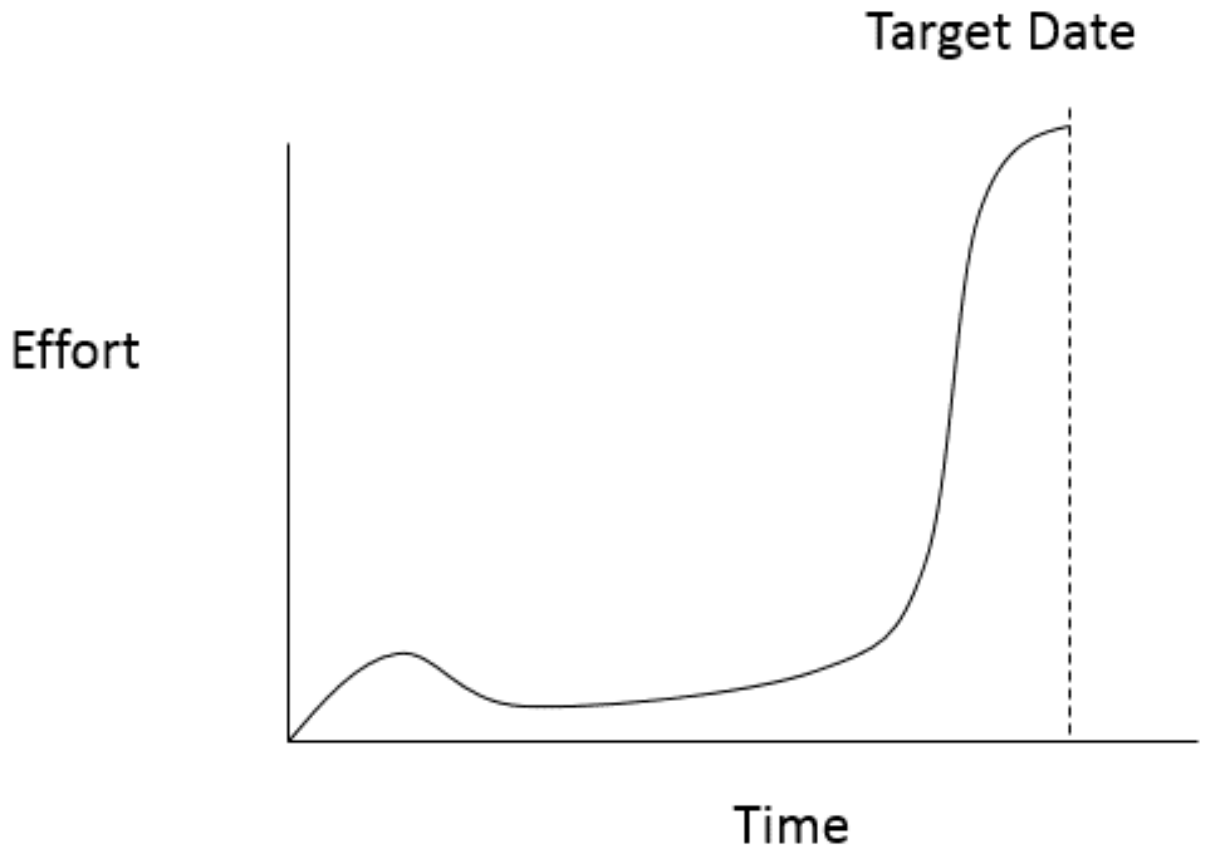

Target Date
Effort
Time

ACME::Foo::Bar → ACME/Foo/Bar.pm

os.path → os/path.py

@INC

/usr/lib/perl5/site_perl/5.22.1/x86_64-linux-thread-multi

/usr/lib/perl5/site_perl/5.22.1

/usr/lib/perl5/vendor_perl/5.22.1/x86_64-linux-thread-multi

/usr/lib/perl5/vendor_perl/5.22.1

/usr/lib/perl5/5.22.1/x86_64-linux-thread-multi

/usr/lib/perl5/5.22.1

Compiled modules

- > find /usr/lib/perl5 -name "*.pmc" | wc -l
  2
- > find /usr/lib/python3.4 -name "*.pyc" | wc -l
  793

