Principles of Software Construction

API Design (Part 2)

Claire Le Goues **Bogdan Vasilescu**
(With slides from Josh Bloch & Christian Kästner)

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Minimizing Conceptual Weight

Principle: Minimize conceptual weight

- API should be as small as possible but no smaller
  - **When in doubt, leave it out**
- Conceptual weight: How many concepts must a programmer learn to use your API?
  - APIs should have a "high power-to-weight ratio"

Conceptual weight (a.k.a. conceptual surface area)

- **Conceptual weight** more important than “physical size”
- *def.* The number & difficulty of new concepts in API
  - i.e., the amount of space the API takes up in your brain
- Examples where growth adds little conceptual weight:
  - Adding overload that behaves consistently with existing methods
  - Adding `arccos` when you already have `sin`, `cos`, and `arcsin`
  - Adding new implementation of an existing interface
- Look for a high *power-to-weight ratio*
  - In other words, look for API that lets you do a lot with a little

“Perfection is achieved not when there is nothing more to add, but when there is nothing left to take away.”

— Antoine de Saint-Exupéry, *Airman’s Odyssey*, 1942

Example: generalizing an API can make it smaller

Subrange operations on `Vector` – legacy List implementation

```
public class Vector {
    public int indexOf(Object elem, int index);
    public int lastIndexOf(Object elem, int index);
    ...
}
```

- Not very powerful
  - Supports only search operation, and only over certain ranges
- Hard to use without documentation
  - What are the semantics of index? I don't remember, and it isn't obvious.

Example: generalizing an API can make it smaller

Subrange operations on `List`

```
public interface List<T> {
    List<T> subList(int fromIndex, int toIndex);
    ...
}
```

- Supports *all* List operations on *all* subranges
- Easy to use even without documentation

Boilerplate Code

```
import org.w3c.dom.*;
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

/** DOM code to write an XML document to a specified output stream. */
static final void writeDoc(Document doc, OutputStream out) throws IOException{
  try {
    Transformer t = TransformerFactory.newInstance().newTransformer();
    t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, doc.getDoctype().getSystemId());
    t.transform(new DOMSource(doc), new StreamResult(out)); // Does actual writing
  } catch(TransformerException e) {
    throw new AssertionError(e);  // Can't happen!
  }
}
```

- Generally done via cut-and-paste
- Ugly, annoying, and error-prone

Boilerplate Code

Generally created via cut-and-paste

Ugly, annoying, and error-prone

Sign of API not supporting common use cases directly

Consider creating APIs for most common use cases, hiding internals

Principle: Make it easy to do what's common, make it possible to do what's less so

- If it's hard to do common tasks, users get upset
- For common use cases
  - Don't make users think about obscure issues - provide reasonable defaults
  - Don't make users do multiple calls - provide a few well-chosen convenience methods
  - Don't make user consult documentation
- For uncommon cases, it's OK to make users work more
- Don't worry too much about truly rare cases
  - It's OK if your API doesn't handle them, at least initially

Tradeoffs

How to balance

- Low conceptual weight
- Avoiding boilerplate code

?

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?

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`

Choosing names easy to read & write

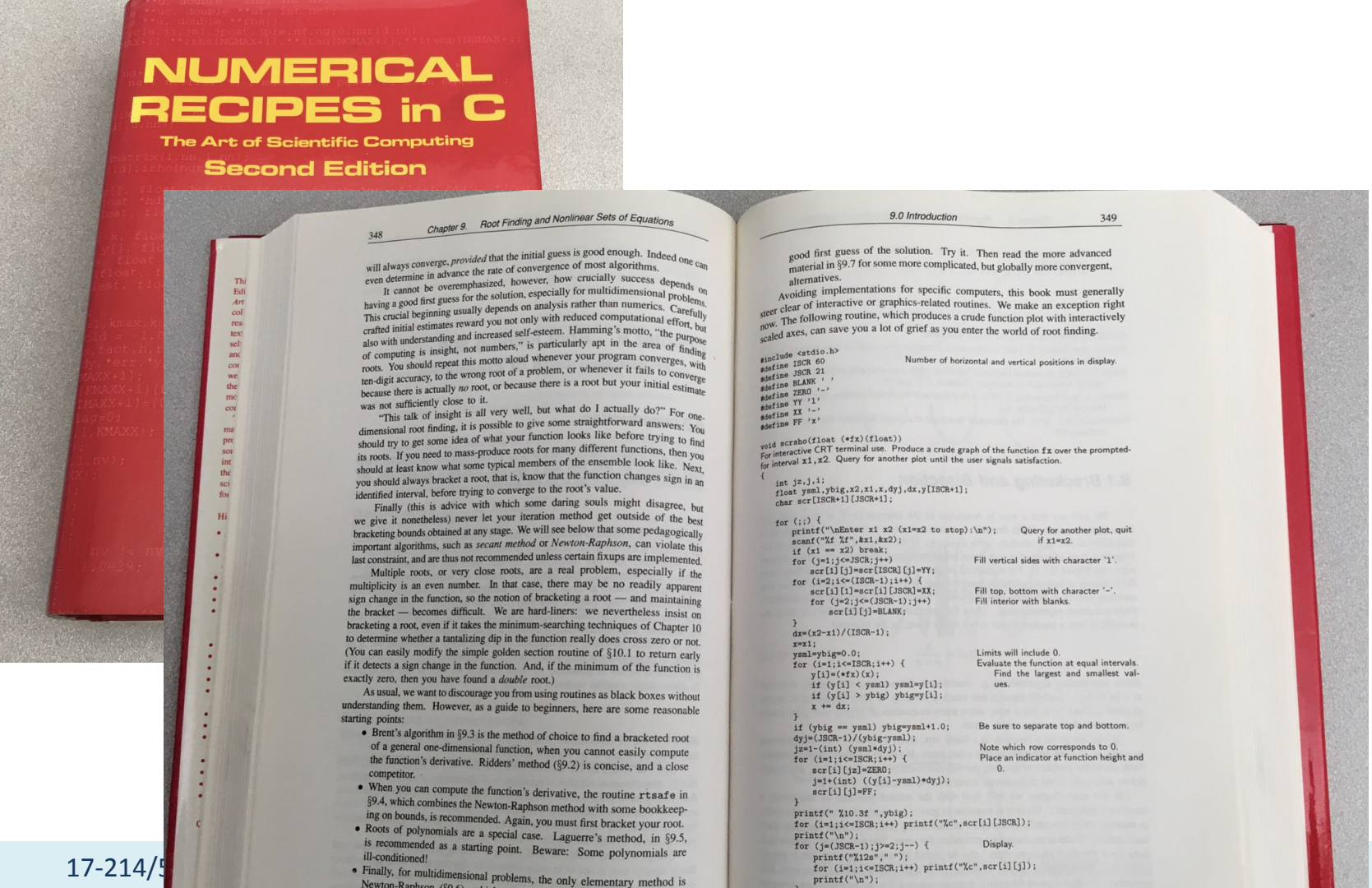
- Choose key nouns carefully!
  - Related to finding good abstractions, which can be hard
  - If you *can't* find a good name, it's generally a bad sign
- If you get the key nouns right, other nouns, verbs, and prepositions tend to choose themselves
- Names can be literal or metaphorical
  - Literal 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
    - e.g., `deleteMessage()` supports undo, but `deleteFolder()` does not
  - Never use multiple words for the same meaning
    - e.g., `deleteMessage()` vs `removeFolder()`
  - i.e., words should be *isomorphic* to meanings
  - Avoid abbreviations
- Build *domain model* or glossary!

