

Principles of Software Construction

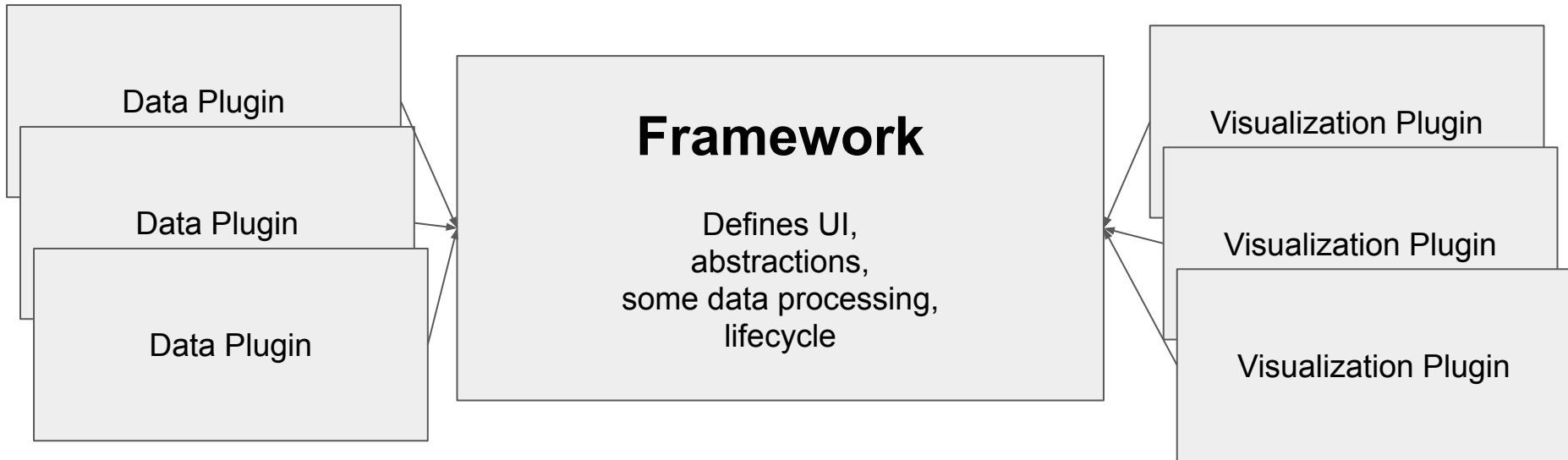
API Design (Part 2)

Christian Kästner Vincent Hellendoorn
(With slides from Josh Bloch)



Midterm Recap

Homework 6 Released



HW6: Map-Based Data Visualizations?

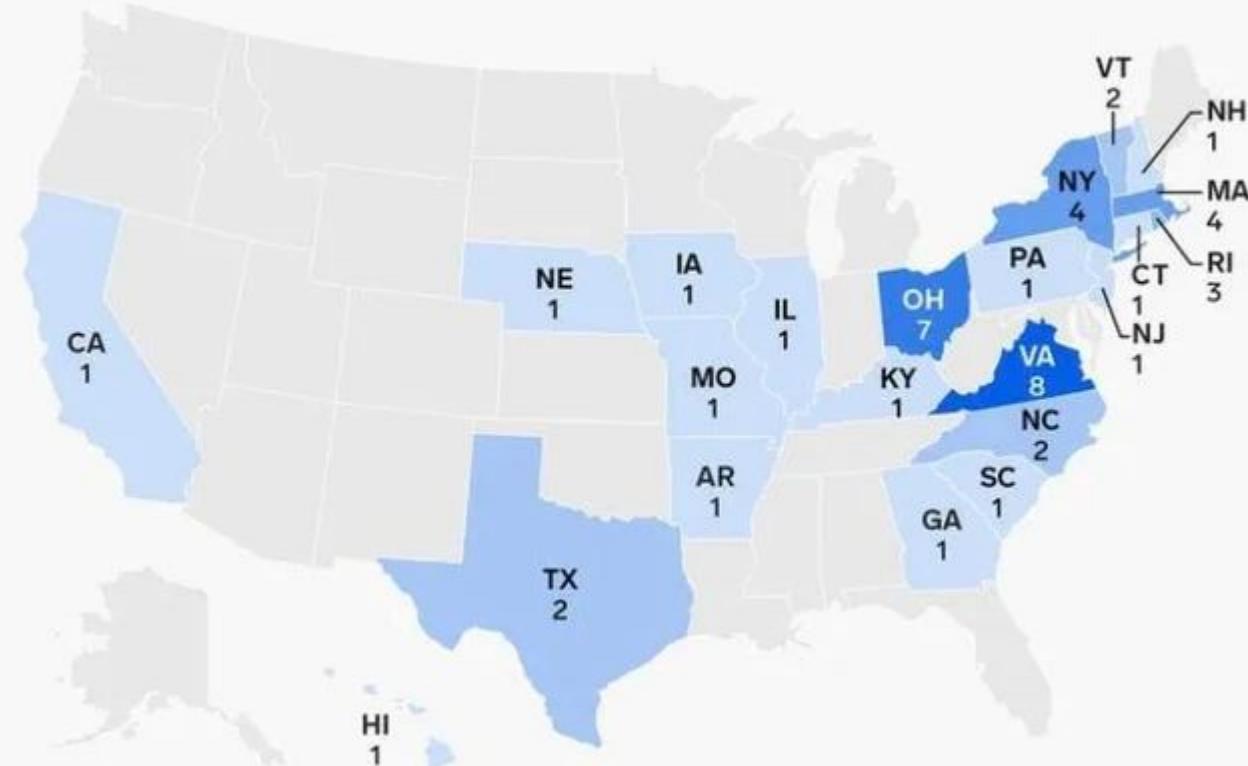
State, county, or country data

Data from many sources

Visualization as map image, table, google maps

Animations for time series data

States that produced the most presidents



BUSINESS INSIDER

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

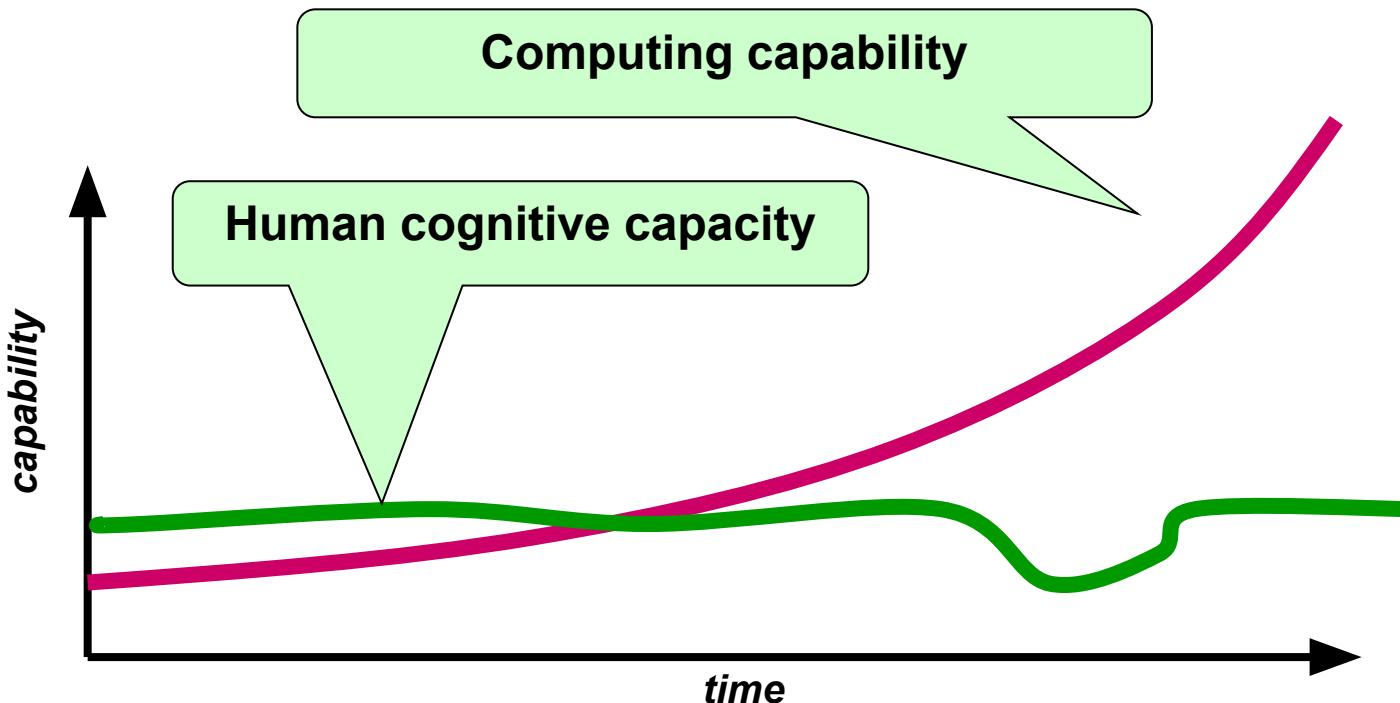
Teamwork

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

	<i>Small scale:</i> One/few objects	<i>Mid scale:</i> Many objects	<i>Large scale:</i> Subsystems
<i>Design for understanding change/ext. reuse robustness ...</i>	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* strcpy(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 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 component: com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:199)
at com.ibm.rmi.io.ValueHandlerImpl.readValue(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.io.CDRInputStream.read_value(CDRInputStream.java:1429)
at com.ibm.ejs.sm.beans._EJSRemoteStatelessPmiService_Tie._invoke(_EJSRemoteStatelessPmiService_Tie.java:515)
at com.ibm.CORBA.idl.IfaceForwarder._doForwardInvoke(IfaceForwarder.java:515)