Perl 6 features

- Ünicöde
- Authors
- Versions

Packaging

> rpm -ql perl-Net-Telnet-3.04-5.4.noarch

/usr/lib/perl5/vendor_perl/5.22.1/Net

/usr/lib/perl5/vendor_perl/5.22.1/Net/Telnet.pm

/usr/share/doc/packages/perl-Net-Telnet

/usr/share/doc/packages/perl-Net-Telnet/ChangeLog

/usr/share/doc/packages/perl-Net-Telnet/README

/usr/share/man/man3/Net::Telnet.3pm.gz

Long names

Foo::Bar:auth<cpan:nine>:ver<0.3>:api<1>

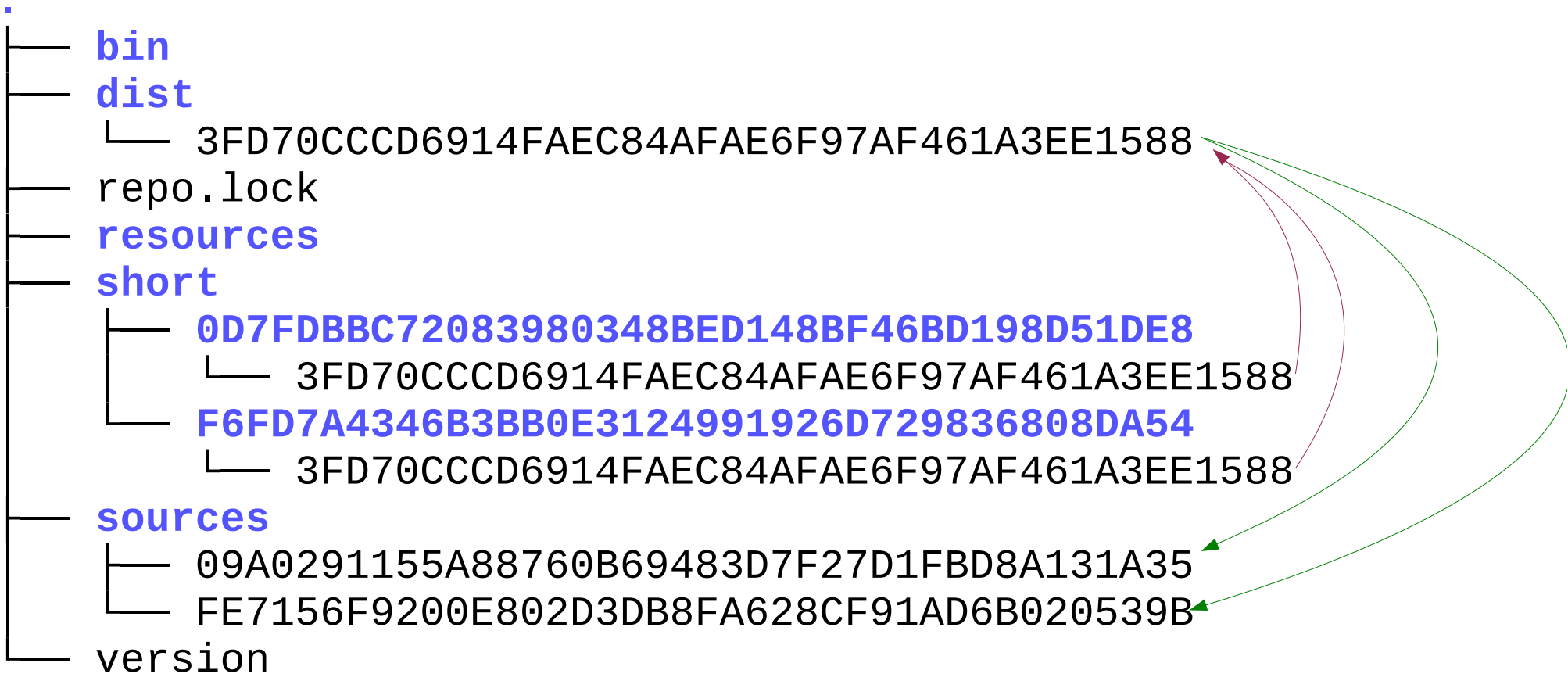
$*REPO

```
role CompUnit::Repository {
    has CompUnit::Repository $.next-repo is rw;

    method need(CompUnit::DependencySpecification $spec,
                CompUnit::PrecompilationRepository $precomp,
                CompUnit::PrecompilationStore :@precomp-stores)
        returns CompUnit:D
        { ... }

    method loaded()
        returns Iterable
        { ... }

    method id()
        returns Str
        { ... }
}
```

```
> perl6 -Ilib -e 'dd $*REPO'
CompUnit::Repository::FileSystem.new("/home/nine/lib")
> perl6 -e 'dd $*REPO'
CompUnit::Repository::Installation.new("/home/nine/.perl6/2015.12-395-g9c0f96f")
```

Matching

```
use Inline::Perl5:auth(/.*:nine/):ver(v0.2+);

CompUnit::DependencySpecification.new(
    :short-name<Inline::Perl5>,
    :auth-matcher(/.*:nine/),
    :version-matcher(v0.2+),
);

'Inline::Perl5' eq 'Inline::Perl5'
and 'cpan:nine' ~~ /.*:nine/
and v0.3 ~~ v0.2+
and 1 ~~ True
```

