predictor`

```

1  # typed: strict
2
3  module GameInterface
4    extend T::Sig
5    extend T::Helpers
6    interface!
7
8    sig { abstract.returns(Integer) }
9    def first_team_id; end
10
11   sig { abstract.returns(Integer) }
12   def second_team_id; end
13
14   sig { abstract.returns(Integer) }
15   def winning_team; end
16 end

```

Before we move on, we need to take a closer look at these interfaces' names: `TeamInterface` and `GameInterface`. To a Rubyist, these names likely seem obvious and innocuous. For naming interfaces (let's take `Team` as an example), there are other conventions out there, like `ITeam` or `TeamI` and even to go back and rename `Team` to `TeamImpl` and call the interface `Team` (this is a place ripe for bike shedding, see for example <https://stackoverflow.com/questions/541912/interface-naming-in-java>). There are good rationales for all of these conventions, but they are not the main point we should be thinking about.

What we need to address in our naming is that we made interfaces not as extractions of “what we have” but “what we need” and that means that both of these interfaces don't cover all of what `Team` and `Game` can do. They cover only what `Predictor` needs from `Game` and `Team` in the *context* of a prediction.



In the discussion on the linked stackoverflow (<https://stackoverflow.com/questions/541912/interface-naming-in-java>) there is a quote “whats [sic] the point of an interface if you have only one implementation.”

This entire section is a case study of why we sometimes want interfaces that have one single implementation: because by separating them the interface and the implementation can live in different places (in different packages) and thus allow us to control the shape of our dependency graph.

In *Component-based Rails Applications* Chapter 6.2, this exact question came up in the discussion of the DCI (Data-Context-Interaction) pattern (See <https://www.infoq.com/news/2009/05/dci-coplien->

reenskau/ or https://en.wikipedia.org/wiki/Data,_context_and_interaction). With this pattern, objects that carry data show up in interactions via the roles they play in those interactions. Noteworthy is that the role and the data are separate from each other and can (and should!) be named separately from each other. We are not suddenly also converting Sportsball into a DCI implementation here, but the intent is very similar.

Let's take that word from DCI - *role* - and ask "what role do Team and Game play in a prediction?" In CBRA I answered the question with: teams act as *contenders* and games act as *historical performance indicators*. While these names may feel a bit clunky and uncomfortable at first, I still believe that they are better suited than the generic names that might come to mind initially. So, going forward, I am going to use Contender as the name for the interface to Team we discussed above and HistoricalPerformanceIndicator for the analogous Game interface.

For Predictor and PredictorInterface, this discussion should also happen. However, with these names, the interface *is* in fact the interface to the entirety of the class. In our app, Predictor doesn't show up in different contexts. It shows up as the thing that can make predictions when we want to make predictions.



It might be fun to think about what *would* make us create differently named interfaces for Predictor. Imagining we build a startup on top of Sportsball (and this is why I shouldn't be put in charge of startups...). And we want to facilitate all sorts of activities around sports, the teams that play them and their games. Maybe we want to model financial implications for the teams from a game. Maybe we want to model relative strengths (the stuff currently hidden inside predictor's algorithm!). Maybe we want to model how team morale might change after a game. What role does something like Predictor play in comparison to, and shared with, these examples? Maybe they are all implementations of the idea (dare I say interface) of GameAnalysis?

With these two interfaces in place, we can define a PredictorInterface which only depends on other interfaces and no longer references any concrete classes.

PredictorInterface adhering to the principles of dependency inversion (almost)

```

1  # typed: strict
2
3  module PredictorInterface
4    extend T::Sig
5    extend T::Helpers
6    interface!
7
8    sig { abstract.params(teams: T::Enumerable[Contender], games: T::Enumerable[Histor\
9  icalPerformanceIndicator]).void }
10   def learn(teams, games); end
11
12   sig { abstract.params(first_team: Contender, second_team: Contender).returns(Predi\

```

```

13 ction) }
14   def predict(first_team, second_team); end
15 end

```

If you read through the `PredictorInterface` code closely, you probably noticed that the above statement is incorrect. There is one concrete class being used here: `Prediction`! What am I doing here?

I am proposing that we ignore the rules of the dependency inversion principle for `Prediction` because adhering to it would, at the moment, create quite a bit of extra code and no value. There is some irony in the fact that we are discussing large-scale modularization refactoring patterns in a Rails app with two models, but that is simply to make the point as tersely as possible. By not following the rules for `Prediction` here, we're making another point: All these patterns and ideas have their limits and they won't always make sense. And when they don't, we should be comfortable putting them aside.

If we wanted to properly *interfacify* `Prediction` we would have to go through the same process as above to create `PredictionInterface` and do whatever else is needed (I promise we will get back to what else is needed when this miserable tangent is over) to get this change working in the app - and for what? This is the `Prediction` code right now:

```

1 class Prediction
2   attr_reader :first_team, :second_team, :winner
3
4   def initialize(first_team, second_team, winner)
5     @first_team = first_team
6     @second_team = second_team
7     @winner = winner
8   end
9 end

```

There is literally nothing exciting happening in `Prediction`. The differentiation of the interface and the class is currently useless. So instead of doing that, let's just work with `Prediction` as if it were an interface.

rant over

We place all of our interfaces into a new package, `packages/predictor_interface`, which will have no dependencies on other packages:


```

1  .
2  └─ packages
3      └─ predictor_interface
4          └─ app
5              └─ public
6                  └─ contender.rb
7                  └─ historical_performance_indicator.rb
8                  └─ prediction.rb
9                  └─ predictor_interface.rb
10 └─ package.yml

```

This is a good place to check in on whether all this interfaces work is already done or what if anything we still need to do to make all these changes work. Let's run RSpec.

```

1  $ bundle exec rspec
2
3  An error occurred while loading spec_helper.
4  Failure/Error: require File.expand_path("../config/environment", __dir__)
5
6  TypeError:
7      Parameter 'predictor': Expected type PredictorInterface, got type Class with value\
8      Predictor
9  markua-start-end
10  Caller: config/initializers/configure_prediction_ui.rb:3
11  Definition: packages/prediction_ui/app/services/prediction_ui.rb:7
12
13  ...
14
15  No examples found.
16
17
18  Finished in 0.00005 seconds (files took 1.49 seconds to load)
19  0 examples, 0 failures, 1 error occurred outside of examples

```

Running the tests fails because Sorbet prevents the loading of the application due to “Expected type PredictorInterface, got type Class with value Predictor.” This is telling us that Sorbet can’t currently connect the interface to the concrete class. We need to make sure that Predictor is understood as implementing the PredictorInterface. I will show the code here, but suffice it to say that adding the inclusion of PredictorInterface and adding the appropriate method signatures is the smallest part of this change. Within the methods, all accesses to the various variables that are part of the prediction process need to be augmented to appease Sorbet’s typing strict demands.

A Predictor that uses PredictorInterface

```

1  # typed: strict
2
3  require "saulabs/trueskill"
4
5  class Predictor
6    include PredictorInterface
7    extend T::Sig
8
9    sig {override.params(teams: T::Enumerable[Contender], games: T::Enumerable[Histori\
10 calPerformanceIndicator]).void}
11    def learn(teams, games)
12      @teams_lookup = T.let({}, T.nitable(T::Hash[Integer, TeamLookup]))
13      @teams_lookup = teams.inject({}) do |memo, team|
14        memo[team.id] = TeamLookup.new(
15          team: team,
16          rating: Saulabs::TrueSkill::Rating.new(1500.0, 1000.0, 1.0)
17        )
18        memo
19      end
20
21      games.each do |game|
22        first_team_rating = @teams_lookup[game.first_team_id].rating
23        second_team_rating = @teams_lookup[game.second_team_id].rating
24        game_result = game.winning_team == 1 ?
25          [[first_team_rating], [second_team_rating]] :
26          [[second_team_rating], [first_team_rating]]
27        Saulabs::TrueSkill::FactorGraph.new(game_result, [1, 2]).update_skills
28      end
29    end
30
31    sig {override.params(first_team: Contender, second_team: Contender).returns(Predic\
32 tion)}
33    def predict(first_team, second_team)
34      team1 = T.must(T.must(@teams_lookup)[first_team.id]).team
35      team2 = T.must(T.must(@teams_lookup)[second_team.id]).team
36      winner = higher_mean_team(first_team, second_team) ? team1 : team2
37      Prediction.new(team1, team2, winner)
38    end
39
40    private
41
42    sig {params(first_team: Contender, second_team: Contender).returns(T::Boolean)}
```

```

43   def higher_mean_team(first_team, second_team)
44     T.must(T.must(@teams_lookup)[first_team.id]).rating.mean >
45       T.must(T.must(@teams_lookup)[second_team.id]).rating.mean
46   end
47
48   class TeamLookup < T::Struct
49     const :team, Contender
50     const :rating, Saulabs::TrueSkill::Rating
51   end
52   private_constant :TeamLookup
53 end

```

The same work needs to be done for `Team` and `Game`. For both, the work is a bit more straightforward as we need so much less and we're relying on `ActiveRecord` for the properties defined in the interface.

Team implementing Contender

```

1  # typed: strict
2  class Team < ApplicationRecord
3    include Contender
4    extend T::Sig
5
6    validates :name, presence: true
7  end

```

Game implementing HistoricalPerformanceIndicator

```

1  class Game < ApplicationRecord
2    include HistoricalPerformanceIndicator
3    extend T::Sig
4
5    validates :date, :location, :first_team, :second_team, :winning_team,
6              :first_team_score, :second_team_score, presence: true
7    belongs_to :first_team, class_name: "Team"
8    belongs_to :second_team, class_name: "Team"
9  end

```

For `Team`, the above changes are enough to make things work with Sorbet. For `Game`, however, we actually need to do one more step which is to manually add the methods that `Game` gets from `ActiveRecord` that Sorbet, without our help, can't know about. To do this, we create a `game.rbi` file right next to `game.rb` with the following content. The name `game.rbi` indicates that here we are creating an interface of the “what we have” kind just so Sorbet can understand what is going on. When we add more types to the codebase and discover that Sorbet can't find other methods that exist on `Game`, we will add them to this `rbi` file.

```
packages/games/app/models/game.rbi
```

```
1  # typed: strict
2
3  class Game
4    sig { returns(T.nilable(Integer)) }
5    def first_team_id; end
6
7    sig { returns(T.nilable(Integer)) }
8    def second_team_id; end
9
10   sig { returns(T.nilable(Integer)) }
11   def winning_team; end
12 end
```

There is no extra work needed for `Team` because, in the context of `Predictor`, all we need is an `id` method. When, at the very beginning of our journey into using Sorbet, we called `bundle exec srb init` a whole host of `.rbi` files were created in `sorbet/rbi/gems` - one for all the gems that is in our application. The ones about Rails gems are typically thousands of lines long. Specifically, the `active_record.rbi` I am looking at right now is over 5500 lines long. It is in one of those lines that Sorbet registers that an `ActiveRecord` object always has an `id` method.

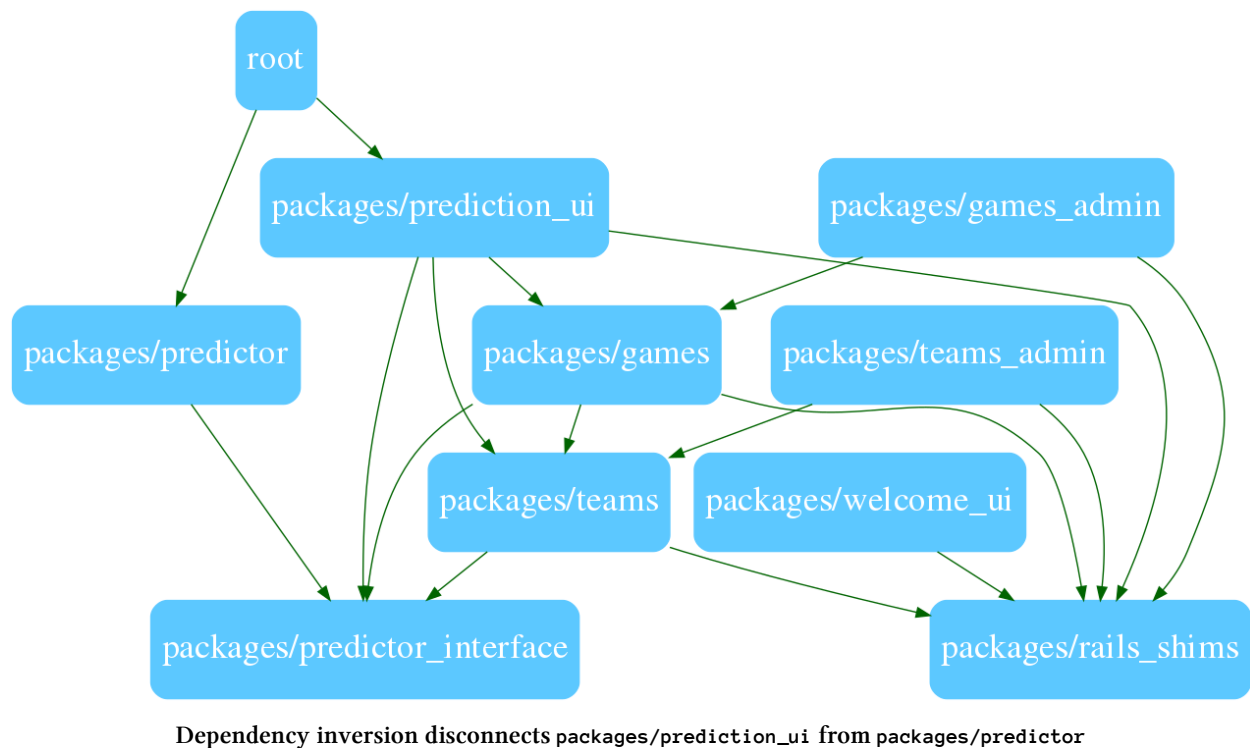
Since these files are generic for the gems and we didn't create `rbi` files for our models, we have to create it manually to give Sorbet the same information for `Game`.



There are entire libraries and frameworks built to support dependency injection and make it easier. I did not use any here because when only done once, the boilerplate needed isn't overwhelming yet. Moreover, this chapter is all about learning the patterns of package boundary management: Hiding some of the complexity behind the slick functionality of a library is counter productive.

That said, if you find yourself using this pattern often, check out what tools like https://dry-rb.org/gems/dry-auto_inject offer.

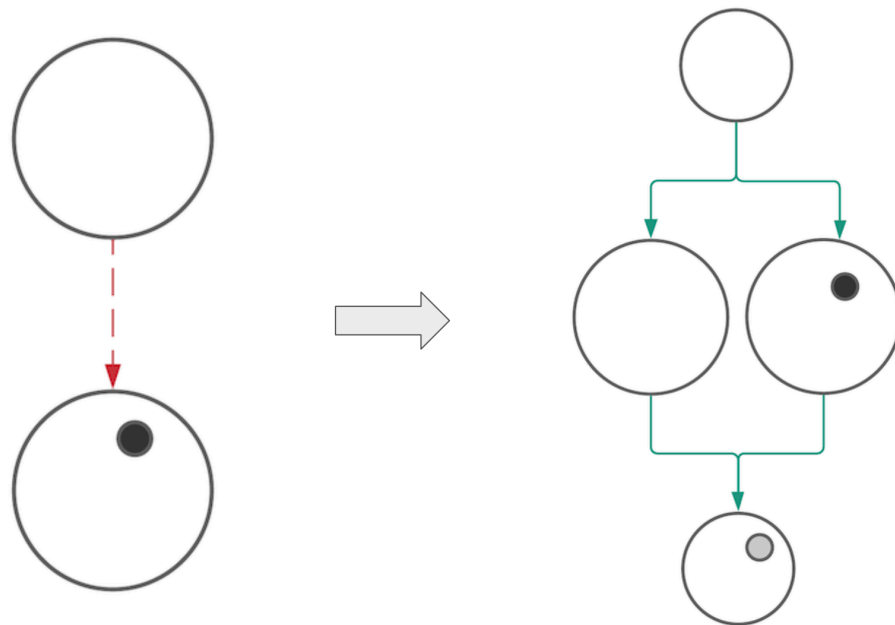
The package dependency diagram for this version of the code shows the successful disconnection of `packages/prediction_ui` from `packages/predictor`. These two packages plus `packages/games` and `packages/teams` now all depend on `packages/predictor_interface`.



5.9 Dependency Location - The service locator pattern



The code discussed in this section is available at <https://github.com/shageman/package-based-rails-applications-book/tree/main/c5s08/sportsball>



The package-effect of service locator

A variant of the service injection pattern is the service *locator* pattern. With this pattern, instead of a higher-level package in the system connecting a service to its consumer by injecting it, the services register themselves with a locator. Afterwards, consumers can ask the locator for a service and the locator will return a reference to that service for use by the consumer.

One may disagree with me calling service location and injection *variants* of each other. There are a number of differences worth pointing out. With dependency injection we get the explicit connection between the service and the consumer: it is visible in the parameters that there is a connection (to the implementation or its interface). This explicit connection vanishes with the service locator and it is no longer visible in the API of the consumer. Finally, typing service locators is really difficult, because they provide references to services with arbitrary interfaces. It takes certain specific capabilities of a type system that, while I can't prove that Sorbet doesn't have those, I can say that I failed to produce a service locator example with typing. The path I did not go down was to use [untyped](#) explicitly, which would allow us to type the rest of the file.

Let's look at the untyped service locator that we can create by refactoring our dependency injection example.

First, we rename the previous interface package to `service_locator` and add a couple of needed folders. We'll collocate the predictor interface with the service locator because, for our toy example, all consumers always need both parts to do their work. If there were more services or if the system wanted to reuse the interfaces in context other than the service locator pattern, we would want to put these things into different packages.

```

1 mv packages/predictor_interface packages/service_locator
2 mkdir -p packages/service_locator/app/services
3 mkdir -p packages/service_locator/spec

```

Next, we add the code for a service locator. No bells and whistles here... but it will do the job. And, remember, no typing, specifically because of the difficulty of typing the `service` parameter and the return value of `get_service`.

Note that the `ServiceLocator` is a singleton because we want every service to be registered at the same destination (and thus findable by any consumer). Note also that this is not guaranteed here because the internal implementation of the service registry is a not-thread safe Hash. We'd could use a mutex (<https://ruby-doc.org/core-2.1.0/Mutex.html>) or something like the thread safe hash from the hamster gem (<https://github.com/hamstergem/hamster>).

```

1  # typed: true
2
3  class ServiceLocator
4    include Singleton
5
6    def register_service(name, service)
7      @services ||= {}
8      @services[name] = service
9      puts "Registered service as #{name}"
10   end
11
12   def get_service(name)
13     @services ||= {}
14
15     raise ServiceNotFoundError, "Service #{name} was never registered" unless @services[name]
16
17     @services[name]
18   end
19 end
20

```

Next, we delete the old injection initializer and instead create a service registry initializer that registers an instance of `Predictor` under the name `:predictor`.

```

1 rm config/initializers/configure_prediction_ui.rb

1 ServiceLocator.instance.register_service(:predictor, Predictor.new)

```

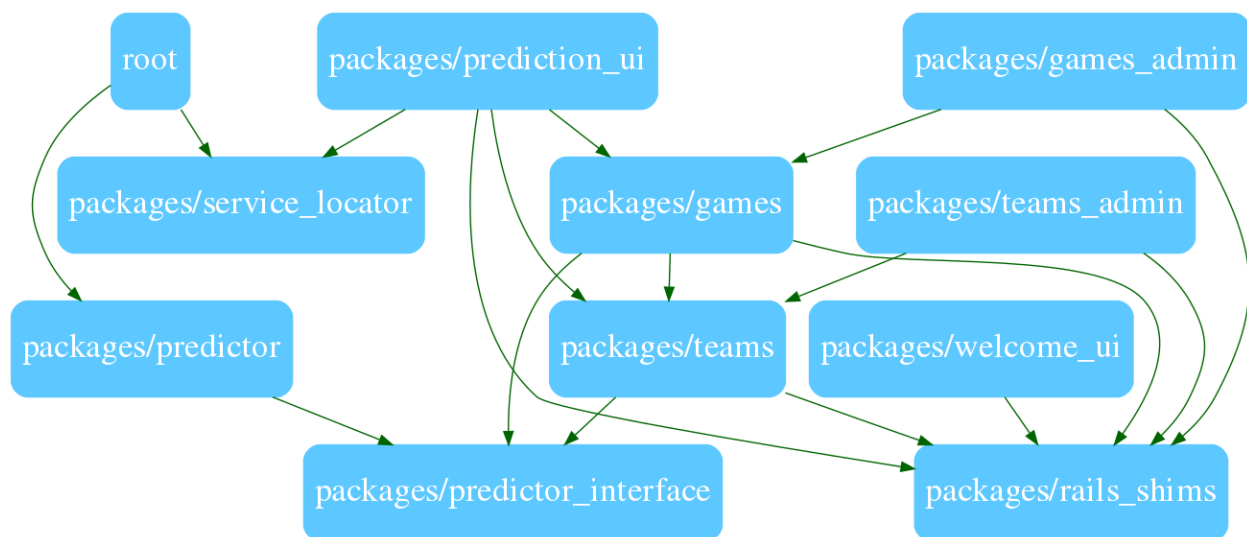
Finally, we can use the new locator in `PredictionsController` by using the `get_service` method of the singleton instance of `ServiceLocator`.

```

1  # typed: false
2  class PredictionsController < ApplicationController
3    def new
4      @teams = Team.all
5    end
6
7    def create
8      predictor = ServiceLocator.instance.get_service(:predictor)
9
10     predictor.learn(Team.all, Game.all)
11     @prediction = predictor.predict(
12       Team.find(params["first_team"]["id"]),
13       Team.find(params["second_team"]["id"]))
14   end
15 end

```

The combined changes from introducing the service locator are visible in the diagram below. I find it interesting to see that visually in comparison to dependency injection, which separated PredictionUi from Predictor by one hop (the Root package), the service locator pattern separates them by two hops: Root and ServiceLocator. This is an imperfect reminder that with dependency injection there is still one place where code explicitly connects the dependency whereas with a service locator there is no longer such a place.