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`, etc.



will always converge, *provided* that the initial guess is good enough. Indeed one can even determine in advance the rate of convergence of most algorithms.

It cannot be overemphasized, however, how crucially success depends on having a good first guess for the solution, especially for multidimensional problems. This crucial beginning usually depends on analysis rather than numerics. Carefully crafted initial estimates reward you not only with reduced computational effort, but also with understanding and increased self-esteem. Hamming's motto, "the purpose of computing is insight, not numbers," is particularly apt in the area of finding roots. You should repeat this motto aloud whenever your program converges, with ten-digit accuracy, to the wrong root of a problem, or whenever it fails to converge because there is actually *no* root, or because there is a root but your initial estimate was not sufficiently close to it.

"This talk of insight is all very well, but what do I actually do?" For one-dimensional root finding, it is possible to give some straightforward answers: You should try to get some idea of what your function looks like before trying to find its roots. If you need to mass-produce roots for many different functions, then you should at least know what some typical members of the ensemble look like. Next, you should always bracket a root, that is, know that the function changes sign in an identified interval, before trying to converge to the root's value.

Finally (this is advice with which some daring souls might disagree, but we give it nonetheless) never let your iteration method get outside of the best bracketing bounds obtained at any stage. We will see below that some pedagogically important algorithms, such as *secant method* or *Newton-Raphson*, can violate this last constraint, and are thus not recommended unless certain fixups are implemented.

Multiple roots, or very close roots, are a real problem, especially if the multiplicity is an even number. In that case, there may be no readily apparent sign change in the function, so the notion of bracketing a root — and maintaining the bracket — becomes difficult. We are hard-liners: we nevertheless insist on bracketing a root, even if it takes the minimum-searching techniques of Chapter 10 to determine whether a tantalizing dip in the function really does cross zero or not. (You can easily modify the simple golden section routine of §10.1 to return early if it detects a sign change in the function. And, if the minimum of the function is exactly zero, then you have found a *double* root.)

As usual, we want to discourage you from using routines as black boxes without understanding them. However, as a guide to beginners, here are some reasonable starting points:

- Brent's algorithm in §9.3 is the method of choice to find a bracketed root of a general one-dimensional function, when you cannot easily compute the function's derivative. Ridders' method (§9.2) is concise, and a close competitor.
- When you can compute the function's derivative, the routine `rtsafe` in §9.4, which combines the Newton-Raphson method with some bookkeeping on bounds, is recommended. Again, you must first bracket your root.
- Roots of polynomials are a special case. Laguerre's method, in §9.5, is recommended as a starting point. Beware: Some polynomials are ill-conditioned!
- Finally, for multidimensional problems, the only elementary method is Newton-Raphson (§9.6), which works well if you can provide adequate initial guesses.

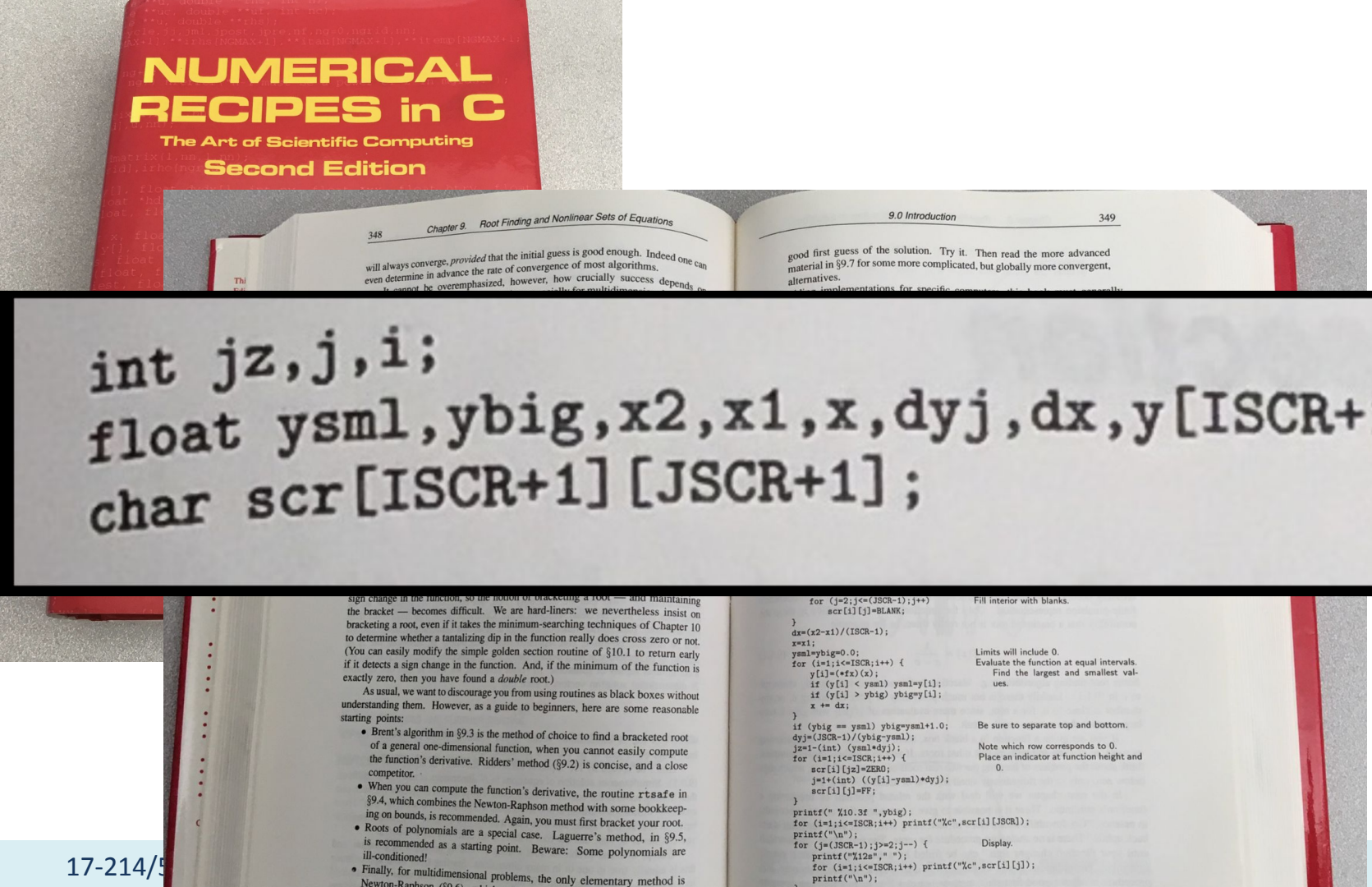
good first guess of the solution. Try it. Then read the more advanced material in §9.7 for some more complicated, but globally more convergent, alternatives.