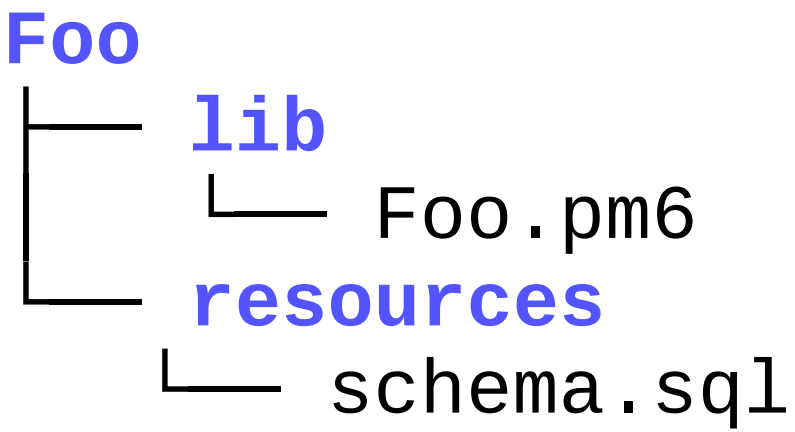

.
bin
dist
3FD70CCCD6914FAEC84AFAE6F97AF461A3EE1588
repo.lock
resources
short
0D7FDBBC72083980348BED148BF46BD198D51DE8
3FD70CCCD6914FAEC84AFAE6F97AF461A3EE1588
F6FD7A4346B3BB0E3124991926D729836808DA54
3FD70CCCD6914FAEC84AFAE6F97AF461A3EE1588
sources
09A0291155A88760B69483D7F27D1FBD8A131A35
FE7156F9200E802D3DB8FA628CF91AD6B020539B
version

