Principles of Software Construction

API Design (Part 2)

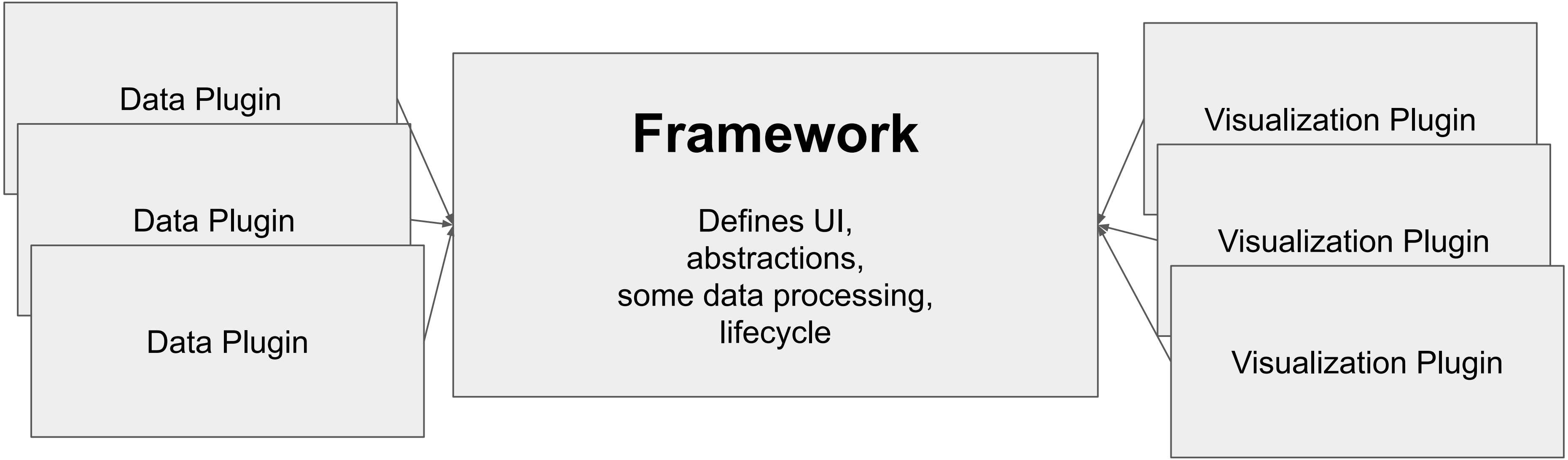
Christian Kästner Vincent Hellendoorn
(With slides from Josh Bloch)

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Midterm Recap

Homework 6 Released



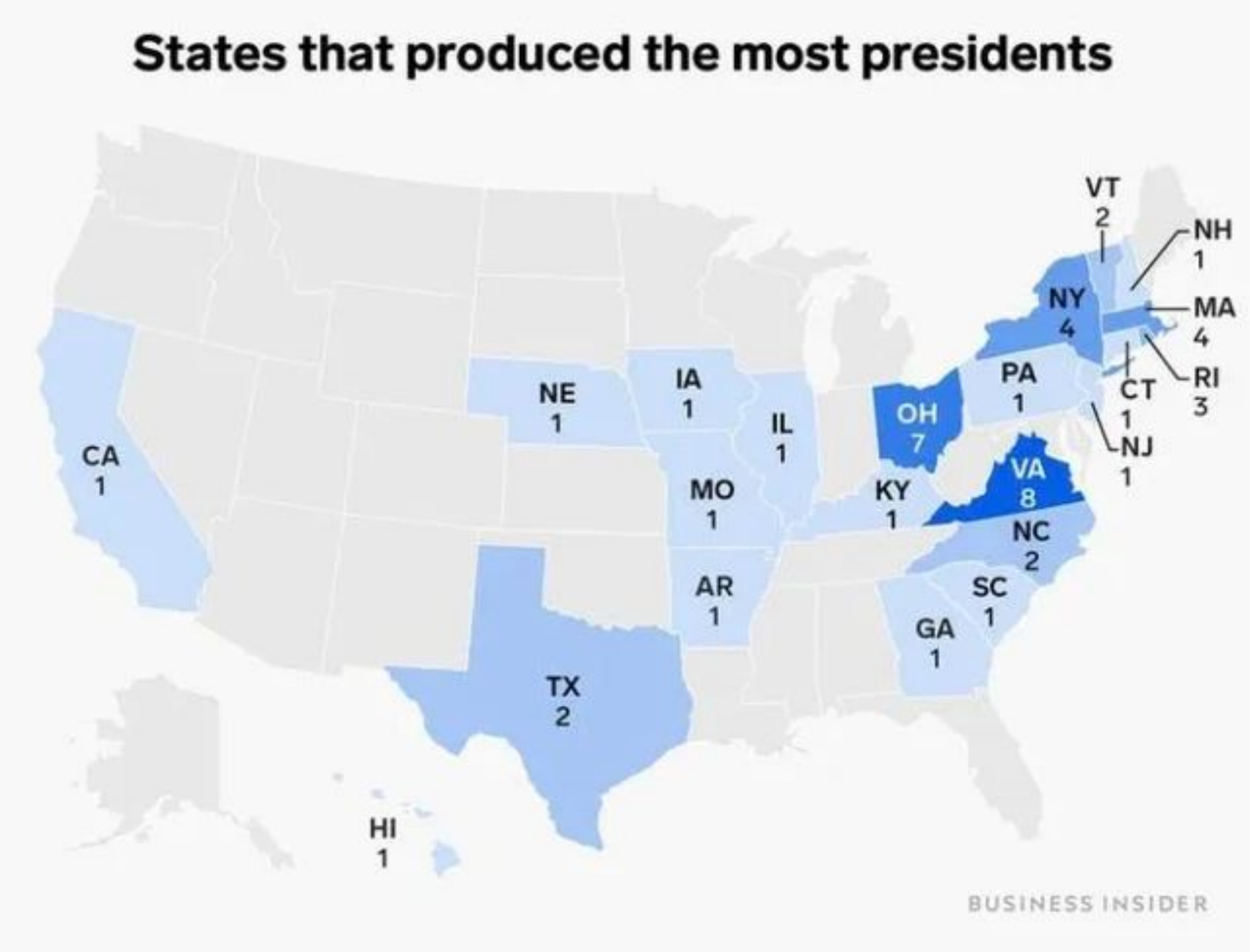
HW6: Map-Based Data Visualizations?