Don't let your output become your de facto API

- Document the fact for the future
- Provide programmatic 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

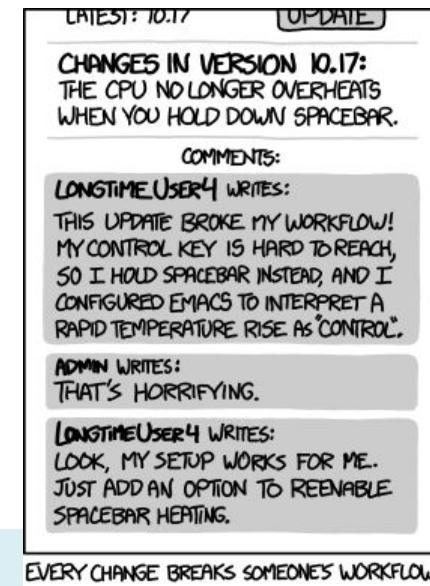
Can add new interfaces, classes

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



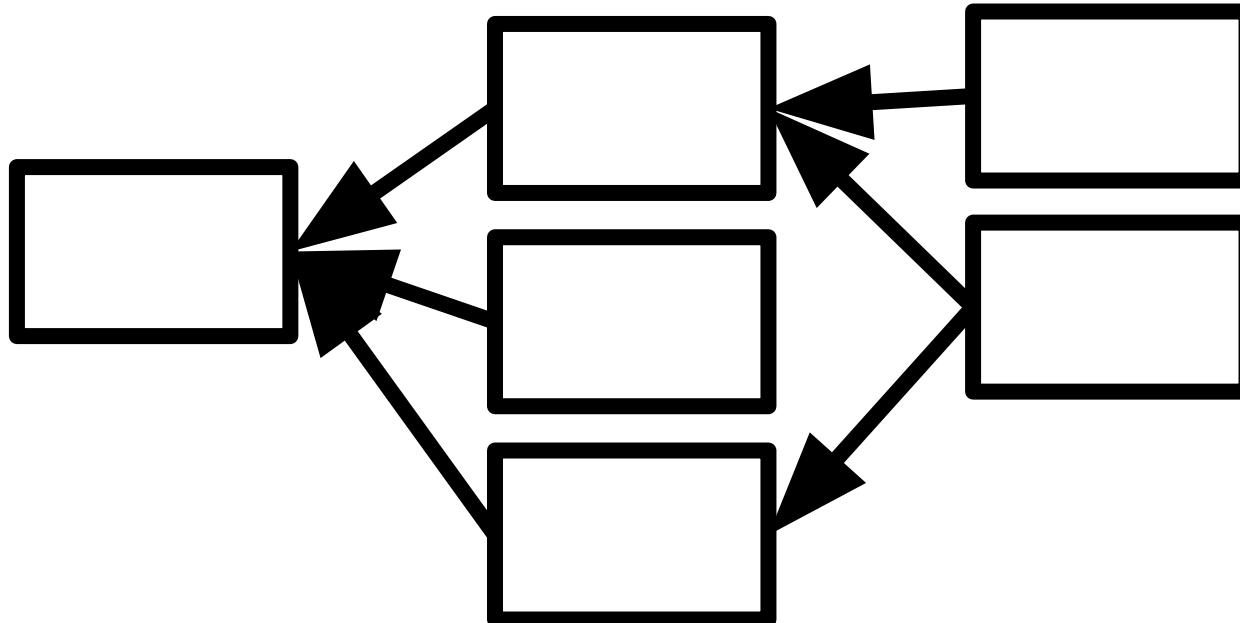
Breaking Changes

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

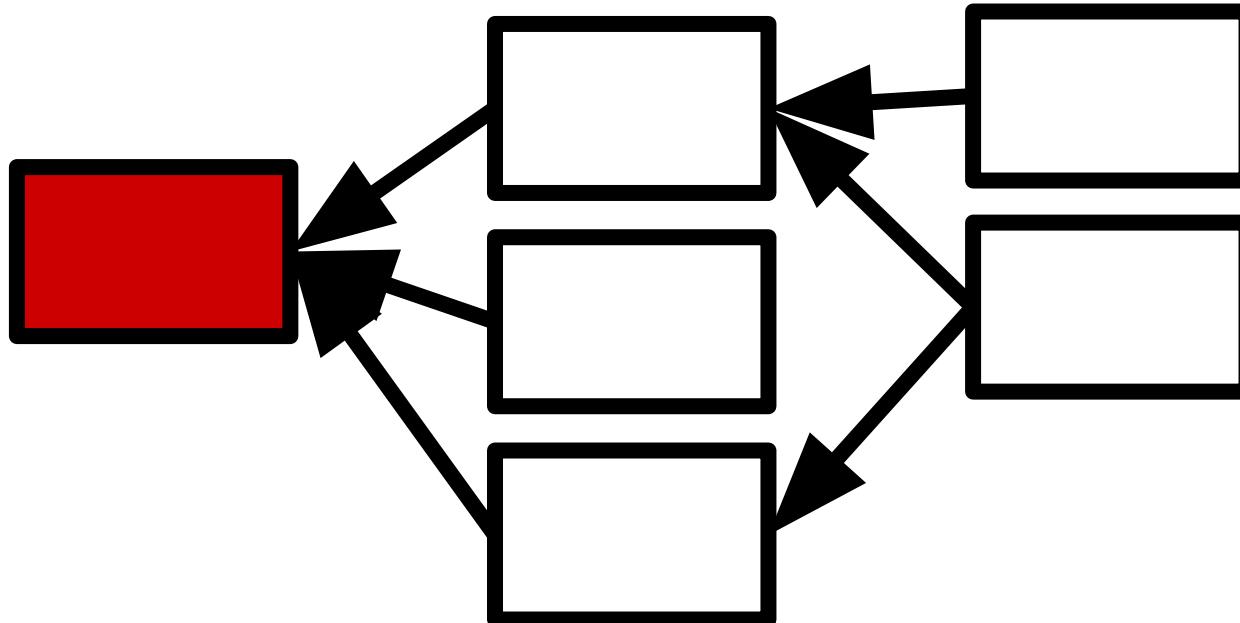
Clients may need to change their implementation when they update

or even migrate to other library

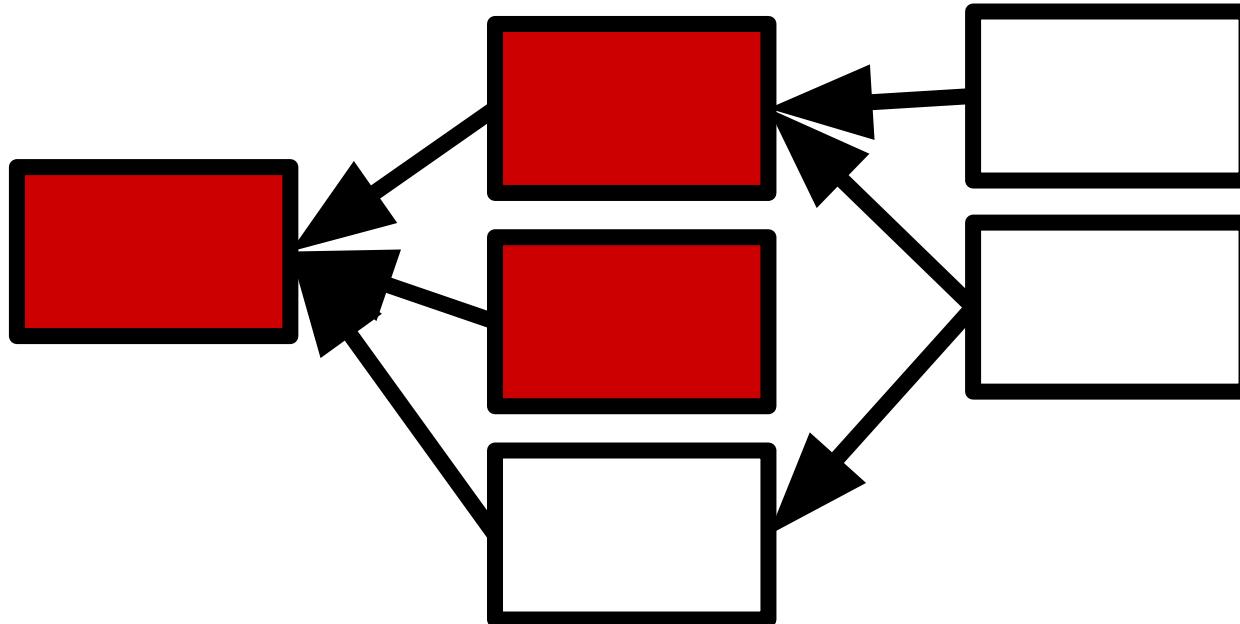
May cause costs for rework and interruption, may ripple through ecosystem



