Principles of Software Construction: Objects, Design, and Concurrency

Organizing Systems at Scale: Modules, Dependencies, Breaking Changes

Bogdan Vasilescu

Quiz: API Design lecture

On Canvas, Lecture 18

Administrative

Exam Thursday.

HW6 release.

Email BV+JA+MD, please!

Where we are

Design for	*Small scale:* One/few objects	*Mid scale:* Many objects	*Large scale:* Subsystems
understanding	Subtype Polymorphism ✓	Domain Analysis ✓	GUI vs Core ✓
change/ext.	Information Hiding, Contracts ✓	Inheritance & Del. ✓	Frameworks and Libraries ✓, **APIs** ✓
reuse	Immutability ✓	Responsibility Assignment, Design Patterns, Antipattern ✓	**Module systems, microservices**
robustness	Types	Promises/ Reactive P. ✓	Testing for Robustness
...	Unit Testing ✓	Integration Testing ✓	CI ✓, DevOps, Teams

REST APIs
(REpresentational State Transfer)

REST (or RESTful) API

API of a web service “that conforms to the constraints of the REST architectural style.”

Uniform interface over HTTP requests

Send parameters to URL, server responds with the representation of a resource (JSON, XML common)

Stateless: Each request is self-contained

Language independent, distributed

REST API Design

All the same design principles apply

Document the API, input/output formats and error conditions!

CRUD Operations

Path correspond to nouns, not verbs, nesting common:

- `/articles`, `/state`, `/game`
  `/articles/:id/comments`

GET (receive), POST (submit new), PUT (update), and DELETE requests sent to those paths

Parameters for filtering, searching, sorting, e.g., `/articles?sort=date`

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();
app.use(bodyParser.json()); // JSON input
app.get('/articles', (req, res) => {
  const articles = [];
  // code to retrieve an article...
  res.json(articles);
});
app.post('/articles', (req, res) => {
  // code to add a new article...
  res.json(req.body);
});
app.put('/articles/:id', (req, res) => {
  const { id } = req.params;
  // code to update an article...
  res.json(req.body);
});
app.delete('/articles/:id', (req, res) => {
  const { id } = req.params;
  // code to delete an article...
  res.json({ deleted: id });
});
app.listen(3000, () => console.log('server started'));
```

REST Specifics

- JSON common for data exchange: Define and validate schema -- many libraries help
- Return HTTP standard errors (400, 401, 403, 500, …)
- Security mechanism through SSL/TLS and other common practices
- Caching common
- Consider versioning APIs `/v1/articles`, `/v2/articles`

Module Systems

Traditional Library Reuse

Static/dynamic linking against binaries (e.g., .DLLs)

Copy library code into repository

Limitations?

Package Managers

Refer to library releases by name + version

Immutable storage in repository

Dependency specification in repository

Package manager downloads / updates dependencies

    Maven, npm, pip, cargo, nuget, …

Release libraries to package repository

To allow all of the previous things, we need Module Systems

Foundation for distributing and reusing libraries

Packaging code (binary / source)

Linking against code in a module without knowing internals

Traditional Approach in Java: Packages and Jar Files

Packages structure name space, avoid naming collisions (edu.cmu.cs17214…)

Public classes are globally visible

- package visibility to hide within package
- no way to express visibility to select packages

.jar files bundle code (zip format internally)

- Java can load classes from all .jar files in classpath
- Java does not care where a class comes from, loads first that matches name

Classpath established at JVM launch

Packages enough?

```
edu.cmu.cs214.santorini
edu.cmu.cs214.santorini.gui
edu.cmu.cs214.santorini.godcards
edu.cmu.cs214.santorini.godcards.impl
edu.cmu.cs214.santorini.logic
edu.cmu.cs214.santorini.utils
```

Toward Module Systems

Stronger encapsulation sometimes desired

Expose only select public packages (and all public classes therein) to other modules

Dynamic adding and removal of modules desired

OSGi (most prominently used by Eclipse)

- Bundle Java code with Manifest
- Framework handles loading with multiple classloaders

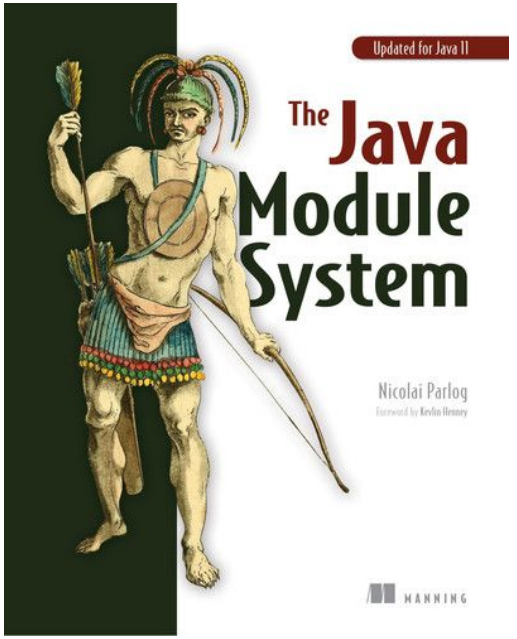
```
Bundle-Name: Hello World
Bundle-SymbolicName: org.wikipedia.helloworld
Bundle-Description: A Hello World bundle
Bundle-ManifestVersion: 2
Bundle-Version: 1.0.0
Bundle-Activator: org.wikipedia.Activator
Export-Package:
org.wikipedia.helloworld;version="1.0.0"
Import-Package:
org.osgi.framework;version="1.3.0"
```

Java Platform Module System

Since Java 9 (2017); built-in alternative to OSGi

Modularized JDK libraries itself

Several technical differences to OSGi (e.g., visibility vs access protection, handling of diamond problem)




```
module A {
     exports org.example.foo;
     exports org.example.bar;
}
module B {
     require A;
}
```

Toward JavaScript Modules

Traditionally no module concept, import into flat namespace – no way to avoid conflicts if two modules had a variable with the same name

Creating own namespaces with closures/module pattern

```
<html>
<header>
<script type="text/javascript" src="lib1.js"></script>
<script type="text/javascript">
  var x = 1;
</script>
<script type="text/javascript" src="lib2.js"></script>
...
```

The Module Pattern

Learning
Patterns
By Lydia Hallie and Addy Osmani

```
var myRevealingModule = (function () {
    var privateVar = "Ben Cherry",
        publicVar = "Hey there!";

    function privateFunction() {
        console.log( "Name:" + privateVar );
    }

    function publicSetName( strName ) {
        privateVar = strName;
    }

    function publicGetName() {
        privateFunction();
    }

    // Reveal public pointers to
    // private functions and properties
    return {
        setName: publicSetName,
        greeting: publicVar,
        getName: publicGetName
    };
})();

myRevealingModule.setName( "Paul Kinlan" );
```

The Module Pattern

```
<html>
<header>
<script type="text/javascript" src="lib1.js"></script>
<script type="text/javascript">
  const m1 = (function () {
    const export = {}
    const x = 1;
    export.x = x;
    return export;
  }());
</script>
<script type="text/javascript" src="lib2.js"></script>
...
```

Node.js Modules (CommonJS)

Function `require()` to load other module, dynamic lookup in search path

Module: JavaScript file, can write to export object

```
var http = require('http');

exports.loadData = function () {
  return http....
};
```

```
var surprise = require(userInput);
```

Node uses Module Pattern Internally

```
function loadModule(filename, module, require) {
  var wrappedSrc =
    '(function(module, exports, require) {' +
      fs.readFileSync(filename, 'utf8') +
    '})(module, module.exports, require);';
  eval(wrappedSrc);
}
```

ES2015 Modules (Similar to TypeScript)

Syntax extension for modules (instead of module pattern)

Explicit imports / exports

Static import names (like Java), supports better reasoning by tools