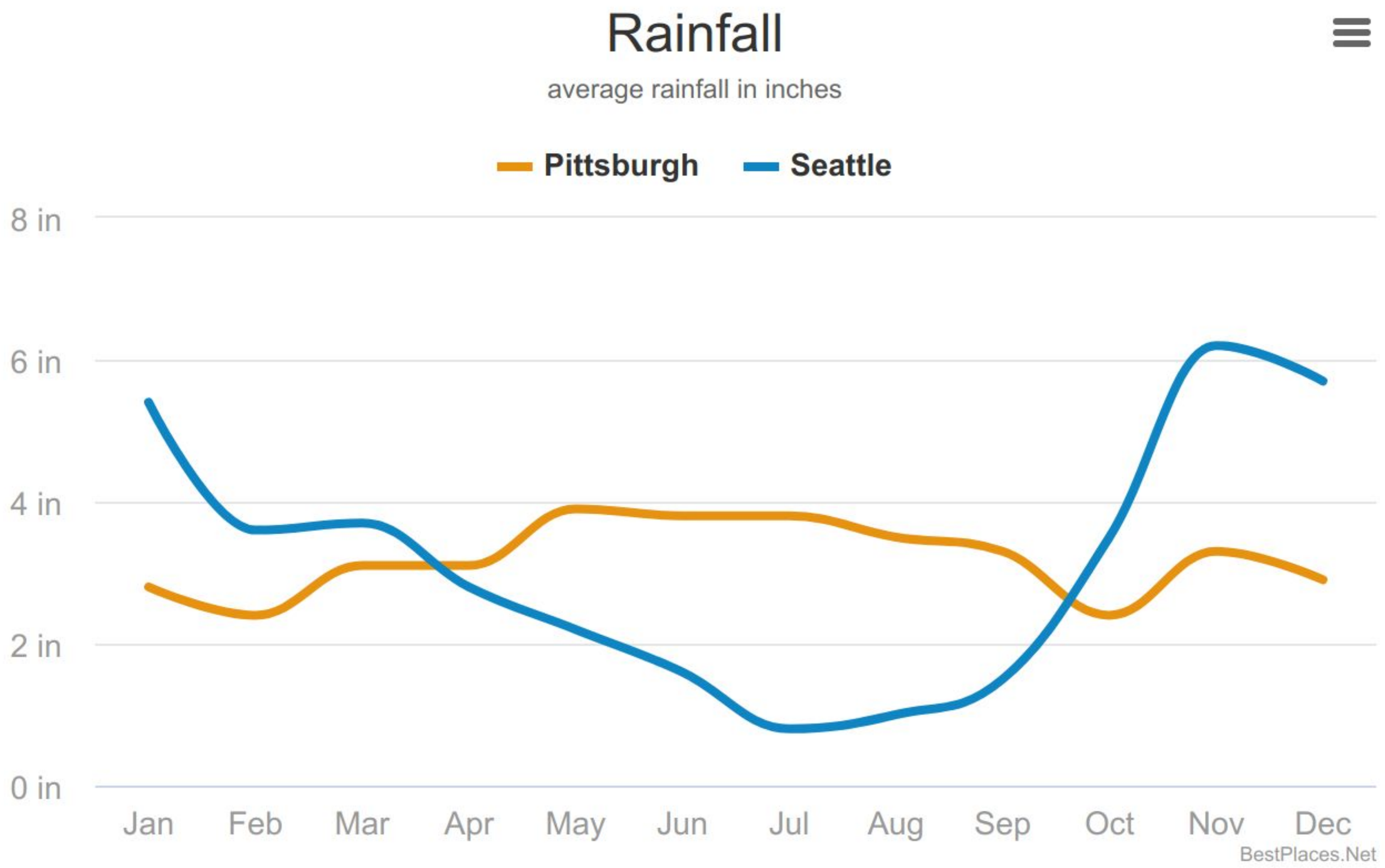
State, county, or country data

Data from many sources

Visualization as map image, table, google maps

Animations for time series data




Rainfall
average rainfall in inches
Pittsburgh
Seattle
8 in
6 in
4 in
2 in
0 in
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
BestPlaces.Net

A Word on Teamwork

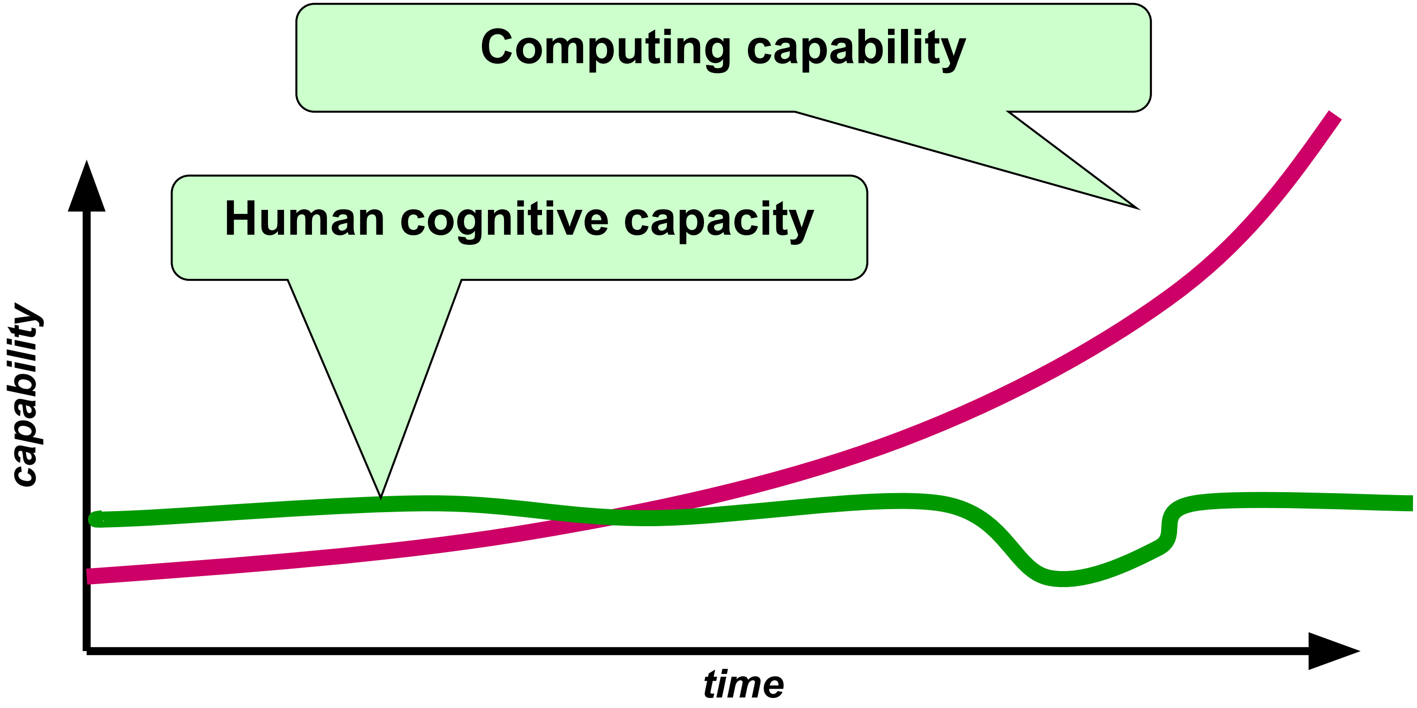
Teamwork

Teamwork essential in software projects

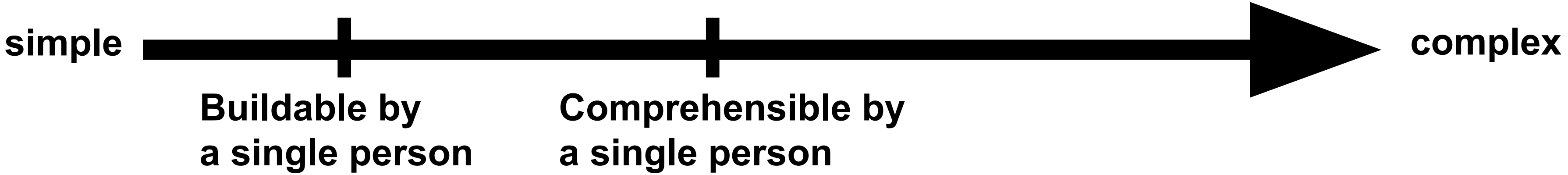
Teamwork needed to scale available work and available skills

Teamwork is a key motivation for
- design for understandability, documentation, etc

The limits of exponentials



Building Complex Systems



- Division of Labor
- Division of Knowledge and Design Effort
- Reuse of Existing Implementations

Student Teams

Different abilities, different motivations, different commitment, different habits

Teams should self-organize

Agree on how to communicate in the team: Email? Text?

Plan, assign responsibilities, and write them down!

Replan and adjust, help each other, communicate frequently

Team Citizenship

Not everybody will contribute equally to every milestone -- that's okay

But be good team citizen!

Be responsive and responsible

Stick to commitments, work on assigned tasks

When problems, reach out, replan, communicate early, be proactive

Common Sources of Team Conflict

- Different team members have different working patterns and communication preferences
  - e.g., start early vs close to deadline
  - e.g., plan ahead vs try and error
  - e.g., react to every notification vs reduce distractions and read email once a day
  - discuss and set explicit expectations; talk about conflicts
- Different abilities, unexpected difficulties
  - work in pairs, plan time for rework and integration
  - replan, contribute to teams in different ways
  - work around it, it's the team's responsibility
- Unreliable team members, poor team citizenship
  - e.g., not starting the work in agreed time, not responding, not attending meetings
  - have written clear deliverables with deadlines
  - talk about it within team, talk to course staff, report poor team citizenship -> grade adjustment

Reporting Poor Team Citizenship

Form on Canvas at end of semester

Provide evidence (point to plan.md, git logs, communication)

We'll adjust grading if needed

Practice Teamwork Beyond This Course

Teamwork not emphasis in this course

Many other courses are very explicit about teaching and supporting teamwork in large groups

- Examples: 17-313 (Foundations of SE), 17-445 (ML in Production), 17-356 (SE for Startups), many capstone courses

Today: API Design (continued)

Where we are

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

Outline

- Introduction to API Design
- The Process of API Design
- Information Hiding and Minimizing Conceptual Weight
- Naming
- Other API Suggestions
- Breaking Changes

Naming

Names Matter – API is a little language

Naming is perhaps the single most important factor in API usability

- Primary goals
  - **Client code should read like prose** (“easy to read”)
  - **Client code should mean what it says** (“hard to misread”)
  - **Client code should flow naturally** (“easy to write”)
- To that end, names should:
  - be largely self-explanatory
  - leverage existing knowledge
  - interact harmoniously with language and each other

Good and Bad Examples?