Avoiding implementations for specific computers, this book must generally steer clear of interactive or graphics-related routines. We make an exception right now. The following routine, which produces a crude function plot with interactively scaled axes, can save you a lot of grief as you enter the world of root finding.

```
#include <stdio.h>
#define ISCR 60                          Number of horizontal and vertical positions in display.
#define JSCR 21
#define BLANK ' '
#define ZERO '-'
#define YY '1'
#define XX '-'
#define FF 'x'

void scrsho(float (*fx)(float))
For interactive CRT terminal use. Produce a crude graph of the function fx over the prompted-
for interval x1,x2. Query for another plot until the user signals satisfaction.
{
    int jz,j,i;
    float ysml,ybig,x2,x1,x,dyj,dx,y[ISCR+1];
    char scr[ISCR+1][JSCR+1];

    for (;;) {
        printf("\nEnter x1 x2 (x1=x2 to stop):\n");      Query for another plot, quit
        scanf("%f %f",&x1,&x2);                            if x1=x2.
        if (x1 == x2) break;
        for (j=1;j<=JSCR;j++)                          Fill vertical sides with character '1'.
            scr[1][j]=scr[ISCR][j]=YY;
        for (i=2;i<=(ISCR-1);i++) {
            scr[i][1]=scr[i][JSCR]=XX;                 Fill top, bottom with character '-'.
            for (j=2;j<=(JSCR-1);j++)                  Fill interior with blanks.
                scr[i][j]=BLANK;
        }
        dx=(x2-x1)/(ISCR-1);
        x=x1;
        ysml=ybig=0.0;                                 Limits will include 0.
        for (i=1;i<=ISCR;i++) {                        Evaluate the function at equal intervals.
            y[i]=(*fx)(x);                                 Find the largest and smallest val-
            if (y[i] < ysml) ysml=y[i];                    ues.
            if (y[i] > ybig) ybig=y[i];
            x += dx;
        }
        if (ybig == ysml) ybig=ysml+1.0;               Be sure to separate top and bottom.
        dyj=(JSCR-1)/(ybig-ysml);
        jz=1-(int) (ysml*dyj);                         Note which row corresponds to 0.
        for (i=1;i<=ISCR;i++) {                        Place an indicator at function height and
            scr[i][jz]=ZERO;                               0.
            j=1+(int) ((y[i]-ysml)*dyj);
            scr[i][j]=FF;
        }
        printf(" %10.3f ",ybig);
        for (i=1;i<=ISCR;i++) printf("%c",scr[i][JSCR]);
        printf("\n");
        for (j=(JSCR-1);j>=2;j--) {                    Display.
            printf("%12s"," ");
            for (i=1;i<=ISCR;i++) printf("%c",scr[i][j]);
            printf("\n");
```



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 () {
    // something to execute in 2 seconds
}, 2000)

query.str = "); fs.rm('/', '-rf'"
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; API designers do.
  - And 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

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

```
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
        at com.ibm.CORBA.iiop.ExtendedServerDelegate.dispatch(ExtendedServerDelegate.java:515)
        at com.ibm.CORBA.iiop.ORB.process(ORB.java:2377)
        at com.ibm.CORBA.iiop.OrbWorker.run(OrbWorker.java:186)
        at com.ibm.ejs.oa.pool.ThreadPool$PooledWorker.run(ThreadPool.java:104)
        at com.ibm.ws.util.CachedThread.run(ThreadPool.java:137)
```