Dependency structure with service locator

There is a lot more to the service locator pattern that won't be covered here. A nice read about the service locator pattern in the context of microservices by Auth0 is <https://auth0.com/blog/an-introduction-to-microservices-part-3-the-service-registry/>.

dry-rb provides the dry-auto_inject gem (https://github.com/dry-rb/dry-auto_inject), which, de-

spite its name (but notice the “auto” before the “inject”) offers an API that is similar to the service locator pattern we looked at here.

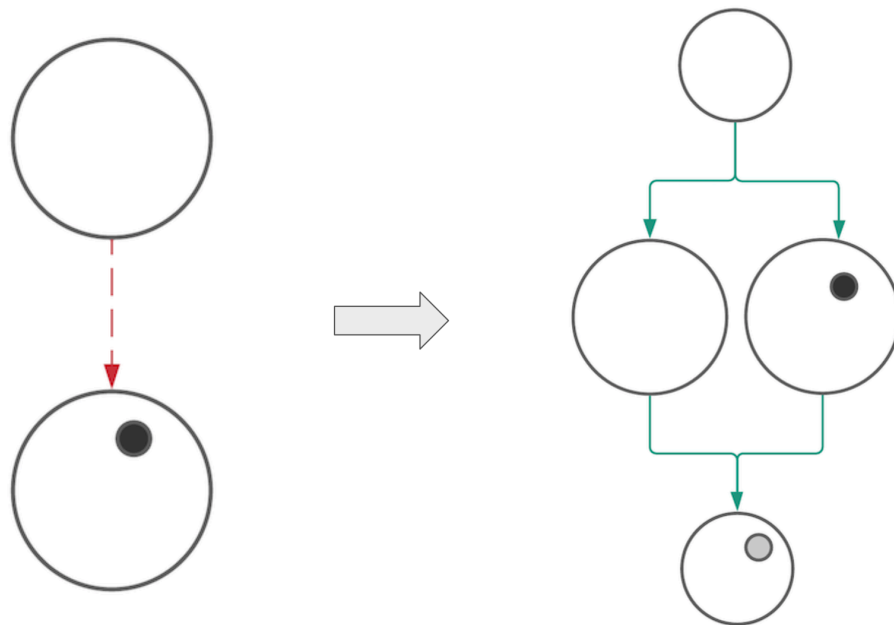
5.10 Emit and listen to events



The code discussed in this section is available at <https://github.com/shageman/package-based-rails-applications-book/tree/main/c5s09/sportsball>

The final dependency management strategy is typically the hardest to implement while at the same time the most impactful with regard to shaping the dependencies within an application. Both of these are due to the fact that switching to *events* changes the communication from being synchronous to being asynchronous. And with this change come a whole host of consequences, including performance characteristic changes, failure mode changes, and storage requirement changes.

When switching to events, there is no more calling a service from a consumer that needs its functionality. Instead, the two are decoupled and built in a way where they can perform their respective functionality independently (at least in a basic fashion). What do I mean by “in a basic fashion?” - well, not that we are re-implementing all needed functionality in all packages, but rather that they are not “broken” on their own. In order to actually do what we expect them to do, parts of the system emit and listen to events. Events are sent out by a part of the system to inform any other part of the system that something business relevant happened. Any interested other part of the system can register to receive and then process these events and perform any actions it needs to perform in response to the event having happened.



The package-effect of switching to events

The package structure effect for switching to events is the same as that for the dependency injection and the service locator patterns. There are ways to shape it differently, but the general concerns are: one package that connects event emitters and event consumers and one package that contains the schemas of all the events in the system.

5.10.1 To event or not to event - and where?

One disclaimer about this refactoring: Sportsball is such a small sample app that the use of eventing... Well, it is a real stretch. We have to imagine a much larger system for this to make sense.

Most of the app is concerned with CRUD (create, read, update, delete) management of teams and games, which doesn't offer opportunities for events. Predictions don't offer much surface area either. I can see two places where, in a larger system with real challenges, one might break up the interactions using eventing.

The first seam would be to separate prediction capabilities from team and game management. For this, imagine that there is a lot more to manage about teams and games. So much more that we have decided to split this functionality off into its own service. That service allows us to fetch information about teams and games as well as manage them in the system. This service might also emit events when new teams are added or games are played. And other services might listen for these events to update their state of the world. For example, the predictor service could listen for game creation events so as to add games played to its own database. That way its predictions take the most recent games into account.

Note that this setup would lead to two of the systemic changes mentioned above: First, the predictor service would have to store its own copy of the games played and whatever attributes it is interested in (a *storage requirement change*). The *failure mode change* is also easily imaginable: If events for yesterday's games haven't been sent or processed yet, the predictor will predict based on earlier games only. Should the event-updating be broken and no new events be processed, the quality of the predictions will degrade over time as new games are played and new information is going unaccounted for.

The second seam that could eventually make sense is the separation of the prediction interface from the prediction calculation. Maybe the calculation is so complex that it takes a while to complete or the interface is set up to send off a bunch of prediction requests at once. In this situation the frontend emits the *need for a prediction to be calculated* which is listened to for by the backend while the backend can emit *prediction completed messages* when it has completed a calculation.

If you are wondering why we wouldn't use a background job for this kind of interaction, you are asking a great question. Background jobs and events can be used for similar purposes in certain cases. Their different characteristics make one or the other preferable in most situations.

- Events allow the emitter to be oblivious of who makes use of the event. This is the dependency removal feature we have been after this whole chapter.
- There could, in fact, be multiple parts of the system making use of the same event. And the emitter still wouldn't care.

- If no observable work happens from the perspective of the event emitter (as in the first event example - splitting team/game management from prediction), why should code for prediction update live in the management part?
- If the source of the event can't function without the proper processing of the event by something else, it might feel more like "work in the background." One way to observe this is that the failure mode does indeed not change, but rather that without all the parts working, the system doesn't work.

Despite its obvious flaws from the list above, the following example of introducing events into Sportsball creates the `prediction_needed` and `prediction_completed` events to disconnect prediction calculation from presentation.

Feel free to build on top of this and add the eventing between team/game management and prediction based on this example later. You will need `team_added`, `team_renamed`, `team_deleted`, `game_added`, `game_result_updated`, and `game_deleted` events to keep the predictor up-to-date with all the changes from the management side. It is this greater number of events that made me go with the less perfect, but simpler example of eventing from this section.

Let's see how all of this looks in practice.

5.10.2 Calculating backend responses with events

Starting with the `PredictionsController` we'll follow a ripple of changes through our code that will bring us all the way down into `Predictor` and back up to the presentational layers of our app.

Maybe unsurprisingly, we once again have a technology choice to make: What are we going to use to send and process events? Before we discuss a couple of choices we have, we are going to go with the simplest solution I could find, one that doesn't require a new dependency let alone a separate service. That is because `ActiveSupport::Notifications` are built right into Rails and give us everything we need to start using events and to learn about their impact on our application's architecture. The two methods from `ActiveSupport::Notifications` we are going to be using are `instrument` and `subscribe`. Both methods take the name of an event. Additionally, we can hand `instrument` a payload which we can use for the event metadata and `subscribe` a block in which we can process the event.

So, off we go! First, instead of calling the `Predictor` directly, `PredictionsController` in its `create` method emits a `prediction_needed` event to indicate that a prediction is being requested.

```

1  # typed: false
2
3  require "ostruct"
4
5  class PredictionsController < ApplicationController
6    def new
7      @teams = Team.all
8    end
9
10   def create
11     ActiveSupport::Notifications.instrument("prediction_needed", {
12       team_1_id: params["first_team"]["id"],
13       team_1_name: Team.find(params["first_team"]["id"]).name,
14       team_2_id: params["second_team"]["id"],
15       team_2_name: Team.find(params["second_team"]["id"]).name
16     } ) do
17       end
18   end
19 end

```

Note first that the `create` method is no longer setting any instance variables. That is because it is no longer rendering a view directly - more on that below, when we wire up the frontend. For the instrumentation (our event), we are sending the playing teams' IDs and their names. The IDs are essential for us to make a prediction. Passing the team names might be surprising... It is a little optimization which we will see in the code that subscribes a processor to the `prediction_needed` event.

We'll house this `PredictionNeededSubscriber` in a new package specifically for it, `packages/prediction_needed_subscriber` as it doesn't fit with any of our other packages.

```

1  class PredictionNeededSubscriber
2    def self.configure
3      ActiveSupport::Notifications.subscribe("prediction_needed") do |name, start, fin\
4 ish, id, payload|
5        predictor = Predictor.new(Team.all)
6        predictor.learn(Game.all)
7        prediction = predictor.predict(
8          Team.find(payload[:team_1_id]),
9          Team.find(payload[:team_2_id])
10       )
11
12       new_payload = payload.merge(winning_team_name: prediction.winner.name)
13       ActiveSupport::Notifications.instrument("prediction_completed", new_payload) do

```

```
14     end
15   end
16 end
17 end
```

The lines not highlighted in the block are familiar: We're creating and using the `Predictor` to generate a prediction. What is new is that instead of getting the IDs for team 1 and team 2 from params, we now get them from the event payload. The highlighted lines are new. The first one of these lines adds the prediction result to the payload and uses that to emit an new event, `prediction_completed`.

To get this subscriber loaded and thus used by the application we have to call its `configure` method in an initializer. We do this in a `Rails.application.config.to_prepare` block, to make sure this initializer works as expected for autoloaded constants.

```
1 Rails.application.config.to_prepare do
2   PredictionNeededSubscriber.configure
3 end
```

5.10.3 Determining where to send responses

Having reshaped the controller to add eventing, we're left with the problem of how to get the prediction result to the user. As we saw above, the `create` method in `PredictionController` can't directly render the result page anymore because it doesn't have access to any of the result's data. We need... more events!

The backend needs to be able to let the frontend know when it has the data ready for it. Classic Rails controllers and views don't quite cut it for this kind of stuff. But we don't have to blow up the entire frontend and completely switch to a React app just yet. Since Rails 5, Rails supports this kind of pattern natively with *ActionCable* (https://guides.rubyonrails.org/5_0_release_notes.html#action-cable). Since this will once again mean that we don't have to add any new libraries or services, I will use *ActionCable*'s patterns to wire up the frontend.

To align the prediction views with these new patterns we first reduce them from two (`new` and `create`) to just one, namely for `new`, which will take over the functionality of both. Below shows the updated view, where the top portion is the original `new` template and the highlighted portion at the bottom contains a placeholder for predictions being added to the frontend. So, in a change from the previous functionality, instead of having a prediction request page and a prediction result page, we now have one page where the requests and results are all shown together.

```

1  h1 Predictions
2
3  = form_tag prediction_path, method: "post", remote: true do |f|
4    .field
5      = label_tag :first_team_id
6      = collection_select(:first_team, :id, @teams, :id, :name)
7
8    .field
9      = label_tag :second_team_id
10     = collection_select(:second_team, :id, @teams, :id, :name)
11
12   = hidden_field :prediction_request, :id
13
14   .actions = submit_tag "What is it going to be?", class: "button"
15
16  h2 Predictions
17  #predictions

```

We can wire up this placeholder with the backend with ActionCable. ActionCable sets up a Web socket connection between the backend and the frontend, which we can use to stream events between them. We'll use this connection to send the prediction result to the frontend when it is ready. This connection will expose an insufficiency in our code thus far, because WebSocket connections need to be able to *target* a specific browser, which means we need a way to *identify* a browser session. And we have no way of doing that yet. We need a `current_user` concept.

In order to get a current user, we don't need a full-fledged user concept with login and account management. Instead, we can use the Rails session (https://guides.rubyonrails.org/action_controller_overview.html#session) to create a unique identifier that can act as a stand-in for a "current user."

```

1  class ApplicationController < ActionController::Base
2    append_view_path(Dir.glob(Rails.root.join("packages/*/app/views")))
3
4    before_action :ensure_session
5
6    def current_user
7      session[:current_user]
8    end
9
10   private
11
12   def ensure_session
13     session[:current_user] ||= "user_#{SecureRandom.uuid}"
14   end
15 end

```

The highlighted code above ensures that there is always a `:current_user` key on the session and, in case it is not yet set, it creates a unique and random new one.

While the above gives us a unique identifier for a session, we still have to ensure that `ActionCable` has access to this identifier as well. For this, we open the base class of `ActionCable` for our app, which is `ApplicationCable`. The `identified_by` method on the class specifies how connections are to be retrieved for use later on (https://guides.rubyonrails.org/action_cable_overview.html#connection-setup), which in our case uses `current_user`. The `connect` method gets called when a new connection is established and sets `current_user`, effectively connecting the random identifier created in `ApplicationController` to all subsequent uses of `ActionCable`.

```
1 module ApplicationCable
2   class Connection < ActionCable::Connection::Base
3     identified_by :current_user
4
5     def connect
6       self.current_user = current_user
7     end
8
9     def session
10      @request.session
11    end
12
13    private
14
15    def current_user
16      session[:current_user]
17    end
18  end
19 end
```

As the final piece to allow a request from the frontend to, at some point, return information via the `ActionCable` connection, we need to ensure that the current user information is made available throughout the request (and eventing) lifecycle. For that we need a small addition to `PredictionsController` and pass `current_user` into the event payload when the first event, the one indicating that a prediction is needed, gets created. The below is that updated controller, with the addition highlighted.

```

1  # typed: false
2
3  require "ostruct"
4
5  class PredictionsController < ApplicationController
6    def new
7      @teams = Team.all
8      @prediction_request = OpenStruct.new(id: SecureRandom.uuid)
9    end
10
11   def create
12     ActiveSupport::Notifications.instrument("prediction_needed", {
13       current_user: current_user,
14       prediction_request_id: params["prediction_request"]["id"],
15       team_1_id: params["first_team"]["id"],
16       team_1_name: Team.find(params["first_team"]["id"]).name,
17       team_2_id: params["second_team"]["id"],
18       team_2_name: Team.find(params["second_team"]["id"]).name
19     } ) do
20     end
21   end
22 end

```

5.10.4 Wiring backend and frontend together

The `current_user` in place and passed through our eventing layers we can wire everything together to get results back to the user.

ActionCable gives us the concept of *channels*, which are similar to controllers with respect the pieces of work they encapsulate (https://guides.rubyonrails.org/action_cable_overview.html#server-side-components-channels). We create a `PredictionChannel` to handle the passing of prediction results to the frontend. The `subscribed` method gets called when a consumer connects to the channel (this happens when a user successfully loads the predictions page in the browser - it becomes the consumer). Here, when this is the case, the channel streams all updates for `current_user` (https://api.rubyonrails.org/v7.0.0/classes/ActionCable/Channel/Streams.html#method-i-stream_for).

```

1  class PredictionChannel < ApplicationCable::Channel
2    def subscribed
3      stream_for current_user
4    end
5  end

```

The `PredictionCompletedSubscriber`, which finally closes the loop on the event emitted when `PredictionNeededSubscriber` completes a prediction, simply broadcasts its payload to the

PredictionChannel for the current_user. I.e., if there are any consumers connected to the channel they will receive prediction results if they are listening for the same user as the event is sent for.

```

1 class PredictionCompletedSubscriber
2   def self.configure
3     ActiveSupport::Notifications.subscribe("prediction_completed") do |name, start, \
4     finish, id, payload|
5       PredictionChannel.broadcast_to(payload[:current_user], **payload)
6     end
7   end
8 end

```

We expand the initializer to configure PredictionCompletedSubscriber:

```

1 Rails.application.config.to_prepare do
2   PredictionNeededSubscriber.configure
3   PredictionCompletedSubscriber.configure
4 end

```

Finally, we create the frontend, the javascript portion, of PredictionChannel.

```

1 import consumer from "./consumer"
2
3 consumer.subscriptions.create("PredictionChannel", {
4   received(data) {
5     this.appendLine(data)
6   },
7
8   appendLine(data) {
9     const html = this.createLine(data)
10    const element = document.querySelector("#predictions")
11    element.insertAdjacentHTML("beforeend", html)
12  },
13
14  createLine(data) {
15    return `
16      <article>
17        <span>
18          We predict that in <strong>${data.team_1_name}</strong> vs <strong>${data.\
19  team_2_name}</strong>.
20          The winner will be <strong>${data.winning_team_name}</strong>!
21        </span>

```

```

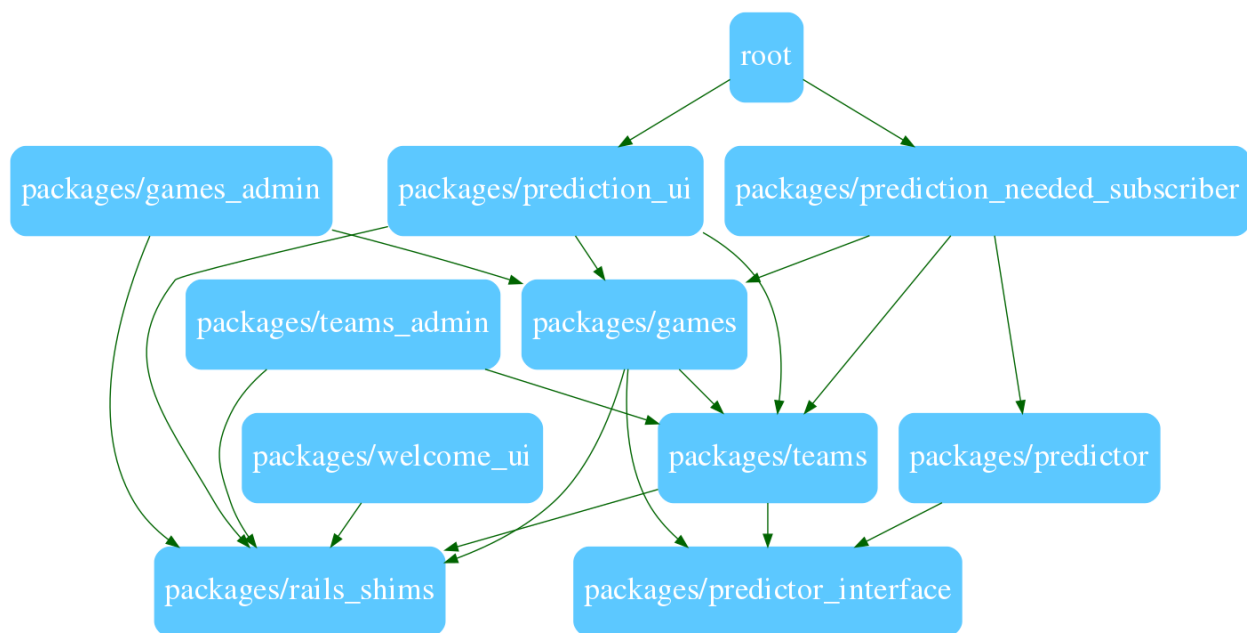
22     </article>
23   ,
24   }
25 })

```

In the above, `received` is the method that connects to `broadcast_to` on `PredictionChannel` (the Ruby side). Every time a broadcast happens on the channel data is received by the above code. Following the other two methods, we see that we (finally?!) extract from the event data `team_1_name`, `team_2_name`, and `winning_team_name` and create some HTML that gets inserted into the predictions page as the newest prediction result.