%?RESOURCES

%?RESOURCES<libraries/p5helper>

%?RESOURCES<icons/foo.png>

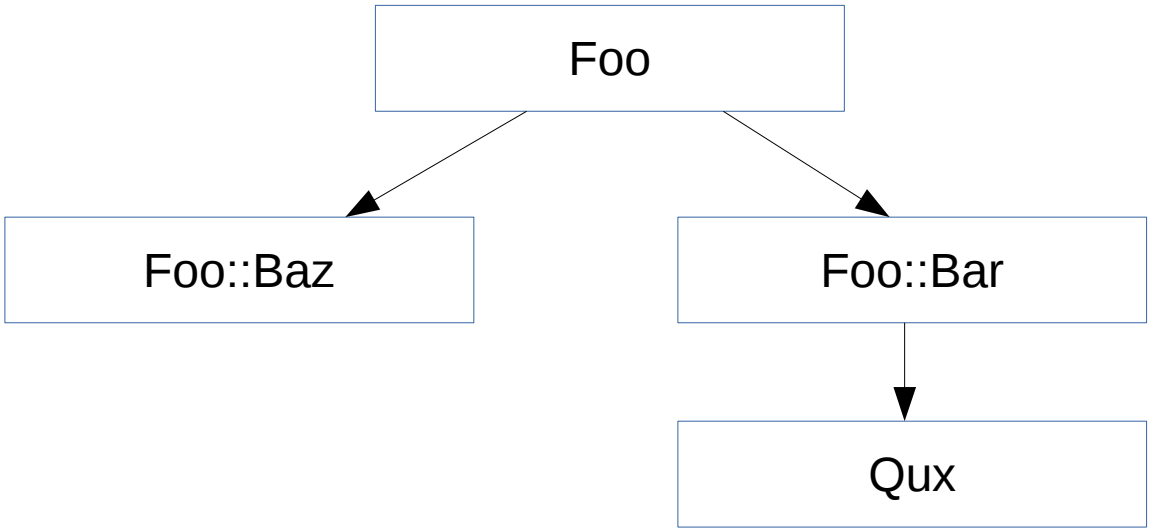
%?RESOURCES<schema.sql>



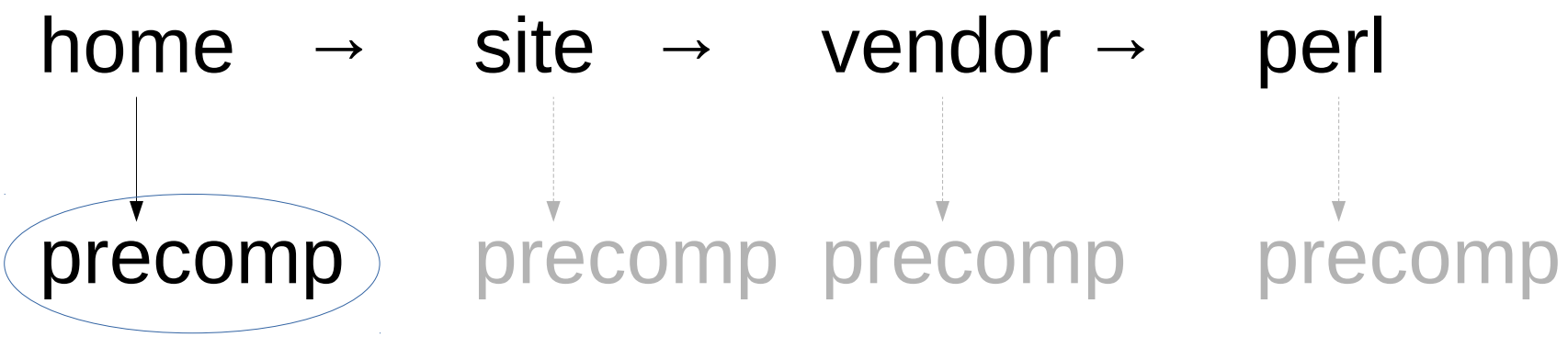
```
unit class CompUnit::Repository::DependencyTracker
    does CompUnit::Repository;

CompUnit::RepositoryRegistry.use-repository($?CLASS.new);

my %seen := SetHash.new;
END say %seen;

method need(
    CompUnit::DependencySpecification $spec,
    CompUnit::PrecompilationRepository $precomp?
) {
   %seen{$spec.Str}++;
   self.next-repo.need($spec,
CompUnit::PrecompilationRepository::None);
}

method id() { 'dependencytracker' }
method loaded() { [] }
```

```
unit class CompUnit::Repository::Panda
    does CompUnit::Repository;

my $prev-repo = $*REPO.repo-chain[*-1];
$prev-repo.next-repo = CompUnit::Repository::Panda.new;

method need(
    CompUnit::DependencySpecification $spec,
    CompUnit::PrecompilationRepository $precomp
) {
    run('panda', 'install', $spec.short-name);
    $prev-repo.next-repo = CompUnit::Repository;
    LEAVE {
        $prev-repo.next-repo = self;
    }
    $*REPO.need($spec)
}

method id() { 'panda' }
method loaded() { [] }
```