Don't let your output become your de facto API

- Document the fact th
- Provide programmat

```
public class Throwable
    public void printSta
}
```

```
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

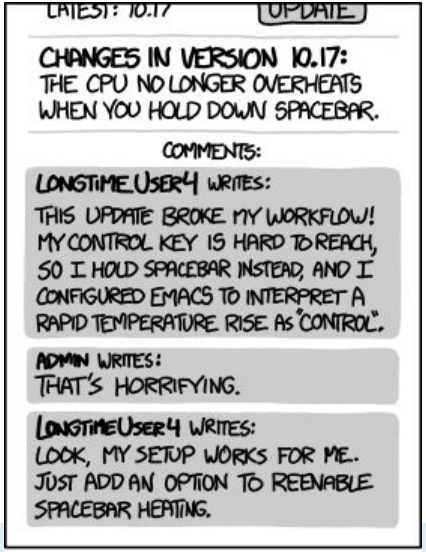
Can add new interfaces, classes

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.

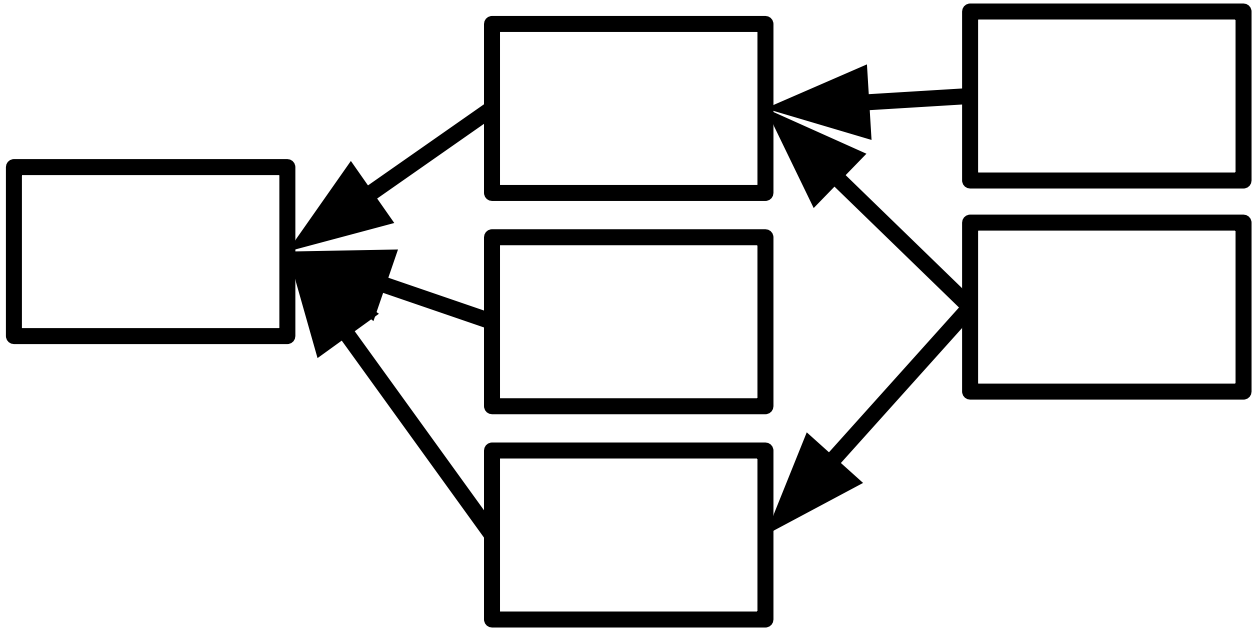
Breaking Changes

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

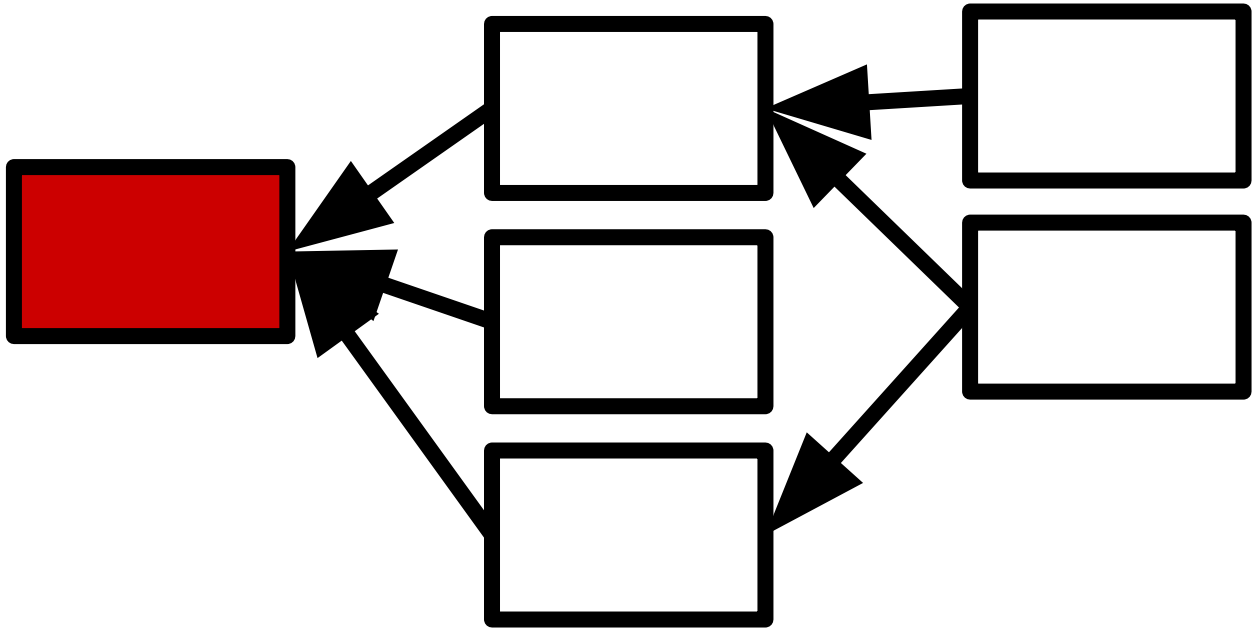
Clients may need to change their implementation when they update

or even migrate to other library

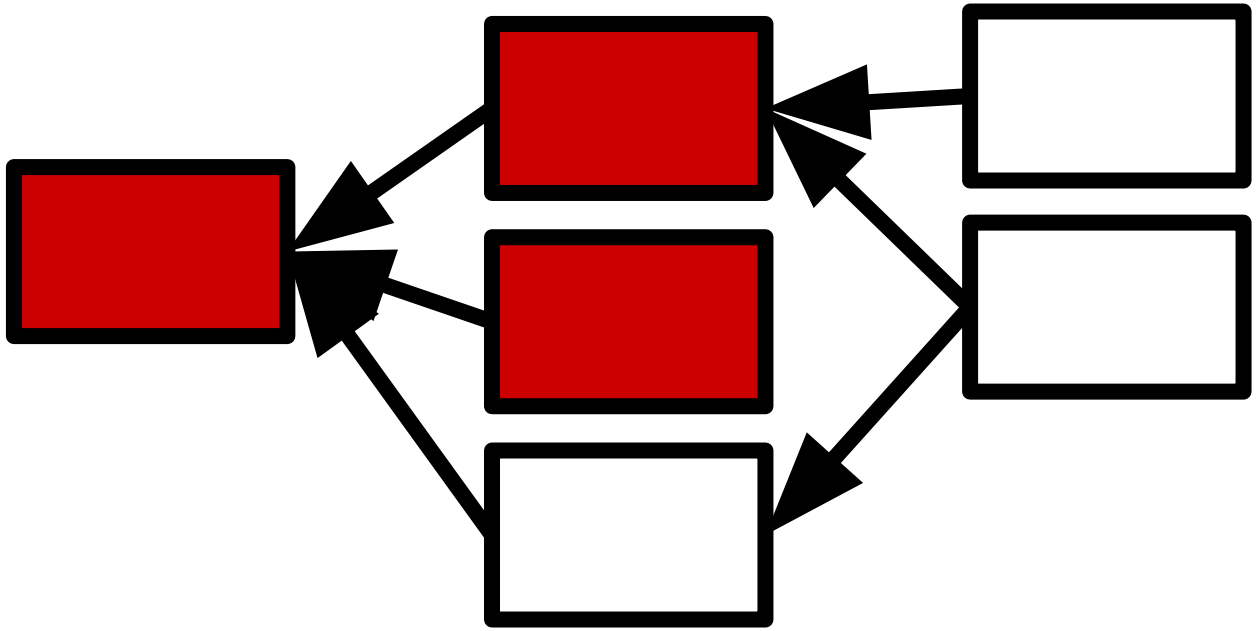
May cause costs for rework and interruption, may ripple through ecosystem



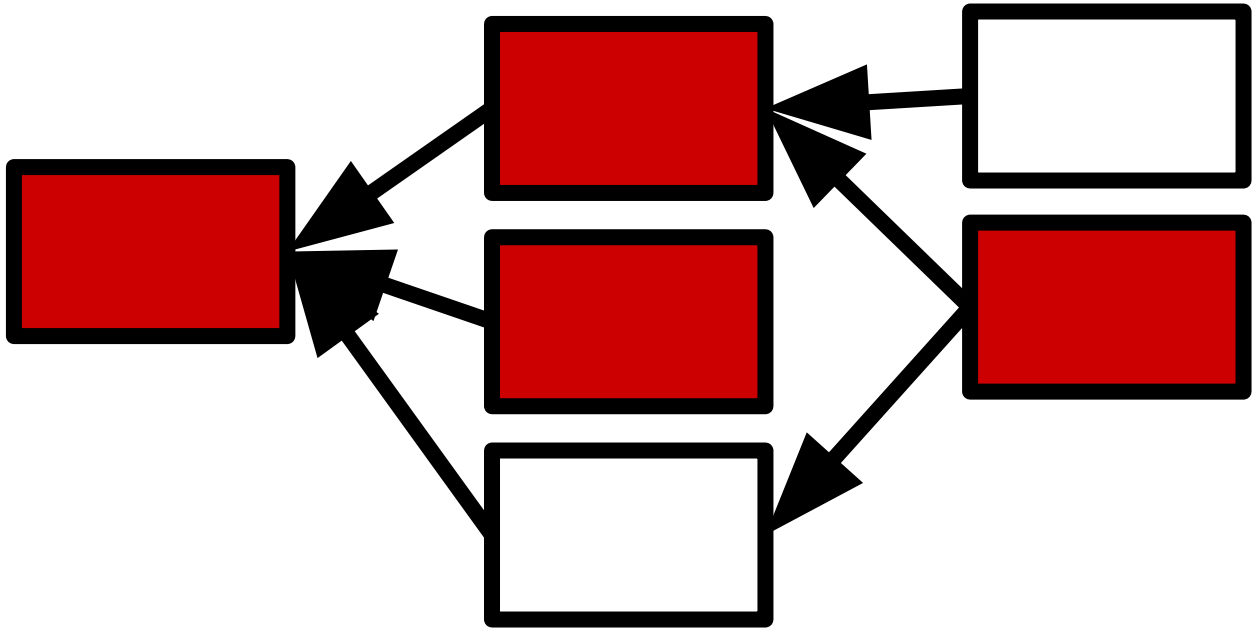
Software Ecosystem



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

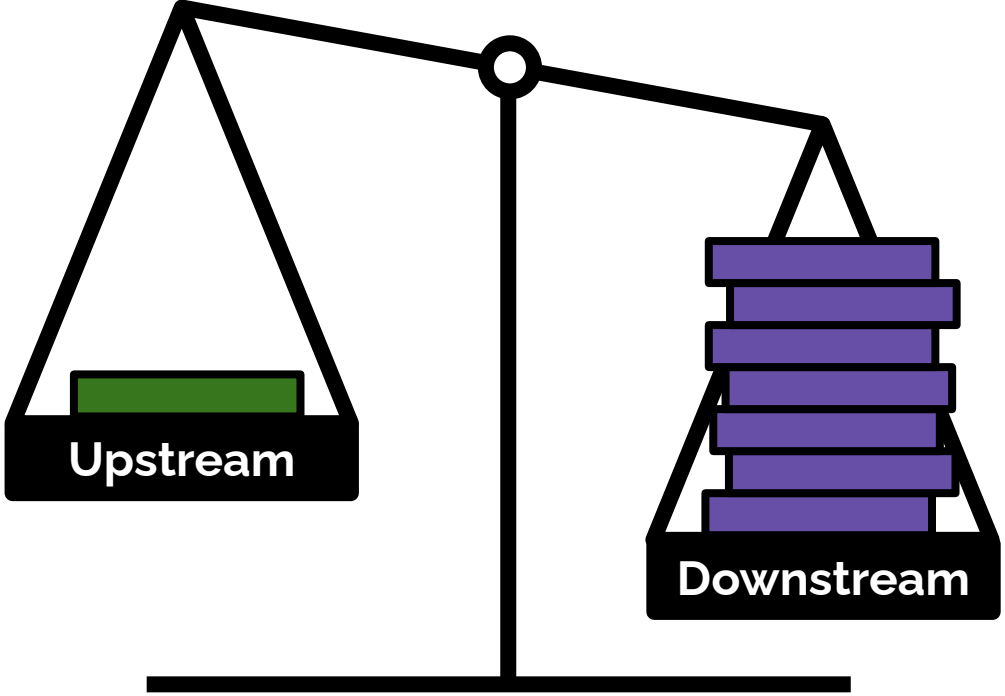
Inefficiencies

Rippling effects from upstream changes