```
import { Location } from './location'
import { Game } from './game'
import { Board } from './board'
// module code
export { Worker, newWorker }
```

JavaScript Modules and Packages

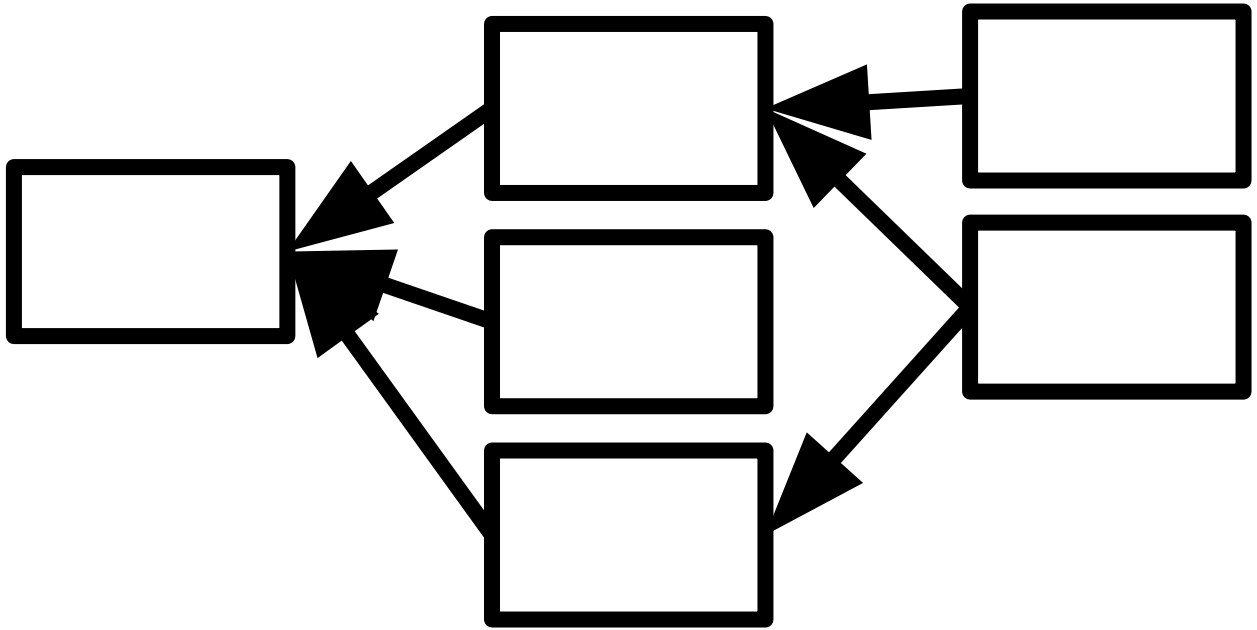
Modules always decide what to export (values, functions, classes, …) -- everything else only visible in module.

Directory structure only used for address in import.

Packages typically have one or more modules and a name and version.

Dependency graphs and dependency problems

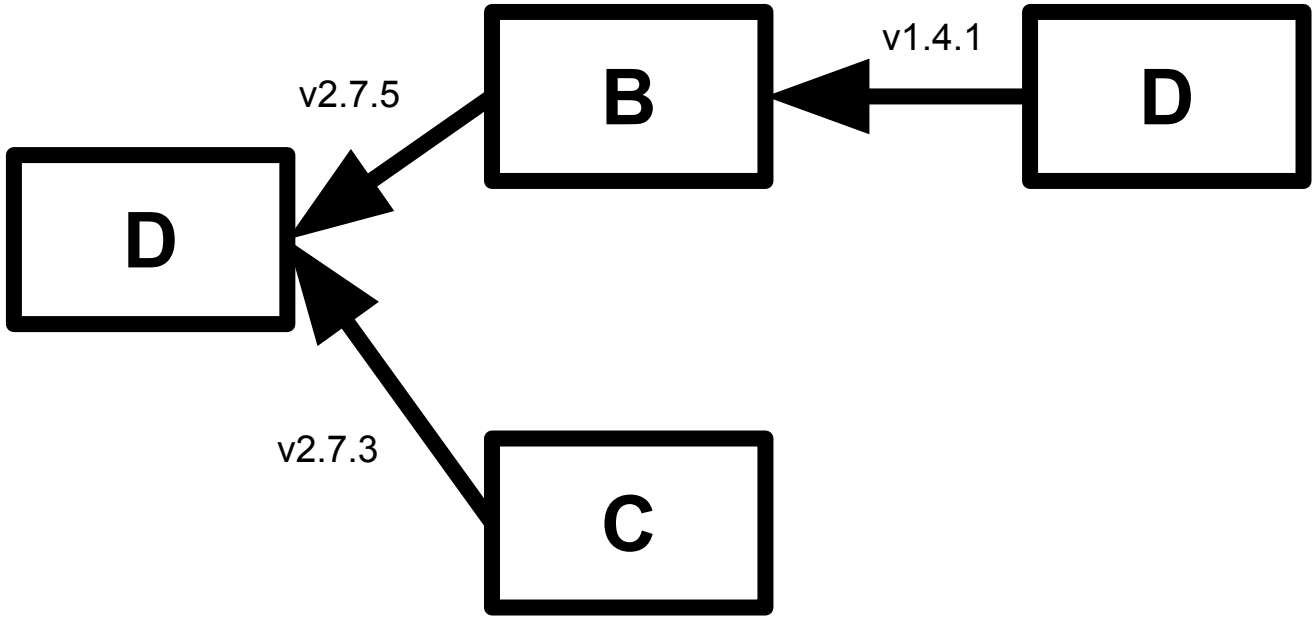
Dependency Graphs



Acyclic

Versioned dependency edges

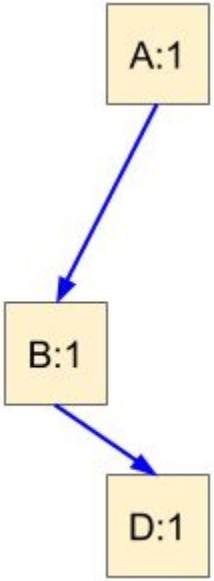
Dependency Graphs



Acyclic

Versioned dependency edges

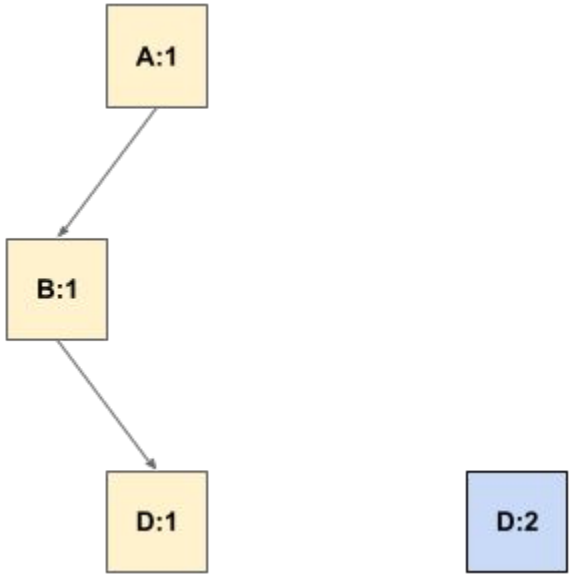
Dependency Graphs



Consider three libraries: A, B, and D, each at version 1.
A:1 depends on B:1 which depends on D:1.
In the beginning, everyone is happy.

Example from https://jlbp.dev/what-is-a-diamond-dependency-conflict

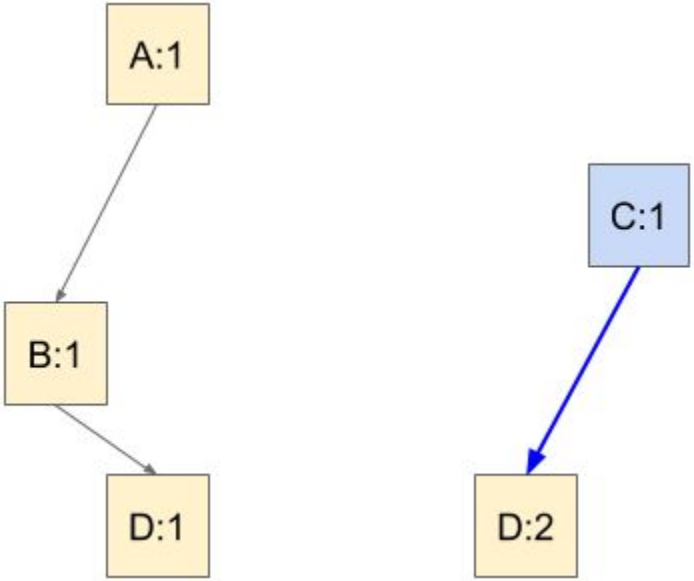
Dependency Graphs



Now D introduces version 2, which adds some features but also removes some features. No problems yet.

Example from https://jlbp.dev/what-is-a-diamond-dependency-conflict

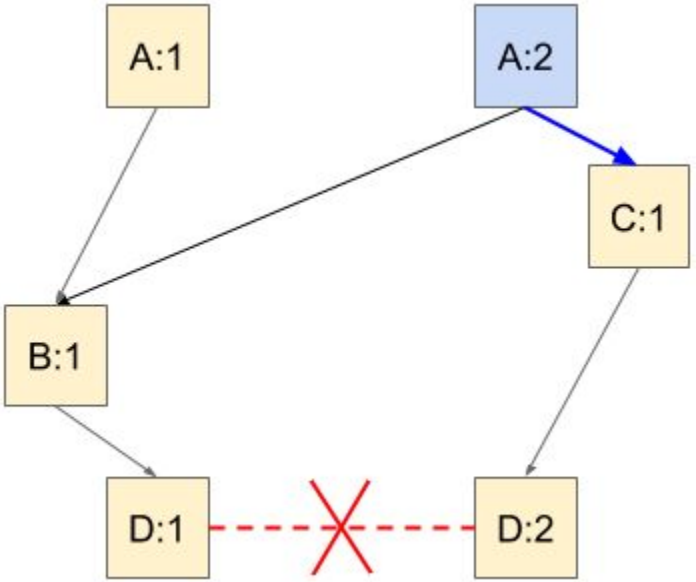
Dependency Graphs



C comes along and decides to depend on version 2 of D because it's the latest and greatest. Everything is still ok at this point.

Example from https://jlbp.dev/what-is-a-diamond-dependency-conflict

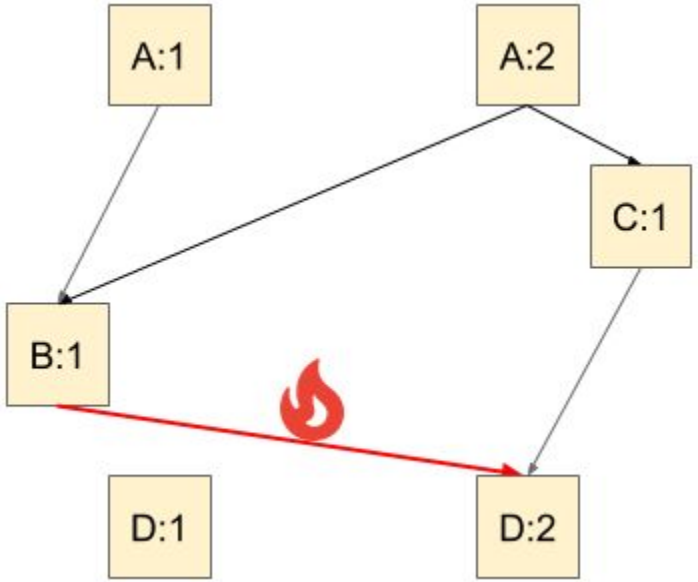
The Diamond Problem



Now A wants to add a dependency on C. This creates a diamond dependency conflict. B:1 can only work with D:1, while C:1 can only work with D:2, so no matter which version of D you choose, the program will blow up.