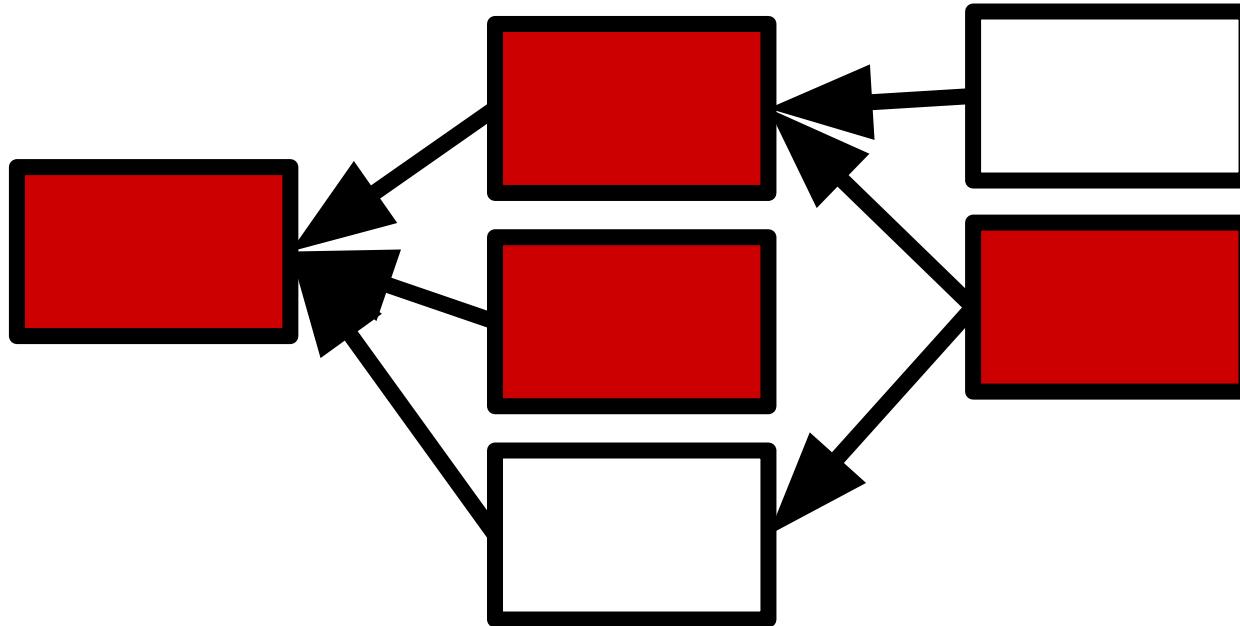
Software Ecosystem



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

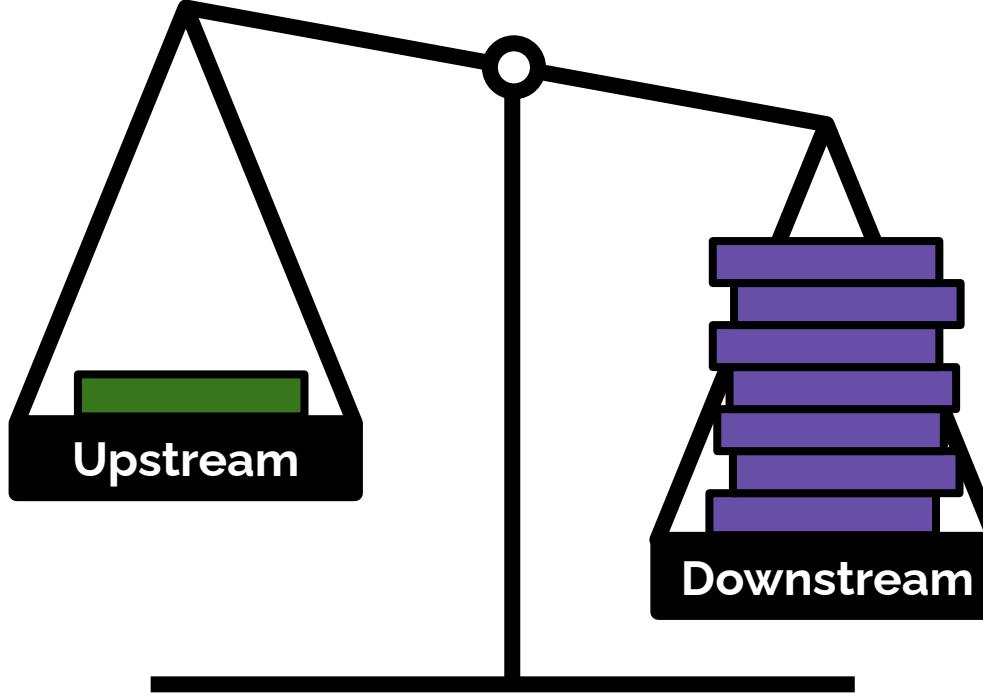
Inefficiencies

Rippling effects from upstream changes

Technical debt, style

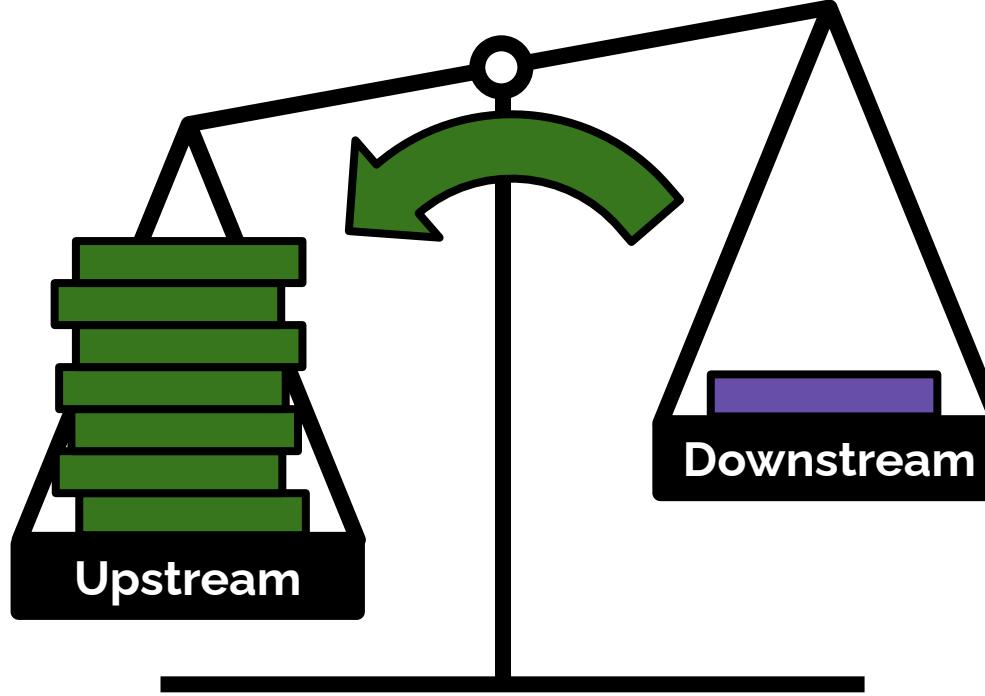
Breaking changes cause costs

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