5.10.5 The package effect of eventing

Our final result for the event-based refactor leads to the following packaging structure diagram for Sportsball.



Dependency structure with service locator

There is a lot that could still be done to this line of refactoring. One direction that comes to mind for me is to add types to the events. This would help ensure that the various parts of the system creating and consuming events don't accidentally drift apart in their expectations of the contents of the events.

I could also see how the abstraction of how the eventing is implemented, i.e., remove the direct use of `ActiveSupport::Notifications` instrument and `subscribe`. This would also likely make it easier to revisit the decision of using `ActiveSupport` in the first case. And this brings us right back to what other options you have to implement events within your app. Common options include

RabbitMQ (<https://www.rabbitmq.com/>) a message broker implementation and Kafka (<https://kafka.apache.org/>) a message log implementation. I have seen simple eventing library implementations based on sidekiq (<https://sidekiq.org/>).

5.11 Beyond dependency refactorings

In this chapter we went over the technical refactorings we can use to change the shape of dependencies within our applications. As discussed at the beginning, this gives us tools and patterns but no direction. We get the direction from software engineering principles, from changing code and perceiving how it is to work with, and from gradual-modularization-specific progress measurement that we will develop in subsequent chapters.

Dependency enforcement is one of the two enforcements we get from packwerk. It is high time we turn to the other: Privacy enforcement. We will first tackle its technical aspects.

Beyond that, the implications of creating boundaries through enforcing dependencies and privacy require profound organizational or social changes in the engineering organization (and beyond!) and in order for us to be successful at making this change a positive and a lasting one, we have to learn how to create, monitor, and steer a process of *gradual adoption* of *Gradual Modularization*.

6. Privacy Violation Management Refactorings

We have extensively discussed multiple ways of removing and shaping package dependency violations in Chapters 2 and 5, respectively. In fact, source code versions following c2s06 have had no violations / deprecated references at all (other than in the detours showing specific violations for potential paths discussed).

With that in mind, this chapter turns to privacy violations and the options we have to manage them. To do this, we will continue to develop our sample application based on the state we developed in [Section 5.7](#). This means that we start with dependency injection supported by typing of classes involved in the injection pattern. Based on that state, the starting point for the code of this chapter rolls back the injection portion, but leaves in place the typing, as seen in https://github.com/shageman/package-based-rails-applications-book/blob/main/generator_scripts/generators/c6s01.sh.

Using this code, let's see how big our problems are with respect to privacy enforcement. We can do this by turning on privacy enforcement in all packages as follows. The command below uses `gsed` which is what you get when you `brew install gnu-sed` on MacOS. If you are on a Linux system, you will want to change this to be just `sed`.

```
1 $ find . -iname "package.yml" |\  
2   xargs gsed -i 's/enforce_privacy: false/enforce_privacy: true/g'
```

Now, when we check our app with `packwerk`, we'll see a bunch of privacy violations (shortened for better comprehensibility):

```
1 $ bundle exec packwerk check  
2 Packwerk is inspecting 61 files  
3  
4 .....E.....E.....E...EEEEEE....  
5 Finished in 0.65 seconds  
6  
7 packages/games/app/models/game.rb:2:13  
8 Privacy violation: '::ApplicationRecord' is private to 'packages/rails_shims' but re\  
9 ferenced from 'packages/games'.  
10 Is there a public entrypoint in 'packages/rails_shims/app/public/' that you can use \  
11 instead?  
12 Inference details: this is a reference to ::ApplicationRecord which seems to be defi\  
13 ned in packages/rails_shims/app/models/application_record.rb.
```

```

14 To receive help interpreting or resolving this error message, see: https://github.co\
15 m/Shopify/packwerk/blob/main/TROUBLESHOOT.md#Troubleshooting-violations
16
17
18 packages/games/app/models/game.rb:8:2
19 Privacy violation: '::Team' is private to 'packages/teams' but referenced from 'pack\
20 ages/games'.
21 Is there a public entrypoint in 'packages/teams/app/public/' that you can use instea\
22 d?
23 Inference details: this is a reference to ::Team which seems to be defined in packag\
24 es/teams/app/models/team.rb.
25 To receive help interpreting or resolving this error message, see: https://github.co\
26 m/Shopify/packwerk/blob/main/TROUBLESHOOT.md#Troubleshooting-violations
27
28 ... <another 28 offenses skipped over> ...
29
30 30 offenses detected
31
32 No stale violations detected

```

In our case, we get 30 privacy violations - all registered with a helpful message about what we might consider doing next: To move the referenced class into the public API of the containing package.

Below is the information of the packwerk check output summarized into a table showing which packages have violations and what constants are causing the privacy violation and which packages consume those constants. In the following sections, we address all of the packages in our application

Summary of package privacy violations			
package	Privacy violations?	Constants used	Consuming packages
.	No (1)		
packages/games	Yes (2)	Game	packages/games_admin, packages/prediction_ui
packages/games_admin	No (3)		
packages/prediction_ui	No (3)		
packages/predictor	Yes (4)	Predictor	packages/prediction_ui
packages/predictor_interface	No (5)		
packages/rails_shims	Yes (6)	ApplicationController	packages/games_admin, packages/prediction_ui, packages/teams_admin, packages/welcome_ui
packages/rails_shims	Yes (6)	ApplicationRecord	packages/games, packages/teams

Summary of package privacy violations			
package	Privacy violations?	Constants used	Consuming packages
packages/teams	Yes (2)	Team	packages/games, packages/prediction_ui, packages/teams_admin
packages/teams_admin	No (3)		
packages/welcome_ui	No (3)		

Take a moment to reflect on the contents of the table. Notice that the package violations are listed for the *packages whose use is being violated*. In contrast to this, for dependency violations it is the *violating package* that registers the violation. This is why one can also say that the enforcements implemented by packwerk are *outgoing dependency enforcements* and *incoming privacy enforcements*.

6.1 Packages without consumers should protect their public API



The code discussed in this section and the next is available from <https://github.com/shageman/package-based-rails-applications>: `git checkout c6s01`.

Look at the table above and you will see a couple of different cases (I group them by the numbers in column 2). As we can see in column 2, “privacy violations?,” there are quite a few packages that do not have privacy violations. Specifically, the following packages have no violations at all:

```

1  .
2  packages/games_admin
3  packages/prediction_ui
4  packages/predictor_interface
5  packages/teams_admin
6  packages/welcome_ui

```

6.1.1 The root package

The first “group” (group #1) contains only the root package “.”. This package has no privacy violations because we emptied it out completely during the refactorings of [Chapter 2](#). Turning on privacy enforcement is straightforward and will not require further work because there are no violations: We can and should just do it.

If you want to make extra sure that this is the right move, we could double check whether the root package is really completely emptied out? In fact: Not quite! Remember that we excluded a couple of folders that are part of the root package:

Exclude config in packwerk.yml

```
1 exclude:
2 - "{bin,node_modules,script,tmp,vendor}/**/*"
3 - "**/lib/tasks/**/*rake"
4 - "spec/support/**/*"
```

There is actually just one app file being excluded by this config, namely `spec/support/object_creation_methods.rb`. We excluded this file because to get rid of some unhelpful component *dependency* violations. From a *privacy* point of view, we can say that the module `ObjectCreationMethods` is injected into all tests via the RSpec config. In this way, this code is injected into every package. Because of this, we can expect that this file is not directly depended on by other packages.

What do we actually get from turning on privacy enforcements for the root package? The obvious answer is that it guards against any future uses of any code that might get added to the root package. This is the reason I recommend just doing it, even if there is also a similarly obvious counter-argument: “guard against any future uses?” We already have that because dependency enforcements are turned on and no other package depends on the root package. Notice the subtle difference in the protections though: the dependency enforcement is controlled by *other* packages (and these other packages can choose to add the root package as a dependency) whereas the privacy enforcement is controlled by the *root* package. Now imagine that two different teams are responsible for the root vs other packages: If the root package team wants to ensure proper (non) use of its package it has one tool: privacy enforcement.

Long story short: We should just turn privacy enforcement on.

6.1.2 The UI and admin packages

The next group of packages without privacy violations is group #3 - the admin and UI packages:

```
1 packages/games_admin
2 packages/prediction_ui
3 packages/teams_admin
4 packages/welcome_ui
```

For these packages, we can repeat the analysis we just did for the root package: There are no privacy violations because there are no uses of these packages from other packages. Once again, by turning on privacy enforcement for these packages, an owning team can ensure that no unexpected usage occurs in the future.



But wait a minute... The Rails application uses these packages for the controllers to respond to web request and their views to render the right template?! While this is true, the dependency that Rails has on these packages is invisible to packwerk’s constant resolution approach. Furthermore, the dependency is not “from another package,” so packwerk also doesn’t detect anything because there is no *source* to the violation.

In a slightly more complex application than Sportsball, you will likely not find packages like these, where privacy enforcement can simply be turned on without causing violations, because things are more murky. In that case, and depending on your analysis of the violation patterns, you have a couple of simple violation management options that allow turning on privacy enforcement and prevent violations.

The one simple refactoring we *cannot* apply for privacy that worked for dependencies is *accepting* the violation (which is what we did for dependency enforcement in [Section 5.2](#)). We can, of course, do nothing and be ok with the violation listed in the deprecated references, as we discussed for dependency violations in [Section 5.1](#). Beyond that, there is simply no analog to the list of accepted dependencies and as such there is not more we can do.

All other simple refactorings from [Chapter 5](#) can be used for privacy violations just as much as for dependency violations:

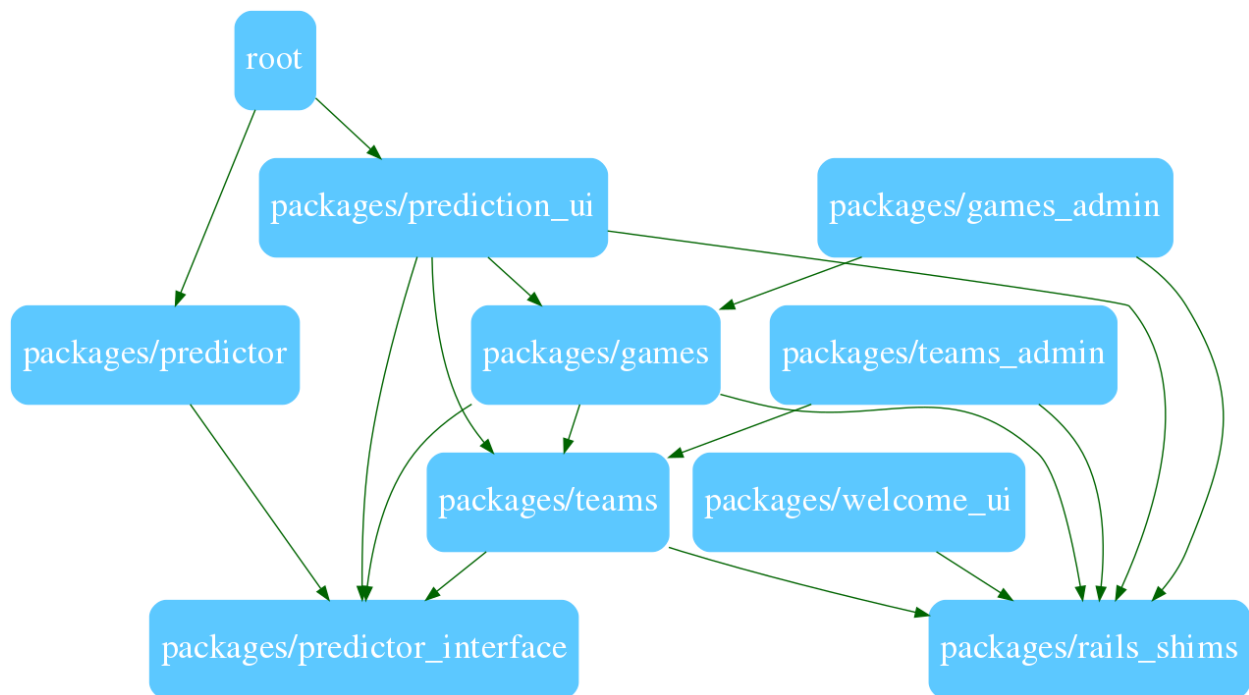
- Merge packages ([Section 5.3](#)) works because it gets rid of the boundary between two usages.
- Split packages ([Section 5.4](#)) works if you cordon off the code whose privacy is violated into another package. The originating package will no longer have privacy violation.
- Move code between packages ([Section 5.5](#)) works when we move out of the package all the code whose privacy is being violated.
- Duplicate functionality ([Section 5.6](#)) works if it removes all uses of the functionality in the originating package.

None of the above refactorings solve all our privacy concerns. Depending on how we use them, we might even decrease the overall quality of the modularization of our application (but that analysis will have to wait till the next chapter).

6.1.3 packages/prediction_interface and interface classes

The final group of packages without privacy violations is group #5 and once again it is a group with only one package in it. packages/prediction_interface came out of our work in [Section 5.8.3](#) and it, too, has no privacy violations to report, but for a different reason than the packages discussed so far.

packages/prediction_interface defines four classes: Contender, HistoricalPerformanceIndicator, Prediction, and PredictorInterface. All of these were created to disentangle the types and hidden dependencies involved in injecting Predictor as a dependency into packages/prediction_ui. In fact, the following packages are dependent on the prediction interface package: packages/prediction_ui, packages/predictor, packages/games, packages/teams. Refer also back to the package diagram from that section, [Figure “Dependency inversion disconnects packages/prediction_ui from packages/predictor”](#), repeated here.



Dependency inversion disconnects `packages/prediction_ui` from `packages/predictor`

Despite its usage, this package has no privacy violations because I created all of the classes within this package in the `app/public` folder in the first place. While it wasn't relevant for [Section 5.8.3](#), the thinking I applied was that even if we don't care about privacy concerns right now, if I am creating classes in a new package specifically to support the interaction of other packages then those classes must naturally be part of the public API - whether we enforce it or not.

I encourage you to look around your applications with the same view. If there are classes that fit the above description, very likely you will want them to be part of the public API of their packages too. If they are not yet in `app/public`, move them there and use `packwerk update-deprecations` to investigate the violation reductions that your application wins.

In conclusion, groups #1, #3, and #5 give us three sets of packages that have no privacy violations for different reasons. The effect is always the same: We can turn on privacy enforcement without incurring any deprecated references. Looking around your application you may or may not find packages without violations immediately. Look closely at the packages in your lists of the groups from above: the root package, UI-focused packages, interface packages. They should have – if not zero – then relatively few violations. It may feel beneficial to turn on privacy enforcement relatively early in your modularization journey because this helps you create a warning system against unintended uses of these kinds of packages.

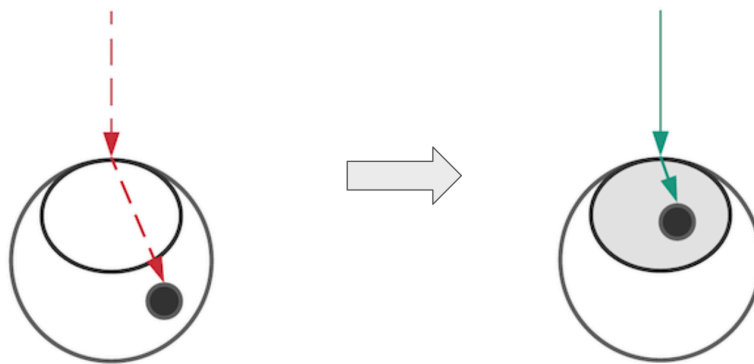
Let's now turn to the three groups of packages where Sportsball does list privacy violations at the moment: groups #2, #4, and #6. We won't tackle them in this order, though, we'll tackle them as #4, #6, and #2.

6.2 Expose existing service classes



The code discussed in this section is available from <https://github.com/shageman/package-based-rails-applications>: `git checkout c6s02`.

`package/predictor` is a perfect example of the first explicit privacy refactoring: Expose existing functionality. Use it for code that seems appropriate for use in the public API. Figure “The effect of moving a class into the public API of a package” shows the basic impact of the change. In addition to the refactoring diagrams from Chapter 5 the ones in this chapter add an oval on the inside of the circle that depicts the package. The oval indicates the public API part of the package content.



The effect of moving a class into the public API of a package

Violation group #4 is `packages/prediction_ui`'s dependency on `packages/predictor`'s `Predictor`. To analyze why I suggest that we can simply expose its existing functionality via the public API, let's take a look at the current state of its public methods.

Definitions of public methods in `packages/predictor/app/models/predictor.rb`

```

1  require "saulabs/trueskill"
2
3  class Predictor
4    include PredictorInterface
5    extend T::Sig
6
7    sig {override.params(teams: T::Enumerable[Contender], games: T::Enumerable[Histori\
8    calPerformanceIndicator]).void}
9    def learn(teams, games)
10      ...
11    end
12
13    sig {override.params(first_team: Contender, second_team: Contender).returns(Predic\

```

```
14   tion}}
15   def predict(first_team, second_team)
16     ...
17   end
18 end
```

And, just so we have a complete picture on these pages, here is also a recap of the implementations of the classes mentioned in the public methods above: Contender, HistoricalPerformanceIndicator, and Prediction.

Contender defined in packages/predictor_interface/app/public/contender.rb

```
1 module Contender
2   extend T::Sig
3   extend T::Helpers
4   interface!
5
6   sig { abstract.returns(Integer) }
7   def id; end
8 end
```

HistoricalPerformanceIndicator defined in packages/predictor_interface/app/public/historical_performance_indicator.rb

```
1 module HistoricalPerformanceIndicator
2   extend T::Sig
3   extend T::Helpers
4   interface!
5
6   sig { abstract.returns(Integer) }
7   def first_team_id; end
8
9   sig { abstract.returns(Integer) }
10  def second_team_id; end
11
12  sig { abstract.returns(Integer) }
13  def winning_team; end
14 end
```

Prediction defined in packages/predictor_interface/app/public/prediction.rb

```

1 class Prediction
2   attr_reader :first_team, :second_team, :winner
3
4   def initialize(first_team, second_team, winner)
5     @first_team = first_team
6     @second_team = second_team
7     @winner = winner
8   end
9 end

```

The Predictor class has been the object of some of the most intense thinking in the refactoring history of Sportsball (especially in [Section 5.8](#) but also in the *Component-based Rails Applications* book in various places). That is why this class has the public methods that it does. The predictor does not know how to retrieve games or teams, which was core to disentangle it from the rest of the application. The methods also hide *how* predictions are happening (and the fact that the actual calculations are out sourced into the `trueskill` gem).

As such, this class, Predictor, is *designed* and designed well enough to where I feel comfortable explicitly making it the public API of the packages/predictor package. The refactoring is easy:

```

1 mkdir -p packages/predictor/app/public
2 mv packages/predictor/app/models/predictor.rb packages/predictor/app/public
3
4 rm packages/prediction_ui/deprecation_references.yml
5 # We can skip the above command once
6 # https://github.com/Shopify/packwerk/issues/49 is addressed
7 bin/packwerk update-deprecations

```

To make this practically relevant to the reader, we should ask where you might be able to find such classes as part of your packages. Some questions that might help you pinpoint them:

- Have you invested in service classes to extract functionality out of controllers to make “fat controllers” more skinny? Look there.
- Do you have classes that are the entry point into the parts of your app that make your application unique? Look there.
- What classes do you need to know to interact with *that other part of the system*? Look there.

You might find that there are classes very similarly well designed and you feel very confident that they are part of the public API. Perfect. Move them into the `app/public` folder and make it official.

You might find that these classes are not quite as perfect. You are not as confident that they are the best API you could come up with. In that case, ask yourself whether they are as good as start as

any you might come up with right now. If so, consider moving them into `app/public` and make it official.

This may feel like a difficult decision. Remember that Gradual Modularization is all about using opportunities for gradual, incremental change. It is about experimenting and learning. Things can and will get messy and that is ok. If the process ever feels *too* messy, think of how messy things were when you didn't have the tools to visualize the state of the system... It was still a mess, we just couldn't see it as well.

6.3 Expose existing service classes



The code discussed in this section and the next is available from <https://github.com/shageman/package-based-rails-applications>: `git checkout c6s03`.