Example from https://jlbp.dev/what-is-a-diamond-dependency-conflict

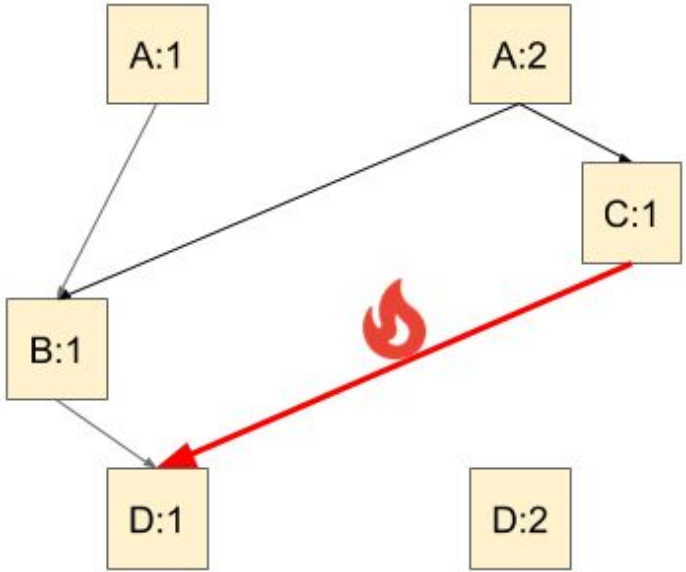
The Diamond Problem



If we choose D:2, then B blows up.

Example from https://jlbp.dev/what-is-a-diamond-dependency-conflict

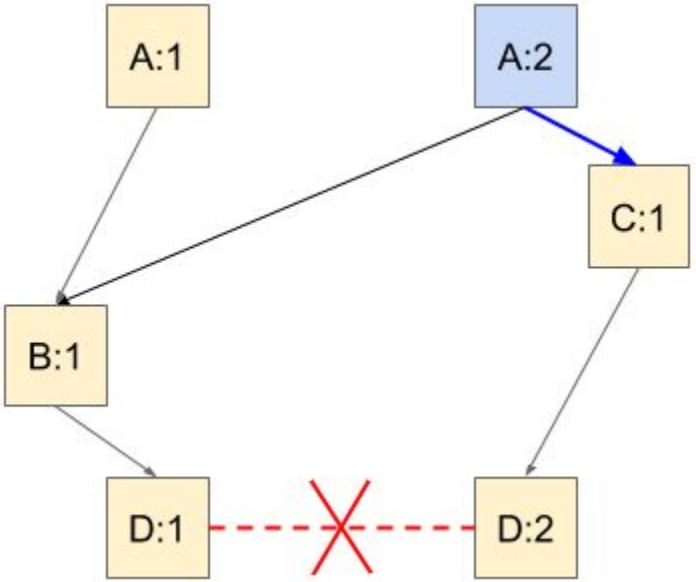
The Diamond Problem



If we choose D:1, then C blows up.

Example from https://jlbp.dev/what-is-a-diamond-dependency-conflict

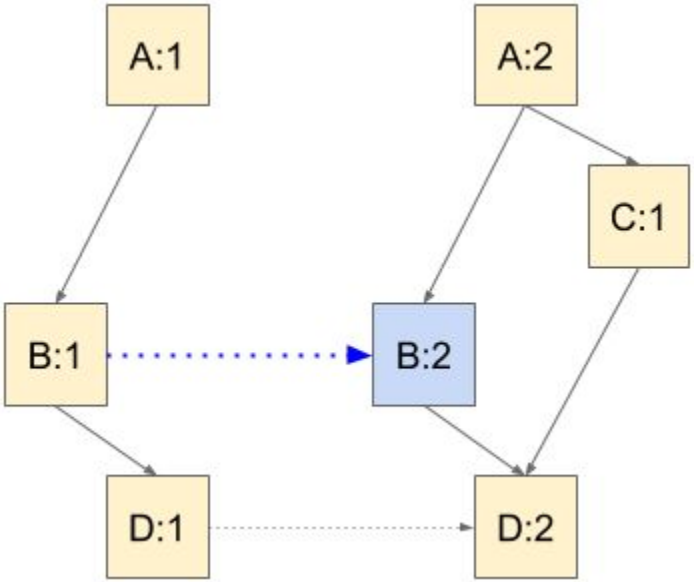
The Diamond Problem



Diamond dependency conflicts are particularly difficult to solve because the changes required to solve them can't be made by either the root of the dependency tree (here, A) or the common library in conflict (here, D); the updates need to be made in one or more intervening libraries (here, B). At the same time, the library that needs updates (library B) has no incentive to make the change.

Example from https://jlbp.dev/what-is-a-diamond-dependency-conflict

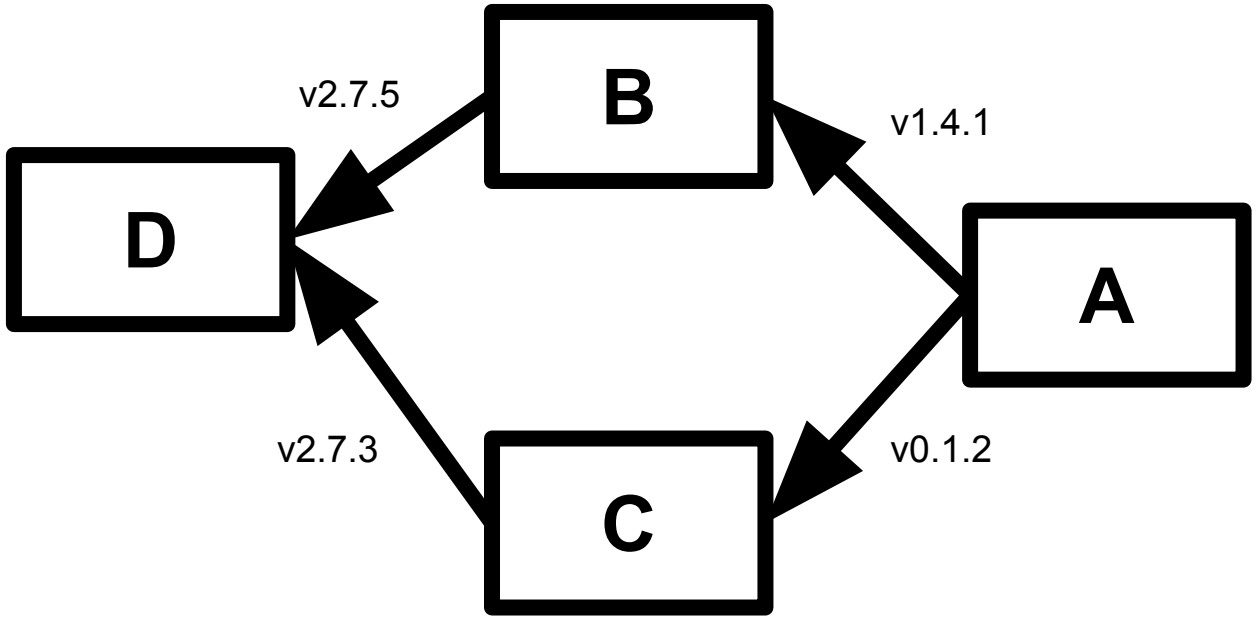
The Diamond Problem



In order to move the ecosystem forward, B needs to create a new version 2, which is compatible with D:2. Then, A can form a successful diamond.

Example from https://jlbp.dev/what-is-a-diamond-dependency-conflict

The Diamond Problem



What now?

Summary: Modules

Encapsulation at Scale

Decide which of many classes or packages to expose

Building a dependency graph between modules

Breaking Changes

Backward Compatible Changes

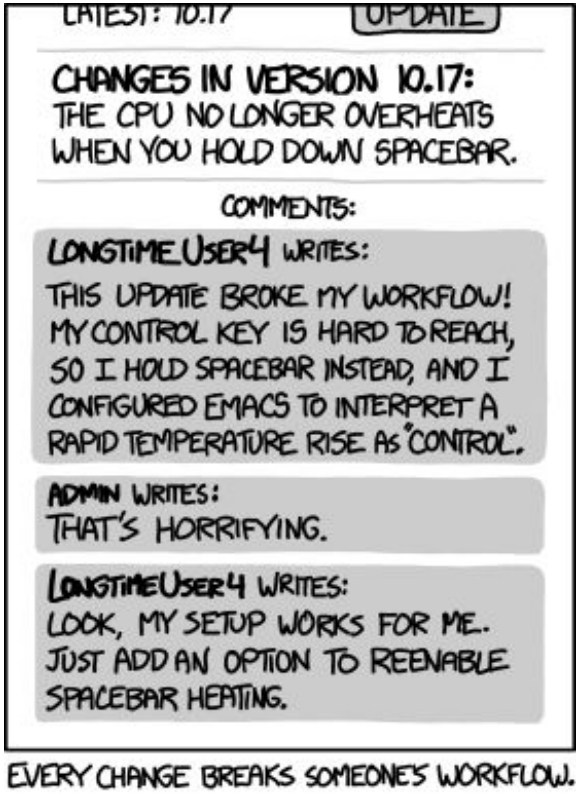
Can add new interfaces, classes

Can add methods to APIs,
but cannot change interface implemented by clients

Can loosen precondition and tighten postcondition,
but no other contract changes

Cannot remove classes, interfaces, methods

Clients may rely on undocumented behavior and even bugs



EVERY CHANGE BREAKS SOMEONE'S WORKFLOW.