Technical debt, style

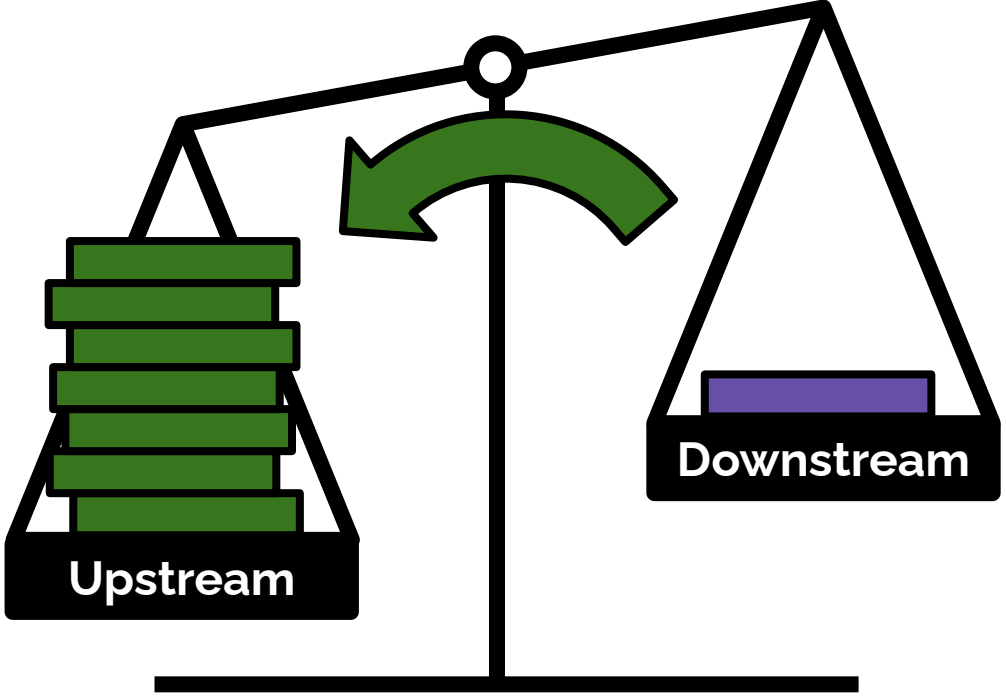
Breaking changes cause costs

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

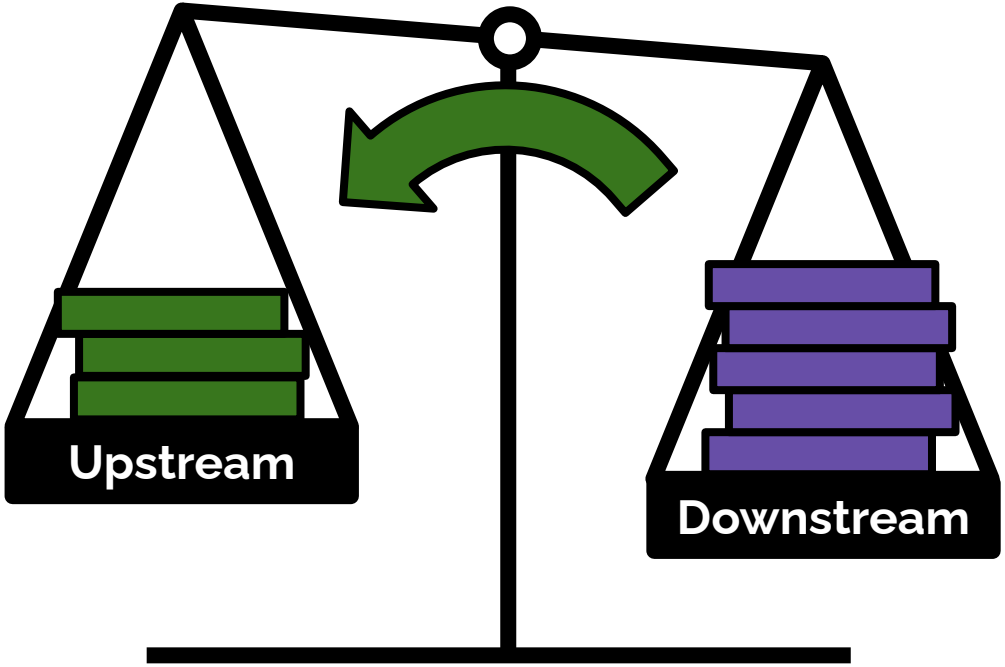


By default, rework and interruption costs for downstream users

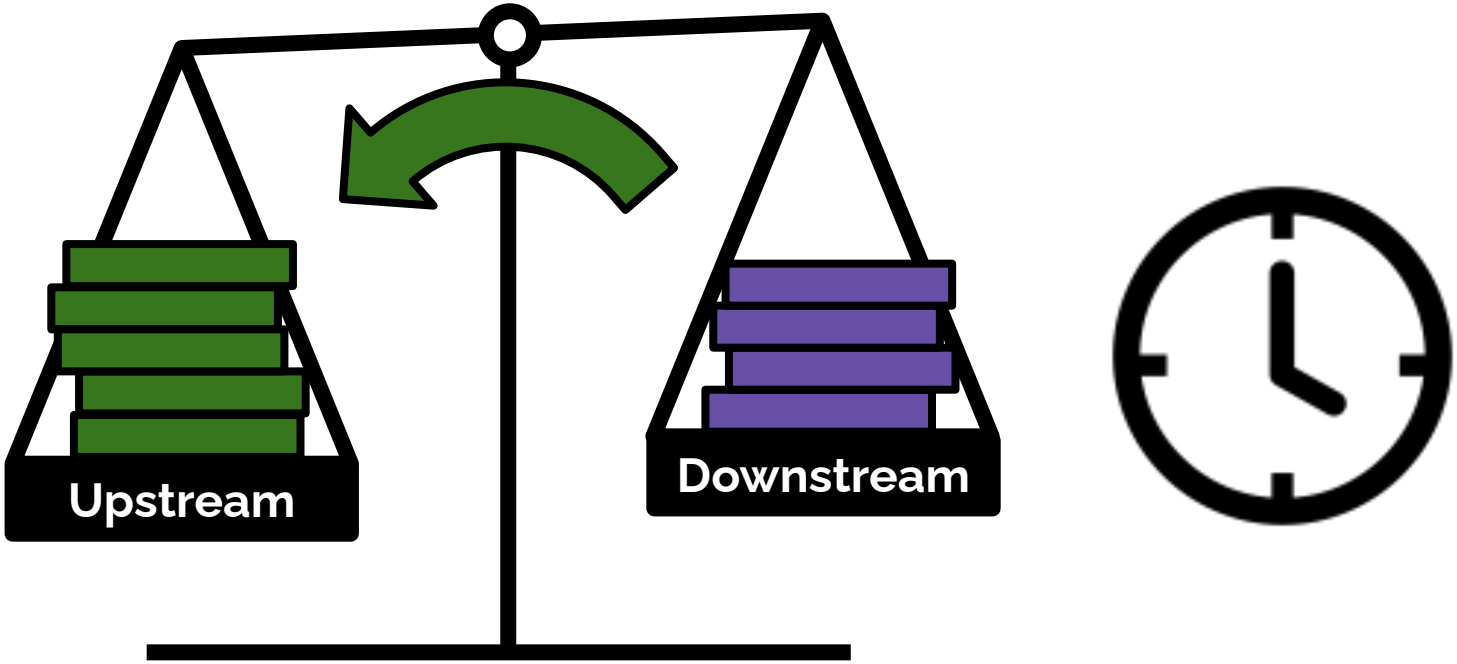
How to reduce costs for downstream users?



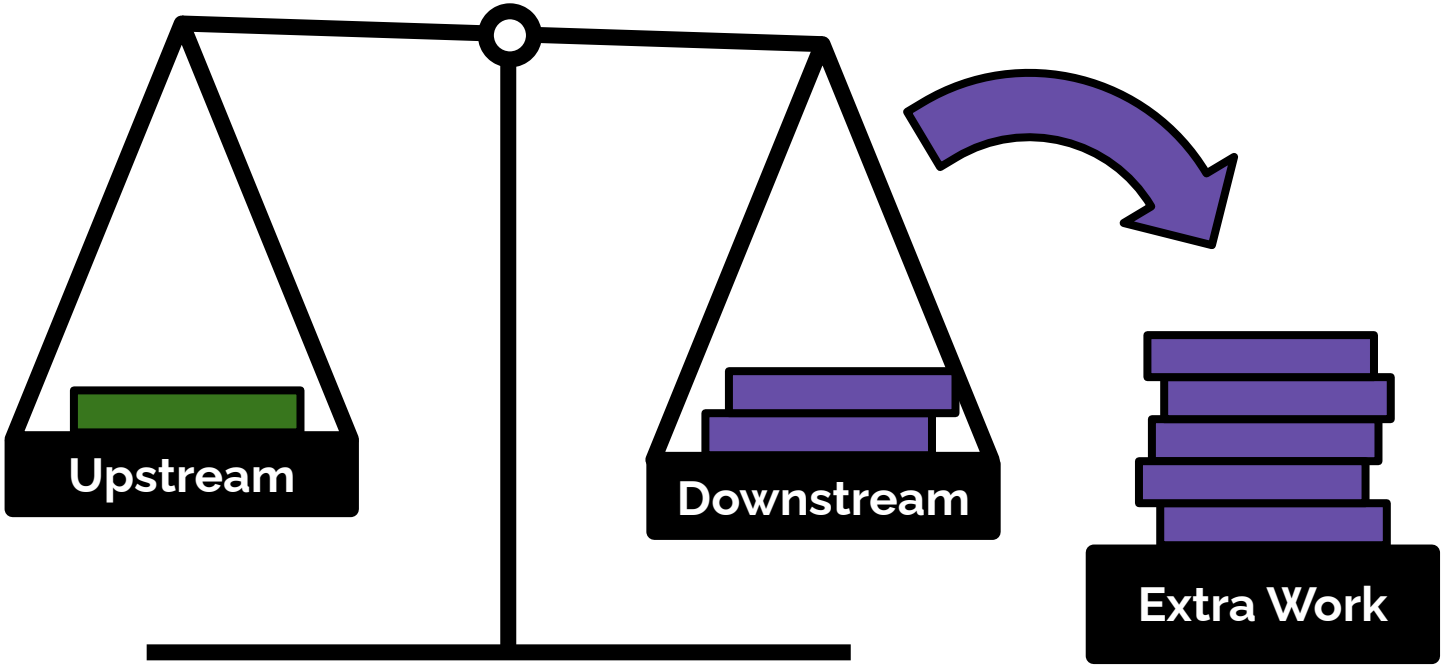
Not making a change
(opportunity costs, technical debt)



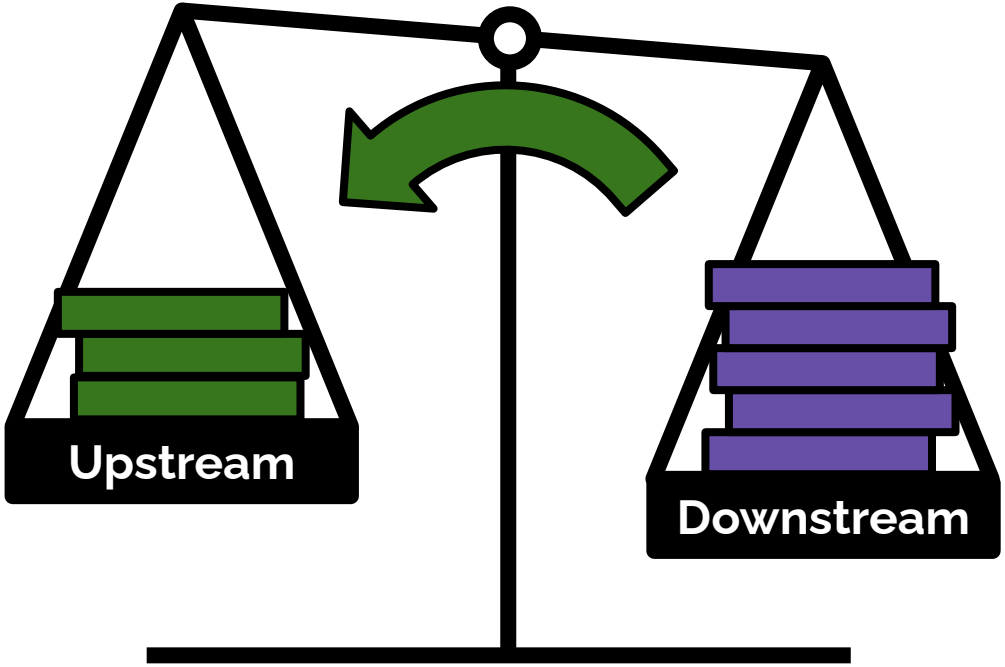
Announcements
Documentation
Migration guide



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:

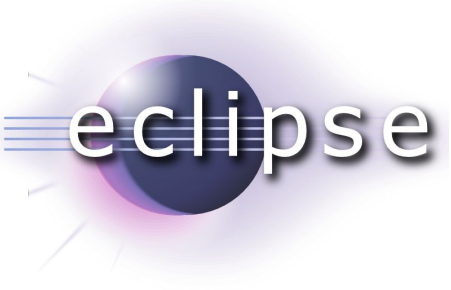
1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add functionality in a backwards compatible manner, and
3. PATCH version when you make backwards compatible bug fixes.

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



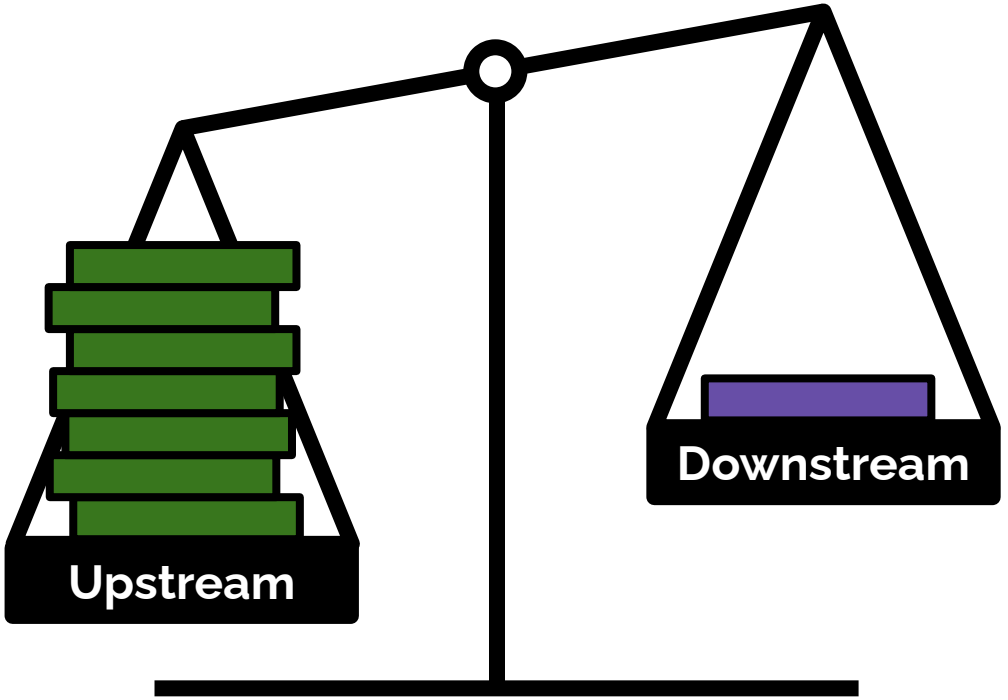
Backward compatibility to reduce costs for **clients**