Dependencies



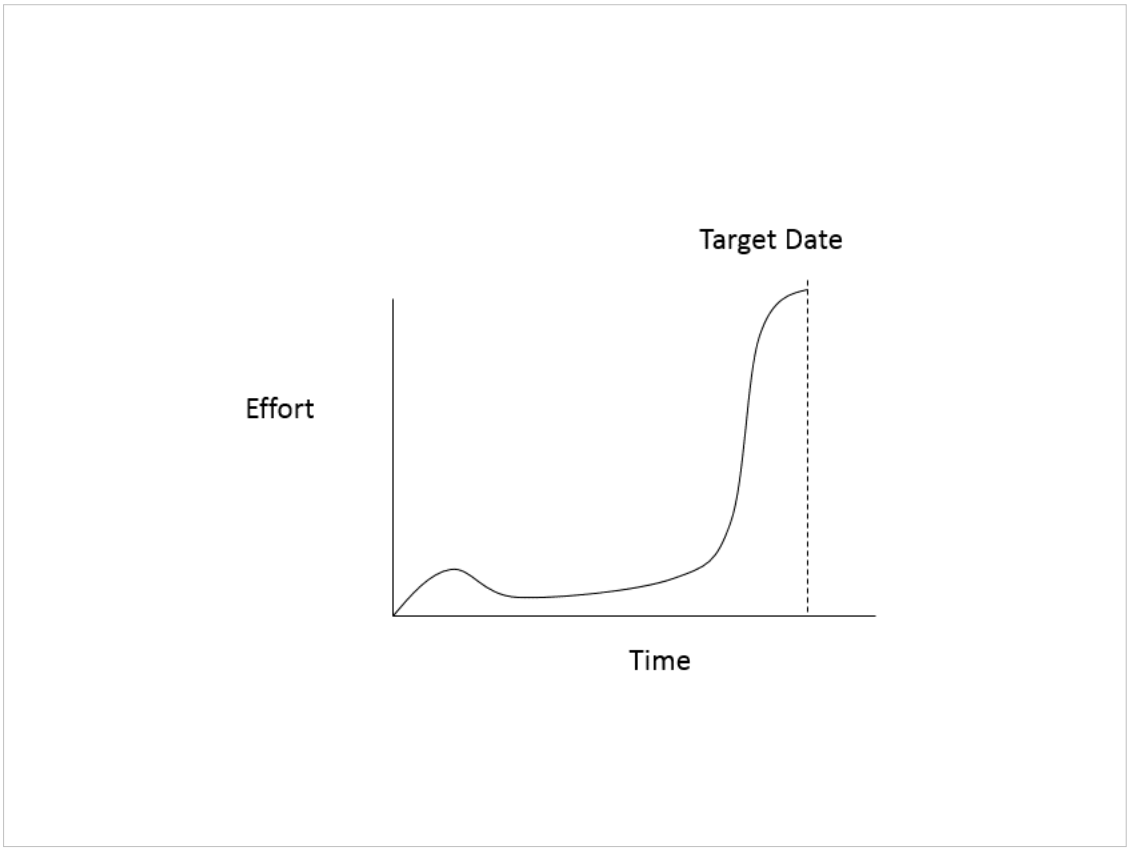
Precomp stores



Thank You!

http://niner.name/talks/
http://github.com/niner/

We did it!
The great experiment came to conclusion.
We took 15 years, longer than anyone ever before to undisputably proof, that the student syndrome is unavoidable.



15 years of time to complete the project and still when the self-picked deadline arrived, we had to rush to get the final pieces in place.

One of those pieces was a completely redesigned framework for module installation and loading.
Unfortunately it was also one of the biggest sources of pain right after the big release.
How comes?
Why did it take 15 years to design and implement this module framework?
Shouldn't this be a rather mundane part compared to the rest of Perl 6?

Before I answer those questions, let's have a look at how languages like Perl 5 or Python handle module installation and loading.

ACME::Foo::Bar → ACME/Foo/Bar.pm

os.path → os/path.py

In those languages, module names have a 1:1 relation with file system paths.
We simply replace the double colons with slashes and add a .pm

Note that these are relative paths.
Both Python and Perl 5 use a list of include paths, to complete these paths.
In Perl 5 they are available in the global @INC array.

@INC

/usr/lib/perl5/site_perl/5.22.1/x86_64-linux-thread-multi
/usr/lib/perl5/site_perl/5.22.1
/usr/lib/perl5/vendor_perl/5.22.1/x86_64-linux-thread-multi
/usr/lib/perl5/vendor_perl/5.22.1
/usr/lib/perl5/5.22.1/x86_64-linux-thread-multi
/usr/lib/perl5/5.22.1

Each of these include directories is checked for whether it contains a relative path determined from the module name.
If the shoe fits, the file is loaded.

Of course that's a bit of a simplified version.
Both languages support caching compiled versions of modules.
So instead of just the .pm file Perl 5 first looks for a .pmc file.
So does Python with .pyc files.

Compiled modules

- > find /usr/lib/perl5 -name "*.pmc" | wc -l
  2
- > find /usr/lib/python3.4 -name "*.pyc" | wc -l
  793

While the support is there, to my knowledge almost no one uses this possibility in Perl 5 while in Python it's quite common to precompile modules on installation.

Module installation in both cases means mostly copying files into locations determined by the same simple mapping.

This system is easy to explain, easy to understand, simple and robust.
In other words, it sounds pretty much perfect.
So Perl 6 should probably just follow these well established examples and do the same, shouldn't it?

Perl 6 features

- Ünicöde
- Authors
- Versions

I would say "yes", if it weren't for some features that they lack and that Perl 6 wants to provide:
* Unicode module names
* Modules published under the same names by different authors
* Having multiple versions of a module installed

Why would you want this madness?
The only language that restricts itself to 26 Latin characters is Latin. Even English has diacritics for many loan words, at least if they're written correctly. With a 1:1 relation between module names and file system paths, you enter a world of pain once you try to support Unicode on multiple platforms and file systems.

Then there's sharing module names between multiple authors. This one may or may not work out well in practice. I can imagine using it for example for publishing a module with some fix until the original author includes the fix in the "official" version.

Finally there's multiple versions. Usually people who need certain versions of modules reach for local::lib or containers or some home grown workarounds. They all have their own disadvantages. None of them would be necessary if applications could just say, hey I need good old, trusty version 2.9 or maybe a bug fix release of that branch.

If you had any hopes of continuing using the simple name mapping solution, you probably gave up at the versioning requirement. Because, how would you find version 3.2 of a module when looking for a 2.9 or higher?
Any ideas?