Choosing names easy to read & write

- Choose key nouns carefully!
  - Related to finding good abstractions, which can be hard
  - If you *can't* find a good name, it's generally a bad sign
- If you get the key nouns right, other nouns, verbs, and prepositions tend to choose themselves
- Names can be literal or metaphorical
  - Literal names have literal associations: e.g., **matrix** suggests inverse, determinant, eigenvalue, etc.
  - Metaphorical names enable reasoning by analogy: e.g., **mail** suggests send, cc, bcc, inbox, outbox, folder, etc.

Vocabulary consistency

- Use words consistently throughout your API
  - Never use the same word for multiple meanings
  - Never use multiple words for the same meaning
  - i.e., words should be *isomorphic* to meanings
  - Avoid abbreviations
- Build *domain model* or glossary!

Discuss these names

- `get_x()` vs `getX()`
- `Timer` vs `timer`
- `isEnabled()` vs. `enabled()`
- `computeX()` vs. `generateX()`?
- `deleteX()` vs. `removeX()`?

Good names drive good design

- Be consistent
  - `computeX()` vs. `generateX()`?
  - `deleteX()` vs. `removeX()`?
- Avoid cryptic abbreviations
  - Good: `Font, Set, PrivateKey, Lock, ThreadFactory, TimeUnit, Future<T>`
  - Bad: `DynAnyFactoryOperations, _BindingIteratorImplBase, ENCODING_CDR_ENCAPS, OMGVMCID`

Names drive development, for better or worse

- Good names drive good development
- Bad names inhibit good development
- Bad names result in bad APIs unless you take action
- **The API talks back to you. Listen!**

Another way names drive development

- Names may remind you of another API
- Consider **copying** its vocabulary and structure
- People who know other API will have an easy time learning yours
- You may be able to develop it more quickly
- You may be able to use types from the other API
- You may even be able to share implementation

Avoid abbreviations except where customary

- Back in the day, storage was scarce & people abbreviated everything
  - Some continue to do this by force of habit or tradition
- Ideally, use complete words
- But sometimes, names just get too long
  - If you must abbreviate, do it tastefully
  - **No excuse for cryptic abbreviations**
- Of course you should use `gcd`, `Url`, `cos`, `mba`, etc.

Grammar is a part of naming too

- Nouns for classes
  - `BigInteger`, `PriorityQueue`
- Nouns or adjectives for interfaces
  - `Collection`, `Comparable`
- Nouns, linking verbs or prepositions for non-mutative methods
  - `size`, `isEmpty`, `plus`
- Action verbs for mutative methods
  - `put`, `add`, `clear`

Names should be regular – strive for symmetry

- If API has 2 verbs and 2 nouns, support all 4 combinations, unless you have a very good reason not to
- Programmers will try to use all 4 combinations, they will get upset if the one they want is missing

```
addRow       removeRow
addColumn    removeColumn
```

What’s wrong here?

```
public class Thread implements Runnable {
    // Tests whether current thread has been interrupted.
    // Clears the interrupted status of current thread.
    public static boolean interrupted();
}
```

What's wrong here?

```
var timeoutID = setTimeout(function[, delay, arg1, arg2, ...]);
var timeoutID = setTimeout(function[, delay]);
var timeoutID = setTimeout(code[, delay]);

setTimeout(function () {
    // nice fast code here
},2000) // run after 2 seconds

setTimeout(`writeResults(${query.str})`, 100)
```

Don’t mislead your user

- Names have implications
- **Don’t violate *the principle of least astonishment***
- Can cause unending stream of subtle bugs

```
public static boolean interrupted()
```

Tests whether the current thread has been interrupted. The interrupted status of the thread is cleared by this method....

Don’t lie to your user outright

- **Name method for what it does, not what you wish it did**
- **If you can’t bring yourself to do this, fix the method!**
- Again, ignore this at your own peril

```
public long skip(long n) throws IOException
```

Skips over and discards n bytes of data from this input stream. The skip method may, for a variety of reasons, end up skipping over some smaller number of bytes, possibly 0. This may result from any of a number of conditions; reaching end of file before n bytes have been skipped is only one possibility. The actual number of bytes skipped is returned…

Use consistent parameter ordering

- An egregious example from C:
  - `char* strncpy(char* dest, char* src,  size_t n);`
  - `void    bcopy(void* src,  void* dest, size_t n);`

Use consistent parameter ordering

- An egregious example from C:
  - `char* strncpy(char* dest, char* src,  size_t n);`
  - `void    bcopy(void* src,  void* dest, size_t n);`
- Some good examples:
  - `java.util.Collections` – first parameter always collection to be modified or queried
  - `java.util.concurrent` – time always specified as long delay, TimeUnit unit

Good naming takes time, but it’s worth it

- Don’t be afraid to spend hours on it; I do.
  - And I still get the names wrong sometimes
- Don’t just list names and choose
  - Write out realistic client code and compare
- Discuss names with colleagues; it really helps.

Other API Design Suggestions

Apply principles of user-centered design

e.g., "Principles of Universal Design"

- Equitable use: Design is useful and marketable to people with diverse abilities
- **Flexibility in use:** Design accommodates a wide range of individual preferences
- **Simple and intuitive use**: Use of the design is easy to understand
- Perceptible information: Design communicates necessary information effectively to user
- Tolerance for error
- Low physical effort
- Size and space for approach and use

Principle: Favor composition over inheritance

```
// A Properties instance maps Strings to Strings
public class Properties extends HashTable {
    public Object put(Object key, Object value);
    …
}
public class Properties {
    private final HashTable data = new HashTable();
    public String put(String key, String value) {
        data.put(key, value);
    }
    …
```

Principle: Minimize mutability

- Classes should be immutable unless there's a good reason to do otherwise
  - Advantages: simple, thread-safe, reusable
  - Disadvantage: separate object for each value

Bad: `Date`, `Calendar`

Good: `LocalDate`, `Instant`, `TimerTask`

Antipattern: Long lists of parameters

- Especially with repeated parameters of the same type

```
HWND CreateWindow(LPCTSTR lpClassName, LPCTSTR lpWindowName,
  DWORD dwStyle, int x, int y, int nWidth, int nHeight,
  HWND hWndParent, HMENU hMenu, HINSTANCE hInstance,
  LPVOID lpParam);
```

- Long lists of identically typed params harmful
  - Programmers transpose parameters by mistake; programs still compile and run, but misbehave
- Three or fewer parameters is ideal
- Techniques for shortening parameter lists: Break up method, parameter objects, Builder Design Pattern

What's wrong here?

```
// A Properties instance maps Strings to Strings
public class Properties extends HashTable {
    public Object put(Object key, Object value);

    // Throws ClassCastException if this instance
    // contains any keys or values that are not Strings
    public void save(OutputStream out, String comments);
}
```

Principle: Fail fast

- Report errors as soon as they are detectable
  - Check preconditions at the beginning of each method
  - Avoid dynamic type casts, run-time type-checking

```
// A Properties instance maps Strings to Strings
public class Properties extends HashTable {
    public Object put(Object key, Object value);

    // Throws ClassCastException if this instance
    // contains any keys or values that are not Strings
    public void save(OutputStream out, String comments);
}
```