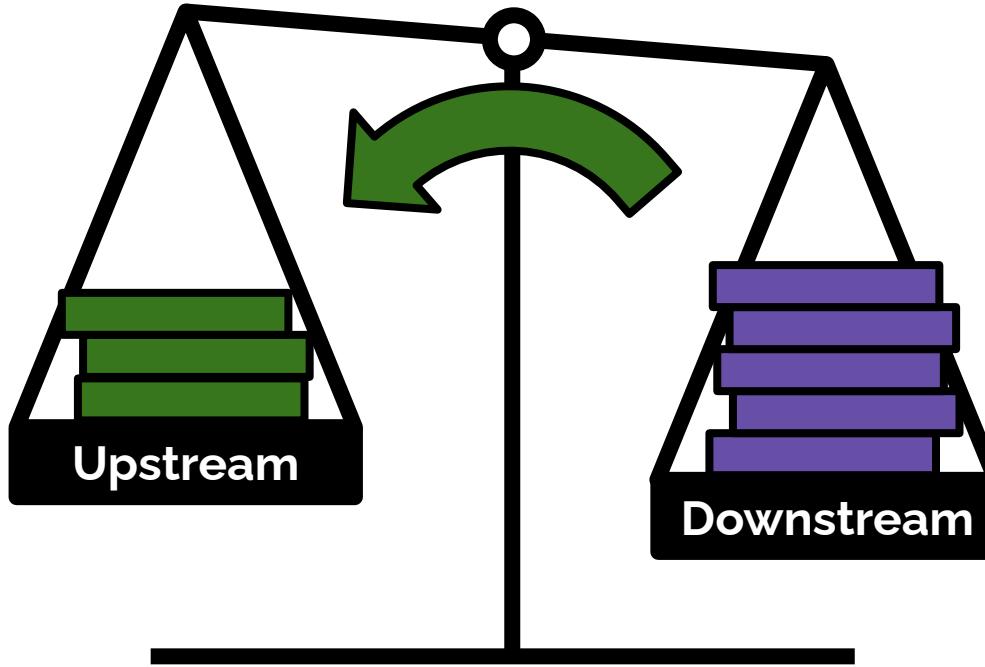


By default, rework and interruption costs for downstream users

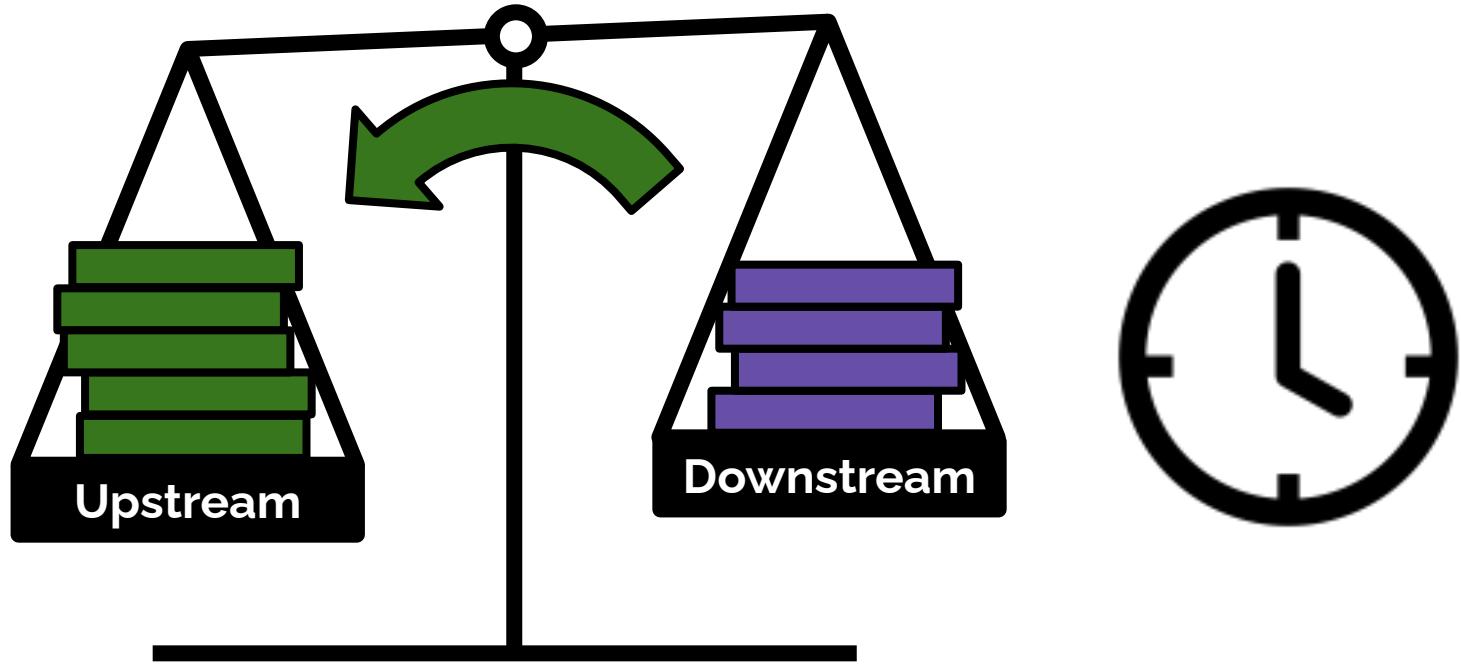
How to reduce costs for downstream users?



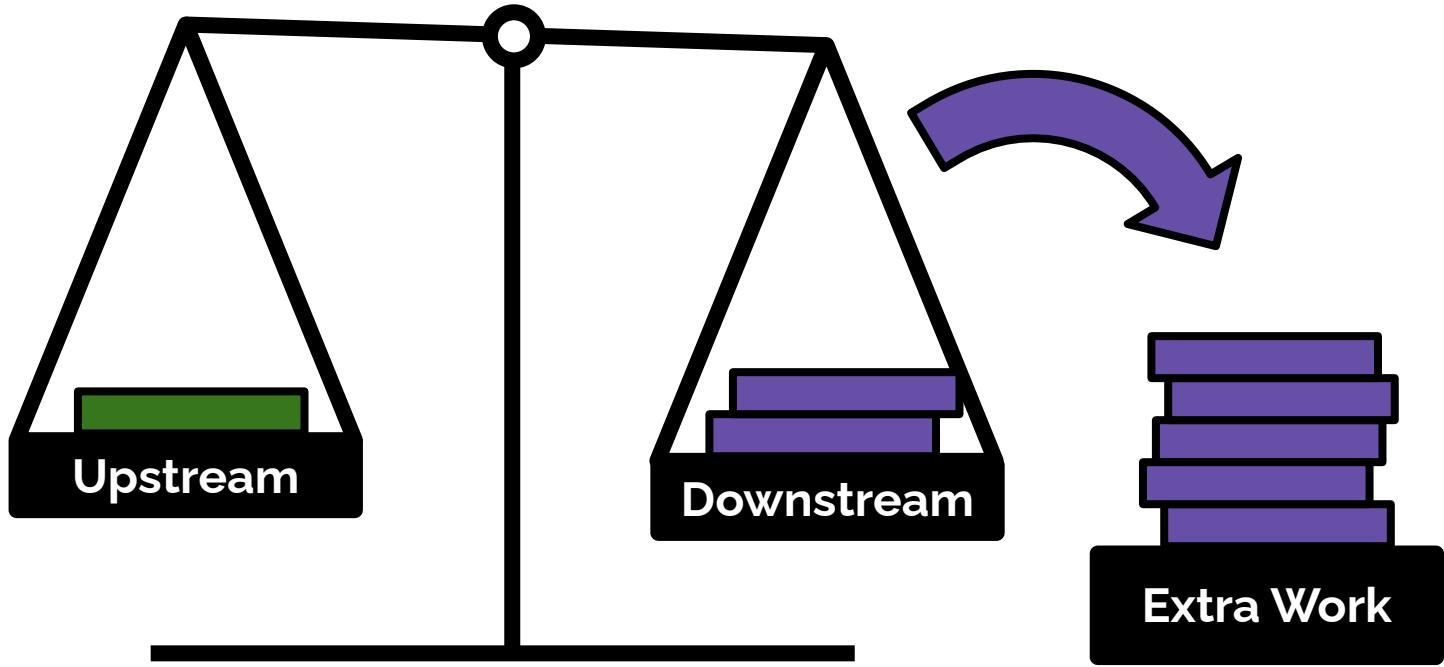
Not making a change
(opportunity costs, technical debt)



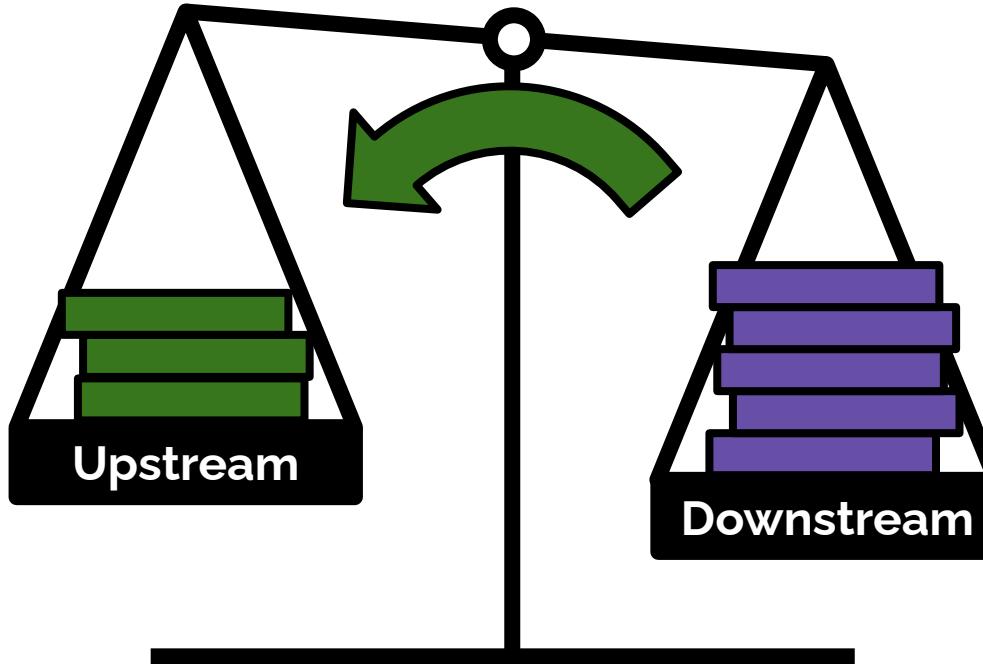
Announcements
Documentation
Migration guide