"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

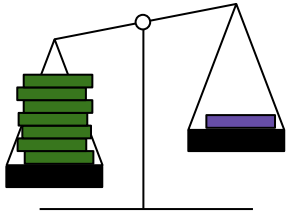
Values

Backward compatibility for clients



Yearly synchronized coordinated releases

Backward compatibility for clients



Willing to accept high costs + opportunity costs

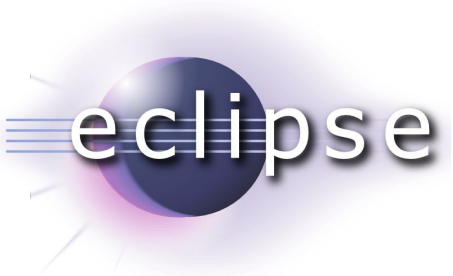
Educational material, workarounds

API tools for checking

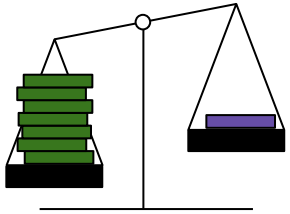
Coordinated release planning

No parallel releases

Upstream



Backward compatibility for clients



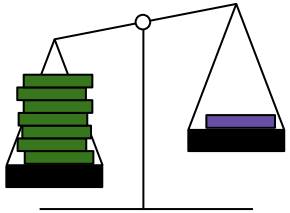
Convenient to use as resource

Yearly updates sufficient for many

Stability for corporate users

Downstream

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

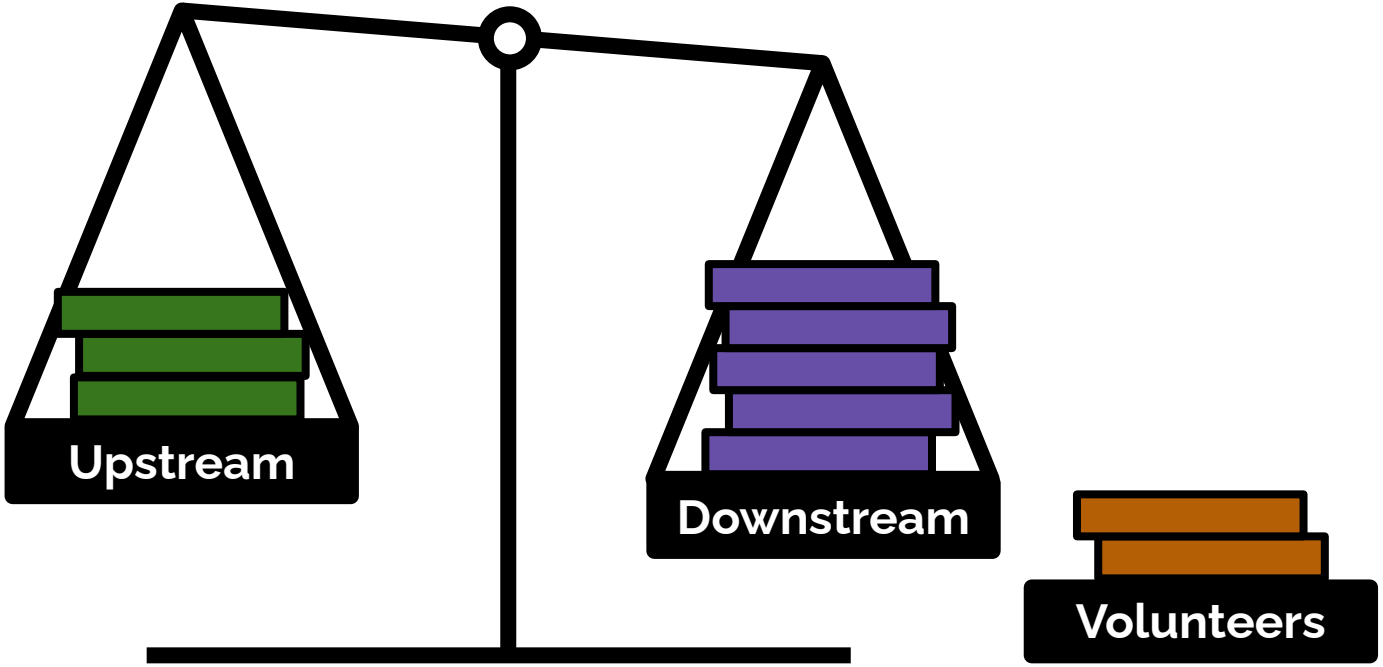
> “Typically, if you have hip things, then you get also people who create new APIs on top ... to create the next graphical editing framework or to build more efficient text editors. ... And these things don’t happen on the Eclipse platform anymore.”