Throw exceptions on exceptional conditions

- Don’t force client to use exceptions for control flow
- Conversely, don’t fail silently

```
void processBuffer (ByteBuffer buf) {
  try {
    while (true) {
      buf.get(a);
      processBytes(a, CHUNK_SIZE);
    }
  } catch (BufferUnderflowException e) {
    int remaining = buf.remaining();
    buf.get(a, 0, remaining);
    processBytes(a, remaining);
  }
}
```

```
ThreadGroup.enumerate(Thread[] list)

// fails silently: “if the array is too
   short to hold all the threads, the
   extra threads are silently ignored”
```

Java: Avoid checked exceptions if possible

- Overuse of checked exceptions causes boilerplate

```
try {
  Foo f = (Foo) g.clone();
} catch (CloneNotSupportedException e) {
    // Do nothing. This exception can't happen.
}
```

Antipattern: returns require exception handling

- Return zero-length array or empty collection, not null

```java
package java.awt.image;
public interface BufferedImageOp {
  // Returns the rendering hints for this operation,
  // or null if no hints have been set.
  public RenderingHints getRenderingHints();
}
```

- Do not return a String if a better type exists

Don't let your output become your de facto API

- Document the fact that output formats may evolve in the future
- Provide programmatic access to all data available in string form

```
public class Throwable {
   public void printStackTrace(PrintStream s);
}
```

```
org.omg.CORBA.MARSHAL: com.ibm.ws.pmi.server.DataDescriptor; IllegalAccessException  minor code: 4942F23E  comp
        at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:199)
        at com.ibm.rmi.iiop.CDRInputStream.read_value(CDRInputStream.java:1429)
        at com.ibm.rmi.io.ValueHandlerImpl.read_Array(ValueHandlerImpl.java:625)
        at com.ibm.rmi.io.ValueHandlerImpl.readValueInternal(ValueHandlerImpl.java:273)
        at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:189)
        at com.ibm.rmi.iiop.CDRInputStream.read_value(CDRInputStream.java:1429)
        at com.ibm.ejs.sm.beans._EJSRemoteStatelessPmiService_Tie._invoke(_EJSRemoteStatelessPmiService_Tie.ja
```

Don't let your output become your de facto API

- Document the fa[…]
  the future
- Provide programm[…]
  string form

```
public class Throwable
    public void printSt
}
```

```
public class Throwable {
    public void printStackTrace(PrintStream s);
    public StackTraceElement[] getStackTrace();
}

public final class StackTraceElement {
    public String  getFileName();
    public int     getLineNumber();
    public String  getClassName();
    public String  getMethodName();
    public boolean isNativeMethod();
}
```

Documentation matters

“Reuse is something that is far easier to say than to do. Doing it requires both good design and very good documentation. Even when we see good design, which is still infrequently, we won't see the components reused without good documentation.”

– D. L. Parnas, *Software Aging. Proceedings of the 16th International Conference on Software Engineering, 1994*

Contracts and Documentation

- APIs should be self-documenting
  - Good names drive good design
- Document religiously anyway
  - All public classes
  - All public methods
  - All public fields
  - All method parameters
  - Explicitly write behavioral specifications
- Documentation is integral to the design and development process

REST APIs

REST API

API of a web service

Uniform interface over HTTP requests

- Send parameters to URL, receive data (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`

Breaking Changes

Backward Compatible Changes

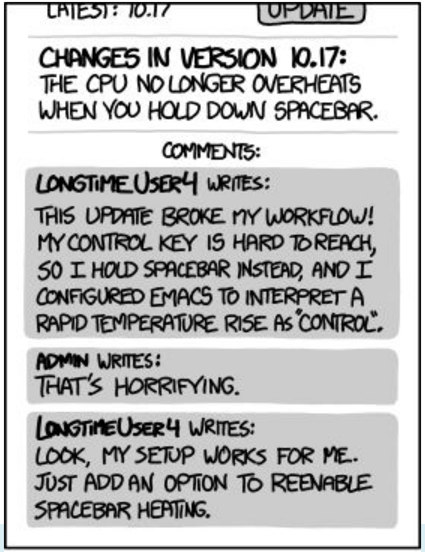
Can add new interfaces, classes

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.

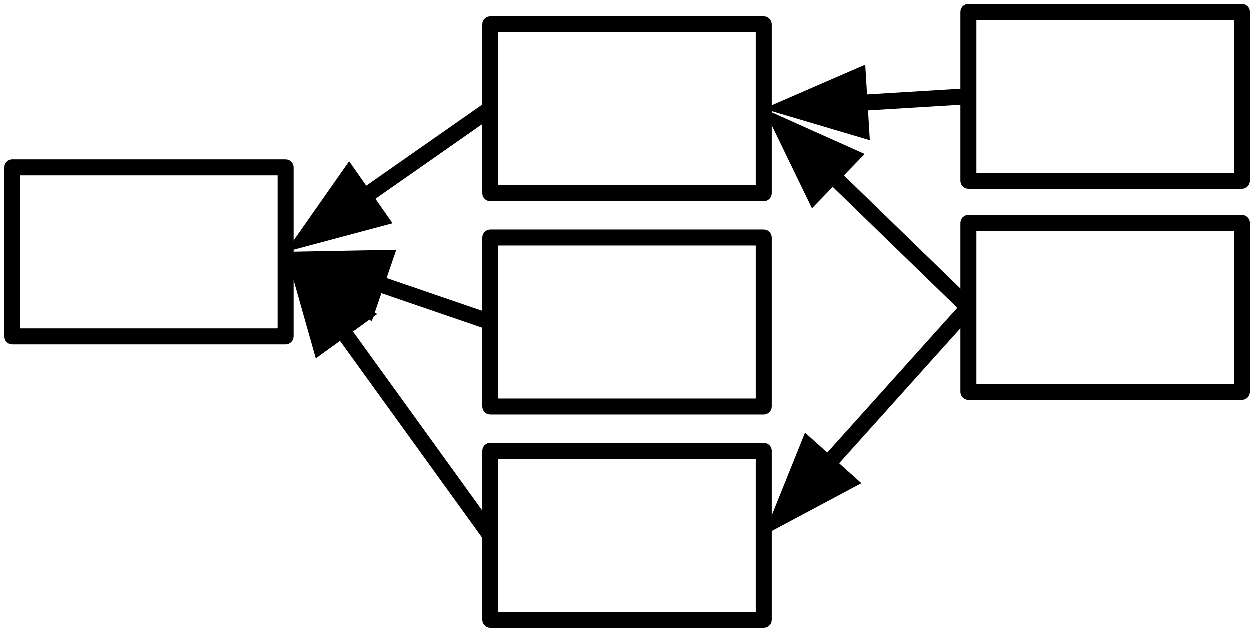
Breaking Changes

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

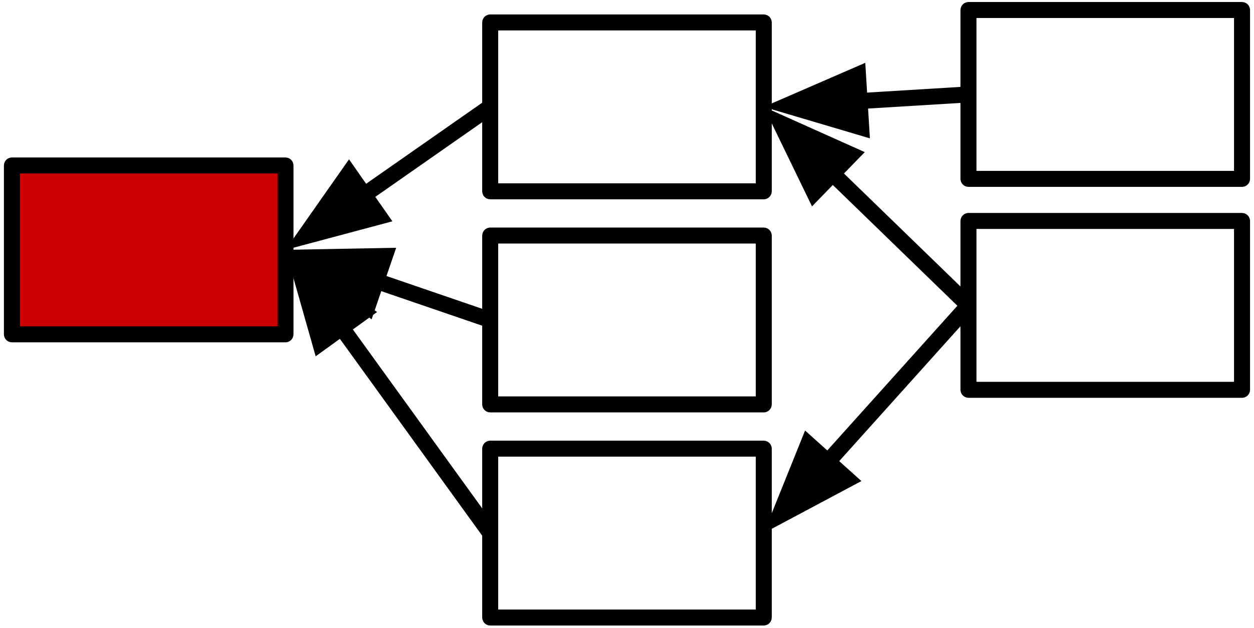
Clients may need to change their implementation when they update