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

Cost distributions and practices are community dependent





Backward compatibility to
reduce costs for **clients**

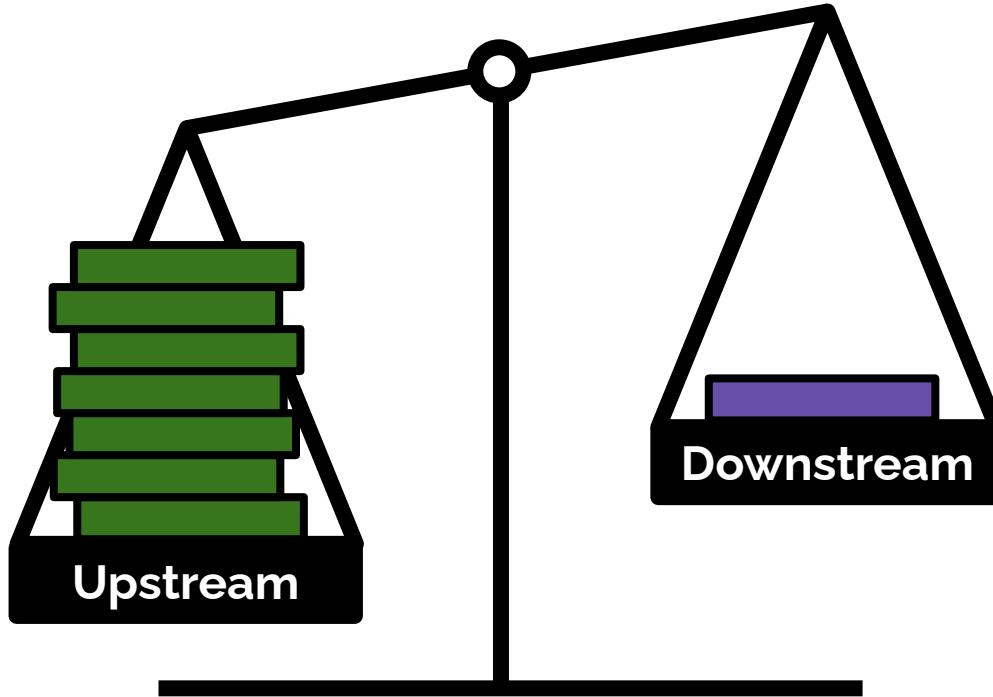
*“API Prime Directive: When
evolving the Component API
from release to release, do
not break existing Clients”*

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

Values



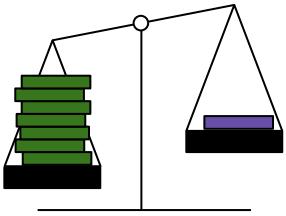
Backward compatibility
for clients



Yearly synchronized
coordinated releases



Backward compatibility
for clients

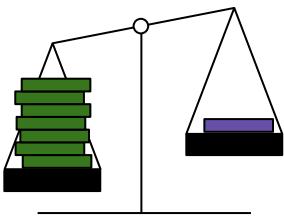


- Willing to accept high costs + opportunity costs
- Educational material, workarounds
- API tools for checking
- Coordinated release planning
- No parallel releases

Upstream



Backward compatibility
for clients

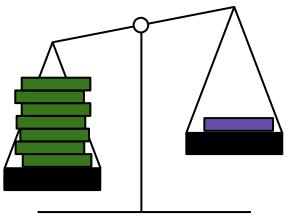


Convenient to use as resource
Yearly updates sufficient for many
Stability for corporate users

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.”





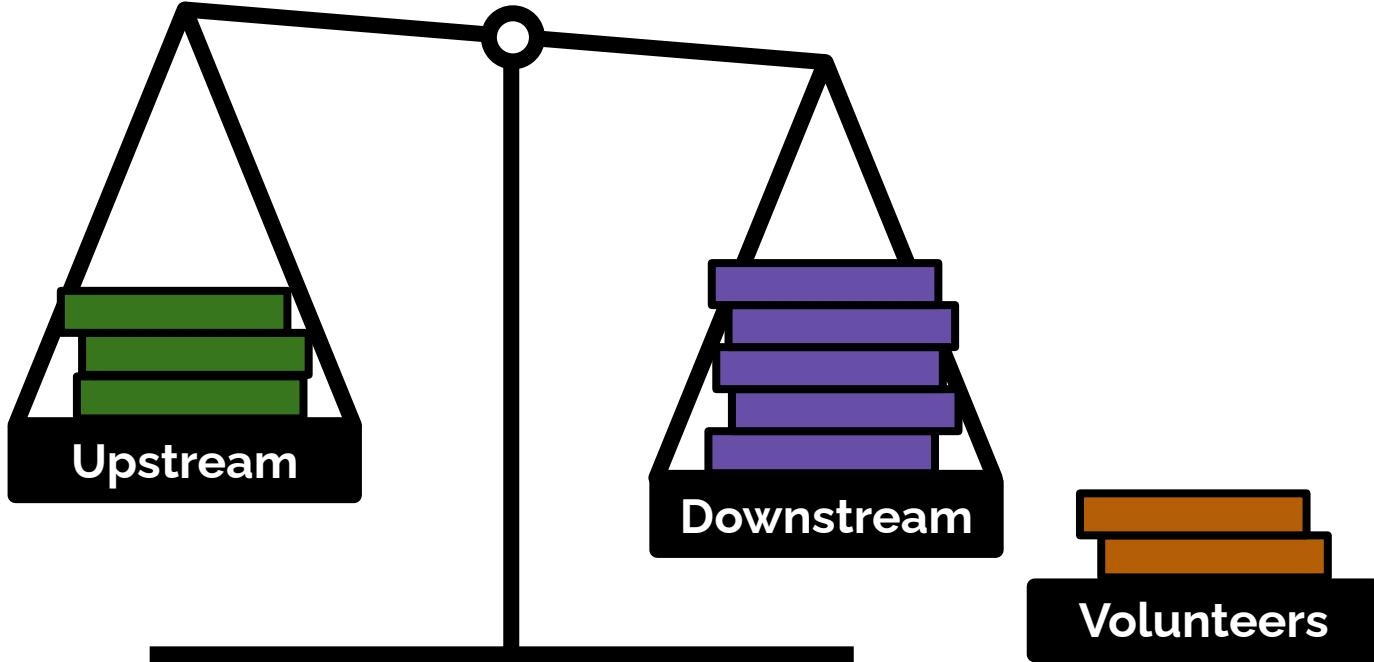
Ease for **end users** to install
and update packages

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

Values



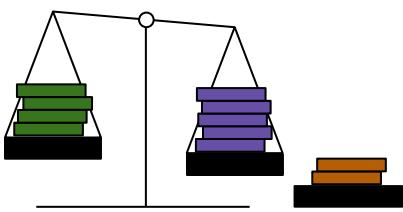
Timely access to
current research
for end users



Continuous synchronization,
~1 month lag



Timely access to
current research
for end users



Snapshot consistency within the ecosystem (not outside)

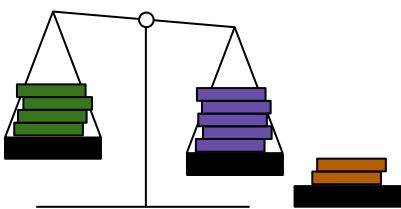
Reach out to affected downstream developers: resolve before release

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

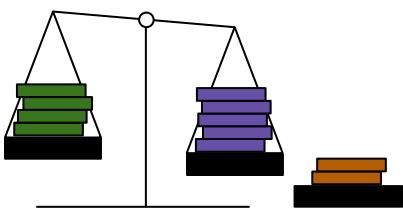
Dependency = collaboration

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.”





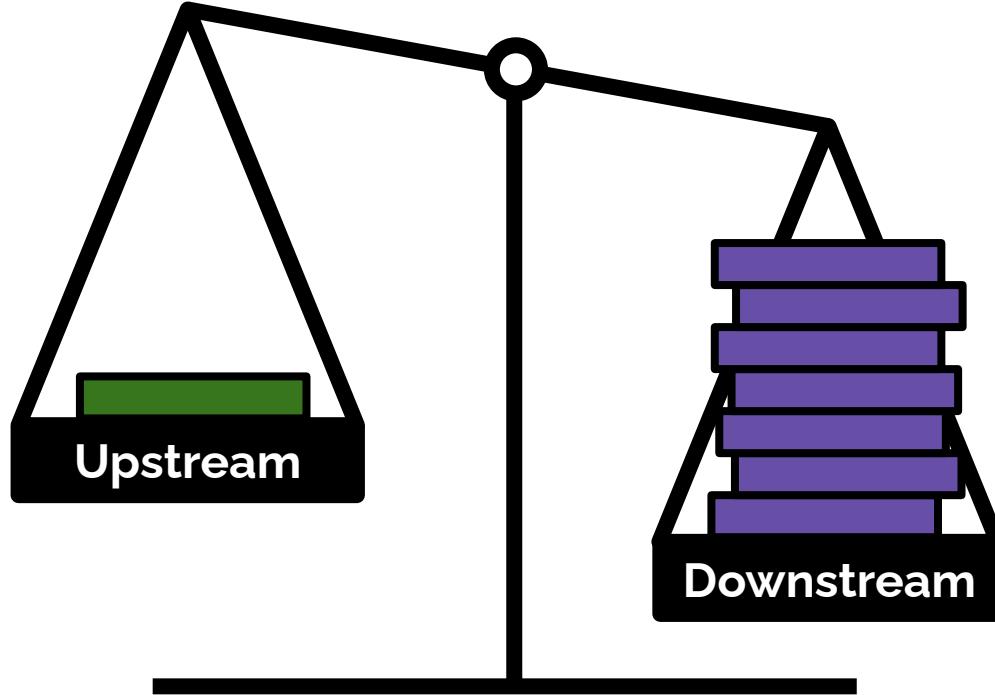
Easy and fast for **developers** to publish and use packages