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]

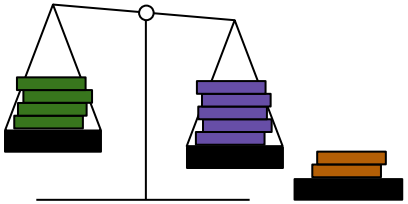
Values

Timely access to current research for end users



Continuous synchronization, ~1 month lag

Timely access to current research for end users



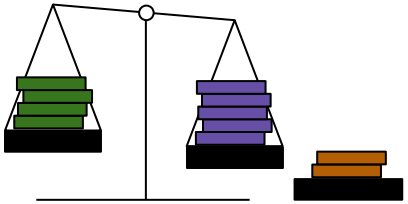
Snapshot consistency within the ecosystem (not outside)

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

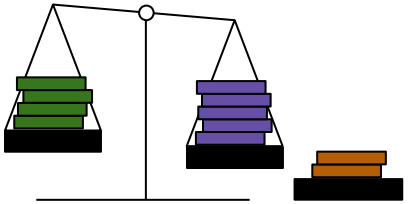
Urgency when upstream package updates

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.”

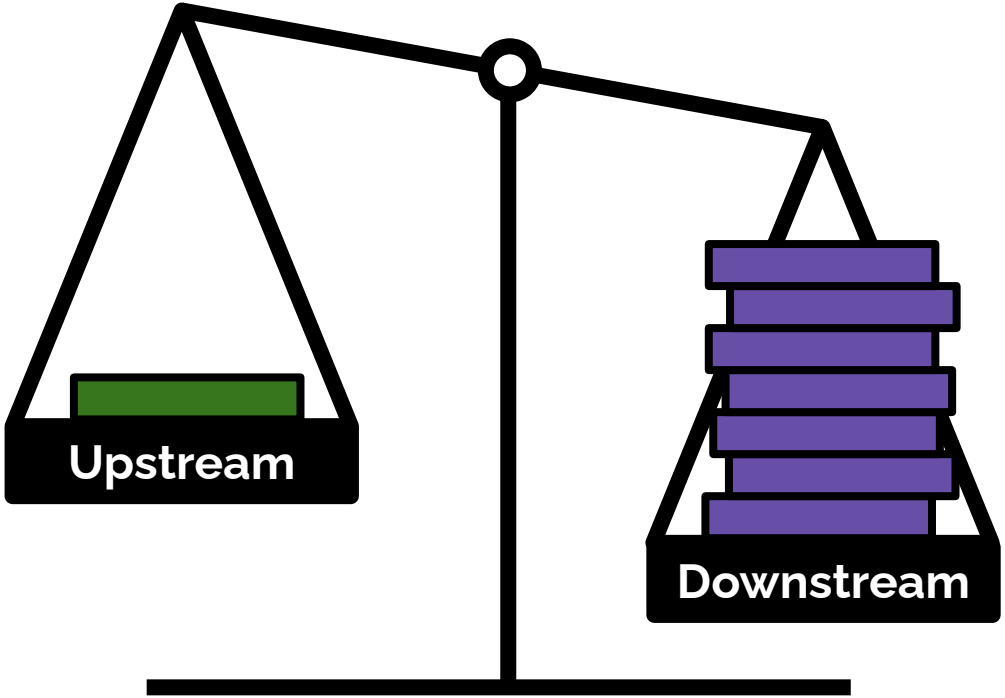

node
JS
TM

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

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

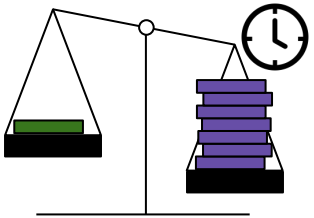
Values

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

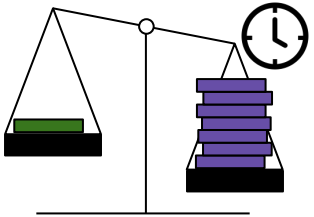