    or even migrate to other library

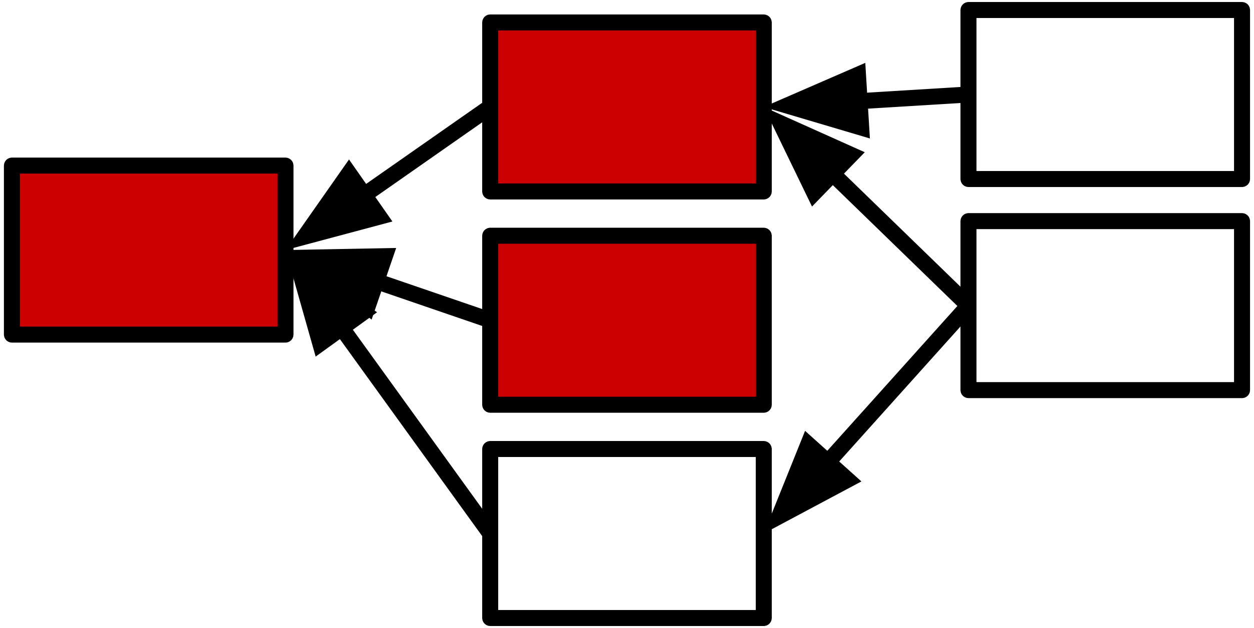
May cause costs for rework and interruption, may ripple through ecosystem