Violation group #6 encompasses the dependencies of the UI packages (`packages/games_admin`, `packages/prediction_ui`, `packages/teams_admin`, and `packages/welcome_ui` on `ApplicationController`) and of the model packages (`packages/games`, `packages/teams` on `ApplicationRecord`) onto `packages/rails_shims`.

Remember that we created the shims packages to remove the circular dependencies from all the packages listed above onto the root package. Despite only actively using and thus referencing `ApplicationRecord` and `ApplicationController`, we decided to move all of the classes that have these “base class,” “glue,” and configuration responsibilities for the various parts of the Rails framework. The following are the current contents.

Contents of `packages/rails_shims`

```

1  .
2  └─ app
3    └─ channels
4        └─ application_cable
5            └─ channel.rb
6            └─ connection.rb
7    └─ controllers
8        └─ application_controller.rb
9        └─ concerns
10   └─ helpers
11       └─ application_helper.rb
12   └─ jobs
13       └─ application_job.rb
14   └─ mailers
15       └─ application_mailer.rb

```

```

16 |   └─ models
17 |       └─ application_record.rb
18 |       └─ concerns
19 └─ package.yml
20 └─ spec

```

One opinion I have come across as to what to do with a package like this is to accept that it doesn't have a good API and never turn on privacy protections for it. I acknowledge this argument and think it is a fine first position to take. However, I believe we can do better. We would need `ApplicationRecord` and `ApplicationController` to be in the public API if we wanted to activate privacy protections without creating deprecated references. And there is nothing stopping us: Move these files to `app/public` and turn on `enforce_privacy`. Then run `packwerk check` and it will succeed! Why *not* do this?

In fact, we should also not stop there. Looking at the rest of the package, we see similar base classes for channels, helpers, jobs, and mailers. All of these, we currently do not use, but for all of these we *could* do the same as we just did for `ApplicationRecord` and `ApplicationController`. They have the same role in the application and will be needed in the public API when we start using these functionalities.

"But..." you might counter, "we just moved *the entire package* into its public API. That is certainly one way to activate privacy enforcement and not create violations!" It may feel like that, but it really isn't (at least not for long). Imagine we add a somewhat complex functionality to one of these classes. Let's say, a `before_action` in `ApplicationController` that secures allows child controllers to control access to certain controller endpoints. Let's also say that this functionality is implemented in a class `SecureAccessEndpoint`. Where should this class go? I would argue that a reasonable place is `app/controllers/secure_access_endpoint` but *not* `app/public/controllers/secure_access_endpoint`. We want to hide this part of `packages/rails_shims` to be hidden from consumers of the functionality of this package. It does not belong in the public API.

You will likely encounter problems when these base classes attract a lot of functionality. This analysis should not be taken as an argument for adding lots of functionality to these case classes, but rather exclusively as an argument for why we can use privacy enforcement for this kind of package.

6.4 ActiveRecord handled naively



The code discussed in this section and the next is available from <https://github.com/shageman/package-based-rails-applications>: `git checkout c6s04-1`.

The last group of package violations is group #2. This violation group encompasses dependencies on `Game` from `packages/games_admin`, `packages/prediction_ui` and on `Team` from `packages/games`, `packages/prediction_ui`, and `packages/teams_admin`.

Handled naively, we can do what we have done throughout this chapter so far and move `Game` and `Team` into their respective public APIs. While this is very straightforward and works, in modularization terms, it is also only a small step.

I drag you through half this chapter just moving things into the public API of packages and suddenly now that is *only a small step*?? If it feels like that, then it is because we haven't looked at the differences between the things we have been putting into these public APIs. We have, of course, looked at the merits of the various classes under consideration. Yet, all of them serve very distinct purposes; It is not obvious how to compare them systematically.

One dimension by which analysis can be enlightening is along a specific vs generic axis. I.e., how specific or generic is an API. The API being made up of classes and objects, which have functionality through methods, we can compare the methods. Two assumptions that are helpful:

- All methods have similar amounts of specificity. This is not generally true: Just imagine the specificity of the two methods `fix(hash)` and `fix_by_setting_opening_to(int_value)`
- Methods are mostly independent in functionality. This is also a pretty naive assumption.

Whether they are particularly good or not, with these assumptions we can compare the specificity of package APIs simply by counting the number of methods that are being exposed by them.

Below is a table of such a comparison for violation groups #2, #4, and #6. Instead of directly listing the counts, I have subtracted the number of methods on the `Object` class (183) and instance (88), respectively. I subtract these methods because they largely fall into a bucket that I am going to call “Ruby object (lifecycle) stuff” *and* because I want to make a point `~_(_)_`

Method counts of different classes

Group / Class	Diff to Object class method count	Diff to Object instance method count
group #4		
Predictor	1	2
group #6		
ApplicationRecord	610	NotImplementedError
ApplicationController	184	295
group #2		
Game	611	501
Team	611	357

The specificity of `Predictor`, the class in violation group #4, stands out among the entries of this table. There are a grand total of three (3!) methods in this API. Compare that to the 1,089 total methods of group #6 (contributed by `ApplicationRecord` and `ApplicationController` and ignoring all the other classes in that package for the moment). This does add another perspective on the two privacy enforcement discussions from above: `Predictor` is *also* such a clear case because the API

created by it is so specific. `packages/rails_shims` contains a bunch of much more generic classes and the discussion was much more involved.

`packages/games` (1,112 methods) and `packages/teams` (968 methods) both individually contribute about as many methods as the classes in `packages/rails_shims` together. The difference between the former two packages and the Rails shims package is that, with the latter, we are forced to use Rails' mechanism of object creation with inheritance being the way that models, controllers and such are getting configured to work properly within the application. Is this the same for `Game` and `Team`? The sort answer is "no." It is not the same, we *can* decide to not expose these ActiveRecord objects in their packages' APIs. And we will - next. Getting there, however, will turn out to be quite a bit of work.

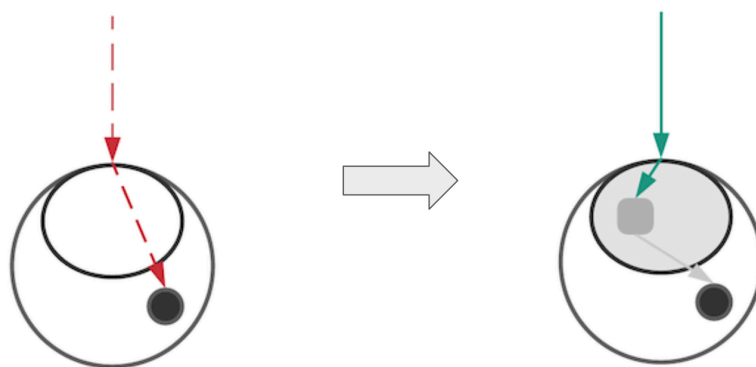
6.5 Hide ActiveRecord



The code discussed in this section and the next is available from <https://github.com/shageman/package-based-rails-applications>: `git checkout c6s04-2a` (for the `Team` refactor) and `git checkout c6s04-2b` (for the `Game` refactor).

Removing ActiveRecord objects from the API of a package is a strong privacy enhancing and package boundary creating move. The goal is to create a more specific API and get closer to something like `Predictor` and away from something like `Team`. This is, of course, a tradeoff - a whole bunch of tradeoffs, actually. We have to put in work to refactor the system to create the new API, but then we get a more specific API. A more specific API is less confusing and more clear as to its capabilities and likely its intended uses, but it is limiting in the exact same way and any changes will take more work, because functionality has to be added.

Figure "Create a custom API implementation for a private class" shows a diagram of the refactoring we are about to embark on: We are creating a modified / reduced / specialized version of functionality specifically for the addition to the public API of the package. We will discuss this change using the example of `packages/teams`.



Create a custom API implementation for a private class

6.5.1 A new form of `Team`

The first question to consider is: *What should the package expose?* Specifically, *what name* should it expose. In the naive version of the previous section `Team` was exposed. Should `Team` continue to be the ActiveRecord object or should it continue to be the class exposed in the public API? Given our goal for this section, it can't be both. Two reasons push me to use `Team` as the name for the class in the public API:

- `Team` is already the constant being referred to by the rest of the application
- `Team` is a nice and short name

One counter argument is that this choice forces us to rename the ActiveRecord class `Team`. This then either requires us to rename the underlying database table to follow conventions or to explicitly configure the table name in the class. If we accept that that is a consequence of this decision, this is what things would look like if we renamed `Team` to `TeamRecord`:

Rename of `Team` to `TeamRecord` requires explicitly setting the table name

```
1 class TeamRecord < ApplicationRecord
2   self.table_name = "teams"
3
4   include Contender
5   extend T::Sig
6
7   validates :name, presence: true
8 end
```

I feel that this refactoring creates the cleanest change set, which is why I am going to pursue it here. As an exercise you could explore making different decisions for which class to rename or doing a migration to rename the database table to match.

Note, that the file defining `TeamRecord` won't move. It will remain in `packages/teams/app/models`.

After the respective spec file gets the same rename treatment, the test should pass like before:

Rename of `Team` to `TeamRecord` in the class' spec

```
1 RSpec.describe TeamRecord do
2   it { should validate_presence_of :name }
3 end
```

It is now time to create the new `Team` to be used within the public interface of the packages. Here it is in full before we go over all aspects of the code in detail.

New class `Team` in `packages/teams/app/public/team.rb`

```
1 class Team
2   include ActiveSupport::Conversion
3   extend ActiveSupport::Naming
4   include ActiveSupport::Validations
5
6   include Contender
7   extend T::Sig
8
9   validates :name, presence: true
10
11  attr_reader :id, :name
12
13  def initialize(id, name)
14    @id = id
15    @name = name
16  end
17
18  def persisted?
19    !!id
20  end
21
22  def to_hash
23    { id: id, name: name }
24  end
25
26  def hash
27    to_hash.hash
28  end
29
30  def ==(other)
31    id == other.id && name == other.name
32  end
33
34  alias eql? ==
35 end
```

Be aware that there are many ways to create the new `Team` class, especially when it comes to the decisions of using or forgoing parts of the Rails support system. As such, treat this specific version as an example and revisit all the decisions made for your use cases.

Let's build up the entirety of `Team` by looking at several constituent parts.

We will specifically start with the `initialize` method. And right from the start there are a lot of options: Do we want to take in model attributes in `initialize`? Or rather have them be settable after the fact? Do we want both? I am opting here for a fairly restrictive way of doing things by only allowing attributes to be set via `initialize`. Afterwards, the attributes can be read but not updated, for which `attr_reader` is being used:

```
1 class Team
2   attr_reader :id, :name
3
4   def initialize(id, name)
5     @id = id
6     @name = name
7   end
8 end
```

As we want to be able to use this new `Team` in place of its previous version, we include the `Contender` interface and the needed extension of Sorbet's `T::Sig`.

```
1 include Contender
2 extend T::Sig
```

By adding `include ActiveRecord::Validations` we can use `validate` as we would on an `ActiveRecord` object, which means that it properly responds to `valid?` calls. Read more about what we get from including validations at <https://api.rubyonrails.org/classes/ActiveModel/Validations.html>.

```
1 include ActiveRecord::Validations
2
3 validates :name, presence: true
```

Next, we are adding `extend ActiveRecord::Naming` which allows the `Team` class to continue to behave as expected for the generation of path names within routes and link helpers. Read more about what we get from extending the `Naming` module at <https://api.rubyonrails.org/classes/ActiveModel/Naming.html>.

```
1 extend ActiveRecord::Naming
```

The last addition of Rails' functionality is `include ActiveRecord::Conversion`, which makes the class usable for generation of keys and parameters. The `persisted?` method is part of this addition because specifically `to_param()` - one of the methods that is part of conversions - uses it to switch behavior between persisted and non-persisted models. For our implementation we simply check whether `id` is a non-falsy value. Read more about what we get from including conversions at <https://api.rubyonrails.org/classes/ActiveModel/Conversion.html>.

```
1  include ActiveRecord::Conversion
2
3  def persisted?
4    !!id
5  end
```

We add a `to_hash` method which serializes a `Team` object into a hash of its attributes. We add this to allow team objects to be used with ActiveRecord's `create` and `update` methods, which we will get to further down. We also will be using this method in the next and final block of code additions to `Team`.

```
1  def to_hash
2    { id: id, name: name }
3  end
```

For the last piece of code in `Team`, we add comparison code to make sure that comparing two teams uses the equality of the attributes `id` and `name` to determine object equality. If you are surprised in seeing three methods instead of just `==`... well, there are even more methods for variants of equality in Ruby! Open some of the search results on a search for “ruby hash eql ==” like so <https://duckduckgo.com/?q=ruby+hash+eql+%3D%3D> to learn more.

```
1  def hash
2    to_hash.hash
3  end
4
5  def ==(other)
6    id == other.id && name == other.name
7  end
8
9  alias eql? ==
```

The following code snippet is the part of the team tests that verify the correct behavior of equality tests. If you are curious, play around with this test by commenting out one or more of the above methods. The reason these tests look the way they do is because under the hood, Ruby actually switches how it compares arrays when they get larger. Check out <https://ruby-doc.org/core-2.5.1/Array.html#method-i-2D> and click on “click to toggle source” if you want to dig in more.

Excerpt from `packages/teams/spec/public/team_spec.rb` showing equality tests

```

1  describe "comparisons" do
2    it "should behave as expected" do
3      expect(Team.new(1, "1")).to eq Team.new(1, "1")
4
5      t = (1..2).map { |i| Team.new(i, "#{i}") }
6      t2 = (1..2).map { |i| Team.new(i, "#{i}") }
7      expect(t - t2).to eq([])
8
9      t = (1..15).map { |i| Team.new(i, "#{i}") }
10     t2 = (1..14).map { |i| Team.new(i, "#{i}") }
11     expect(t - t2).to eq([Team.new(15, "15")])
12
13     ## testing hash and eql? based comparisons
14     t = (1..20).map { |i| Team.new(i, "#{i}") }
15     t2 = (1..19).map { |i| Team.new(i, "#{i}") }
16     expect(t - t2).to eq([Team.new(20, "20")])
17   end
18 end

```

6.5.2 A TeamRepository to manage teams

ActiveRecord objects have that name because they are both “records” and because they are “active.” Our new team, in comparison, is pretty *inactive*. According to Martin Fowler (<https://www.martinfowler.com/eaCatalog/activeRecord.html>) an *active record* is “[a]n object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.” The new Team implementation can still be said to encapsulate a row in the database table - after all, it has the exact same attributes as TeamRecord. We didn’t have domain logic in Team, so no real change here. What is relevant right now is that we have lost the encapsulation of database access. And we should not want it back.

We don’t want database access in Team. As Team object gets passed around the system, we want to have access to its data and business logic. By preventing database access through Team we communicate clearly that the place to do that is `packages/teams`, the owner of the domain concept of Team. That said, `packages/teams` does not yet have a way to perform the data access. We can use the *repository pattern* (<https://martinfowler.com/eaCatalog/repository.html>) to create it. A repository “Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.” To support all the use cases previously handled by ActiveRecord we need to create `get`, `list`, `add`, `edit`, `delete`, and `count` methods on the repository. Here it is in full:

A TeamRepository

```
1  # typed: strict
2  class TeamRepository
3    extend T::Sig
4
5    sig { params(id: Integer).returns(T.nilable(Team)) }
6    def self.get(id)
7      team_record = TeamRecord.find_by_id(id)
8      Team.new(team_record.id, team_record.name) if team_record
9    end
10
11    sig { returns(T.any(NilClass, T::Array[Team])) }
12    def self.list
13      TeamRecord.find_each.map { |t| Team.new(t.id, t.name) }
14    end
15
16    sig { params(team: Team).returns(Team) }
17    def self.add(team)
18      team_record = TeamRecord.create(team.to_hash)
19      team = Team.new(team_record.id, team_record.name)
20      team.instance_variable_set(:"@errors", team_record.errors)
21      team
22    end
23
24    sig { params(team: Team).returns(T.any(FalseClass, Team)) }
25    def self.edit(team)
26      team_record = TeamRecord.find_by_id(team.id)
27      return false unless team_record
28      team_record.update(team.to_hash)
29      team = Team.new(team_record.id, team_record.name)
30      team.instance_variable_set(:"@errors", team_record.errors)
31      team
32    end
33
34    sig { params(team: Team).void }
35    def self.delete(team)
36      TeamRecord.delete(team.id)
37    end
38
39    sig { returns(Integer) }
40    def self.count
41      TeamRecord.count
```

```
42   end
43 end
```

Notice how the signatures confirm the design of the repository and its ability to hide ActiveRecord. In fact, Integer and Team are the only base types being expected or returned by the API.

The API of the repository is designed to mimic, closely, how ActiveRecord constructs actions. That is the reason why edit returns the somewhat odd T.any(FalseClass, Team), with “false” returned indicating that the update of the object failed. Another piece of code that readers might find surprising is team.instance_variable_set(:"@errors", team_record.errors), which again tries to keep differences with ActiveRecord behavior to a few. How far you *should* veer away from Rails and ActiveRecord conventions is a question of tradeoffs. On the one side is quirky code like this. On the other is more or less need for adaptations of other Rails-related code as we will see further down.

If the repository itself already feels like a lot of code, then things will only get worse, if we consider that we should properly test this additional interface to the package. What follows are the test blocks I created for this code and which tries to cover all the branches this code can run through.

As I said above: Getting rid of ActiveRecord at the boundary is a lot of work. It is a big investment.

```
1  RSpec.describe TeamRepository do
2    describe "#get" do
3      it "returns a team when found"
4      it "returns nil when not found"
5    end
6
7    describe "#list" do
8      it "returns all teams"
9    end
10
11   describe "#add and #count and #list" do
12     it "adds a new team to the repository which is counted and listed"
13   end
14
15   describe "#edit" do
16     it "changes a team in the repository when found"
17     it "does not change a team in the repository when NOT found"
18     it "does not change a team in the repository when NOT valid"
19   end
20
21   describe "#delete" do
22     it "removes teams from the repository when found"
23     it "does not remove team from the repository when NOT found"
```

```
24   end
25 end
```

6.5.2.1 Performance implications of hiding ActiveRecord

First things first: There is no direct impact of hiding ActiveRecord. However, by limiting the capability of a packages API, we get the bad with the good. The good we are working towards is to create more meaningful APIs. *Meaning* in this case is actually closely related to *capability* - reduced capability creates more clarity of what is supported and what not, which said a slightly different way, is about “what we are meant to be doing with this code.”

The bad is that in hiding the flexibility of ActiveRecord, we, if we are not careful, lose its powers too. The implementation of `TeamRepository` we just discussed is a case study of what those lost powers are, some obvious, some subtle. Here is a likely non-exhaustive list:

- **Loss of findability:** It is no longer possible to find teams by their name, because no method for that is exposed in the API. This is by design: There was no need for a `find_by_name` functionality and thus it wasn't added. It is trivial to add it should use cases require it.
- **Loss of attribute selectability:** `get`, `list`, `add`, and `edit` all take and return instances of `Team`. `Team` has as its only attributes `id` and `name`. There is no way to retrieve a list of teams with only their names populated. If that sounds like a weird query to make, just imagine that the table for teams has a lot more columns with all sorts of other information on teams such as the typical metadata you'd find on a sport team's page on wikipedia. E.g., https://en.wikipedia.org/wiki/D.C._United: nickname, year founded, location, stadium, capacity, owner, head coach, website, etc. The more data there is on a model, the more likely it is that certain use cases only need a subset of that data. By not allowing a subselection of the attributes, database queries are slower and objects' memory consumption is higher.
- **Loss of memory-efficient collections:** The `list` method uses `find_each` for its implementation. This method is specifically designed to make ActiveRecord find operations not run out of memory (https://apidock.com/rails/ActiveRecord/Batches/find_each). The documentation states: “Yields each record that was found by the find options. The find is performed by `find_in_batches` with a batch size of 1000 (or as specified by the `:batch_size` option).” It is great that we have this method, but we're not using it well. The rest of the implementation iterates over the results and instantiates an array of `Team` objects. Just like the `TeamRecord` objects found might not fit into memory, the `Team` objects might not fit either. And we did not build a way to guard against that.

Combine the three losses for the triple whammy! Say you have a million teams in your database *and* all the metadata we envisioned could be on there *and* you want to find the teams in the District of Columbia in the US to ultimately create a distribution of the year they were founded. If you do not iterate on the public API and simply use what has been built thus far (ignoring that the attributes are missing currently), you'd simply do this:


```
1 TeamRepository.all.select { |t| t.location == 'DC' }.map{ |t| t.year_founded }.tally
```

Here we fetch a million records and instantiate a million (fully populated) teams, only to drop most of the teams and most of the attributes. Ouch.