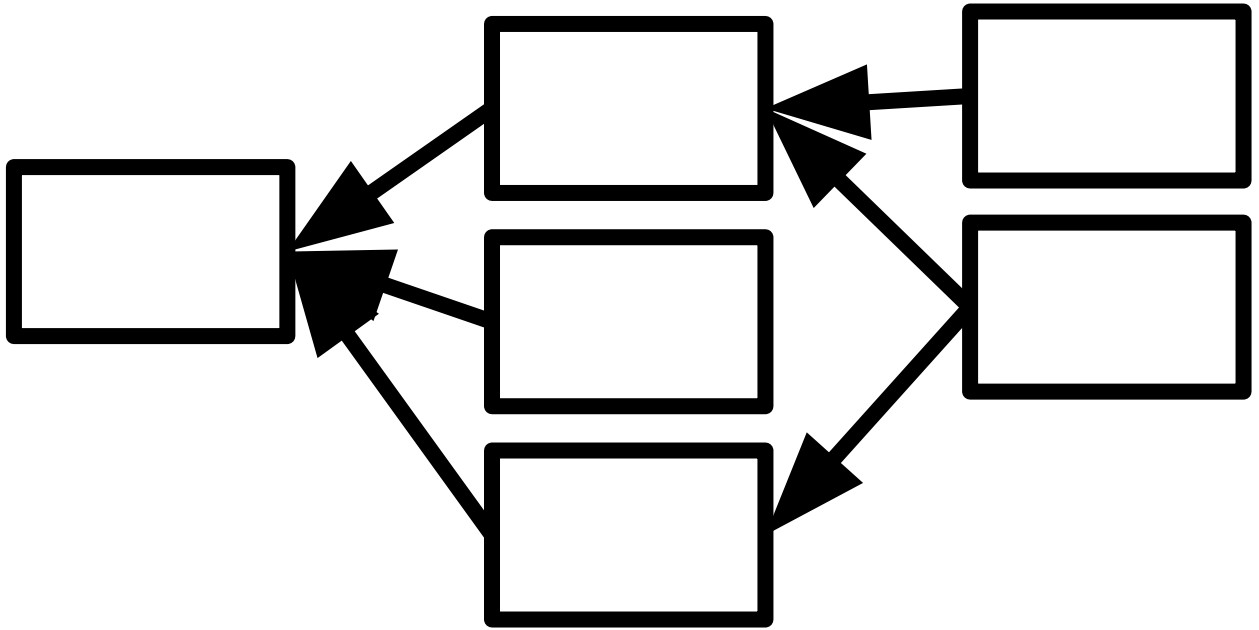
Breaking Changes

Not backward compatible (e.g., renaming/removing method)

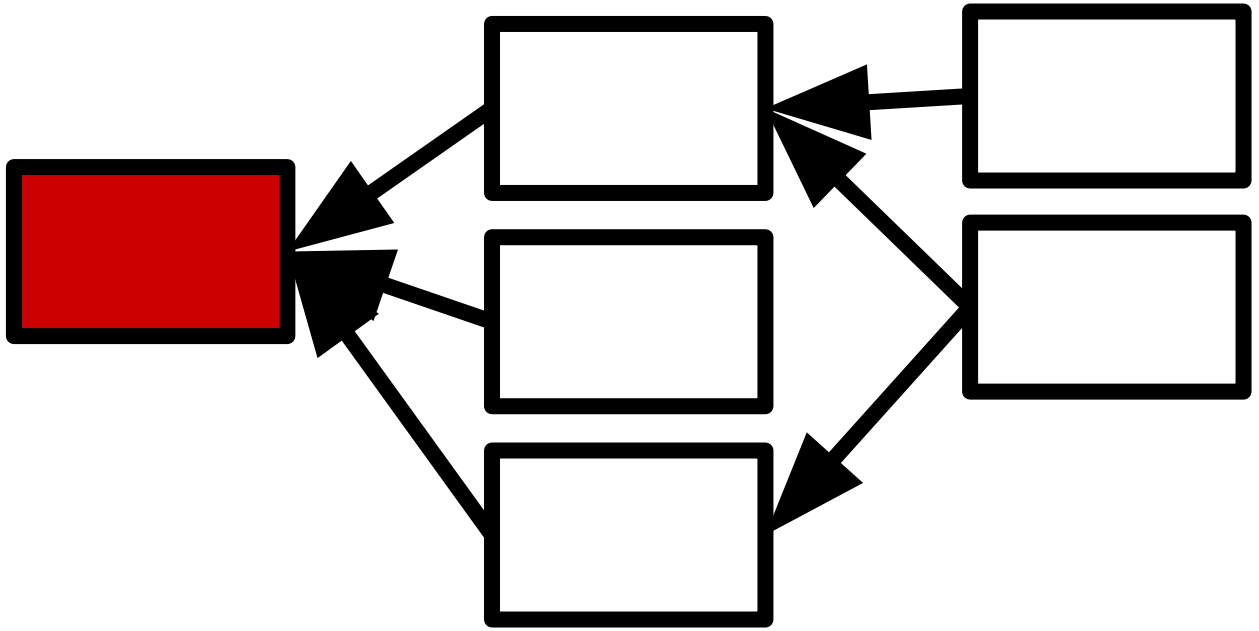
Clients may need to change their implementation when they update

    or even migrate to other library

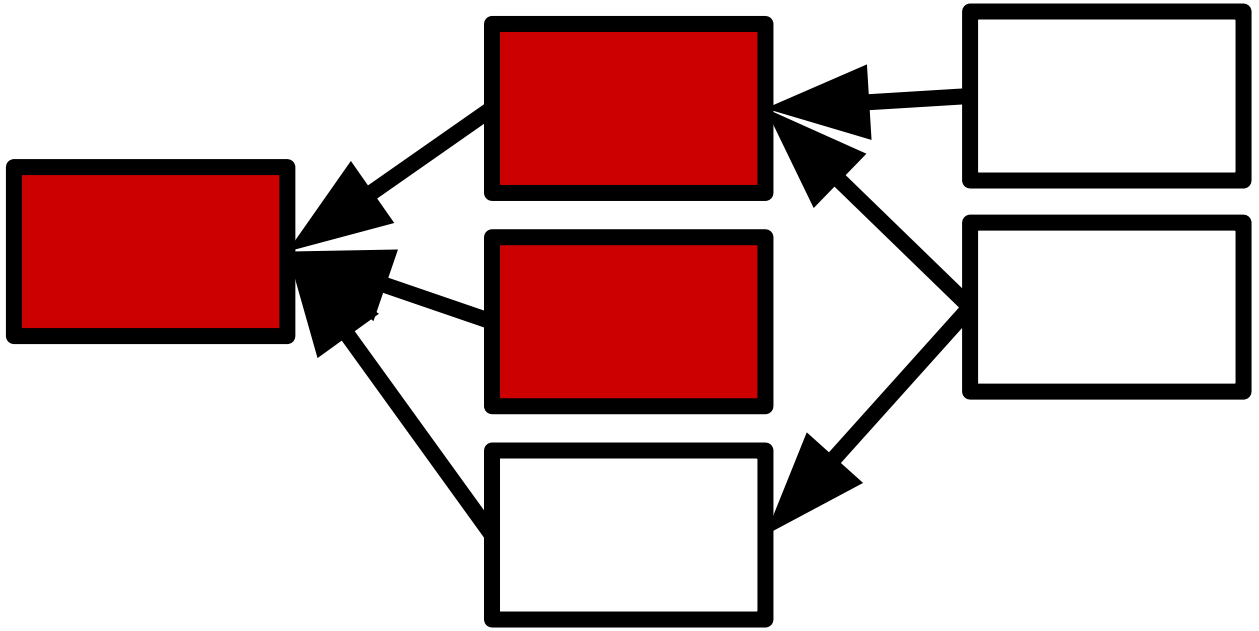
May cause costs for rework and interruption, may ripple through ecosystem



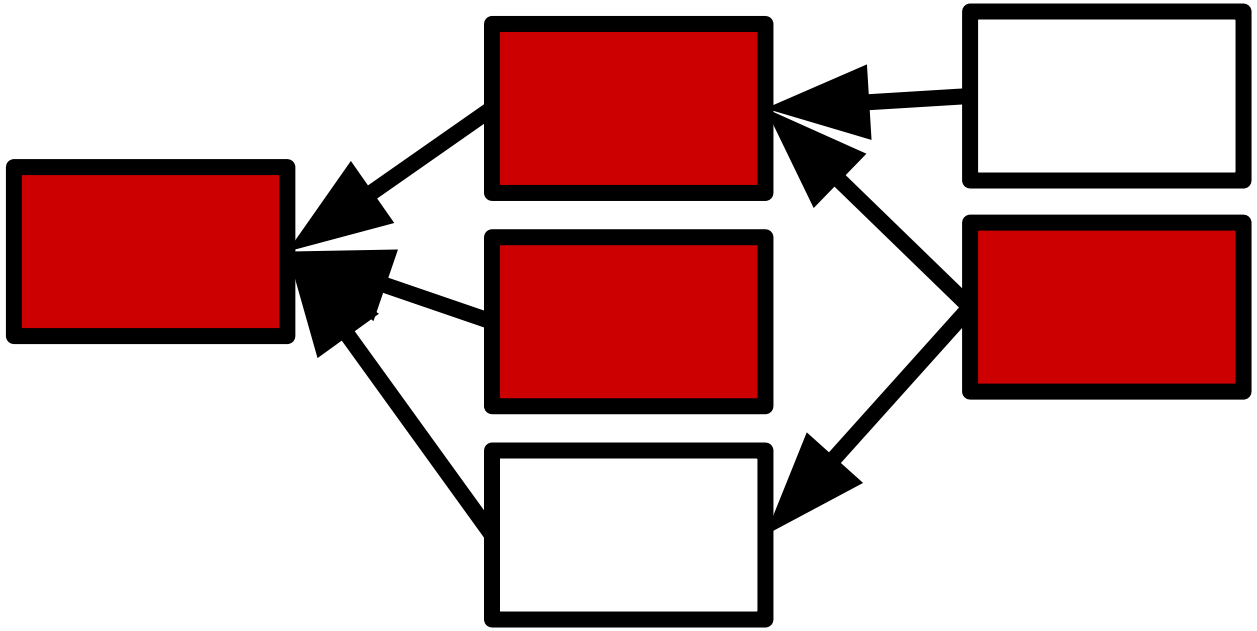
Software Ecosystem



Breaking Changes



Breaking Changes



Breaking Changes

Breaking changes can be hard to avoid

Need better planning? (Parnas' argument)

Requirements and context change

Bugs and security vulnerabilities

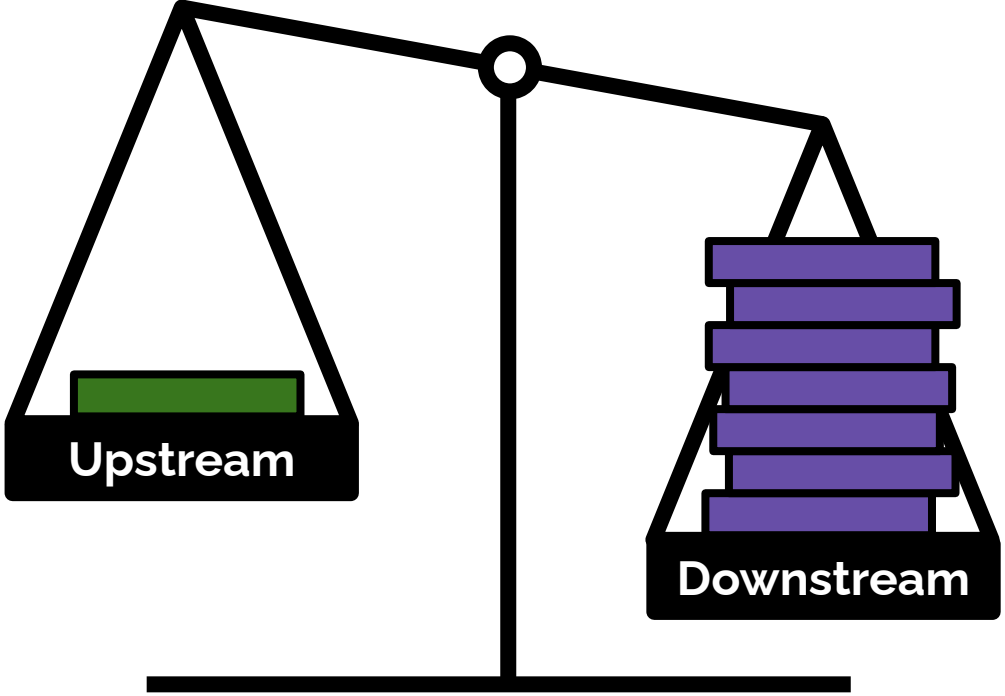
Inefficiencies

Rippling effects from upstream changes

Technical debt, style

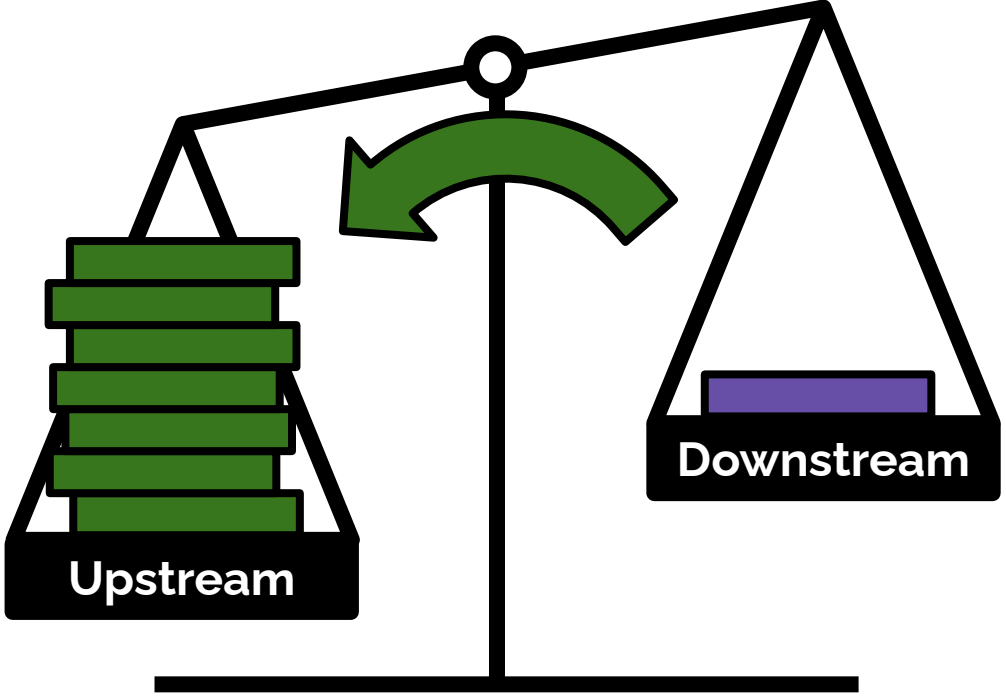
Breaking changes cause costs

But cost can be paid by different participants and can be delayed

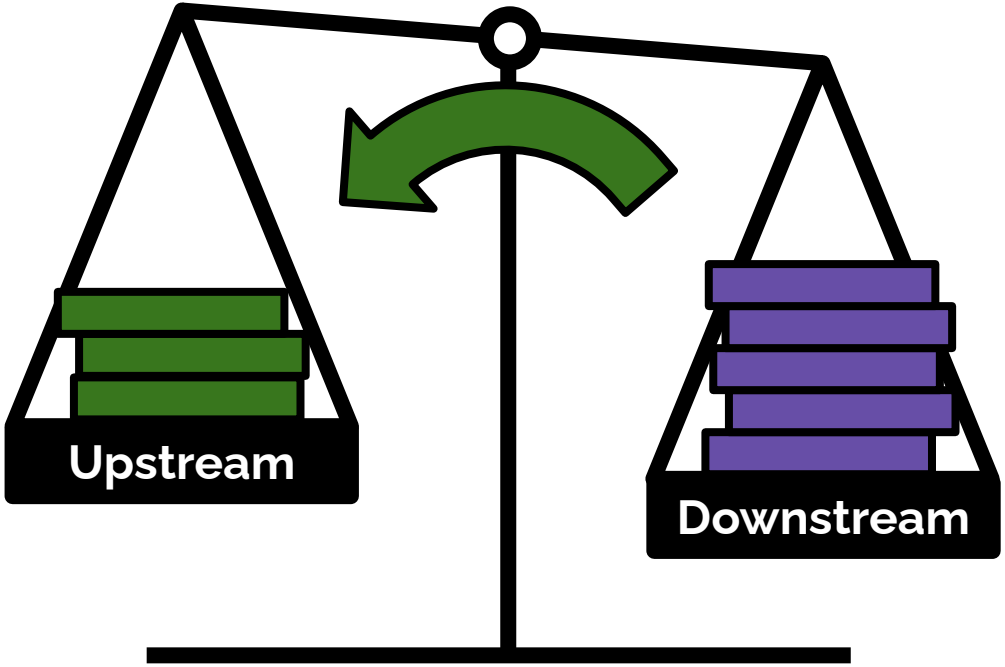


By default, rework and interruption costs for downstream users

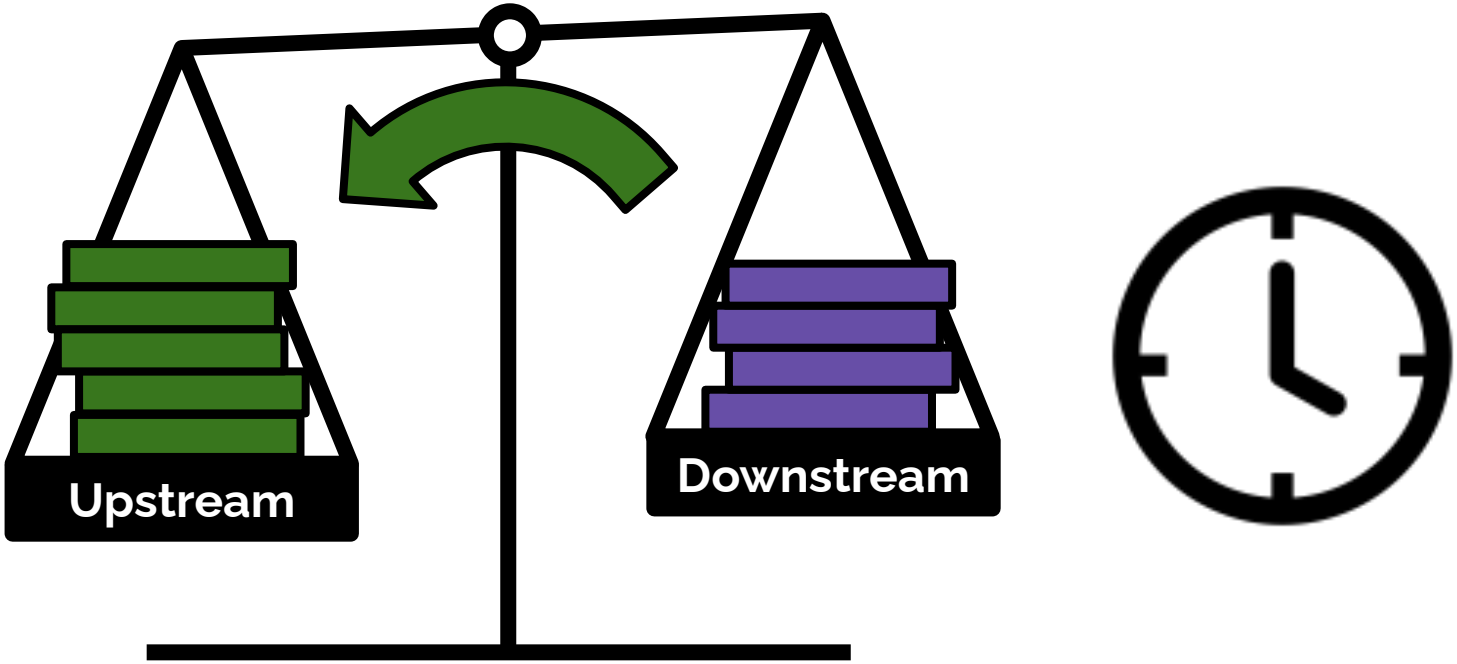
How to reduce costs for downstream users?



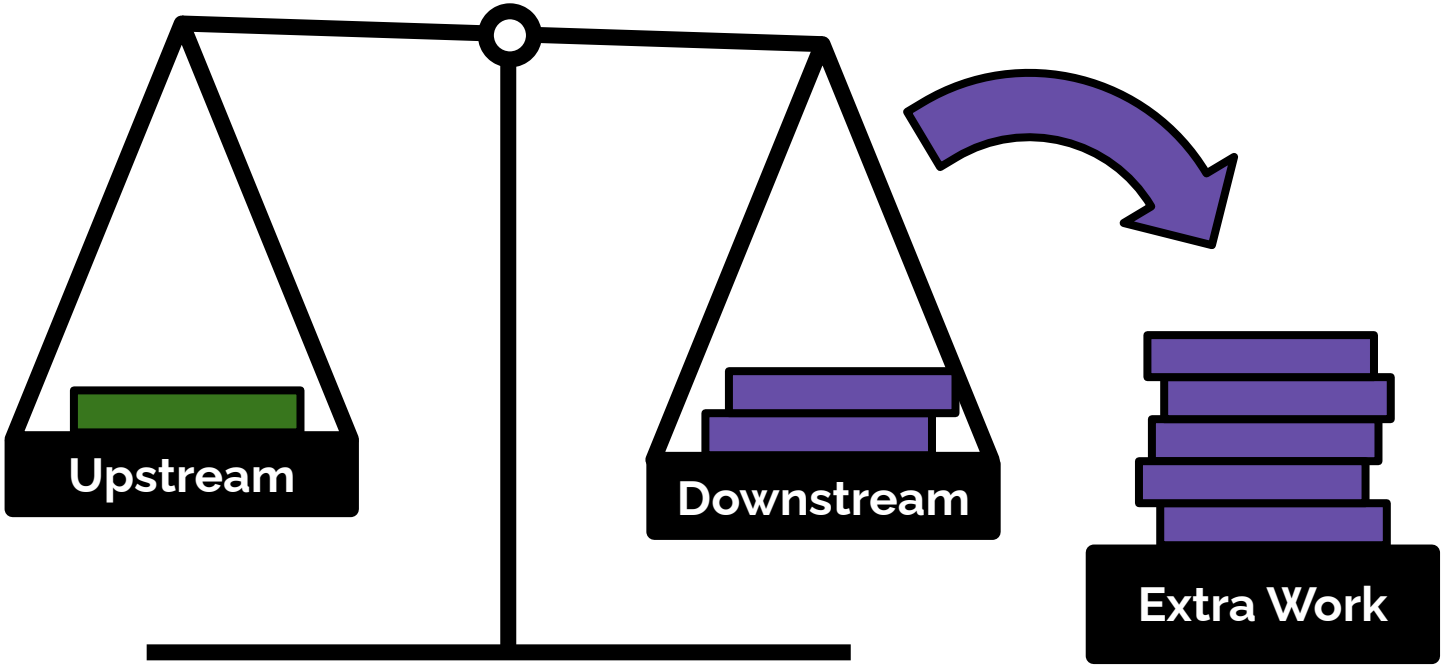
Not making a change
(opportunity costs, technical debt)



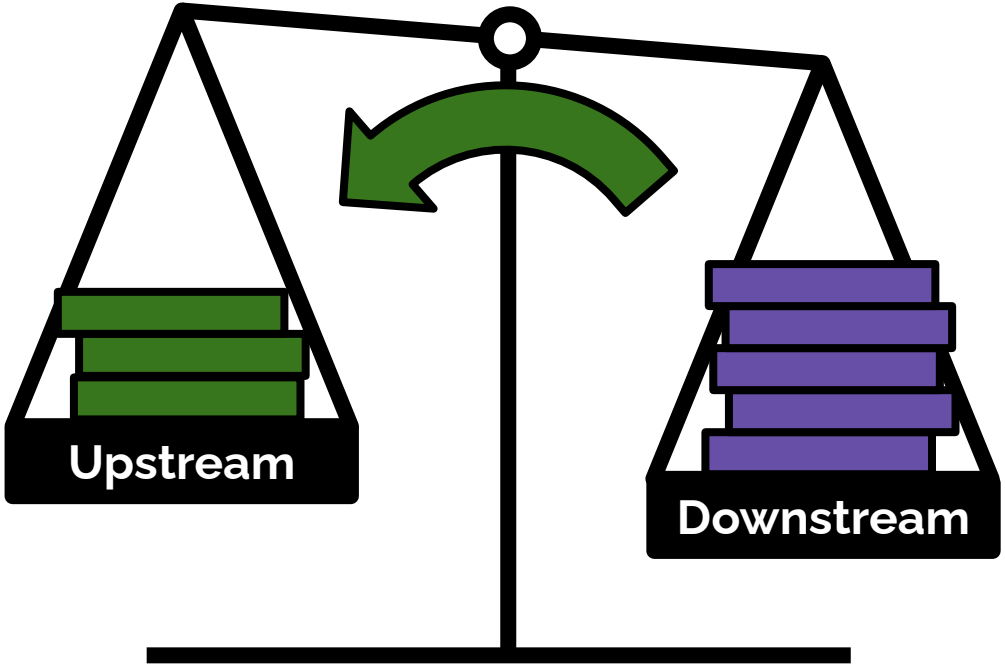
Announcements
Documentation
Migration guide



Parallel maintenance releases
Maintaining old interfaces (deprecation)
Release planning



Avoiding dependencies
Encapsulating from change



Influence development

Semantic Versioning

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

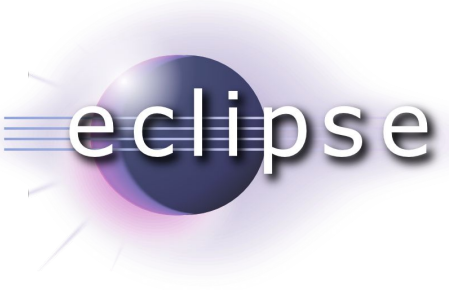
1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add functionality in a backwards compatible manner, and
3. PATCH version when you make backwards compatible bug fixes.

Code status	Stage	Rule	Example version