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

Values



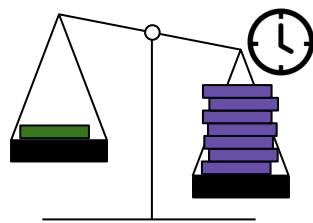
Easy and fast to
publish and use
for developers



Decoupled pace, update
at user's discretion



Easy and fast to
publish and use
for developers

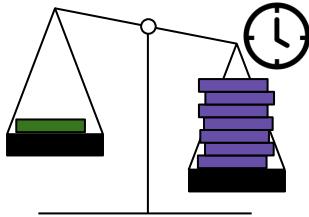


Breaking changes easy
More common to remove technical
debt, fix APIs
Signaling intention with SemVer
No central release planning
Parallel releases more common

Upstream



Easy and fast to publish and use for developers



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

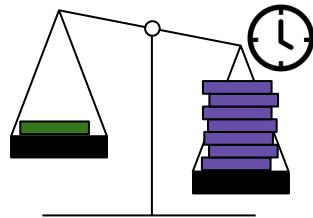
Choice to stay up to date

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

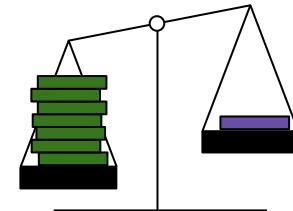
Friction

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

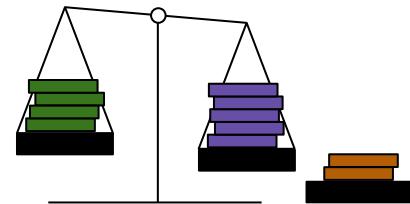
Contrast



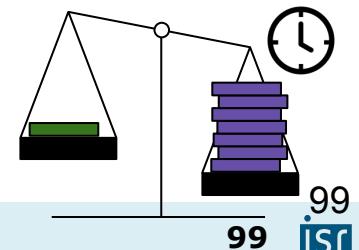
Backward compatibility
for clients



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 slavishly, but...
 - Don't violate them without good reason

Principles of Software Construction: Objects, Design, and Concurrency

Design for Robustness: Distributed Systems

Christian Kästner



Vincent Hellendoorn



Outline

- Intro to distributed systems
- Robustness and Failures
- Testing large/distributed systems
 - Mocks, Stubs

Where we are

	<i>Small scale:</i> One/few objects	<i>Mid scale:</i> Many objects	<i>Large scale:</i> Subsystems
<i>Design for understanding change/ext. reuse robustness ...</i>	Subtype Polymorphism ✓ Information Hiding, Contracts ✓ Immutability ✓ Types Unit Testing ✓	Domain Analysis ✓ Inheritance & Del. ✓ 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

Recap: Designing for Robustness

- Single-threaded, local systems:
 - Problems are (usually) deterministic
 - Checked vs. unchecked exceptions
- Key ideas:
 - ???

Recap: Designing for Robustness

- Single-threaded, local systems:
 - Problems are (usually) deterministic
 - Checked vs. unchecked exceptions
- Key ideas:
 - Provide explicit control-flow for normal and abnormal execution
 - Error handling and recovery for the latter
 - Unit testing to increase confidence
 - Cover both typical and atypical/boundary paths

Recap: Designing for Robustness

- Concurrent, local systems:
 - Non-determinism from thread ordering, asynchronous returns
 - Errors can occur at any shared mutable state
- Key ideas:
 - ???

Recap: Designing for Robustness

- Concurrent, local systems:
 - Non-determinism from thread ordering, asynchronous returns
 - Errors can occur at any shared mutable state
- Key ideas:
 - Reduce mutable state
 - Use atomicity, synchronization everywhere else
 - Organize asynchrony with promises
 - Especially natural in a single-threaded environment

Designing for Robustness

- Key ideas:
 - Provide explicit control-flow for normal and abnormal execution
 - Error handling and recovery for the latter
 - Test normal and abnormal execution

Designing for Robustness

- Key ideas:
 - Provide explicit control-flow for normal and abnormal execution
 - Error handling and recovery for the latter
 - Test normal and abnormal execution
- Until now, most of the program was under our control
 - What if something goes wrong and it's not our fault?
 - What if the system is too big to test?

What is a distributed system?

“A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.”

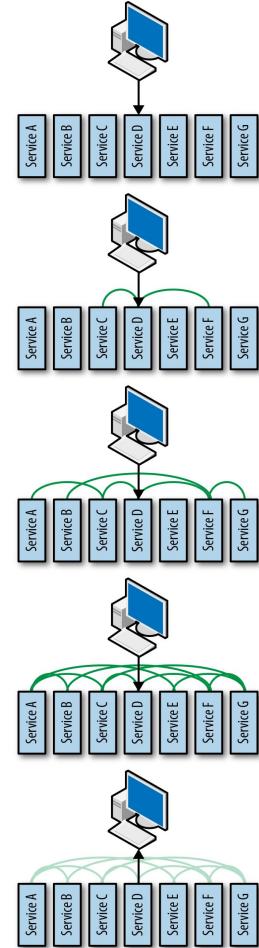
-- Leslie Lamport

What is a distributed system?

- Multiple system components (computers) communicating via some medium (the network) to achieve some goal
- “**Concurrent**” (shared-memory multiprocessing) vs. **Distributed**
 - **Agents:** **Threads** vs. **Processes**
 - Processes typically spread across multiple computers
 - Can put them on one computer for testing
 - **Communication:** **changes to Shared Objects** vs. **Network Messages**

Distributed systems

- A collection of autonomous systems working together to form a single system
 - Enable scalability, availability, resiliency, performance, etc ...



Designing for Robustness

- Concurrent, distributed systems:
 - Non-determinism risks almost everywhere
 - Left-pad gone? Better not rebuild your apps.
 - DB busy? Queries could time out.
 - Use any API? Prepare for down-time
 - Errors can occur at any external call
- Key ideas:
 - ???

What will you do if

- An API your data plugin uses is temporarily down?
 - Or returns a surprising error code

Retry!

- Maybe wait a bit.
 - How Long? How often?

Retry!

- Exponential Backoff
 - Retry, but wait exponentially longer each time
 - Assumes that failures are exponentially distributed
 - E.g., a 10h outage is extremely rare, a 10s one not so crazy
 - E.g.:

```
const delay = retryCount => new Promise(resolve =>
    setTimeout(resolve, 10 ** retryCount));

const getResource = async (retryCount = 0, lastError = null) => {
  if (retryCount > 5) throw new Error(lastError);
  try {
    return apiCall();
  } catch (e) {
    await delay(retryCount);
    return getResource(retryCount + 1, e);
  }
}
```

17- } https://www.bayanbennett.com/posts/retrying-and-exponential-backoff-with-promises/

Retry!

- Still need an exit-strategy
 - Learn [HTTP response codes](#)
 - Don't bother retrying on a 403 (go find out why)
 - Use the API response, if any
 - Errors are often documented -- e.g., GitHub will send a “rate limit exceeded” message

```
const delay = retryCount => new Promise(resolve =>
    setTimeout(resolve, 10 ** retryCount));

const getResource = async (retryCount = 0, lastError = null) => {
  if (retryCount > 5) throw new Error(lastError);
  try {
    return apiCall();
  } catch (e) {
    await delay(retryCount);
    return getResource(retryCount + 1, e);
  }
}
```

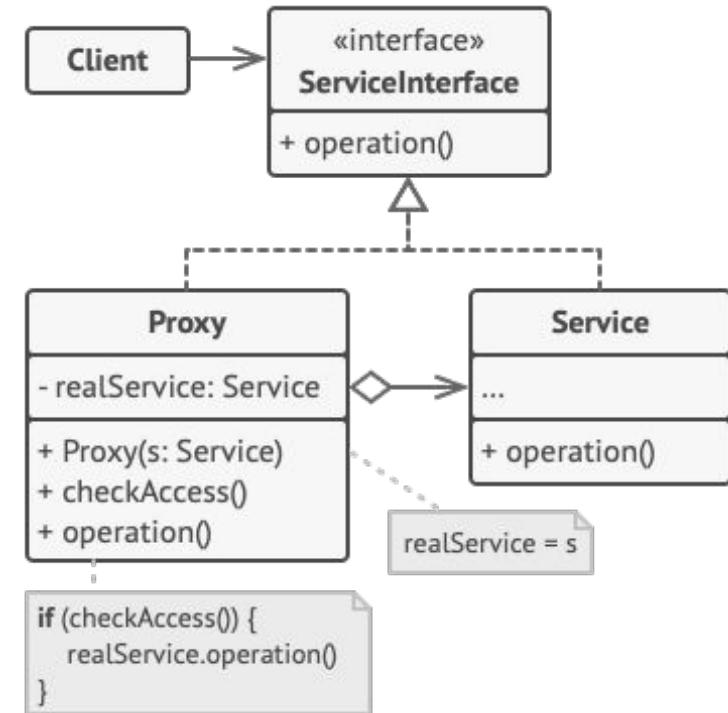
17- } https://www.bayanbennett.com/posts/retrying-and-exponential-backoff-with-promises/

Handling Recovery

- We need a fallback plan
 - Can't just `e.printStackTrace()`
 - What *can* we do?