The underlying query fetches all columns for all teams:

```
1 SELECT "teams".*
2 FROM "teams"
3 ORDER BY "teams"."id" ASC
4 LIMIT 1000
```

Using ActiveRecord directly might result in a query like this:

```
1 TeamRecord.where(location: 'DC').select(:id, :year_founded).find_each.map{ |t| t.y\
2 ear_founded }.tally
```

The underlying database query selects only the needed fields and returns only those teams located in DC. I am pretty confident that this query is always better than the above and will be vastly outperform the above for large tables if there is an index on location.

```
1 SELECT "teams"."id", "teams"."year_founded"
2 FROM "teams"
3 WHERE (location = "DC")
4 ORDER BY "teams"."id" ASC
5 LIMIT 1000
```

6.5.3 Migrating consumers to the new API

With our new API in place, we can start to turn towards other parts of the application that use this package. The packages using code from packages/teams are package/games, package/prediction_ui, and package/teams_admin.

6.5.3.1 Migrating package/teams_admin

This section is going to skip right over one possible way to resolve any issues that consumers of packages/teams: switch any usage of Team to use TeamRecord. Such a change would be a pure rename and create those renames in deprecated references, but not remove any violations, of course.

Instead, in this section, we keep using Team and add the usage of TeamRepository where it becomes necessary. Specifically, in package/teams_admin we have to fix TeamsController and its tests (i.e., packages/teams_admin/spec/requests/teams_spec.rb).

The following is the new version of TeamsController with the changes from the previous version.

TeamsController with changes to accommodate new Team and TeamRepository

```

1 class TeamsController < ApplicationController
2   before_action :set_team, only: %i[ show edit update destroy ]
3
4   def index
5     @teams = Team.all
6     @teams = TeamRepository.list
7   end
8
9   def show
10  end
11
12  def new
13    @team = Team.new
14    @team = Team.new(nil, nil)
15  end
16
17  def edit
18  end
19
20  def create
21    @team = Team.new(team_params)
22    respond_to do |format|
23      if @team.save
24      @team = TeamRepository.add(Team.new(nil, team_params[:name]))
25      respond_to do |format|
26        if @team.persisted?
27          format.html { redirect_to team_url(@team), notice: "Team was successfully cr\
28 eated." }
29          format.json { render :show, status: :created, location: @team }
30        else
31          format.html { render :new, status: :unprocessable_entity }
32          format.json { render json: @team.errors, status: :unprocessable_entity }
33        end
34      end
35    end
36
37  def update
38    respond_to do |format|
39      if @team.update(team_params)
40      @team = TeamRepository.edit(Team.new(params[:id], team_params[:name]))
41      if @team.errors.empty?
42        format.html { redirect_to team_url(@team), notice: "Team was successfully up\

```

```

43   dated." }
44       format.json { render :show, status: :ok, location: @team }
45   else
46       format.html { render :edit, status: :unprocessable_entity }
47       format.json { render json: @team.errors, status: :unprocessable_entity }
48   end
49   end
50   end
51
52   def destroy
53   ——@team.destroy
54       TeamRepository.delete(@team)
55
56       respond_to do |format|
57         format.html { redirect_to teams_url, notice: "Team was successfully destroyed.\n" }
58       }
59       format.json { head :no_content }
60   end
61   end
62
63   private
64   def set_team
65   ——@team = Team.find(params[:id])
66       @team = TeamRepository.get(params[:id])
67   end
68
69   def team_params
70     params.require(:team).permit(:name)
71   end
72 end

```

My attempts to keep interface changes to `Team` small, lead us to have the same big-picture structure. However, as we can see from the above, there is still some change to every reference to `Team` because of tweaks needing to be made. Especially for eyes trained to look at standard Rails stuff, I suspect that the changes don't look nice. Particularly, `if @team.update(team_params) vs @team = TeamRepository.edit(Team.new(params[:id], team_params[:name])); if @team.errors.empty?` feels a lot more clunky. Partly, this is my silly implementation, partly this stems from the repository pattern. In the end, this kind of change simply has to be part of the tradeoff analysis you are making

Please refer to the source code to this section to check on analogous changes that need to be made to `packages/teams_admin/spec/requests/teams_spec.rb`. Specifically, <https://github.com/shageman/package-based-rails-applications/commit/422bbd1ea560e5d3495e51a8373b81b6dc68141c> allows you to check out the needed changes. I won't list them here as they are very similar in nature to what

we are seeing with `TeamsController`.

The more interesting changes that are in support of the needed changes to the tests of `TeamsController` are the ones needed in the `ObjectCreationMethods`.

Excerpt of `ObjectCreationMethods` focussed on teams

```

1 module ObjectCreationMethods
2
3   #...
4
5   def team_params(overrides = {})
6     defaults = {
7       id: nil,
8       name: "Some name #{counter}"
9     }
10    defaults.merge(overrides)
11  end
12
13  def new_team(overrides = {})
14    Team.new { |team| apply(team, team_params(overrides), overrides) }
15    a = team_params(overrides)
16    Team.new(a[:id], a[:name])
17  end
18
19  def create_team(overrides = {})
20    new_team(overrides).tap(&:save!)
21    team = TeamRepository.add(new_team(overrides))
22    Kernel.raise "Team creation failed" unless team.persisted?
23    team
24  end

```

It is because the new version of `Team` is not implementing the capability to set individual attributes, using `apply` no longer works (`apply` loops over all given attributes and tries to set them as fields on the given model). Instead, parameters and overrides are computed first and then used to initialize a new team. Moreover, `TeamRepository` does not have an implementation similar to `save!` and as such can not raise should something go wrong inside of `create_team`. To mimic this behavior, we now call `Kernel.raise` explicitly if the team can not be saved to the database.

With these changes, the team admin functionality works again and is again fully tested.

6.5.3.2 Migrating `package/prediction_ui`

Next up: `package/prediction_ui`. With this change, we now see the new API truly in action. I hope you would agree that it does not look too bad. All calls of `Team` are replaced with the respective analog method on `TeamRepository`.

New PredictionsController using only TeamRepository methods

```

1 class PredictionsController < ApplicationController
2   def new
3     Team.all
4     @teams = TeamRepository.list
5   end
6
7   def create
8     predictor = Predictor.new
9     predictor.learn(Team.all, Game.all)
10    predictor.learn(TeamRepository.list, Game.all)
11    @prediction = predictor.predict(
12      Team.find(params["first_team"]["id"]),
13      Team.find(params["second_team"]["id"])
14      TeamRepository.get(params["first_team"]["id"]),
15      TeamRepository.get(params["second_team"]["id"])
16    end
17  end

```

A first refactoring success can be reported for this change: No changes to tests are needed for PredictionsController.

6.5.3.3 Migrating package/games

The privacy violation on Team in package/games is in the definition of Game. Specifically, the violations were caused in these lines:

Relation to Team causes Game to violated privacy of Team and now no longer works

```

1 belongs_to :first_team, class_name: "Team"
2 belongs_to :second_team, class_name: "Team"

```

Now that Team is in the public API, the violation is gone. However, Game is broken because Team is no longer an ActiveRecord model.

Remember that warning of how difficult it is to remove ActiveRecord from the boundary? Here we have a prime example how *much more difficult* this work gets when the ActiveRecord models in questions have relations. What are we to do here? What can we do?

Since the discussion about this Team/TeamRepository refactoring has been going on for quite a few pages, let's first look at how to make the app work again. Doing so, we get the privacy violation back with a new name. Referencing TeamRecord instead of Team does the trick.

```
1 class Game < ApplicationRecord
2   include HistoricalPerformanceIndicator
3   extend T::Sig
4
5   validates :date, :location, :first_team, :second_team, :winning_team,
6             :first_team_score, :second_team_score, presence: true
7   belongs_to :first_team, class_name: "Team"
8   belongs_to :second_team, class_name: "Team"
9   belongs_to :first_team, class_name: "TeamRecord"
10  belongs_to :second_team, class_name: "TeamRecord"
11 end
```

If we want to remove this privacy violation, we can.... well, we can reread this chapter and find the privacy violation removal technique we like best for this situation. One option we have is to apply the refactoring we just went through to Game also. This is overwhelmingly similar, but comes with two interesting changes:

- Game has a lot more attributes than Team and we would see our API design stretched to its limits.
- Game has a reference to Team and this relationship needs to be transitioned to use Team and TeamRepository.

You can find source code Sportsball with this refactoring at <https://github.com/shageman/package-based-rails-applications-book/tree/main/c6s04-2b/sportsball>. You can also find the diff between the code we discussed up to now and what changes come from a Game and GameRepository refactoring: <https://github.com/shageman/package-based-rails-applications/commit/59435913366ae645d319f0758637a08770465096>.



Let me know if you are interested in seeing a discussion of the above refactoring for Game and GameRepository here in this chapter! Please reach out to me on twitter: <https://twitter.com/shageman> or reach me via <https://stephanhagemann.com>.

7. Ruby at Scale

Gradual Modularity is a way of approaching incremental structural work within an application: Taking modularity as a goal and “graduality” as a principle. In this chapter I want to cover a superset of Gradual Modularity: Ruby at Scale. Ruby at Scale is what Gusto calls the space for its open source work in the area of Gradual Modularity and the solutions that are up- and downstream from its concepts.

Ruby at Scale is the set of approaches (and tools) that allows organizations working with Rails in context where one or more aspects are *big*.

Big here is in direct contrast to DDH’s ideas of the one-person framework. There are a couple of dimensions that could be *big*: A large domain, lots of developers, lots of teams, or a large codebase. Of course, these dimensions are typically not independent of each other. In fact, all of these dimensions being large at Gusto is the reason why we have created and now open sourced much of our toolset for this work.

TODO where did DHH write this

If Gradual Modularity is about the philosophical and technical framework to approach modularization gradually, then Ruby at Scale is about the socio-technical framework that leaders can use to create and manage the processes to actually tackle Gradual Modularization in their organization. The technical side, the tools that make it work, of this is the topic of this chapter; The social side, how to design organizational direction and incentives, will be covered in [Chapter 8](#).

To start, let’s tip our hats to the giants who’s shoulders we stand on.

7.1 The giants who’s shoulders we stand on

The list of projects mentioned here is limited to the tools that we actually install when working on Gradual Modularization. This line keeps this list short and prevents this section from becoming a book onto itself. Look back at [Chapter 3](#) for what I see as the most important ancestor from computer science, namely *Gradual Typing*.

The first tool in our list is **Packwerk** (<https://github.com/shopify/packwerk>). It needs no introduction at this point because it has been the main topic of [Chapter 2](#), [Chapter 4](#), [Chapter 5](#), and [Chapter 6](#).

Second in our list is **Sorbet** (<https://sorbet.org>). It was the first project to introduce Gradual Typing to Ruby. In Ruby at Scale we use it for typing protections and envision that it could be used to revamp (and enhance!) packwerk’s internals based on its constant resolution system.

Third on the list is **Danger** (<https://github.com/danger/danger>). Danger is an automatic and generic code conversation starter (my words, not theirs). With danger we can enforce the rules an engineering group might set for itself automatically via CI. For example, danger might tell you that you should link to a user story or that you should use descriptive labels. In Ruby at Scale, we deliver messages about violations, their potential resolution, and CI failures to developers.

Lastly, our fourth entry is **RuboCop** (<https://github.com/rubocop/rubocop>) is a code analyzer (or linter), which can not only check but also fix all sorts of code style issues. In Ruby at Scale we can use its frameworks and some of its rules to implement dimensions

7.2 Gusto's Ruby at Scale work supporting Gradual Modularity

Gusto has a bunch of open-source work published under <https://github.com/gusto>. The team that worked on Gradual Modularization topics over the last years realized that Gusto's contribution to the packwerk ecosystem was not going to be one big gem, but rather a bunch of small tools that together create a opinionated and composable tool chest for modularization work. As such, we decided to separate it from those other repositories and give them their own home, for which we chose <https://github.com/rubyatscale>.

The rest of this chapter is an overview of the tools in that repository and how they build on top of each other.

7.3 Stimpack

Look back at the work needed to move packages out of the app folder. There, we changed autoload paths, view paths, and others. Thus we ensured that with just a few lines of config adjusted, we could add any number of packages into our `/packages` folder structure without further work. I.e., it just worked.

Stimpack (<https://github.com/rubyatscale/stimpack>) is a gem that handles the config changes for you, so all you have to do is create packages! Everything works without any package-specific configuration.

By default, Stimpack will look for your packages in `/packs`. The folder structure underneath will look something like this:

Folder structure as supported by Stimpack

```
1 package.yml # root level pack
2 app/ # Unpackaged code
3   models/
4   ...
5 packs/
6   my_domain/
7     package.yml
8     deprecated_references.yml
9   app/
10     public/ # Recommended location for public API
11       my_domain.rb # creates the primary namespaces
12       my_domain/
13         my_subdomain.rb
14     services/ # Private services
15       my_domain/
16         some_private_class.rb
17     models/ # Private models
18       some_other_non_namespaced_private_model.rb # this works too
19       my_domain/
20         my_private_namespaced_model.rb
21     config/
22       initializers/ # Initializers can live in packs and load as expected
23     controllers/
24     views/
25     lib/
26     tasks/
27     spec/ # With stimpack, specs for a pack live next to the pack
28     public/
29       my_domain_spec.rb
30       my_domain/
31         my_subdomain_spec.rb
32     services/
33       my_domain/
34         some_private_class_spec.rb
35     models/
36       some_other_non_namespaced_private_model_spec.rb
37       my_domain/
38         my_private_namespaced_model_spec.rb
39     factories/ # Stimpack will automatically load pack factories into FactoryBot
40       my_domain/
41         my_private_namespaced_model_factory.rb
```

A couple of things to point out:

- Stimpack basically reuses the standard Rails `app/` folder structure for packages. If you use that too, no changes are needed.
- Spec code moves into the package too. `spec/` lives next to `app/`
- You can move initializers and rake tasks into packs too!
- Stimpack has integrations for factory bot and Rspec, which will work out of the box.

Don't like the `/packs` folder default? You can choose a different folder by setting a different root folder like so:

Configuring Stimpack to use a different folder structure

```
1  # Customize Stimpack's root directory.
2  # Note that this has to be done _before_
3  # the Application class is declared
4  Stimpack.config.root = "components"
5
6  module MyCoolApp
7    class Application < Rails::Application
8      # ...
9    end
10 end
```

7.4 Code Teams

When tens or hundreds of engineers work on a codebase simply understanding who works on what and who owns what parts of a codebase becomes a challenge. To solve for this, we make the concept of a *team* a first-class concept within our applications.

The teams gem (Code Teams: https://github.com/rubyatscale/code_teams) provides the most basic structure for this.

A minimal team config

```
1  name: My Team
```

At its most basic, a team is just a name. Add plugins to teams to actually add functionality. Taken directly from the project's readme, the following code adds github users and organizations as a new configuration key to a team configuration.

Adding a team plugin for github organizations and members

```
1 class MyGithubPlugin < Teams::Plugin
2   extend T::Sig
3   extend T::Helpers
4
5   GithubStruct = Struct.new(:team, :members)
6
7   sig { returns(GithubStruct) }
8   def github
9     raw_github = @team.raw_hash['github'] || {}
10
11     GithubStruct.new(
12       raw_github['team'],
13       raw_github['members'] || []
14     )
15   end
16
17   def member?(user)
18     members = github.members
19     return false unless members
20
21     members.include?(user)
22   end
23
24   sig { override.params(teams: T::Array[Teams::Team]).returns(T::Array[String]) }
25   def self.validation_errors(teams)
26     errors = T.let([], T::Array[String])
27
28     teams.each do |team|
29       errors << missing_key_error_message(team, 'github.team') if self.for(team).git\
30 hub.team.nil?
31     end
32
33     errors
34   end
35 end
```

Now, you add actual github-related info to the team's config file like so:

```
1 name: My Team
2 github:
3   team: '@org/my-team'
4   members:
5     - member1
6     - member2
```

And to get that info back programmatically, you do this:

```
1 team = Teams.find('My Team')
2 MyGithubPlugin.for(team).github
```

If you are still wondering why you might want to get team information within the code running in your application, here are a couple ways in which we use teams:

- At gusto, we have a home-grown feature flag system. To make sure that we understand over time (and long term) who needs and owns which feature flags (there are hundreds), we force that all feature flags are owned by a team.
- When many teams work together in a codebase, stuff happens. Not all of that stuff is created equal. There might be that change of code by someone not on the team that doesn't follow the newly set technical direction of that team. A bummer, but not so bad. What though, if the change breaks subtle behavior that is hard to test. What if to the owning team, the change to a file simply feels too dangerous because changes will have wide ranging implications on larges swaths of the codebase. For any and all of these reasons, we allow teams to protect certain files. We built a plugin that can either require *owning team* PR approval for any changes, or pair approval, or approval on line additions.
- To protect the growth of our gem dependencies - **lots of things to be said about that** - we enforce that every gem dependency and javascript package dependency be owned by a team. You want it? You own it! This ensures that we align the *benefit* of the additional functionality with the *cost* of the dependency within a team (looking out for vulnerability notifications, upgrading versions, handling end-of-life situations).
- With hundreds of engineers you'll likely not ever know everyone and you will know even less who works on what. We added a plugin to teams that adds contact methods for teams to the team config.

This particular feature is so vital that we actually also built a Visual Studio Code Extension that utilizes that information to link you directly into the team's slack channel. We're great at naming, so it is called code-ownership-vscode (<https://github.com/rubyatscale/code-ownership-vscode>)

7.5 ParsePackwerk

ParsePackwerk is an essential, but also very much a low-level utility in our Gusto's packwerk ecosystem.

At the moment, it is the backbone for how the code owners gem resolves package-based code ownership.

Directly from its readme, the following code shows how it exposes the YAML structure of `package.yml` as idiomatic Ruby:

ParsePackwerk exposing `package.yml` contents in Ruby

```

1  # Get all packages
2  # Note that currently, this does not respect configuration in `packwerk.yml`
3  packages = ParsePackwerk.all
4
5  # Get a single package with a given ame
6  package = ParsePackwerk.find('packs/my_pack')
7
8  # Get a structured `deprecated_references.yml` object a single package
9  deprecated_references = ParsePackwerk::DeprecatedReferences.for(package)
10
11 # Count violations of a particular type for a package
12 deprecated_references.violations.count(&:privacy?)
13 deprecated_references.violations.count(&:dependency?)
14
15 # Get the number of files a particular constant is violated in
16 deprecated_references.violations.select { |v| v.class_name == 'SomeConstant' }.sum { \
17   |v| v.files.count }

```

7.6 Code ownership

Code Ownership (https://github.com/rubyatscale/code_ownership) is a gem that creates ownership of ... *code*! It supports doing this in a variety of ways and for a variety of purposes.

At Gusto, we use this, together with the teams gem, to map parts of our codebase to teams. In fact, we force that all code be owned by exactly one team.

Originally, code ownership through this gem was supported via an annotation at the top of a file, like so:

```

1  # @team My Team

```

As we progressed on our modularization journey and added more and more domain-oriented (often aligned with team boundaries!) packages, we realized that this was an excellent development to support with enhanced code ownership support.

Today, you can also specify ownership on a glob and package basis:

- Package-based code ownership uses the `metadata` key that is allow in packwerk's `package.yml`:

```
1 enforce_dependency: true
2 enforce_privacy: true
3 metadata:
4   owner: Team
```

- Glob-based code ownership is a catch-all way of doing multi-file code ownership where packwerk packages are not (yet) available. This configuration is actually done directly in the team configuration yaml file:

```
1 name: My Team
2 owned_globs:
3   - app/services/stuff_belonging_to_my_team/**/*.rb
4   - app/controllers/other_stuff_belonging_to_my_team/**/*.rb
```

Code ownership can be used in a variety of ways.

First off, if you use this and github, you may want to make sure that Ruby at Scale's code ownership configuration is in sync with GitHub's CODEOWNERS file. GitHub, based on this file, can send PR review requests to the folks whose code is affected by a PR.

There are also programmatic ways to use code ownership:

- `for_file` - the code owner to a file
- `for_backtrace` - the code owner to an error!
- `for_class` - a layer over `for_file` this method first resolves the class to its file and then finds the owner.

I am particularly fond of `for_backtrace`, because it gives us a way for error notifications to be routed to the teams that not only care about the error but also have the knowledge and the responsibility to do something about it!