Signaling intention with SemVer

No central release planning

Parallel releases more common

Upstream

Easy and fast to publish and use for developers



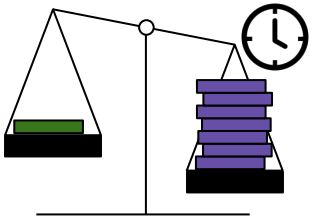
Technology supports using old + mixed revisions; decouples upstream and downstream pace

Choice to stay up to date

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

Downstream

Easy and fast to publish and use for developers



Rapid change requires constant maintenance

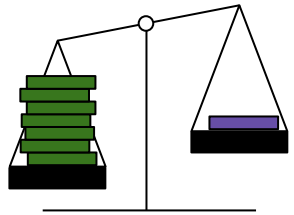
Emphasis on tools and community, often grassroots

Friction

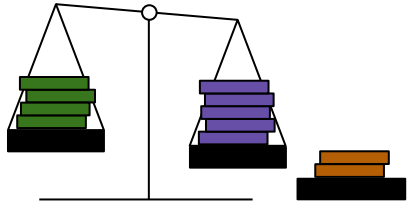
“Last week’s tutorial is out of date today.”

Contrast

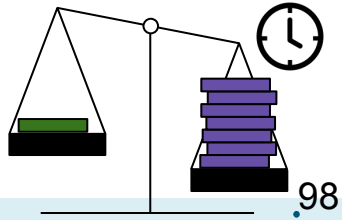
Backward compatibility for clients



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.

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 unconditionally, but…
  - Don't violate them without good reason