Proxy Design Pattern

- Local representative for remote object
 - Create expensive obj on-demand
 - Control access to an object
- Hides extra “work” from client
 - Add extra error handling, caching
 - Uses *indirection*



Example: Caching

```
interface FacebookAPI {  
    List<Node> getFriends(String name);  
}  
class FacebookProxy implements FacebookAPI {  
    FacebookAPI api;  
    HashMap<String, List<Node>> cache = new HashMap...  
    FacebookProxy(FacebookAPI api) { this.api=api;}  
  
    List<Node> getFriends(String name) {  
        result = cache.get(name);  
        if (result == null) {  
            result = api.getFriends(name);  
            cache.put(name, result);  
        }  
        return result;  
    }  
}
```

Example: Caching and Failover

```
interface FacebookAPI {  
    List<Node> getFriends(String name);  
}  
class FacebookProxy implements FacebookAPI {  
    FacebookAPI api;  
    HashMap<String, List<Node>> cache = new HashMap...  
    FacebookProxy(FacebookAPI api) { this.api=api;}  
  
    List<Node> getFriends(String name) {  
        try {  
            result = api.getFriends(name);  
            cache.put(name, result);  
            return result;  
        } catch (ConnectionException c) {  
            return cache.get(name);  
        }  
    }  
}
```

Example: Redirect to Local Service

```
interface FacebookAPI {  
    List<Node> getFriends(String name);  
}  
class FacebookProxy implements FacebookAPI {  
    FacebookAPI api;  
    FacebookAPI fallbackApi;  
    FacebookProxy(FacebookAPI api, FacebookAPI f) {  
        this.api=api; fallbackApi = f; }  
  
    List<Node> getFriends(String name) {  
        try {  
            return api.getFriends(name);  
        } catch (ConnectionException c) {  
            return fallbackApi.getFriends(name);  
        }  
    }  
}
```

Principle: Delegating Recovery

- We need a fallback plan
 - Can't just `e.printStackTrace()`
 - What *can* we do?
- In case of failure, redirect
 - If at all plausible, hand work over to proxy
 - Local data(set), fallback service
 - If not, recruit clean-up service
 - Process, display errors

What will you do if

- An API your data plugin uses is temporarily down?
 - Or returns a surprising error code
- Consider caching
 - E.g., store last Twitter feed, Target shopping card offline
 - Not cheap, select caching mechanism carefully
 - If user-facing: be transparent about offline status

What will you do if

- Your visualization plugin's latest version has a vulnerability?

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 receive updates.
npm [WARN] deprecated chokidar@2.1.8: Chokidar 2 will break on node v14+. Upgrade to chokidar 3 with 15x less dependencies.
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 URLSearchParams 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 use @rollup/plugin-babel.
npm [WARN] deprecated fsevents@1.2.13: fsevents 1 will break on node v14+ and could be using insecure binaries. Upgrade to fsevents 2.
npm [WARN] deprecated uuid@3.4.0: Please upgrade to version 7 or higher. Older versions may use Math.random() in certain 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 URLSearchParams API instead.
npm [WARN] deprecated sane@4.1.0: some dependency vulnerabilities fixed, support for node < 10 dropped, and newer ECMAScript 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
```

Ever looked at NPM Install's output?

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

Vulnerabilities in Distributed Systems

- A lot of software relies on vulnerable code somewhere deep down
 - Often not disclosed/discovered for quite a while
 - By then, it could be everywhere
- What can you do?
 - Routinely check using tools (e.g. dependabot, CI is great)
 - Upgrade/downgrade where possible, ditch bad packages otherwise
 - Area of active research



NPM package with 3 million weekly downloads had a severe vulnerability

What will you do if

- Facebook withdraws its DNS routing information?

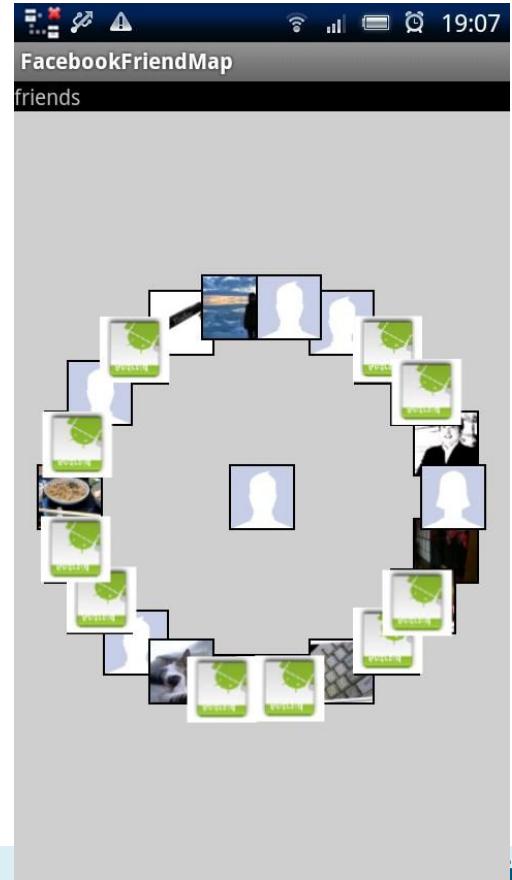
<https://blog.cloudflare.com/october-2021-facebook-outage/>

Testing Distributed Systems

- Challenges:
 - Volatility
 - Real-world effects -- things crashing, delays.
 - Users are hard to simulate
 - Performance
 - Massive databases? Systems with minutes-long start-up times?
 - Very common in ML

For example

- 3rd party Facebook apps
- Android user interface
- Backend uses Facebook data



Testing in real environments



```
void buttonClicked() {  
    render(getFriends());  
}  
List<Friend> getFriends() {  
    Connection c = http.getConnection();  
    FacebookAPI api = new FacebookAPI(c);  
    List<Node> persons = api.getFriends("john");  
    for (Node person1 : persons) {  
        ...  
    }  
    return result;  
}
```

A code snippet in blue and black text. It defines a method 'buttonClicked' that calls 'render(getFriends())'. The 'getFriends' method creates a 'Connection' object, initializes a 'FacebookAPI' object, calls 'api.getFriends("john")' to get a list of nodes, loops through them, and finally returns the result. The code ends with a closing brace at the bottom.

Eliminating Android dependency



```
@Test void testGetFriends() {  
    assert getFriends() == ...;  
}  
List<Friend> getFriends() {  
    Connection c = http.getConnection();  
    FacebookAPI api = new FacebookAPI(c);  
    List<Node> persons = api.getFriends("john");  
    for (Node person1 : persons) {  
        ...  
    }  
    return result;
```

That won't quite work

- **GUI applications process *thousands* of events**
- Solution: automated GUI testing frameworks
 - Allow streams of GUI events to be captured, replayed
- These tools are sometimes called *robots*

Eliminating Android dependency



```
@Test void testGetFriends() {  
    assert getFriends() == ...;  
}  
List<Friend> getFriends() {  
    Connection c = http.getConnection();  
    FacebookAPI api = new FacebookAPI(c);  
    List<Node> persons = api.getFriends("john");  
    for (Node person1 : persons) {  
        ...  
    }  
    return result;
```

How about this one?

A callout box with a black border and white background is positioned to the right of the code. It contains the text "How about this one?". An arrow points from the word "new" in the highlighted line of code to the top edge of the callout box.

Test Doubles

- Stand in for a real object under test
- Elements on which the unit testing depends (i.e. collaborators), but need to be approximated because they are
 - Unavailable
 - Expensive
 - Opaque
 - Non-deterministic
- Not just for distributed systems!



<http://www.kickvick.com/celebrities-stunt-doubles>

How Test Doubles Help

1. Speed: simulate response without going through the API

```
class FakeFacebook implements FacebookInterface {  
    void connect() {}  
    List<Node> getFriends(String name) {  
        if ("john".equals(name)) {  
            List<Node> result=new List();  
            result.add(...);  
            return result;  
        }  
    }  
}
```

How Test Doubles Help

1. Speed: simulate response without going through the API
2. Stability: guaranteed deterministic return, reduces flakiness

```
class FakeFacebook implements FacebookInterface {  
    void connect() {}  
    List<Node> getFriends(String name) {  
        if ("john".equals(name)) {  
            List<Node> result=new List();  
            result.add(...);  
            return result;  
        }  
    }  
}
```

How Test Doubles Help

1. Speed: simulate response without going through the API
2. Stability: guaranteed deterministic return, reduces flakiness
3. Coverage: reliably simulate problems (e.g., return 404)

```
class FakeFacebook implements FacebookInterface {  
    void connect() {}  
    List<Node> getFriends(String name) {  
        if ("john".equals(name)) {  
            List<Node> result=new List();  
            result.add(...);  
            return result;  
        }  
    }  
}
```

How Test Doubles Help

1. Speed: simulate response without going through the API
2. Stability: guaranteed deterministic return, reduces flakiness
3. Coverage: reliably simulate problems (e.g., return 404)
4. Insight: expose internal state

```
class FakeFacebook implements FacebookInterface {  
    void connect() {}  
    List<Node> getFriends(String name) {  
        if ("john".equals(name)) {  
            List<Node> result=new List();  
            result.add(...);  
            return result;  
        }  
    }  
}
```

How Test Doubles Help

1. Speed: simulate response without going through the API
2. Stability: guaranteed deterministic return, reduces flakiness
3. Coverage: reliably simulate problems (e.g., return 404)
4. Insight: expose internal state
5. Development: presume functionality not yet implemented