First release	New product	Start with 1.0.0	1.0.0
Backward compatible bug fixes	Patch release	Increment the third digit	1.0.1
Backward compatible new features	Minor release	Increment the middle digit and reset last digit to zero	1.1.0
Changes that break backward compatibility	Major release	Increment the first digit and reset middle and last digits to zero	2.0.0

https://docs.npmjs.com/about-semantic-versioning

Cost distributions and practices are community dependent


eclipse



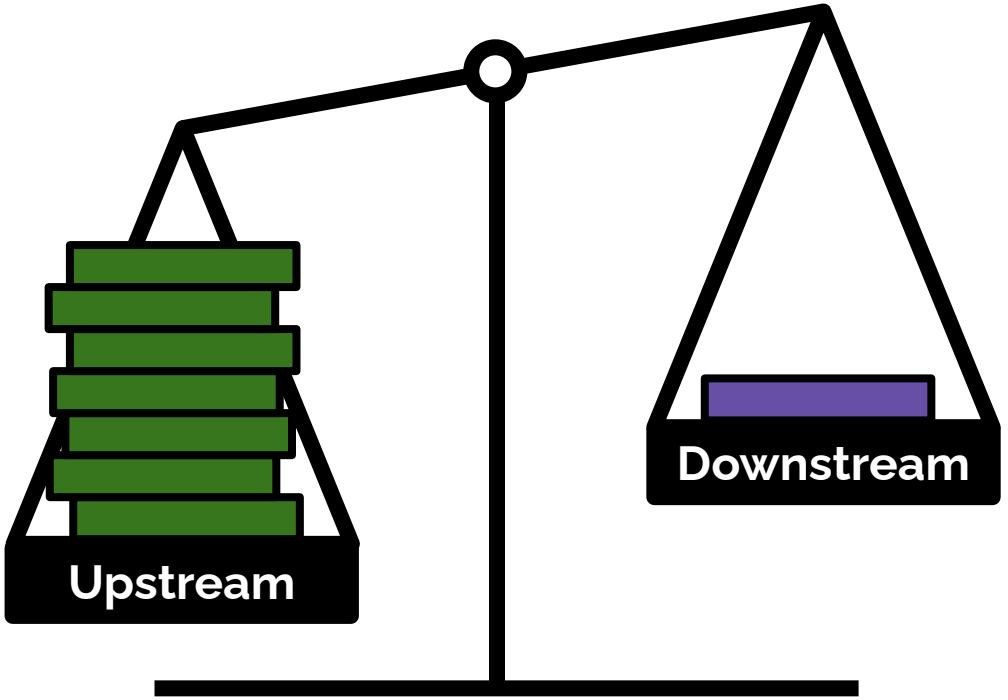
Backward compatibility to reduce costs for **clients**

"API Prime Directive: When evolving the Component API from to release to release, do not break existing Clients"

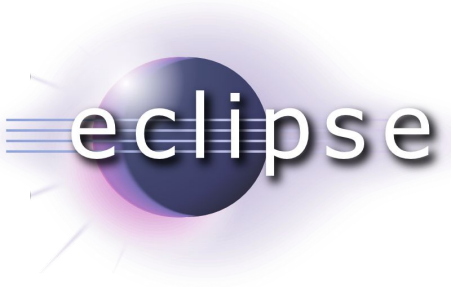
https://wiki.eclipse.org/Evolving_Java-based_APIs

Values

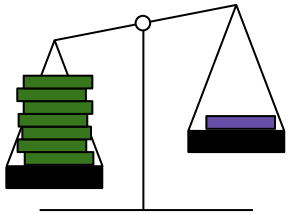
Backward compatibility for clients



Yearly synchronized coordinated releases



Backward compatibility for clients



Willing to accept high costs + opportunity costs

Educational material, workarounds

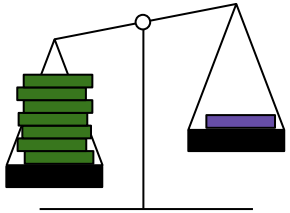
API tools for checking

Coordinated release planning

No parallel releases

Upstream

Backward compatibility for clients

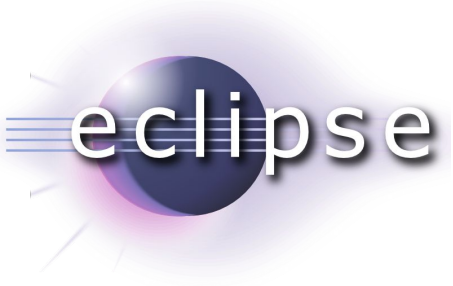


Convenient to use as resource

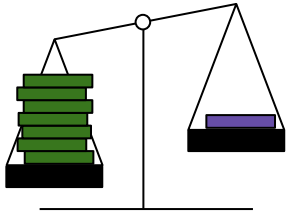
Yearly updates sufficient for many

Stability for corporate users

Downstream



Backward compatibility for clients



Perceived stagnant development and political decision making

Stale platform; discouraging contributors

Coordinated releases as pain points

SemVer prescribed but not followed

Friction

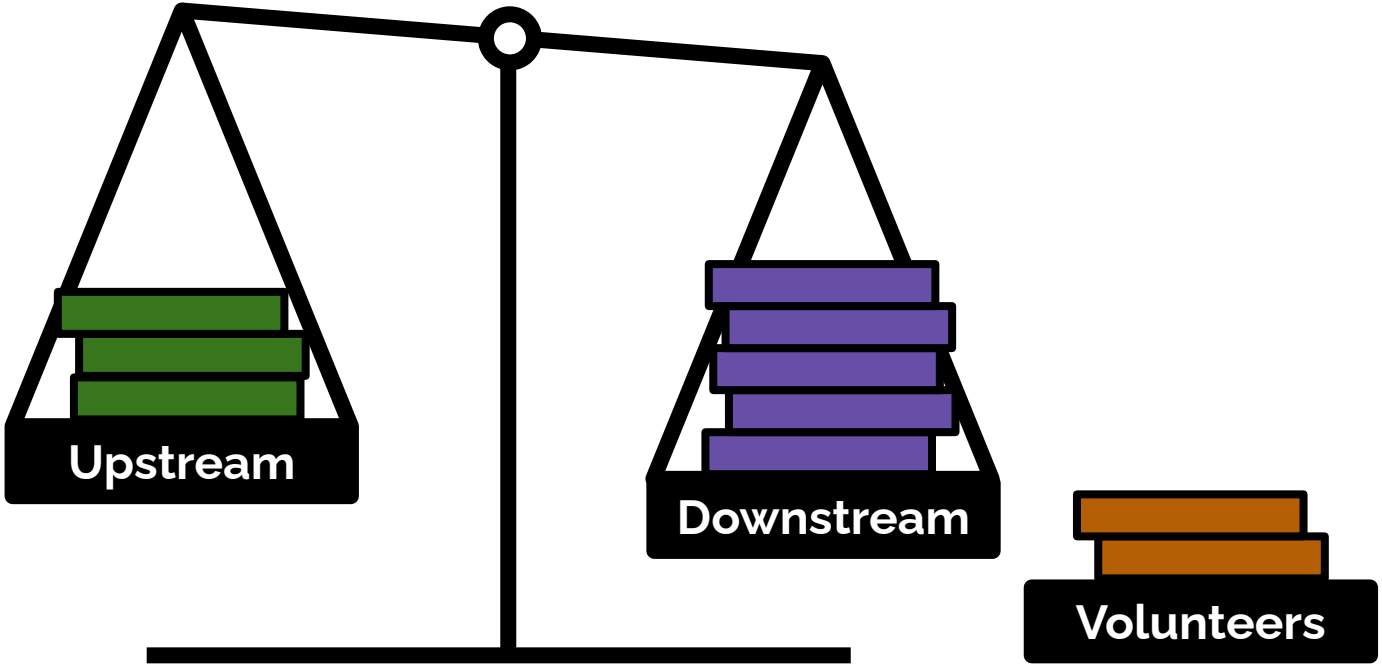