Packaging

```
> rpm -ql perl-Net-Telnet-3.04-5.4.noarch
/usr/lib/perl5/vendor_perl/5.22.1/Net
/usr/lib/perl5/vendor_perl/5.22.1/Net/Telnet.pm
/usr/share/doc/packages/perl-Net-Telnet
/usr/share/doc/packages/perl-Net-Telnet/ChangeLog
/usr/share/doc/packages/perl-Net-Telnet/README
/usr/share/man/man3/Net::Telnet.3pm.gz
```

Popular ideas included collecting information about installed modules in JSON files and as those turned out to be toe nail growing slow, putting the meta data into SQLite databases.
However, these ideas can be easily shot down by introducing another requirement: distribution packages.

Packages for Linux distributions are mostly just archives containing some files plus some meta data.
Ideally the process of installing such a package means just unpacking the files and updating the central package database.
Uninstalling means deleting the files installed this way and again updating the package database.
Changing existing files on install and uninstall makes packagers' lives much harder, so we really want to avoid that.
Also the names of the installed files may not depend on what was previously installed.
We must know at the time of packaging what the names are going to be.

So what does the current attempt at solving all these challenges look like?

Long names

Foo::Bar:auth<cpan:nine>:ver<0.3>:api<1>

Step 0 in getting us back out of this mess is some definitions:

A full module name in Perl 6, a so called long-name consists of the short-name, auth, version and API

On the other side, the thing you install is usually not a single module but a distribution which probably contains one or more modules.

Distribution names work just the same way as module names.

Indeed, distributions often will just be called after their main module.

An important property of distributions is that they are immutable.

Foo:auth<nine>:ver<0.3>:api<1> will always be the name for exactly the same code.

$*REPO

```
role CompUnit::Repository {
   has CompUnit::Repository $.next-repo is rw;

   method need(CompUnit::DependencySpecification $spec,
               CompUnit::PrecompilationRepository $precomp,
               CompUnit::PrecompilationStore :@precomp-stores)
       returns CompUnit:D
       { ... }

   method loaded()
       returns Iterable
       { ... }

   method id()
       returns Str
       { ... }
}
```

In Perl 5 and Python you deal with include paths, pointing to file system directories.
In Perl 6 we call such directories "repositories" and each of these repositories is governed by an object that does the CompUnit::Repository role.
Instead of an @INC array, there's the $*REPO variable.
It contains a single repository object.
This object has a "next-repo" attribute that may contain another repository.
In other words: repositories are managed as a linked list.
The important difference to the traditional array is, that when going through the list, each object has a say in whether to pass along a request to the next-repo or not.
Perl 6 sets up a standard set of repositores, i.e. the "perl", "vendor" and "site" repositories, just like you know them from Perl 5.
In addition, we set up a "home" repository for the current user.

Repositories must implement the "need" method.
A "use" or "require" statement in Perl 6 code is basically translated to a call to $*REPO's "need" method.
This method may in turn delegate the request to the next-repo.

```
> perl6 -Ilib -e 'dd $*REPO'
CompUnit::Repository::FileSystem.new("/home/nine/lib")
> perl6 -e 'dd $*REPO'
CompUnit::Repository::Installation.new("/home/nine/.perl6/2015.12-395-g9c0f96f")
```

- Rakudo comes with several classes that can be used for repositories.
- The most important ones are CompUnit::Repository::FileSystem and CompUnit::Repository::Installation.
- The FileSystem repo is meant to be used during module development and actually works just like Perl 5 when looking for a module.
- It doesn't support versions or auths and simply maps the short-name to a file system path.

- The Installation repository is where the real smarts are.