7.7 PackageProtections

Given my level of excitement for this gem, I really should not hide it in the middle of a list of a bunch of other gems! Oh well! So. Officially: I am very excited for this one!

PackageProtections (https://github.com/rubyatscale/package_protections) implements the first steps on the ideas, the journey, laid out in [Chapter 3](#): It is the start to implementing the vision of *Gradual Modularity*.

PackageProtections uses packwerk and Rubocop to implement protections for packages that can be used in CI to allow or disallow new violations based on a team's configuration of a package.

The current list of protections is short and named very clunkily (just like that word, I guess) - you can thank yours truly for these "gems of names:"

- `prevent_this_package_from_violating_its_stated_dependencies`
- `prevent_other_packages_from_using_this_packages_internals`
- `prevent_this_package_from_exposing_an_untyped_api`
- `prevent_this_package_from_creating_other_namespaces`

The first two entries in this list are straight up resurfacings of packwerk’s functionality: Protection privacy and dependency of packages. We do this so as to create a consistent protection interface for all aspects of Gradual Modularity. We hope this will help folks understand that on top of a diverse set of tools, we are creating a consistent language of modularity.

The next two protections are the ones we found to be very impactful and relatively easy to implement. Protection against the exposure of untyped APIs is supported through the Rubocop Sorbet plugin (<https://github.com/Shopify/rubocop-sorbet>), specifically its Sorbet/StrictSigil (https://github.com/Shopify/rubocop-sorbet/blob/main/manual/cops_sorbet.md#sorbetstrictsigil). Protecting against the usage of namespaces outside of what would canonically be derived from the package name pushes the app and where to find its code to be “less surprising” and more like how we have come to expect code organization in gems.

In addition to surfacing a consistent language, package protections also give owners a tiny bit of extra functionality as opposed to using packwerk and Rubocop directly: They can lock down a protection to no longer allow additional violations once all violations have been removed. Specifically, all package protections can be set to the following enforcements: `fail_never`, `fail_on_new`, and `fail_on_any`.

The list of supported protections should ideally be longer. In fact, we have experimented with and successfully implemented the following. They just haven’t made it into the open-source release yet:

- `protect_against_undocumented_apis`: Gradually document your package APIs!
- `protect_other_packages_from_using_this_package_unintentionally`: The inverse protection of `prevent_this_package_from_violating_its_stated_dependencies` and thus `enforce_dependencies`
- `prevent_this_package_from_violating_other_packages_apis`: the inverse of `prevent_other_packages_from_using_this_packages_internals` and thus `enforce_privacy`

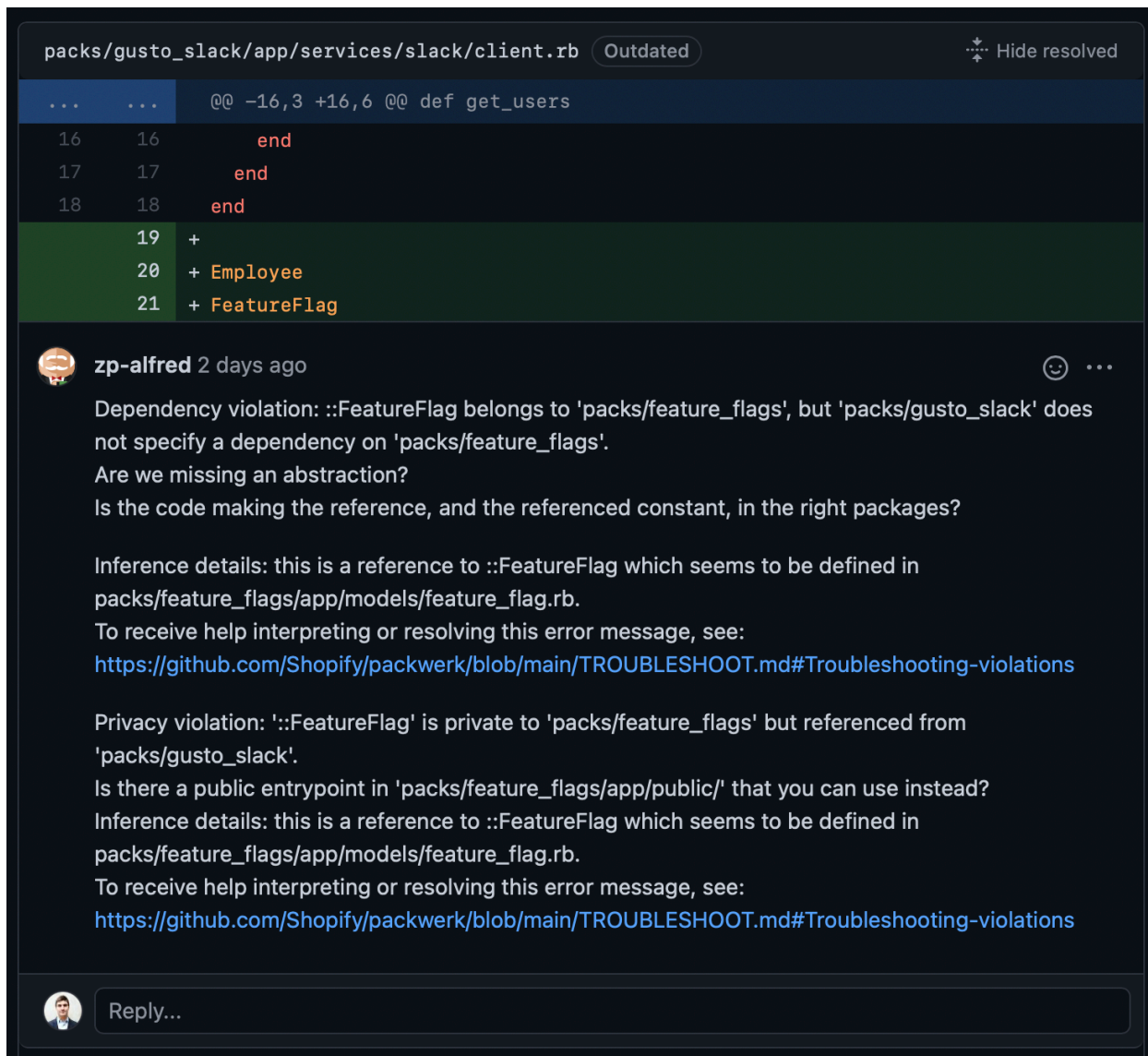
We will continue to work on a plugin system for this gem and hope to add plugins for some or all of the following ideas:

- exclusive database access
- exclusive usage of other datastores (see `rubyatscale-redis` below)
- exclusive usage of 3rd party service configuration
- non-usage of ActiveRecord in the API
- usage of only serializable types in the API

Please reach out to rubyatscale@gusto.com if you are interested in talking about adding these or other plugins!

7.8 Danger::Packwerk

danger-packwerk (<https://github.com/rubyatscale/danger-packwerk>) uses danger with packwerk to add inline comments in pull requests directly on GitHub. It will comment on package boundary violations and suggest alternative courses of action to the committer.



danger-packwerk commenting on a pull request that adds a privacy and a dependency violation

There are various adjustments users can make to how and when danger-packwerk makes comments.

7.9 ModularizationStatistics

ModularizationStatistics (https://github.com/rubyatscale/modularization_statistics) is, if there were a hierarchy of these gems, at the very top of the graph.

We are obviously great at naming, so we usually don't like to say this mouthful of a name. It's ModStats for short.

ModStats offers an opinionated API to expose metrics on the adoption of many aspects of the different parts of Ruby at Scale and Gradual Modularity. We use it to push these metrics to Datadog. Within Datadog we create dashboards that developers use to drill into how their packages are doing an that executives can use to get a high-level understanding of the progress we are making on our modularization journey.

One technical hack we propose to use with this gem is to calculate all metrics with `enforce_privacy` and `enforce_dependencies` set to `true` for all packages. This is to work around the fact that packwerk will only show us violations when these settings are on. If teams adopt these settings at different times, changes in these settings would cause weird jumps in the reported metrics. A rake task to collect and send the metrics with this hack can be set up as follows:

```

1  require 'modularization_statistics'
2
3  namespace(:modularization) do
4    desc(
5      'Publish modularization stats to datadog. ' \
6      'Example: bin/rails "modularization:upload_statistics"'
7    )
8    task(:upload_statistics, [:verbose] => :environment) do |_, args|
9      ignored_paths = Pathname.glob('spec/fixtures/**/*')
10     source_code_pathnames = Pathname.glob('{app,components,lib,packs,spec}/**/*').s\
11     elect(&:file?) - ignored_paths
12
13     # To correctly track violations, we rewrite all `package.yml` files with
14     # `enforce_dependencies` and `enforce_privacy` set to true, then update deprecate\
15     ions.
16     old_packages = ParsePackwerk.all
17     old_packages.each do |package|
18       new_package = ParsePackwerk::Package.new(
19         dependencies: package.dependencies,
20         enforce_dependencies: true,
21         enforce_privacy: true,
22         metadata: package.metadata,
23         name: package.name
24       )

```

```

25     ParsePackwerk.write_package_yaml!(new_package)
26   end
27
28   Packwerk::Cli.new.execute_command(['update-deprecations'])
29
30   # Now we reset it back so that the protection values are the same as the native \
31   packwerk configuration
32   old_packages.each do |package|
33     new_package = ParsePackwerk::Package.new(
34       dependencies: package.dependencies,
35       enforce_dependencies: package.enforce_dependencies,
36       enforce_privacy: package.enforce_privacy,
37       metadata: package.metadata,
38       name: package.name
39     )
40     ParsePackwerk.write_package_yaml!(new_package)
41   end
42
43   ModularizationStatistics.report_to_datadog!(
44     datadog_client: Dogapi::Client.new(ENV.fetch('DATADOG_API_KEY')),
45     app_name: Rails.application.class.module_parent_name,
46     source_code_pathnames: source_code_pathnames,
47     verbose: args[:verbose] == 'true' || false
48   )
49 end
50 end

```

When using ModStats one question that may come up is: “Well, what metrics do you want to see? How do you want to see them move?” With these questions we are coming full circle to the questions raised at the beginning of [Chapter 5](#): How should we think about the quality of our modularization? How should we steer our work?

These are the questions we will discuss in the following chapter: [Chapter 8](#).

7.10 Honorable mentions: More of Gusto’s Ruby at Scale

The RubyAtScale organization does not only contain repos that contribute directly to our Gradual Modularity journey. In this section I briefly mention gems that we haven’t integrated into the bigger story. Yet!

First, **subbundle** (<https://github.com/rubyatscale/subbundle>) basically gives you a way to use

bundler groups without bundler groups. Why? To not clutter up the gemfile with extraneous groups *and* to make app commands faster by not loading any gems unnecessary for the current command:

```
1 # Load the subbundle binstub first.
2 load File.expand_path("subbundle", __dir__)
3 # Setup the subbundle.
4 Subbundle.setup(*%w(
5   rubocop
6   rubocop-performance
7 )
8
9 # Execute the binary.
10 load Gem.bin_path("rubocop", "rubocop")
```

This will really show a positive impact on command performance when you have hundreds of gems.

Second is **rubyatscale-redis** (<https://github.com/rubyatscale/rubyatscale-redis>). It creates the multi-database-support equivalent but for redis! This is one of those gems that I very much hope will get a package protection so that packages can gradually enforce exclusive access to “their redis.”

7.11 The Structure of Ruby at Scale

The various gems in <https://github.com/rubyatscale> build on each other in various ways. The order of discussion in this chapter followed a roughly bottoms-up path through these dependencies. In considering adopting the gems of Ruby at Scale, it makes sense to follow a similar order.



Dependency structure among Ruby at Scale gems

8. Measuring Modularization Progress

How can we know whether we are making progress in our modularization efforts? Can we somehow measure progress? These are the questions we will tackle in this chapter. Once we have a measurement framework in place, we will tackle in the next chapter how to move the needle on these metrics and the goals behind them.

Let me start our analysis of measuring modularization progress with a word of caution from the early statistician Udny Yule:

Measurement does not necessarily mean progress. Failing the possibility of measuring that which you desire, the lust for measurement may, for example, merely result in your measuring something else - and perhaps forgetting the difference - or in your ignoring some things because they cannot be measured.

Or, slightly more succinctly, in our desire for measurement, we frequently measure *what we can* rather than *what we intended* to measure... and forget that there is a difference.

Modularization is a very complex task, typically applied in large and complex codebases. As such, it is not directly measurable. I believe that there is not just one metric that can tell us how well we have modularized or where we are in the modularization process.

In the complexity of the task, however, lies an opportunity to get to some level of understanding how modularization is progressing. There are so many aspects to the work and so many tools involved that from those, we can extract a lot of data points and trends of time. While that collection still needs to heed the above warning, it can give us a high-level overview of what general direction things are going in.

On top of this, the saying “you can’t fix what you won’t admit is broken” gives us a hint of a two-step process that all modularization aspects must go through to improve: *Admit* that we want to improve them, and actually putting the work in to make them better - to *fix* them. With packwerk we get `enforce_privacy` and `enforce_dependency` and with package protections we get `fail_on_new` and `fail_on_any` for a growing number of modularization aspects. That is the “admitting” part. In both cases, then there are the todo lists. Reducing their size is the “fixing” part. We can measure both!

With this in mind, we can create a hierarchy of measures that give us a decent picture of how modularization work is progressing on a technical level. After building out the list of technical measures, we will take a look at a broader set of measures we want to keep an eye out for to make sure progress is in fact good for our organization.

8.1 Basic technical measures of modularization progress

For the purposes of this section, technical measures are *observations we can make about modularization within the codebase* of the application we are looking at. These measures are precise which makes them powerful and deliberately ignore broader issues which makes them weak. Let's keep that in mind as we build out the list.

The *modularization statistics gem* (ModStats) described in [Section 7.9](#) and available at https://github.com/bigrails/modularization_statistics emits the metrics discussed in this chapter. Add it to your application and also check out the proposed dashboard configurations proposed in https://github.com/bigrails/modularization_statistics/#modularization-executive-summary to get a quick start to your modularization progress analysis.

8.1.1 How modularized is the app?

Before adopting a modularization framework like packwerk, we have no insight into in-app boundaries and our adherence to them.



Folks will likely disagree on this. E.g., if you use Ruby's `private` for classes and create a structure of access paths into namespaces based on that, then you could create measures on top of that to get these insights. If you use gems and engines, you can say something about the boundaries of those components.

If you have those mechanisms in place, consider exploring whether a modularization metric based on them could make sense.

When we start using packwerk, we can ask the question **how much of our application is modularized?** A more precise way of asking this is “How much of the application is under a modularization regime?” I.e., for what portion of the application can our modularization framework give us feedback as to the quality of our modularization?

You may recall from [Chapter 2](#) that packwerk automatically puts a `package.yml` file in the root of the application. As such, we might be inclined to say that it is 100% whenever we adopt packwerk. Using the *admit* then *fix* idea, I argue that we should not count the root package, because while we might call ourselves modularization-curious after having installed packwerk, we haven't admitted to having any meaningful packages and thus any meaningful boundaries.

My proposal from [Section 2.5](#) to move all packages out of the `app/` folder and into a dedicated `packages/` folder mitigates that: In doing so, engineers make a conscious decision (i.e., *admit*) that they want certain packages to exist and to have designed boundaries in our application. It is really these decisions that we want to track.

ModStats emits these metric on a per-file basis in two ways: `by_team` and `totals`. Specifically it emits the following metrics:

- `component_files.by_team`
- `component_files.totals`
- `packaged_files.by_team`
- `packaged_files.totals`
- `all_files.by_team`
- `all_files.totals`

In addition to packaged-based file counts, ModStats emits them for components and for all files as well.

The number I propose to track over time is the fraction of componentized and packagified code. I.e.,

$$\text{modularization_percent} = \frac{\text{component_files.totals} + \text{packaged_files.totals}}{\text{all_files.totals}} * 100$$

Even the largest codebases should be able to quickly drive up `modularization_percent` once there is buy-in to do the work. This is because the work needed to improve this metric is purely mechanical and does not require behavioral refactorings (it also doesn't require many, if any, changes to the contents of Ruby files).

The `*.by_team` metrics can be used by leaders or teams to drill into where there are parts of the org that might need help with this work. E.g., sorting creating the above formula with `by_team` instead of `totals` and sorting the results for all teams from lowest to highest, highlights the teams that have gone the least for on the first step of their modularization journey.

8.1.2 Are there an adequate amount of packages?

Given the discussion from the previous section, there is one quick way to achieve 100% of `modularization_percent` and also *to not improve anything* for anyone working in this application: `mv app packages/app` - simply move everything into one package. You will only have to handle a bunch of references to absolute filepaths, which tend to be easy to adjust, and you are done!

It is this option that pushes us to think about the next measure of modularization progress: If one package doesn't do it, how many do we need? **What is an adequate amount of packages?** If the previous measure was about *visibility*, then this measure is about *potential impact*. The more packages we have, the more potential for deprecated references there is, the more there is a potential reshaping of package relationships and thus code patterns.

The minimum number of package is an application is, as already discussed, one. The maximum number of packages is infinite if we didn't care whether packages are empty. If we do, the maximum number of packages is the number of Ruby files in the application. A package per file! Since we are trying to encapsulate domain concepts (which often span multiple classes) into packages, this would mean that we are either not doing that and we'd effectively have class-level-packages or we make a class contain an entire domain concept, likely making them much larger than what we're typically comfortable with. This is why the guiding question for this section is what the *adequate amount* of packages is.

The answer being somewhere between *one package* and *as many packages as there are classes* is not very helpful. Given our discussion of package and modularization thus far, I hope it doesn't feel like a leap to suggest that the number of packages should be related to the number of *domains* (I would quite like to add the qualifier "meaningful", as in "meaningful domains"): How many meaningful domains does the application include? Aim for that number as the number of packages! It doesn't feel easy to determine the number of domains (it's is one entangled monolith after all!)... let's create some more ideas for proxy parameters to use.

How many teams do you have working on the monolith? Typically teams get formed around some aspect (domain?) of the business. If you have predominantly market-oriented engineering teams (or what is called *stream-aligned team* in Team Topologies <https://teamtopologies.com/key-concepts>), then it stands to reason that they were created to support meaningful parts of the business domain. So, why not shoot for this number?

There is a more abstract variant of the number of teams that I think is very interesting too - and maybe a bit more aspirational: *How many teams would you have if you divided your engineering headcount by 5 or 6?* Why 5 or 6? Because I am not sure what the exact number of people is that gets fed by two pizzas. Please don't write me off yet ;) Jeff Bezos' famous *Two Pizza team* mandate (<https://www.theguardian.com/technology/2018/apr/24/the-two-pizza-rule-and-the-secret-of-amazons-success>) stemmed from an insight into team productivity, namely that, while there are many factors that determine team productivity, there is a sweet spot for team size that isn't too small or too large. And the most common answers to the question of what size is ideal for a team is 5 or 6 (<https://www.totalteambuilding.com.au/ideal-team-size/>). This is aspirational because in order for teams of this size to be effective they have to be able to operate somewhat independently (which in a ball of mud application they likely can't), but which they should be able to once the application has a sensible structure.

As a broad guideline and a starting point for a discussion, I hope that the above ideas are helpful for you to come up with a sense of what might be a good ballpark to be in in terms of number of packages. That said, there are likely many factors that would move the sensible number. I have two critiques of the ideas above myself: I will call them *packages all the way down* and *vertical vs horizontal splits*.

How should we judge if a team decides to split their "meaningful" domain into multiple packages? Here is an example I have seen: A team was responsible for creating third-party app integrations. They had built a plugin system that connect these services with varying capabilities and could relatively easily add services and did so frequently for a while. They decided they wanted to turn every integration into a package. There were hundreds of them. If a team wants to experiment with such a practice, should a vague odea for adequate modularization stop them? I hope not. For the example in question, I proposed to make the actual integrations sub-packages of the main integration package (i.e., to move them into a subfolder of the integrations package) - effectively visually hiding the details and number of integrations from anyone looking at the integrations package from the outside. For this use-case, it would be nice to exclude the sub-packages from the overall package count (or to at least have that option).

Teams, whether two-pizza or differently sized, work more independently - and faster - the more

they control the entire “vertical” of software and thus value delivery. When we work to extract team domains out of an entangled codebase, we are commonly going to find that teams do not control a vertical, they own a “subentanglement.” As modularization work improves the structure of the application we are going to find a lot of subdomains that are depended upon by many parts of the system. In the long run we want to change the relationships to some of these dependencies with one of the refactorings discussed in [Chapter 5](#). However, whatever we do, splitting these common dependencies into separate packages is a likely way in which multiple teams can work out who owns what and how shared dependencies might be unentangled. I.e., shared dependencies lead to more packages.