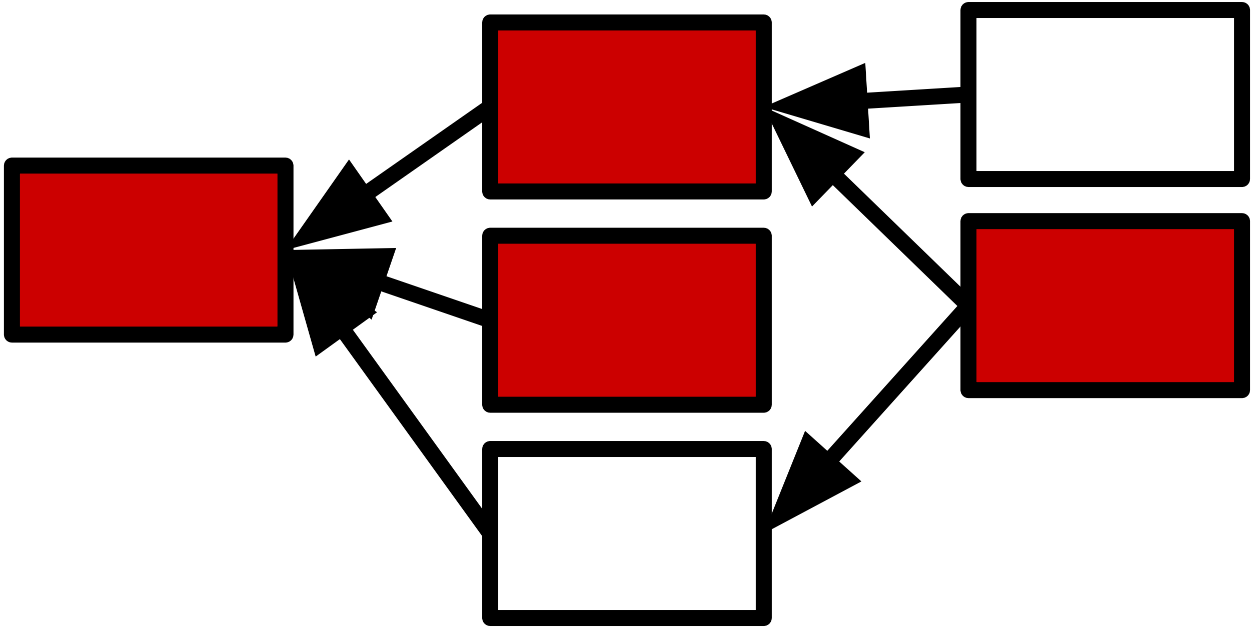
Software Ecosystem



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

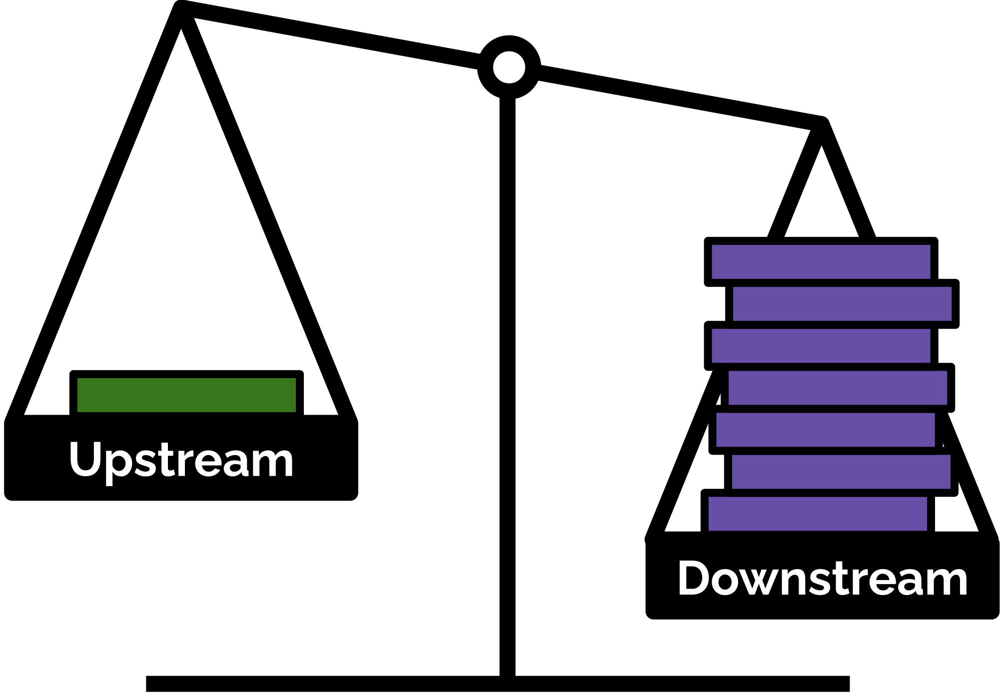
Inefficiencies

Rippling effects from upstream changes

Technical debt, style

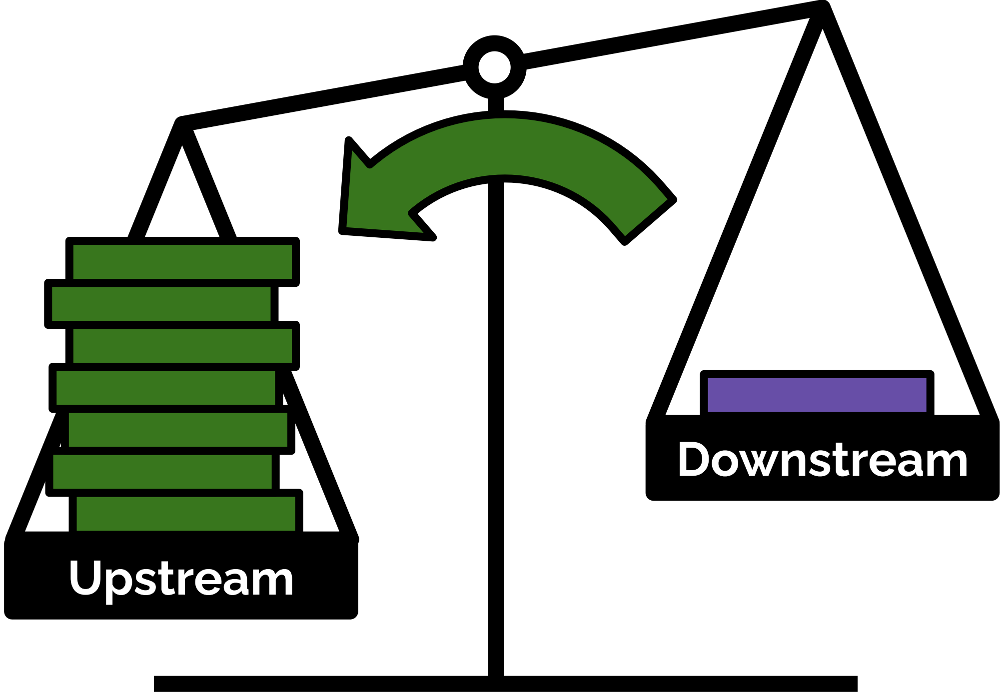
Breaking changes cause costs

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

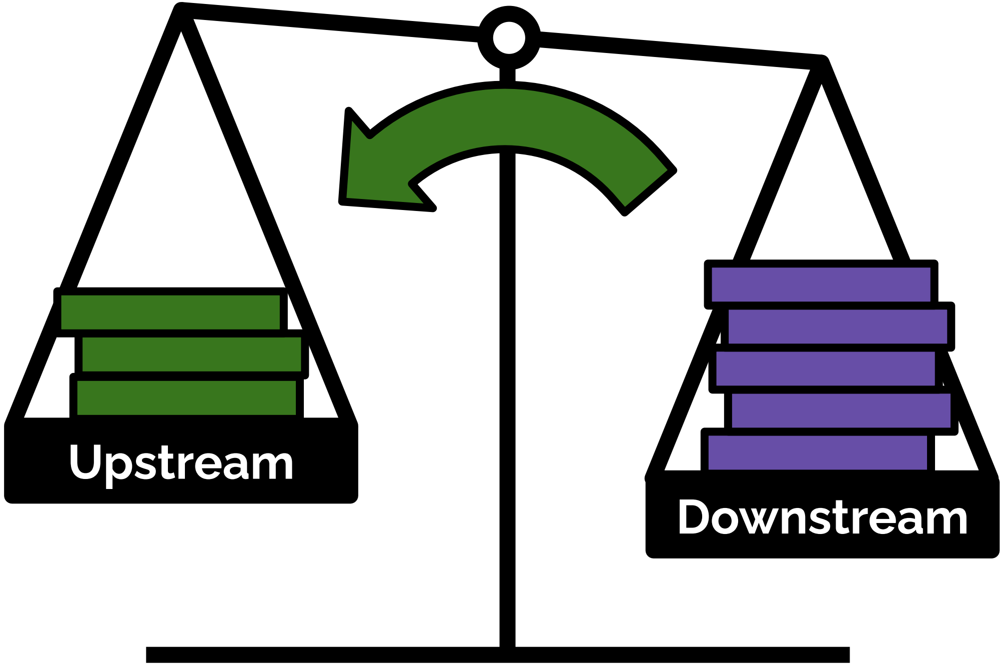


By default, rework and interruption costs for downstream users

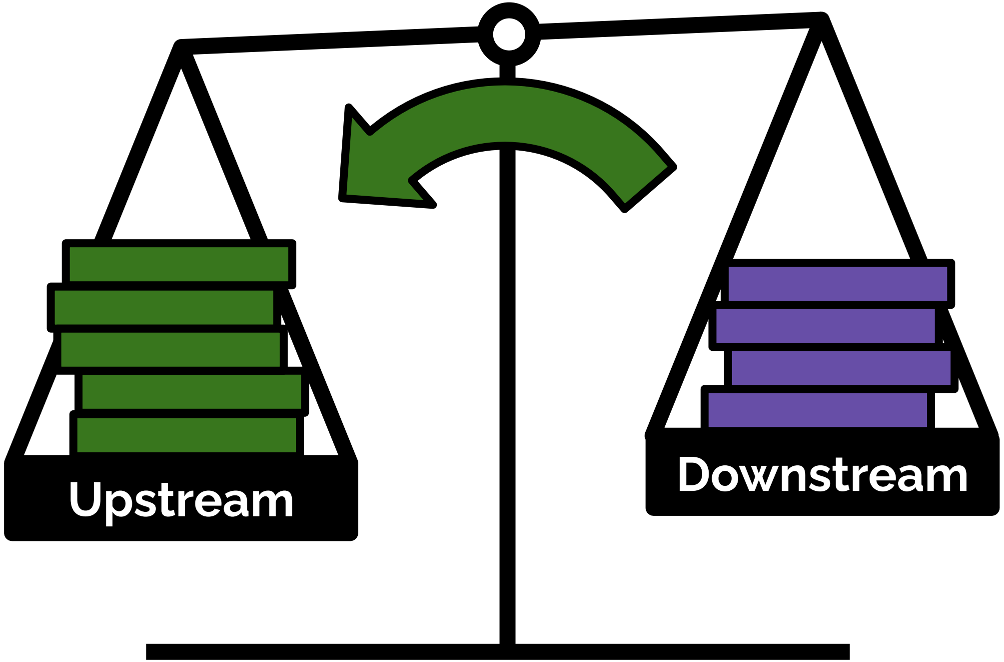
How to reduce costs for downstream users?



Not making a change (opportunity costs, technical debt)