Matching

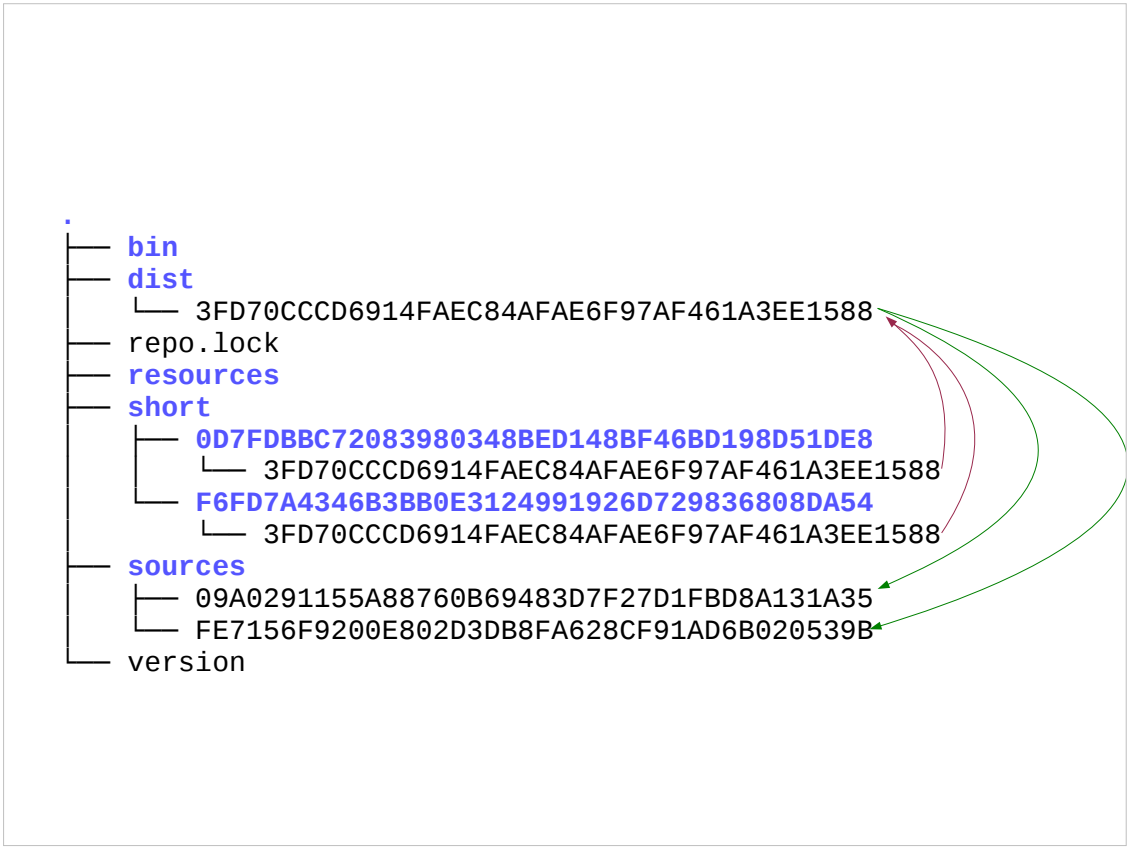
```
use Inline::Perl5:auth(/.*:nine/):ver(v0.2+);

CompUnit::DependencySpecification.new(
   :short-name<Inline::Perl5>,
   :auth-matcher(/.*:nine/),
   :version-matcher(v0.2+),
);

'Inline::Perl5' eq 'Inline::Perl5'
and 'cpan:nine' ~~ /.*:nine/
and v0.3 ~~ v0.2+
and 1 ~~ True
```

When requesting a module, you will usually either do it via its exact long name, or you say something along the lines of "give me a module that matches this filter".

Such a filter is given by way of a CompUnit::DependencySpecification object which has fields for: short-name, auth-matcher, version-matcher and api-matcher.

When looking through candidates, the Installation repository will smart match a module's long name against this DependencySpecification or rather the individual fields against the individual matchers.

Thus a matcher may be some concrete value, a version range or even a regex.



As previously mentioned, loading the meta data of all installed distributions would be prohibitively slow.
Instead, we use the file system as a kind of database.
We store not only a distribution's files but also create indices for speeding up lookups.
One of these indices comes in the form of directories named after the short-name of installed modules.
However thanks to the Unicode issues mentioned, we cannot just use the module names directly.
This is where the now infamous SHA-1 hashes enter the game.
The directory names are the ASCII encoded SHA-1 hashes of the UTF-8 encoded module short-names.

In these directories we find one file per distribution that contains a module with a matching short name.
These files again contain the ID of the dist and the other fields that make up the long name: auth, version and api.
So by reading these files we have a usually short list of auth-version-api triplets which we can match against our DependencySpecification.
We end up with the winning dist's ID, which we use to look up the meta data, stored in a JSON encoded file.
This meta data contains the name of the file in the sources/ directory containing the requested module's code.
This is what we can load.

Finding names for source files is again a bit tricky, as there's still the Unicode issue and in addition the same relative file names may be used by different installed distributions (think versions).
So for now at least, we use SHA-1 hashes of the long-names.

%?RESOURCES

```
%?RESOURCES<libraries/p5helper>
%?RESOURCES<icons/foo.png>
%?RESOURCES<schema.sql>
```

```
Foo
├── lib
│   └── Foo.pm6
└── resources
    └── schema.sql
```

It's not only source files that are stored and found this way.

Distributions may also contain arbitrary resource files.

These could be images, language files or shared libraries that are compiled on installation.

They can be accessed from within the module through the %?RESOURCES hash

As long as you stick to the standard layout conventions for distributions, this even works during development without installing anything.