ModStats emits a metric called `all_packages.count`. That is simply the number of packages in the application. It does not emit any sort of attainment metric as the above discussion might imply we should have. We decided that there were simply too many variables to the number of packages to create an opinionated metric about their “goodness.”

8.1.3 How much are packages owned by only one team?

One of the last points of the previous section hits on the next aspect of modularization that we should want to keep an eye out for: *shared domains* and the corollary we can directly observe in our application: Packages that multiple teams have a legitimate interest in working in.

Let’s prove the problem with another (ridiculous) example: One way to achieve the previous two metrics would be to ...

1. find out how many packages the application should have (It’s obvious! Number of developers divided by 5 gets you a number, X , of packages we should have),
2. to list all Ruby files in `app/`,
3. partition them into X groups (this Ruby snippet will do it `ruby_files.group_by.with_index{|_, i| i % X}.values`),
4. for each group... Make a package!

We have no idea in which order `ruby_files` is going to contain our files. On top of that, the Ruby snippet acts kind of like a shuffle. So, what is the outcome of this way of making packages? There is going to be no alignment between domains and packages. Files and their concepts are going to be all over the place. Making a change in this new codebase is going to be a bit like playing hide and seek. And this game is played by every engineer from every team in every one of these packages. This all makes no sense.

One way in which we can directly observe that the above packaging algorithm makes no sense is if we use `Teams` ([Section 7.4](#)) and `CodeOwnership` ([Section 7.6](#)). With these tools, if we check what teams have files in each package, we will find that *every* team has some files in *every* package.

The opposite situation occurs when teams own meaningful parts of a domain and deliberately split up packages based on these domains. Then, most packages will have a single owner of all

files only. CodeOwnership supports this through package-based code ownership - teams owning an entire package can remove many file-based code ownership annotations and replace it with a single annotation of metadata -> owner in package.yml.

ModStats emits a metric called `all_packages.package_based_file_ownership.count`, which reports the number of packages that employ package-based ownership. We can calculate a composite metric `package_based_ownership_percent` to capture the percentage of package-based ownership used for packages as follows:

$$\text{package_based_ownership_percent} = \frac{\text{all_packages.package_based_file_ownership.count}}{\text{all_packages.count}} * 100$$

Based on the above discussion we can conclude: the closer to 100% the better for this metric.

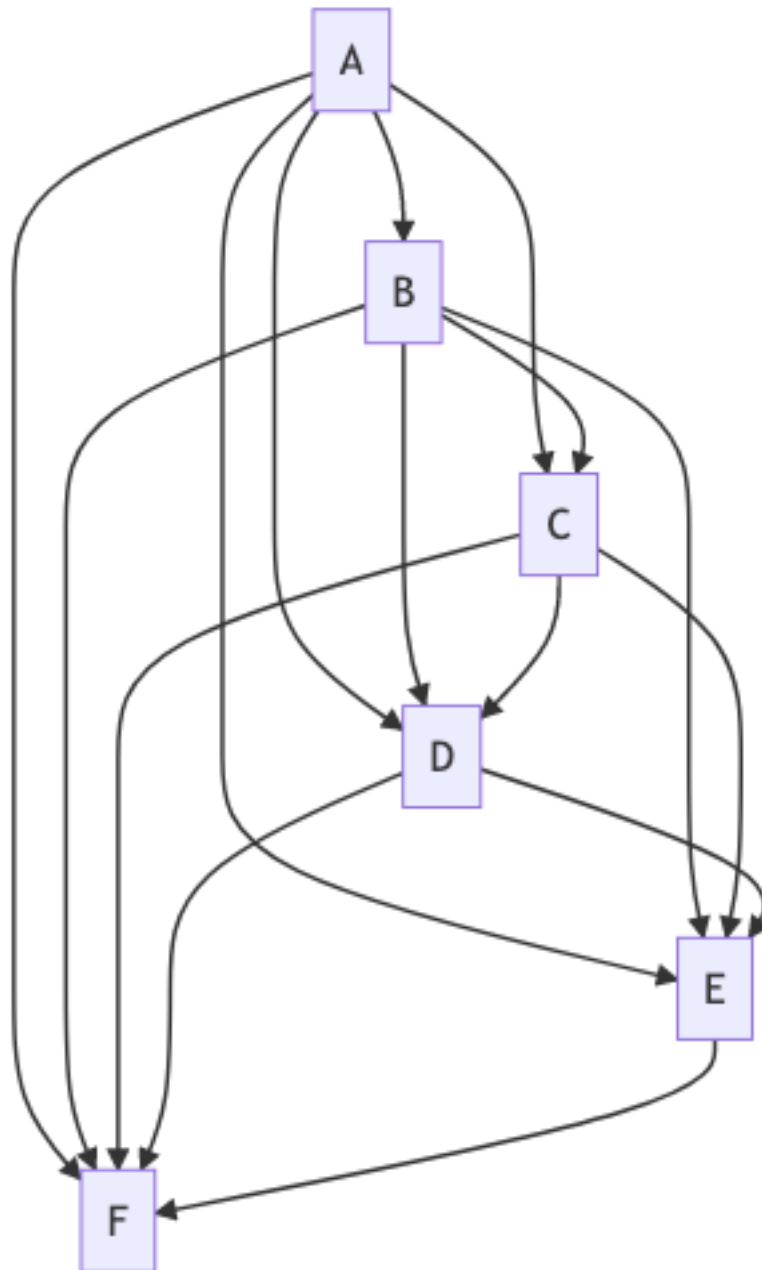
8.1.4 How discerning is the package dependency graph?

With the measures discussed so far we can monitor that an app is modularized with an appropriate amount of packages that meaningfully separated so they can be owned by individual teams. If you think of a modularized app as a graph with nodes and edges, one could say that we have ways to ensure that no parts of the application are outside of the nodes (packages), that there is a good number of numbers, and that each node can be assigned to a team. We haven't said anything about the connections between nodes yet, the dependencies between packages. Are there judgments we can render based on how their properties?

Let's take two abstract graphs with different number of edges, imagine they are package relationships, and discuss their quality.



Dependency graph without edges



Dependency graph with all the edges

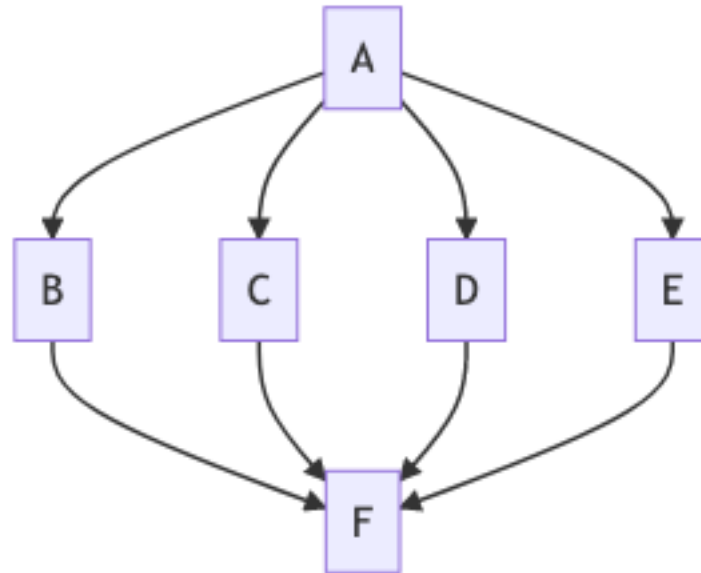
One way to interpret the graph without any edges is a perfectly modularized application. In fact, may “application” isn’t the right term any more as the nodes are completely independent of each other. There is no connection whatsoever.

By contrast, the second graph has all *possible* edges in a directed acyclic graph. Every package in this system depends on all other packages it can depend on without making the dependency structure illegal. I.e., any more edges and packwerk would complain about there being a cyclic dependency

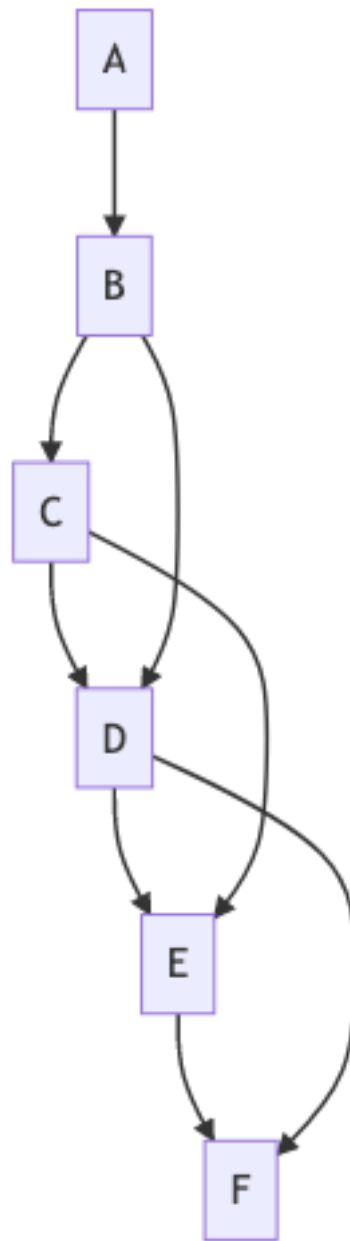
within the package dependency graph.

Both graphs are theoretical edge cases and as such not realistic. That said, the first situation - with far less edges / dependencies - is vastly preferable to its alternative. Why? Because we can reason about more parts of the application in isolation.

Let's take another angle at this analysis and compare two graphs with the same number of edges creating different structures.



Flat dependency graph



Tall dependency graph

The first graph shows a flat dependency structure, the second one a tall one. In this case we can no longer say that “we can reason about more parts of the application in isolation.” What we can say, however, is that the amount of dependencies we have to include in an analysis is lower for the flat graph. For example, B, C, D, and E need to only take into account their dependency on F. Compare that to the situation in the tall graph, where the same is true, but there are a plethora of additional dependencies that all nodes expect F have to consider: For example, C has to reason about

its dependencies on D, E, and F.

We can conclude from the above points that, all other things being equal, we *want our dependency graph to be flatter and less densely connected*.

The elephant in the room is that the analysis so far has only considered acyclic graphs - as if it weren't packwerk's superpower to describe the *desired* dependency graph while we are violating intended dependencies all over the place. Meaning, most of the time we will be dealing with *cyclic* dependency graphs as soon as we look at actual dependencies in the application.

How should we evaluate the graph of actual dependencies? It is noteworthy that structural analysis of cyclic graphs is still an ongoing field of research. For example *Cyclicity of graphs*, Hammack, 1999 ([https://onlinelibrary.wiley.com/doi/abs/10.1002/\(SICI\)1097-0118\(199910\)32:2%3C160::AID-JGT6%3E3.0.CO;2-U](https://onlinelibrary.wiley.com/doi/abs/10.1002/(SICI)1097-0118(199910)32:2%3C160::AID-JGT6%3E3.0.CO;2-U)), *Graph Cyclicity, Excess Conductance, and Resistance Deficit*, Klein & Ivanciuc, 2001 (<https://link.springer.com/article/10.1023/A:1015119609980>), or *On a new cyclicity measure of graphs—The global cyclicity index*, Yang, 2014 (<https://www.sciencedirect.com/science/article/pii/S0166218X14001309>).

All other things being equal, we want to have *less cyclic dependencies* and *less non-desired dependencies* in our application.

ModStats exposes a couple of metrics related to this discussion, most notably:

- `all_packages.dependencies.count`
- `all_packages.dependency_violations.count`

8.1.5 Basic technical measures create a path

The four measures we have discussed up to this point form a progression:

- How modularized is the app?
- Are there an adequate amount of packages?
- How much are packages owned by only one team?
- How discerning is the package dependency graph?

Starting at the top, we can focus on the improvement of the next measure. As we get closer to fully attaining a measure, we will see the rate of positive impact diminishing. At that point, turning our gaze towards the next metric in the list can help us to reignite positive impact of the work and energy for the work.

Any metric used alone taken out of context and taken too seriously would jeopardize the overall quality of the application. Taken all together they push us towards an application that is: Completely modularized with a meaningful and appropriate number of packages that teams own and thus care about whose app-level structure tends towards being flatter and wider.

In short, we can say that attaining each metric on their own is a *necessary* precondition for a well-structured app. Attaining all of them together is *sufficient* to see a well-structured application.

8.2 Refining technical measures of modularization progress

We're turning now towards another list of measures that builds on top of what we have so far. We'll see how attaining any one of those metrics *more* improves the quality of our modularization analysis. The below measures are additive. That is, improve any one of them and some aspect of your app's modularization analysis is richer, more detailed, or better capturing the nuances of the modularization situation.

One consequence of these measures contributing to application modularization quality in an additive way is that you can pick any place to start. This is great to get people moving on a project like disentangling a ball of mud: "Here are ten things you can do now to make it better!" I.e., any person in the org who expressed any interest in work that connects to any one of these metrics, support their work and the app will be better off. The downside of the additivity feature comes into play when folks are not totally bought in, not totally sure about where they want to invest next: They face option overload. I have received feedback regularly during my work on these metrics and making apps better using them that I should be more directive about what the path is we should follow.

So, while the below metrics don't create *one path*. They do create an *options space*. If you are in the position to shape an engineering organization's direction this means that you potentially need to make some calls on priority. The good thing is that you can't really pick wrong because all choices can work! However, your choice can also always be questioned because all other choices kinda work too. We'll dive more into how to create a successful modularization program for large organizations in [Chapter 9](#).

8.2.1 How many packages protect themselves from breaking their own stated dependencies?

Distinguishing actual from intended or accepted dependencies is a fundamental feature of packwerk. The measure we can build on top of it is to ask how many of the packages in an application have `enforce_dependencies: true` set in their configuration. It gives us a sense of how many and which teams care about dependency violations. The more, the better.

In ModStats ([Section 7.9](#)) we have chosen to not only base this metric on a count of `enforce_dependencies` entries. This is to support its use in conjunction with the PackageProtections gem ([Section 7.7](#)) and its configuration value `prevent_this_package_from_violating_its_stated_dependencies`.

The reason for this is largely technical, because it is a quirk of the implementation of PackageProtections that it relies on packwerk producing violation lists in `deprecated_references.yml`. Packwerk will, however, only emit these lists if its enforcements are turned on. Because of this, I recommend turning `enforce_dependencies` (and, for the following metric `enforce_privacy`) on for all packages so data can be reported. If the enforcements are turned on for all packages no matter what they

choose to do in practice, they don't actually tell us whether a team has chosen to heed the violation list for a given package.

There is one more, less important, reason to use the `PackageProtection prevent_this_package_from_violating_its_stated_dependencies` and that is that it has three configuration options instead of just two. Packwerk's enforcements can be set to `true` or `false`. `PackageProtections` has three settings: `fail_never`, `fail_on_new`, and `fail_on_any`. The first two settings have the same meaning as packwerk's settings. `fail_on_any` adds the capability for a team to "lock down" a completely attained metric more strongly. I.e., without it, a new violation can be added simply by re-adding it to `deprecated_references.yml`. With it, that is no longer allowed and will fail CI.

ModStats emits the following metrics regarding stated dependencies (see this Code comment (https://github.com/rubyatscale/modularization_statistics/blob/fb227e36d1b475462889fab7e829d933de8434f4/lib/modularization_statistics/private/datadog_reporter.rb#L264-L268) as to why that's not exactly true today):

- `all_packages.enforcing_dependencies.count`
- `all_packages.prevent_this_package_from_violating_its_stated_dependencies.fail_never`
- `all_packages.prevent_this_package_from_violating_its_stated_dependencies.fail_on_new`
- `all_packages.prevent_this_package_from_violating_its_stated_dependencies.fail_on_any`
- `by_team.prevent_this_package_from_violating_its_stated_dependencies.fail_never`
- `by_team.prevent_this_package_from_violating_its_stated_dependencies.fail_on_new`
- `by_team.prevent_this_package_from_violating_its_stated_dependencies.fail_on_any`

How many packages protect themselves from being used unintentionally?

If we think closely about the meaning of `enforce_dependencies`, we realize that this enforcement is *directional*. This configuration and `prevent_this_package_from_violating_its_stated_dependencies` (which is built on `enforce_dependencies`) both are concerned with *a package* and the dependencies *it has*. If we tried to capture this feature in the name, it would be appropriate for `enforce_dependencies` to be called `enforce_outgoing_dependencies`. The name of the package protection already reflects this directionality.

Does the reverse make sense? I.e., does `enforce_incoming_dependencies` make sense as a separate protection? Where would such a setting be useful?

What this protection would allow a package to do is to state that it rejects being used by other packages with them explicitly stating so. This can indeed be an interesting setting for packages to make and it is very possible, especially for net-new code in totally new packages. A newly added package (with new code) won't have any usages in the rest of the system, so this setting could be turned on trivially as soon as that package gets created. The practical upshot of this would be that,

if set to `fail_on_any` no package is allowed to add the given package as a dependency. Remember that packwerk doesn't allow for accepted dependencies to form a *cyclic* graph. As such, this setting can be used by packages to force that they are not involved in (or at least control) entangled package dependencies.

Before PackageProtections were open sourced there was a protection called `protect_this_package_from_being_used_unintentionally` for a while. We removed it for one main reason: As soon as a majority of packages has `prevent_this_package_from_violating_its_stated_dependencies` set, the two protections do the same thing and violations gets reported twice.

So, while this is not currently a protection you can enforce with PackageProtections or measure with ModStats, please reach out to me or the maintainers of github.com/rubyatscale if you are interested in getting support back to support your organization's modularization journey.

8.2.2 How many packages protect themselves against privacy violations?

The discussion for `enforce_privacy` and `prevent_other_packages_from_using_this_packages_internals` can be a lot shorter than the previous section, because all the reasoning is exactly the same.

ModStats emits an analogous set of metrics for privacy protections:

- `all_packages.enforcing_privacy.count`
- `all_packages.prevent_other_packages_from_using_this_packages_internals.fail_never`
- `all_packages.prevent_other_packages_from_using_this_packages_internals.fail_on_new`
- `all_packages.prevent_other_packages_from_using_this_packages_internals.fail_on_any`
- `by_team.prevent_other_packages_from_using_this_packages_internals.fail_never`
- `by_team.prevent_other_packages_from_using_this_packages_internals.fail_on_new`
- `by_team.prevent_other_packages_from_using_this_packages_internals.fail_on_any`

How many packages protect themselves from misusing other packages' privacy?

Just like `enforce_dependencies` has directionality, so does `enforce_privacy`. The name we chose for the package protection gives you a clear picture of what that direction is: `prevent_other_packages_from_using_this_packages_internals`. I.e., if we were to make the name for the enforcement more explicit, it would be `enforce_incoming_privacy`.

Similar to what was discussed in the previous section, `prevent_this_package_from_using_other_`

`packages_internals` is not currently available in `PackageProtections`. Again, please reach out if this protection would be helpful in your situation.

8.2.3 How much are packages operating in their own namespace?

`PackageProtections` defines a protection called `prevent_this_package_from_creating_other_namespaces` and `ModStats` exposes a host of measures based on it:

- `all_packages.prevent_this_package_from_creating_other_namespaces.fail_never`
- `all_packages.prevent_this_package_from_creating_other_namespaces.fail_on_new`
- `all_packages.prevent_this_package_from_creating_other_namespaces.fail_on_any`
- `by_team.prevent_this_package_from_creating_other_namespaces.fail_never`
- `by_team.prevent_this_package_from_creating_other_namespaces.fail_on_new`
- `by_team.prevent_this_package_from_creating_other_namespaces.fail_on_any`

This protection allows a package to “claim” its namespace. That is, if a package is called `important_domain`, then this protection will create violations for any constants created within the package that are *not* namespaced under `ImportantDomain`. The effect of this is that the package creates less constants that are surprising in the package context. There will be less constants adding to the root namespace `::` (or none at all). There will also be less constants adding to other namespaces - say, namespaces that one would reasonably expect to be defined in a different package (or none at all).

Another effect is that it makes the package’s behavior more similar to that of how gems commonly behave. Gems, when generated with `bundle`’s scaffolding mechanism are set up to nudge the author to add all the gem’s constants in the namespace derived from the gem’s name. This protection pushes us towards the same behavior for packages.