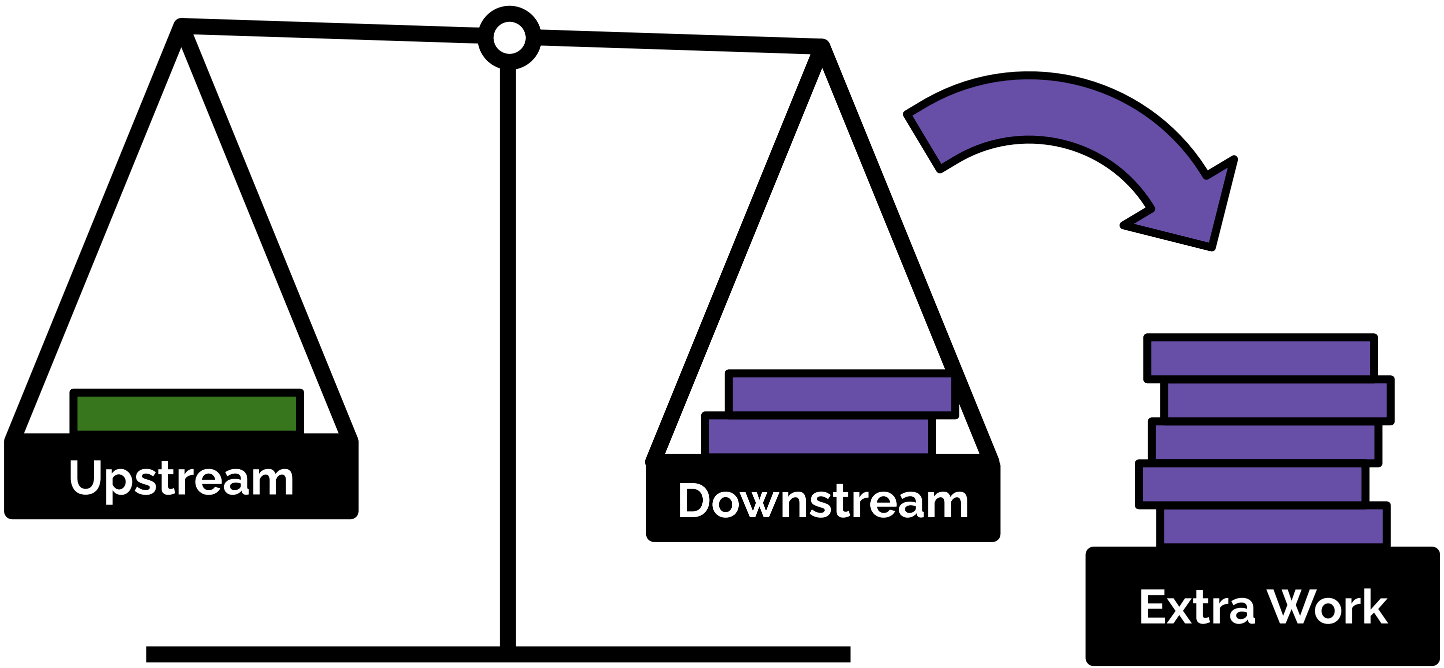
Announcements
Documentation
Migration guide



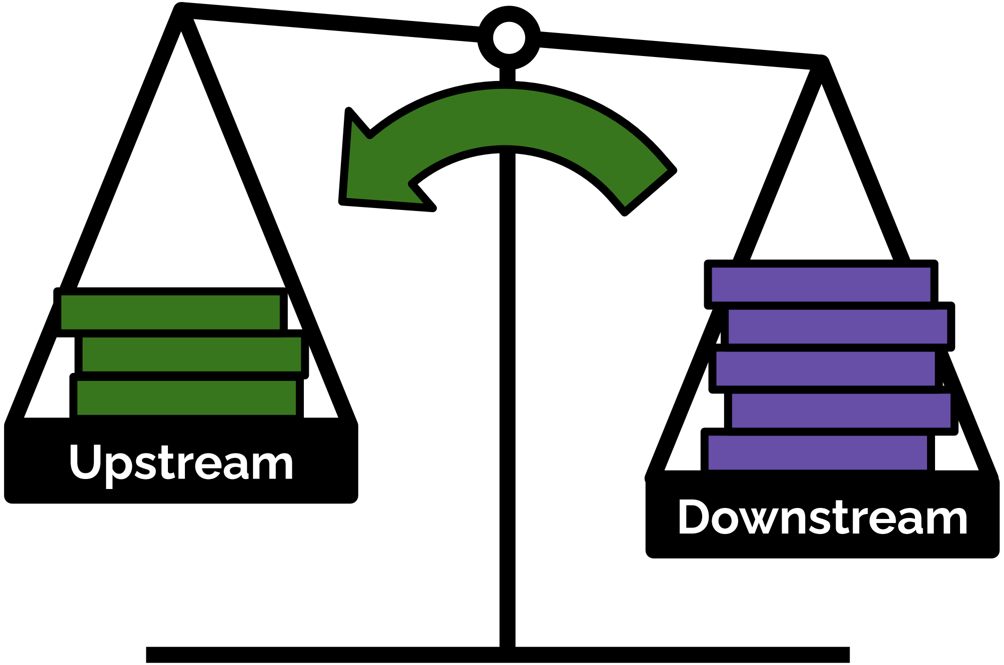
Parallel maintenance releases

Maintaining old interfaces (deprecation)

Release planning



Avoiding dependencies
Encapsulating from change



Influence development

Semantic Versioning

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

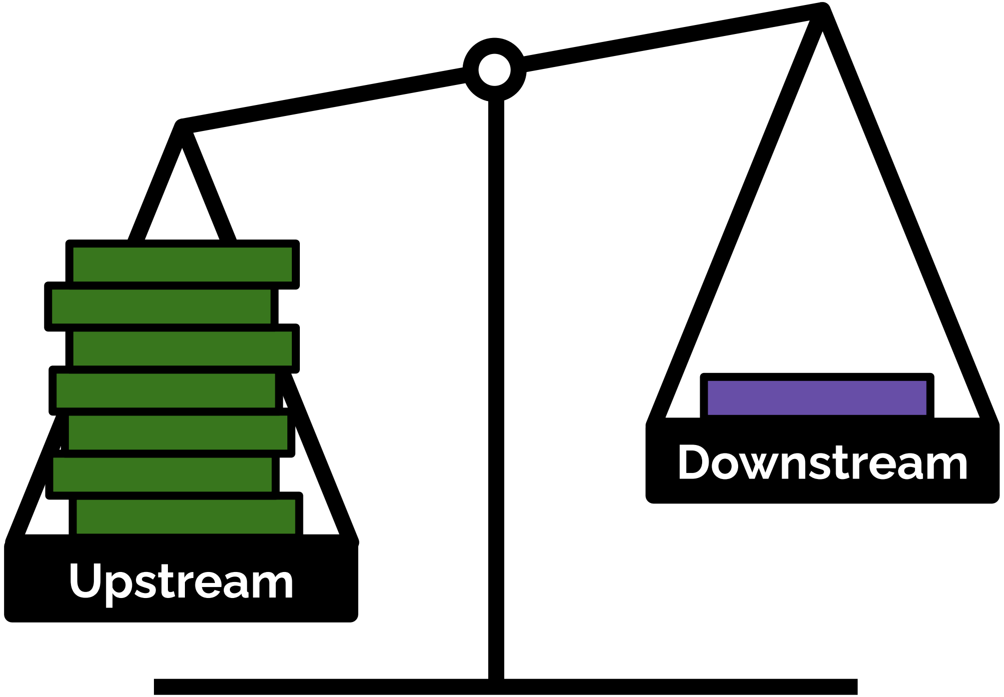
Backward compatibility to reduce costs for **clients**