> “Typically, if you have hip things, then you get also people who create new APIs on top ... to create the next graphical editing framework or to build more efficient text editors. ... And these things don’t happen on the Eclipse platform anymore.”

Ease for **end users** to install and update packages

"CRAN primarily has the academic users in mind, who want timely access to current research" [R10]

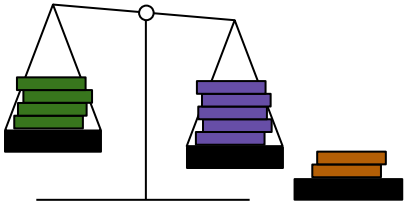
Values

Timely access to current research for end users



Continuous synchronization, ~1 month lag

Timely access to current research for end users



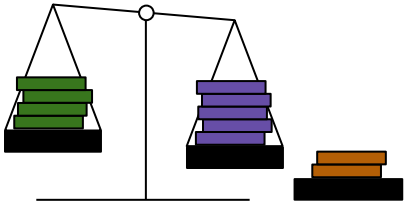
Snapshot consistency within the ecosystem (not outside)

Reach out to affected downstream developers: resolve before release

Gatekeeping: reviews and automated checking against downstream tests

Upstream

Timely access to current research for end users



Waiting for emails, reactive monitoring

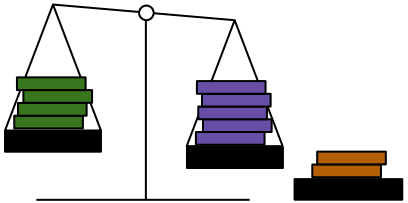
Urgency when upstream package updates

Dependency = collaboration

Aggressive reduction of dependencies, code cloning

Downstream

Timely access to current research for end users



Urgency and reacting to updates as burden vs. welcoming collaboration

Gatekeeping works because of prestige of being in repository

Updates can threaten scientific reproducibility

Friction

“And then I need to [react to] some change ... and it might be a relatively short timeline of two weeks or a month. And that's difficult for me to deal with, because I try to sort of focus one project for a couple weeks at a time so I can remain productive.”

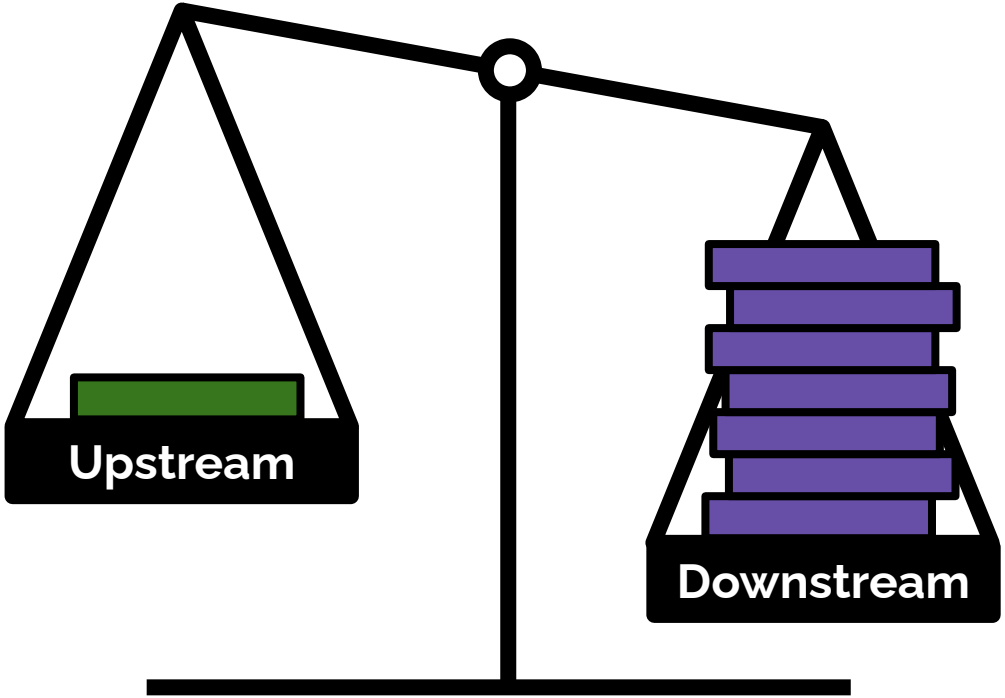

node
JS
TM

Easy and fast for **developers** to publish and use packages

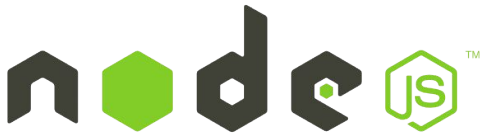
Open to rapid change,
no gate keeping,
experimenting with APIs until they are right

Values