A nice result of this architecture is that it's fairly easy to create special purpose repositories.

```
unit class CompUnit::Repository::DependencyTracker
    does CompUnit::Repository;

CompUnit::RepositoryRegistry.use-repository($?CLASS.new);

my %seen := SetHash.new;
END say %seen;

method need(
    CompUnit::DependencySpecification $spec,
    CompUnit::PrecompilationRepository $precomp?
) {
   %seen{$spec.Str}++;
   self.next-repo.need($spec,
CompUnit::PrecompilationRepository::None);
}

method id() { 'dependencytracker' }
method loaded() { [] }
```

My first example is CompUnit::Repository::DependencyTracker which you put into the repository chain and which simply records what's going on and tells you which modules were loaded.

There are several Perl 5 modules that try to do this, but as far as I could find out, none of them can tell you exactly what was requested, i.e. the DependencySpecification but only what was actually loaded.

```
unit class CompUnit::Repository::Panda
   does CompUnit::Repository;

my $prev-repo = $*REPO.repo-chain[*-1];
$prev-repo.next-repo = CompUnit::Repository::Panda.new;

method need(
   CompUnit::DependencySpecification $spec,
   CompUnit::PrecompilationRepository $precomp
) {
   run('panda', 'install', $spec.short-name);
   $prev-repo.next-repo = CompUnit::Repository;
   LEAVE {
       $prev-repo.next-repo = self;
   }
   $*REPO.need($spec)
}

method id() { 'panda' }
method loaded() { [] }
```

The second example is CompUnit::Repository::Panda.

Just load this module and it will transparently install any missing dependency.

It doesn't do any harm if a module is already installed as it will attach itself to the end of the repository chain and will only be called if no other repository can satisfy the dependency.

While in Perl 5 something similar is quite possible, the result is much less nice and does have significant runtime overhead.

So far, we've looked at loading modules from source files.

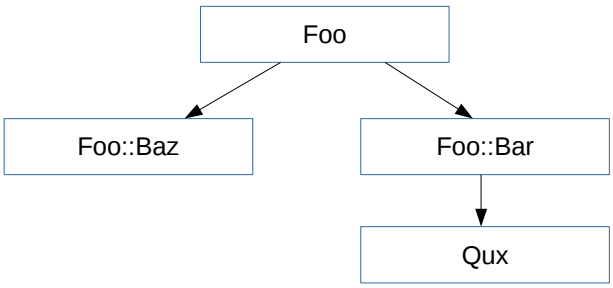
However, just the same as Perl 5 and Python, Perl 6 supports precompiling modules into binaries.

While this is rarely used in Perl 5 and a nice bonus in Python, in Perl 6 it's almost essential.

At least, if you do not want to wait for a minute or two while loading a moderately sized code base.

Yes, compilation of Perl 6 code is slow. It takes forever.

Dependencies



Luckily precompilation at least works quite well in most cases. Yet it comes with its own set of challengess. Loading a single module is easy.

Fun starts when that module has dependencies and those dependencies have again dependencies of their own.

When loading a precompiled file in Perl 6 we need to load the precompiled files of all its dependencies, too.

And those dependencies *must* be precompiled, we cannot load them from source files.

Even worse, the precomp files of the dependencies *must* be exactly the same files we used for precompiling our module in the first place.

To top it off, precompiled files work only with the exact perl6 binary, that was used for compilation.

All of that would still be quite manageable if it weren't for an additional requirement: as a user you expect a new version of a module you just installed to be actually used, don't you?

In other words: if you upgrade a dependency of a precompiled module, we have to detect this and precompile the module again with the new dependency.

Now remember that while we have a standard repository chain, the user may prepend additional repositories by way of "-I" on the command line or "use lib" in the code.
These repositories may contain the dependencies of precompiled modules.
Our first solution to this riddle was that each repository gets it's own precomp store where precompiled files are stored.
We only ever load precomp files from the precomp store of the very first repository in the chain because this is the only repository that has direct or at least indirect access to all the candidates.
If this repository is a FileSystem repository, we create a precomp store in a .precomp directory.
While being the safe option, this has the consequence that whenever you use a new repository, we will start out without access to precompiled files.
Instead, we will precompile the modules used when they are first loaded.
Since the "home" repository is not available when installing a module system wide, the sad truth is, that we will not even use the precomp files created during installation.

Luckily a solution is just around the corner and it will allow us to use precomp files of other repositories as long as there are no changes to their dependencies.
This solution will land in the next few weeks.

Thank You!

http://niner.name/talks/
http://github.com/niner/

To draw a conclusion: Perl 6 tries to go a step or five further than other languages. In the case of module management, we shipped a somewhat reasonable first implementation. There are lots of opportunities for improving the user experience, lots of improvements already in the works and a couple of interesting new possibilities.

Stay tuned and talk to us on #perl6 about where you would want to see this going.

And finally as a little homework exercise: write a CompUnit::Repository that hides an installed module like you would want in a test file to check if everything works if an optional dependency is missing.