"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

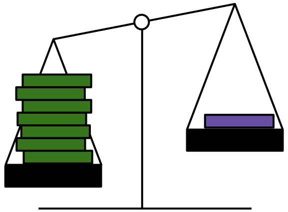
Values

Backward compatibility for clients



Yearly synchronized coordinated releases

Backward compatibility for clients



Willing to accept high costs + opportunity costs

Educational material, workarounds

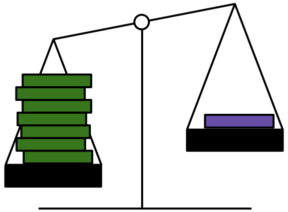
API tools for checking

Coordinated release planning

No parallel releases

Upstream

Backward compatibility for clients



Convenient to use as resource

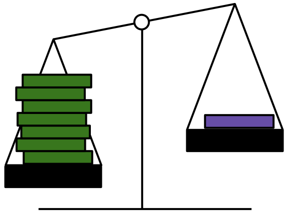
Yearly updates sufficient for many

Stability for corporate users

Downstream

eclipse

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

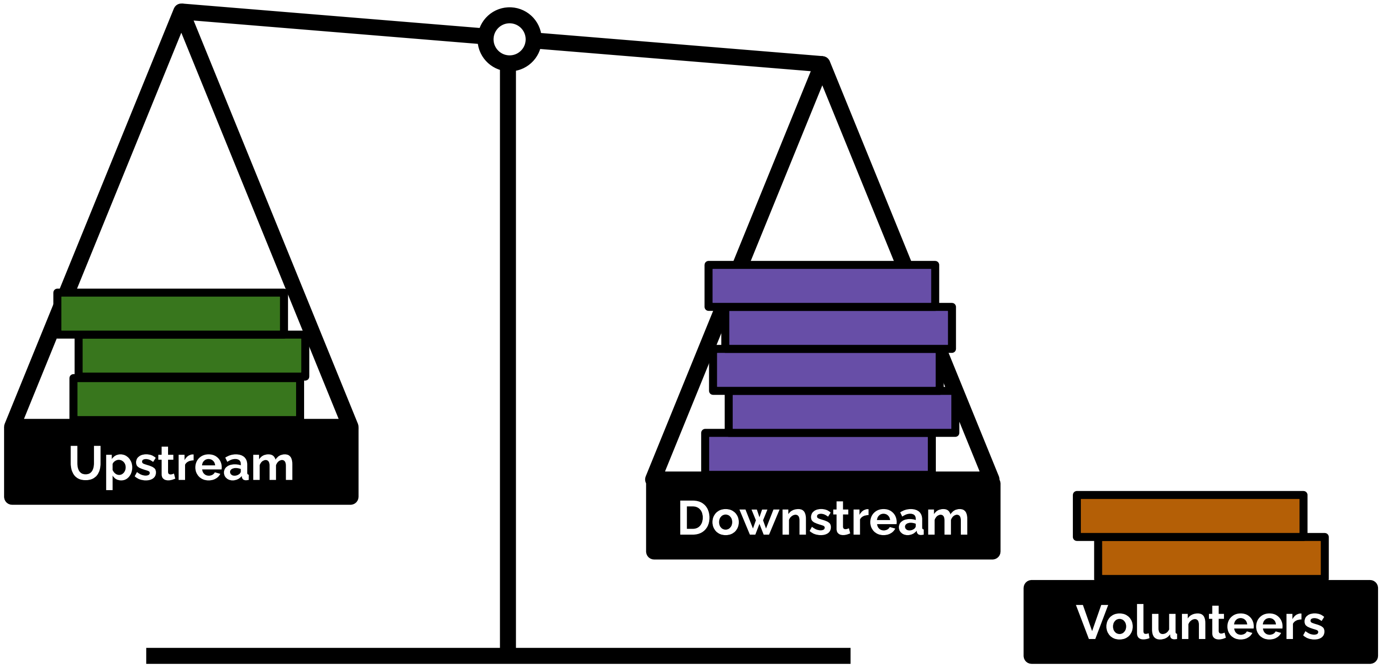
“Typically, if you have hip things, then you get also people who create new APIs on top ... to create the next graphical editing framework or to build more efficient text editors. ... And these things don’t happen on the Eclipse platform anymore.”

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]

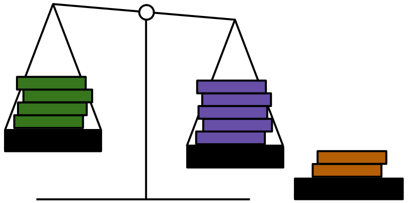
Values

Timely access to current research for end users



Continuous synchronization, ~1 month lag

Timely access to current research for end users



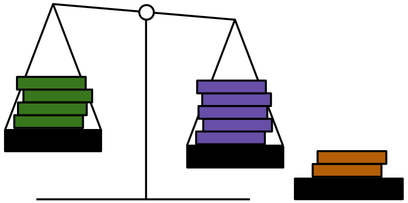
Snapshot consistency within the ecosystem (not outside)

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

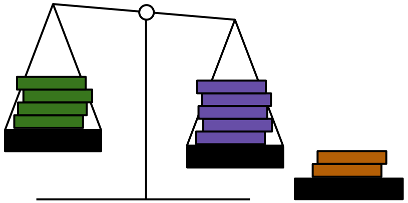
Urgency when upstream package updates

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

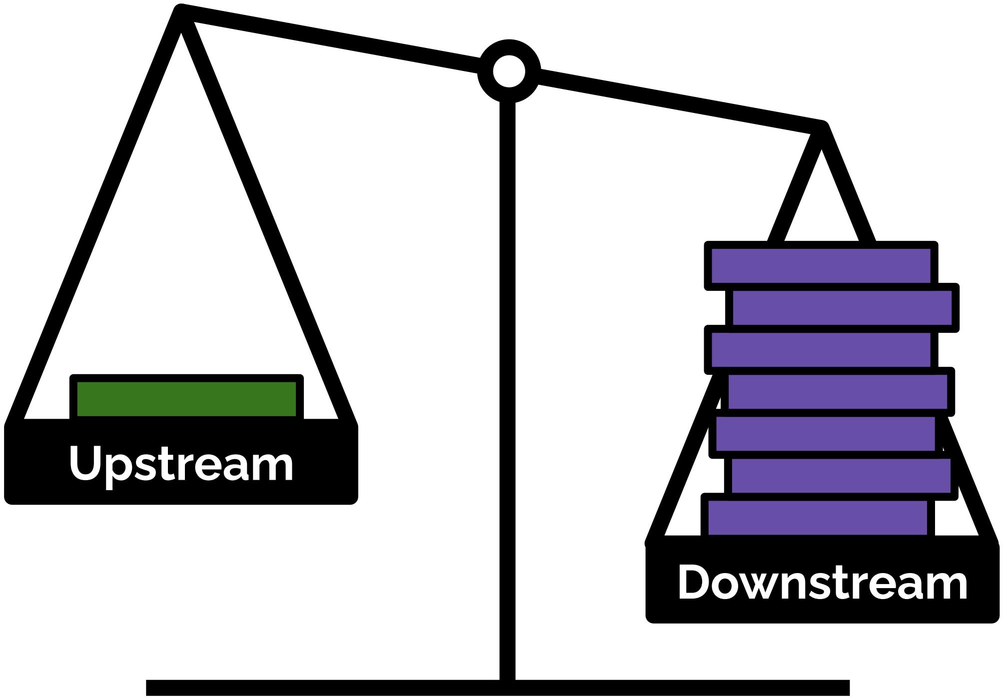
“And then I need to [react to] some change ... and it might be a relatively short timeline of two weeks or a month. And that's difficult for me to deal with, because I try to sort of focus one project for a couple weeks at a time so I can remain productive.”

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

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

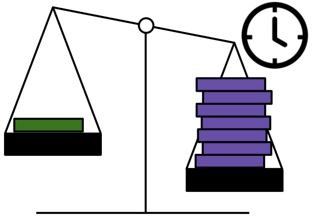
Values

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

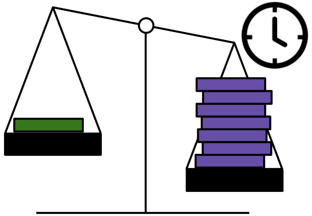
Signaling intention with SemVer

No central release planning

Parallel releases more common

Upstream

Easy and fast to publish and use for developers



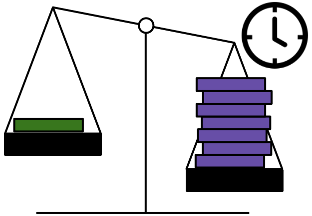
Technology supports using old + mixed revisions; decouples upstream and downstream pace

Choice to stay up to date

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

Downstream

Easy and fast to publish and use for developers



Rapid change requires constant maintenance

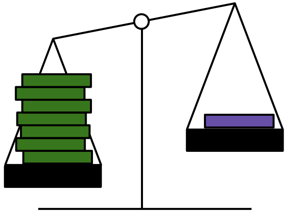
Emphasis on tools and community, often grassroots

Friction

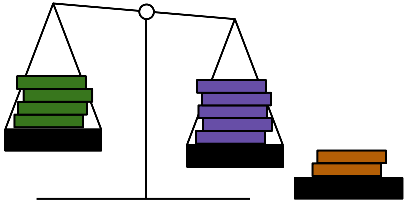
"Last week's tutorial is out of date today."

Contrast

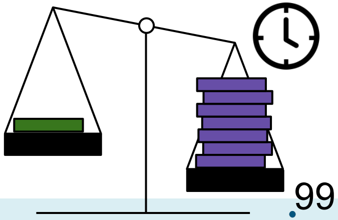
Backward compatibility
for clients



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.

Photo Credit: axi11a (cc)

Lecture summary

- APIs took off in the past thirty years, and gave us super-powers
- Good APIs are a blessing; bad ones, a curse
- API Design is hard
- Following an API design process greatly improves API quality
- Most good principles for good design apply to APIs
    - Don't adhere to them slavishly, but…
    - Don't violate them without good reason