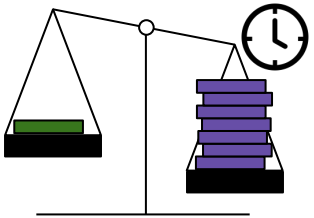
Easy and fast to publish and use for developers



Decoupled pace, update at user's discretion



Easy and fast to publish and use for developers



Breaking changes easy

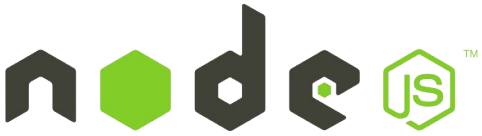
More common to remove technical debt, fix APIs

Signaling intention with SemVer

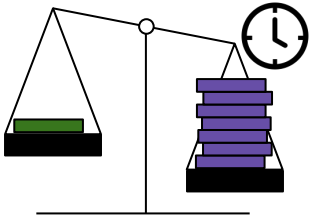
No central release planning

Parallel releases more common

Upstream



Easy and fast to publish and use for developers

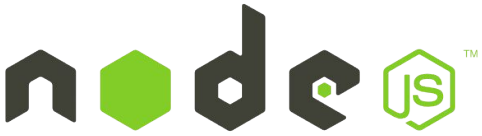


Technology supports using old + mixed revisions; decouples upstream and downstream pace

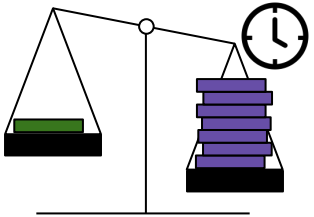
Choice to stay up to date

Monitoring with social mechanisms and tools (e.g., dependabot)

Downstream



Easy and fast to publish and use for developers



Rapid change requires constant maintenance

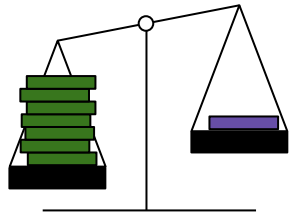
Emphasis on tools and community, often grassroots

Friction

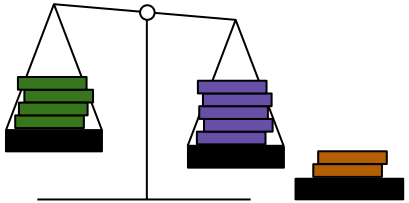
“Last week’s tutorial is out of date today.”

Contrast

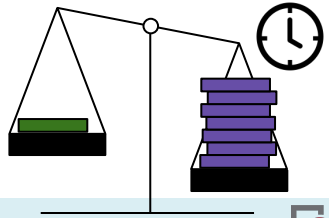
Backward compatibility for clients



Timely access to current research for end users



Easy and fast to publish/use for developers



How to Break an API?

In Eclipse, you don't.

In CRAN, you reach out to affected downstream developers.

In Node.js, you increase the major version number.

Summary

Heavy reliance on dependencies

- Package managers and module systems help organize
- Manage costs and risks of dependencies

Next time:

Modularly organize systems at scale

- Modules
- Distributed systems
- Microservices
- Event-based systems / stream processing

Testing with Stubs and Chaos Engineering

Bonus: Cost of Dependencies

Recall: Ever looked at NPM Install's output?