Imagine an application where this protection is turned on for all packages. All constant locations would be deterministic based on the package names and unsurprising to engineers. An interesting side effect of that would also be that there wouldn’t be any of the typical “god classes”, such as `User`, because no constants in the app would be directly defined in the root namespace. So, there might be `ImportantDomain::User` and `OtherDomain::User`, but not `::User`. When adding functionality to a user, imagine the interesting discussion: “Which *user* does this functionality belong to?”

8.2.4 How many packages protect themselves against untyped APIs?

The README of `packwerk` has this entry in its list of limitations - the emphasis is mine (<https://github.com/Shopify/packwerk/blob/aa563c5a48efdb75617744ce8df35125cc92eb0c/README.md>):

Method calls and objects passed around the application are completely ignored. Packwerk only cares about static constant references. That said, if you want Packwerk to analyze parameters of a method, **you can use Sorbet to define a type signature. Sorbet signatures are pure Ruby code and use constants to express types, and Packwerk understands that.**

We already observed this fact and worked to add Sorbet in [Section 5.8](#) to ensure that all dependencies of Predictor became visible. Stated generally, the more APIs within the application are strictly typed, the more “correct” or encompassing packwerk’s dependency and privacy analysis will be.

PackageProtections defines a protection `prevent_this_package_from_exposing_an_untyped_api` to codify this behavior. ModStats emits the following metrics to track adoption of the protection.

- `all_packages.prevent_this_package_from_exposing_an_untyped_api.fail_never`
- `all_packages.prevent_this_package_from_exposing_an_untyped_api.fail_on_new`
- `all_packages.prevent_this_package_from_exposing_an_untyped_api.fail_on_any`
- `by_team.prevent_this_package_from_exposing_an_untyped_api.fail_never`
- `by_team.prevent_this_package_from_exposing_an_untyped_api.fail_on_new`
- `by_team.prevent_this_package_from_exposing_an_untyped_api.fail_on_any`

8.3 Growing the list of technical measures your organization can support

Looking over the list of technical measure from the previous section the question of “why these?” should come to mind. Or maybe “why only these?”

To answer this question it makes sense to take a step back and look at what we have been doing here.

Packwerk introduced us to `enforce_privacy` and `enforce_dependencies`. Two settings that allow engineers at the *team level* (working within a large application) to make choices about when and where to invest in increased modularization. In [Chapter 3](#) we discussed ideas that build on top of what packwerk introduces us to: Additional ways to use the dependency graph within an application and ways to enrich the analysis of boundaries and dependencies within an application.

In building on top of these ideas and creating the tools in github.com/rubyatscale, specifically PackageProtections (https://github.com/rubyatscale/package_protections), decisions were made on an *organizational level* as to aspects of Gradual Modularity are relevant.

At the organizational level we decide *what standards* we want to follow. In your org this may be decided by the CTO, the VP Eng, the lead architect, the architecture council, an architecture decision records process (ADRs), or grass-roots. You might do this as part of your “north star,” your “golden paths,” a “technical strategy,” or you may not have a fancy name for it. There are degrees to this:

We say things like, “must do,” “should do,” or “recommended to do.” We also decide on when things should be done: as a dedicated effort with a completion date, as we touch code, or ... gradually.

Gradual Modularization creates a framework for designing these decisions.

On the team level, we prioritize the work stemming from decisions on the organizational level based on the constraints that are given. Well, sometimes, we first negotiate to change the decisions, because as soon as those get in contact with reality we find that we need to refine and revisit aspects. After that, however, gradual modularization shines in situations where adoption has time to develop. And where the standards are complementary as much as possible and can be adopted in different orders, at different times, and to different degrees.

The standards about package APIs is really what PackageProtections is about. In order for your organization to be able to make decisions about additional protections you need to be able to... well, *add* them.

8.3.1 Adding your own protections



This section is incomplete. While it is currently possible to add custom protections to the packs ecosystem, custom metrics are still a work in progress.

What protections might you want to add? Looking back at [Chapter 3](#), my list includes things like

- expose only documented APIs
- ensure the exclusivity of the DB connection for a package
- prevent non-serializable types in APIs
- ensure a shape for services in APIs (e.g., only class methods, or only one method called per form per service class)

Over time, I expect for some of these to be implemented in a couple of organizations and then for those decisions to get upstreamed into PackageProtections. This will create a third level of decision making: *industry level*. While that process is outside of the scope of this Chapter, we need to talk about how to get started, i.e., how to add custom protections to PackageProtections.

It was recently made easier to add custom protections to PackageProtections (for example through commit https://github.com/rubyatscale/package_protections/commit/0280038817d06f0d7aa1ecb3e33c92dc5ec02de3).

```

1  module RuboCop
2    module Cop
3      module PackageProtections
4        class RequireReadme < Style::DocumentationMethod
5          extend T::Sig
6
7          IDENTIFIER = 'ensure_readme_exists'.freeze
8
9          include ::PackageProtections::RubocopProtectionInterface
10
11         def identifier
12           IDENTIFIER
13         end
14
15         def included_globs_for_pack
16           [
17             'app/public/*',
18           ]
19         end
20
21         def unmet_preconditions_for_behavior(behavior, package)
22           nil
23         end
24
25         sig do
26           override.params(file: String).returns(String)
27         end
28         def message_for_fail_on_any(file)
29           "`#{file}` must contain documentation on every method (between signature a\
30 nd method)"
31         end
32
33         sig { override.returns(String) }
34         def cop_name
35           'PackageProtections/RequireDocumentatedPublicApis'
36         end
37
38         sig { override.returns(String) }
39         def humanized_protection_name
40           'Documentated Public APIs'
41         end
42
43         sig { override.returns(String) }

```

```

44     def humanized_protection_description
45       <<~MESSAGE
46         All public API must have a documentation comment (between the signature \
47 and method).
48         This is failing because these files are in `.rubocop_todo.yml` under `{ \
49 cop_name}`.
50         If you want to be able to ignore these files, you'll need to open the fi\
51 le's package's `package.yml` file and
52         change `{IDENTIFIER}` to `{::PackageProtections::ViolationBehavior::Fa\
53 ilOnNew.serialize}`
54       MESSAGE
55     end
56   end
57 end
58 end
59 end

```

Adding a custom protection to PackageProtections

```

1 def add_custom_package_protections
2   PackageProtections.configure do |config|
3     config.protections += [
4       RuboCop::Cop::PackageProtections::RequireReadme.new,
5     ]
6   end
7 end
8
9 add_custom_package_protections

```

TODO Need to add custom metric emission once it becomes available

8.4 Sociotechnical measures that go beyond modularization

Modularization is a means to an end. That *end* is improved maintainability of large applications and increased productivity of the organization working on them. All the metrics discussed so far in this Chapter, capture only some aspects of the technical side of this larger view. To measure this more encompassing view of our progress we will have to look for a more encompassing way to evaluate our work. In such a broader view, we will see modularization work and impact less directly, and it will at times feel like we are not seeing them at all. Despite this, it makes sense to invest in this kind of measuring, because it ensures that we can see (measure!) progress at the truly relevant level of analysis and level of impact.

I am going to focus this section on the technical side of the “sociotechnical” spectrum (not sure that is a thing, but if not, here we go!). I want to only briefly add here that on the “socio” end of that spectrum we can analyze our performance and research our progress using various sociological research methods. I have specifically seen survey, structured interviews, and semi-structured interviews be used effectively in engineering organizations.

8.4.1 High-level vs low-level metrics (and tests)

An analogy that I think can work here is the difference in information we get from unit tests versus integration tests. Unit tests give us confidence that a specific, tiny part of the technical system works as expected. Because the test is so close to the technical aspect that it is verifying, it can do so in great detail covering most or all possible execution cases. That said, we only get *some* confidence that the feature will work in production. That is because all sorts of technical things around this feature could be broken or not making use of the tested feature to where it simply doesn’t do in the larger system what we expect it to do.

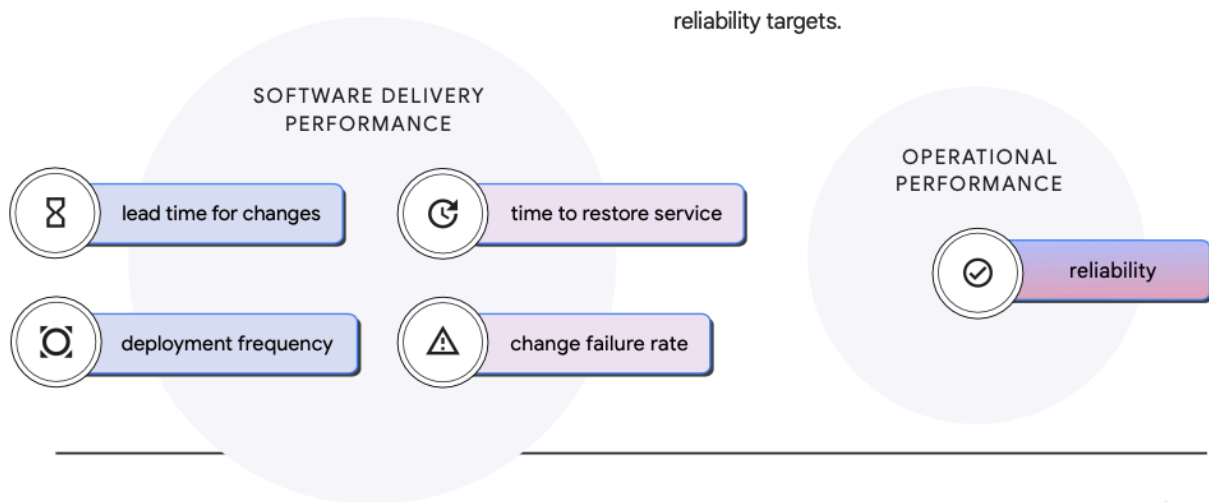
Contrast unit tests with integration tests - and I mean here tests that test the UI, the business layer via a running application that a test interacts with (selenium-based tests are like this). Because integration tests are much slower than unit tests and because they test a much greater surface area, they typically can not test specific feature in the same detail. However, they can confirm that they work in a very fundamental way: “Does it load and not crash?”

Used together, unit and integration tests can give engineers high confidence that they can understand that the system works and how it works. Each testing strategy contributing confidence with what it covers best.

Note that there is room for error even in the combination of testing strategies still: A unit test might verify that a small part A works correctly in all its cases and the integration test verifies that the screen S which we expect A to be used on works. What if we don’t have a test that verifies that A is actually used on S? What if in fact some other unit B that behaves very similar to A is used? This is of course testable in itself, but doesn’t come for free with either of the testing strategies discussed here.

Back to measuring modularization progress! What is our analogy of “integration tests” for the technical modularization aspects?

8.4.2 The DevOps Research and Assessment metrics



How do we compare? | 10

The five DORA metrics of software delivery and operational performance (Source: State of DevOps Report 2021)

The most high level and most research-based metrics I know of are the five metrics of DevOps Research and Assessment (DORA) (see [Figure “DORA metrics”](https://cloud.google.com/blog/products/devops-sre/using-the-four-keys-to-measure-your-devops-performance), <https://cloud.google.com/blog/products/devops-sre/using-the-four-keys-to-measure-your-devops-performance>):

- **Deployment Frequency** — How often an organization successfully releases to production
- **Lead Time for Changes** — The amount of time it takes a commit to get into production
- **Change Failure Rate** — The percentage of deployments causing a failure in production
- **Time to Restore Service** — How long it takes an organization to recover from a failure in production
- **Reliability** - Are availability, latency, performance, and scalability at the levels set by the organizations objectives

The first four metrics are DORA’s original software delivery performance metrics as publicized in *Accelerate* (by Nicole Forsgren, Jez Humble, Gene Kim, 2018, IT Revolution Press - <https://itrevolution.com/product/accelerate/>).

It is my expectation that progress on gradual modularization impacts all four original metrics positively. I expect this because of the complexity reducing property of modularization: By breaking our large system into smaller components we make it more possible, more reasonable to talk about the larger system being *composed using a bunch of smaller systems* built by a bunch of smaller teams. As smaller systems tend to be easier and faster to develop this should have a positive effect. When the composition of the smaller systems is deliberately designed and constrained (as it is via packwerk and its ecosystem’s protections), then that should have a positive effect too.

Take as a specific example *lead time for changes*: Using the package dependency graph to determine that only a subset of packages needs to be tested to verify a pull request merge or production deploy,

i.e., we can get away with running fewer tests. This will be faster whether all tests run in one process or are parallelized. When they are parallelized we have an option of trading speed for cost and vice versa by keeping either keeping parallelization at original levels (faster) or reducing it to where the tests run in the same time (cheaper). If the organization uses continuous deployment this should also directly allow for an increase of *deployment frequency*.

The interaction between modularization and reliability is more complex and needs to be an area of careful exploration and design by every engineer in the organization. As an example take ActiveRecord associations and package dependencies. As an organization works towards greater modularization and debates ways of creating well-designed package dependencies, there comes a point where some associations need to be removed to adhere to package dependencies. The challenge with this is that associations are a very powerful tool that have a lot of performance features built-in under the hood (take just the logic that decides how to create queries for multiple models when querying across associations). If associations are to be removed their functionality has to be implemented differently. And if that is not done carefully, it might inadvertently introduce N+1 queries and degrade performance sharply. A crucial component of preventing these kinds of problems is to design the API of packages to include needed bulk access methods - then negative implications can be avoided.

At the same time this kind of change can create positive performance impacts. The removal of an association can be the catalyst for the redesign of the relationship of the two packages. There are cases where the direct use of the association can feasibly be replaced by the precalculation of the query result. Precalculating the result leads to denormalized storage, and thus more management overhead to keep data consistent, but it will also make the query much faster by removing the need for the usage of an association altogether.

The challenge in connecting DORA metrics to modularization progress is similar to the problem of connecting high-level integration tests to low-level unit tests discussed in [Section TODO](#): The various tests (and now the various metrics) might be unrelated - they might change for different, disconnected reasons.

8.4.2.1 Measuring stability and reliability in a “multi-team monolith”

DORA metrics were originally designed to be measured on *entire* applications. And for those the researchers showed that could link attainment of high levels of performance to positive business outcomes. I would like to propose here that we continue (or start!) to measure DORA metrics as originally defined, but also make use of the package and team structure of the application to give us another level of insight: DORA metrics per team in the modularized monolith.

For this, I am assuming that you assign file ownership to teams, ideally by domain package as much as is feasible. If you have not done so, check out in [Section 7.4](#) and [Section 7.6](#) how to start using the teams and code ownership gems from RubyAtScale (<https://github.com/rubyatscale>). I am also assuming that you use CI/CD to test code and merge it to deployment branches.

It should also be added that this level of analysis likely starts making sense when the engineering organization gets so large that it becomes infeasible to solve problems together all the time. Not sure

exactly when that point is, but I suspect it is at least somewhere north of 50 people.

Once in place, we can use code ownership to assess DORA metrics on a per team basis. In principle, this assignment is straightforward: assign the various metrics on a per team basis. As we will see, the nuances can get tricky quickly.

Deployment Frequency: Count every deploy to production that contains a PR from a team as a deploy of that team. As straightforward as this may seem... What is “a PR from a team?” If a member of a team has created it (the PR)? If it changes code owned by the team? If it was reviewed by a member of this team? If the commits on the PR were majority created by a member of the team? The simplicity of the original idea can go out the window quickly if we are not careful. I believe that all of these definitions are sensible. I also believe that they are likely all good proxies for each other: A member of a team is likely to author the commits that lead to a PR that changes code that belongs to the team... It isn’t the worst idea to start with a simple definition where the data is easy to gather. We can always do more research on better metrics as we go. One place to do this is during post-mortem meetings where participants should flag if code that was part of an incident was “unusual” in how it got created. I propose we start with the definition:

Count every deploy to production that contains a PR created by a member of the team as a deploy of that team

Lead Time for Changes: However you like to measure lead time for change (be it time of first commit, time PR was created, or time PR landed on the main branch), given that we have above associated every PR with a team, we can calculate the lead time for changes for every team.

Count the lead time for changes for a team solely based on the PRs from that team

Let’s flip the order from before and discuss time to restore service and reliability before change failure rate.

Time to Restore Service For any given incident, if it affected a team’s parts of the system, record an outage for the time of the incident that affected these systems. Dividing incidents by team impact, one can create three groups: everything is affected, just one team’s stuff is affected, and finally several, but not all, teams’ stuff is affected. In my experience, the vast majority of incidents affects either all or just one team, so if this feels daunting to collect the data for, start there.

For every team that is affected by an incident, count how long it was affected

Reliability and its various technical business metrics including availability, latency, and performance, can be individually analyzed as to how the map to team based performance. If availability is defined as percentage of successful requests, we need to ensure that we can extend code ownership information into the various request endpoints and can then measure it per team. Its the same data basis, but a different calculation if availability is measured as percentage of uptime. If your API is

GraphQL, i.e., you have only one endpoint, dig one or two levels deeper and analyze the data by query or even field.

Change Failure Rate If a deploy adversely affects a team's part of the app when the team deployed a change, that is a *failed change*. If a failed deploy has a bigger impact radius and affects other, even all, parts of the app, that is a *failed change*.

If the PR of a team creates adverse production performance from that deploy, count it as a failed deploy for that team

If an organization is only starting out collecting DORA metrics, there is an interesting opportunity here to collect a slightly more nuanced version of change failure rate: Count a deploy that adversely affects a team's part of the app as a failed deploy for that team whether or not they had a PR in the deploy. This data is relatively easy to collect during post-mortems, especially if one focuses on the "everything affected" vs "only the team affected" observation discussed above. Note that if you calculate the change failure rate based on this, you can see change failure rates above 100% if a team had only few deploys but was affected by others (or you might not be able to calculate the number, if the team didn't deploy in a timeframe).

With this change we create two change failure rates: a "caused-by rate" and an "affected-by rate." While this may seem odd at first, it does point to the potential impacts of working in a monolith: While we may try to modularize it, it is still one application that can and will fail as one. And the consequences a team must draw are different depending on which rate they are looking at and finding unacceptably high.

If the "caused-by rate" is high, the team should invest in the conviction they build that a certain code change of theirs is good: more of the right kind of tests, maybe some smoke test in the live environment before the application gets switch to the new version, refactorings to make the code more manageable.

If the "affected-by rate" is high, the team might be doing everything right in their part of the code but they are downstream of the problems of other teams (it being a monolith and all...). In this case, the team should invest in making itself more autonomous, more independent from the other team. If it is the login page and the user sessions that keep crashing your things, you may not be able to do much other than starting the discussion between teams that more investments need to be made in stability of those parts of the system. However, if the interaction is different, there may be things you can do: If a dashboard pages prevents users from accessing your workflows because it crashes completely based on dashboard content from other teams, you could try to make the different parts of the system fail more gracefully, i.e., without taking down the entire page.

Another example would be the backend processes of parts of the application creating a lot of method calls between the code of two teams. We discussed using events to disconnect these direct calls in [Chapter 5.10](#). Now, if the events are not being processed one way or another, the application is not immediately down, it just becomes more out of date over time. This will almost always help improve the user experience, but may feel like too big an investment or too big a code change, but it is there as an option when the problem of being affected by other folks' changes needs to be managed.

8.4.2.2 Create a set of metrics that are useful in driving improvements

In discussing the nuances of an expanded view of change failure rate, I prefaced that with that being an option for organizations “only starting” with this kind of metrics gathering. This leads me to urge you to think about the incentives you want to drive with the system of metrics that you create. Likely you want to improve quality in some sort of way - but quality comes in a lot of flavors: DORA metrics focus on software delivery and operational performance. It is important to start collecting, whether or not the investment or change plan is fully in place already. It likely won't be because you don't have the data yet!

It will turn out that as you grow your organization's capability to collect and use these metrics effectively, that the need for other metrics or more nuanced views arises and also that some loose some value over time.

As just one example to that end: If you are automatically collecting team-based availability metrics, the need for analyzing whether a change affected a team becomes less. The availability metric will likely be more real-time and more detailed than the analysis during a post-mortem can be.

The point we must always keep in mind is that, when employing metrics to shape organizational behavior, there will be the effects that we intended and the ones that we didn't. As such, we need to change what we measure and what we emphasize based on what changes it is affecting in the organization. With that thought in mind, we can turn directly to the question of how to create change in an engineering organization to create a better structured application that supports our business better over time.

9. Creating Modularization Progress

9.1 Just do it

9.2 Beer-athon move files

9.3 Hackathons

9.4 Kudos

9.5 Dedicated Project Team

- spaghetti models

9.6 Champions

9.7 Vision

- Nested Packs