```
added 2110 packages from 770 contributors and audited 2113 packages in 141.9

158 packages are looking for funding
  run `npm fund` for details

found 27 vulnerabilities (8 moderate, 18 high, 1 critical)
  run `npm audit fix` to fix them, or `npm audit` for details
```

Recall: Ever looked at NPM Install's output?

```
npm WARN deprecated babel-eslint@10.1.0: babel-eslint is now @babel/eslint-parser. This package will no longer receiv
updates.
npm WARN deprecated chokidar@2.1.8: Chokidar 2 will break on node v14+. Upgrade to chokidar 3 with 15x less dependenc
s.
npm WARN deprecated svgo@1.3.2: This SVGO version is no longer supported. Upgrade to v2.x.x.
npm WARN deprecated querystring@0.2.1: The querystring API is considered Legacy. new code should use the URLSearchPar
s API instead.
npm WARN deprecated @hapi/joi@15.1.1: Switch to 'npm install joi'
npm WARN deprecated rollup-plugin-babel@4.4.0: This package has been deprecated and is no longer maintained. Please u
 @rollup/plugin-babel.
npm WARN deprecated fsevents@1.2.13: fsevents 1 will break on node v14+ and could be using insecure binaries. Upgrade
o fsevents 2.
npm WARN deprecated uuid@3.4.0: Please upgrade  to version 7 or higher.  Older versions may use Math.random() in cert
n circumstances, which is known to be problematic.  See https://v8.dev/blog/math-random for details.
npm WARN deprecated querystring@0.2.0: The querystring API is considered Legacy. new code should use the URLSearchPar
s API instead.
npm WARN deprecated sane@4.1.0: some dependency vulnerabilities fixed, support for node < 10 dropped, and newer ECMAS
ipt syntax/features added
npm WARN deprecated flatten@1.0.3: flatten is deprecated in favor of utility frameworks such as lodash.
npm WARN deprecated urix@0.1.0: Please see https://github.com/lydell/urix#deprecated
npm WARN deprecated @hapi/bourne@1.3.2: This version has been deprecated and is no longer supported or maintained
```

Monitoring for Vulnerabilities

Dependency manager helps knowing what dependencies are used (“bill of materials”)

Various tools scan for known vulnerabilities -- use them

Have a process

Many false positive alerts, not exploitable

Recommended reading:
https://republicans-oversight.house.gov/wp-content/uploads/2018/12/Equifax-Report.pdf

Supply Chain Attacks more common

Intentionally injecting attacks in packages

- Typosquatting: expres
- Malicious updates: us-parser-js

Review all packages? All updates?

Sandbox applications? Sandbox packages?

Using a Dead Dependency?

No more support?

No fixes to bugs and vulnerabilities?

What now?

Open Source Health and Sustainability

Predict which packages will be maintained next year?

- Indicators?

Motivation of maintainers?

Who funds open source?

Commercial dependencies? Commercial support?