```
class FakeFacebook implements FacebookInterface {  
    void connect() {}  
    List<Node> getFriends(String name) {  
        if ("john".equals(name)) {  
            List<Node> result=new List();  
            result.add(...);  
            return result;  
        }  
    }  
}
```

Types of Test Doubles

- Most often talk about Mocks and Stubs
 - Technically, a few other categories, see next slide
- Mocks give you a lot of power
 - Dictate what should be returned when (very broadly construed)
 - Requires framework using reflection
 - E.g., Mockito in Java; Mock functions in Jest*
- Stubs are way simpler; use when possible

*<https://jestjs.io/docs/mock-functions>

Design Implications

- Think about testability when writing code
- When a mock may be appropriate, design for it
- Hide subsystems behind an interface
- Use factories, not constructors to instantiate
- Use appropriate tools
 - Dependency injection or mocking frameworks

What will you do if

- Facebook withdraws its DNS routing information?
 - Fact-of-life: be prepared (test for this)
 - Reduce coupling; don't let someone else's outage cripple your program
 - Like separating your GUI from the backend

<https://blog.cloudflare.com/october-2021-facebook-outage/>

Designing for Robustness

- As a *client* of distributed systems (mainly the Internet):
 - No harm trying again (redundancy)
 - Have a backup plan (resiliency)
 - Maintain awareness of what can go wrong (transparency)
 - HTTP status codes, API documentation, keeping tabs on vulnerabilities

Designing for Robustness

- As a *client* of distributed systems (mainly the Internet):
 - No harm trying again (redundancy)
 - Have a backup plan (resiliency)
 - Maintain awareness of what can go wrong (transparency)
 - HTTP status codes, API documentation, keeping tabs on vulnerabilities
 - **Isolation, isolation, isolation**
 - Use test doubles liberally
 - Rely on protocols to contain and manage failures
 - Never let one module crash another
 - More pointers coming up

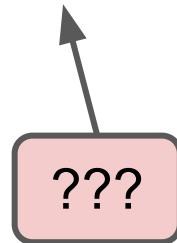
For Application Designers

Some considerations when contributing to *the distributed system*

Why build a distributed system?

Why build a distributed system?

- Unlimited scaling
 - Can be used for capacity or speed
- Geographical dispersion – people and data around the world
- **Robustness** to failures including physical catastrophes



Why build a distributed system?

- Test Santorini all you want, it will die when I turn off my laptop
 - A local server is a Single Point of Failure
- Distributed systems offer robustness through redundancy, duplication
 - Netflix famously unplugs random servers in production

Measuring Robustness

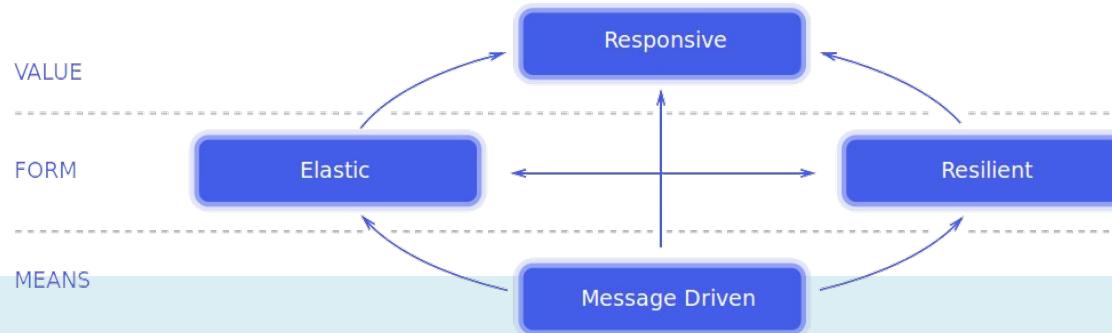
- **Reliability:** works well
 - Often in terms of availability: fraction of time system is working
 - 99.999% available is "5 nines of availability"
- **Performance:** works fast
 - Low latency
 - High throughput
- **Scalability:** adapts well to increased demand
 - Ability to handle workload growth

Robust Distributed System Design

- Consider reading:

<https://www.reactivemanifesto.org>

- Yet another meaning for “Reactive”!
- Short guide identifying key principles
 - Goals: robustness, resilience, flexibility
 - Principles: responsiveness, elasticity, message-driven
 - Patterns/Heuristics: isolation, delegation, verbosity, replication, asynchrony



Principle: Modular Protection

- Errors should be contained and isolated
 - A failing printer should not corrupt a document
 - Handle exceptions locally as much as possible, return useful feedback
 - **Don't do this:**

The screenshot shows an Apache Tomcat 8.0.5 error page with the following details:

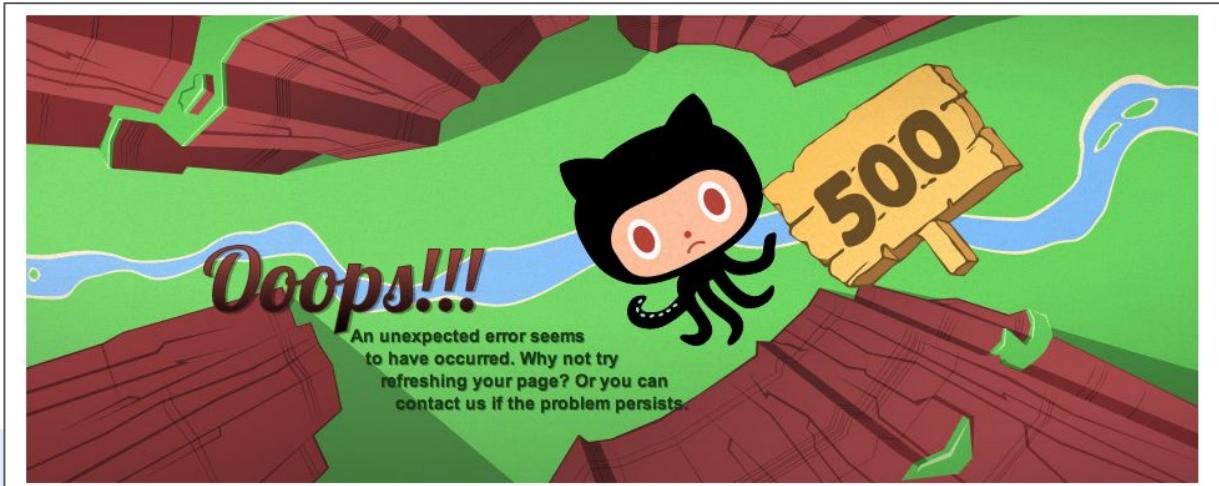
- HTTP Status 500 -**
- type** Exception report
- message**
- description** The server encountered an internal error that prevented it from fulfilling this request.
- exception**

```
java.lang.NullPointerException
    nl.hu.sp.lesson1.dynamicexample.LogoutServlet.doGet(LogoutServlet.java:39)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:618)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:725)
    org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:52)
```
- note** The full stack trace of the root cause is available in the Apache Tomcat/8.0.5 logs.

Apache Tomcat/8.0.5

Principle: Modular Protection

- Online: use HTTP response status codes effectively
 - Don't just hand out 404, 500
 - Unless they really apply
 - Provide and document fall-back options, information
 - Good RESTful design helps



Principle: Delegating Recovery

(Again?)

- Don't make a failing node/module serve a client
 - It needs to clean itself up
 - Forward clients to designated recovery service
 - A bit like the proxy pattern
 - Consider asynchrony
 - Failure is often expensive

Principle: Consider Idempotence

- Idempotency: the same call from the same context should have the same result
 - Hitting “Pay” twice should not cost you double!
 - A resource should not suddenly switch from JSON to XML
 - Makes APIs predictable, resilient

Ensuring Idempotence

- Fairly easy for read-only requests
 - Ensure consistency of read-only data
 - Never attach side-effects to GET requests*
- Also for updates, deletes
 - Not “safe”, because data is mutated
 - Natural idempotency because the target is identified
- How about writing/sending new data?

*<https://twitter.com/rombulow/status/990684463007907840>

Ensuring Idempotence

- How about writing/sending new data?
 - Could fail anywhere
 - Including in displaying success message after payment!
 - POST is not idempotent
 - Use Unique Identifiers
 - Server keeps track of requests already handled

<https://stripe.com/blog/idempotency>

```
curl https://api.stripe.com/v1/charges \
-u sk_test_BQokikJ0vBiI2HlWgH4olfQ2: \
-H "Idempotency-Key: AGJ6FJMKGQIpHUTX" \
-d amount=2000 \
-d currency=usd \
-d description="Charge for Brandur" \
-d customer=cus_A8Z5MHwQS7jUmZ
```

Distributed Systems

There are entire courses on getting these right; not a goal here

But do:

- Understand challenges and solutions to achieving robustness
 - Primarily as a *client* of a distributed system (we all are these days)
 - Test for all scenarios, leveraging test doubles
 - Provide error handling through isolation
- Learn to communicate with, and provide your own, nodes
 - API design, last week
 - Microservices, next week

Principles of Software Construction: Objects, Design, and Concurrency

Organizing Systems at Scale: Modules, Services, Architectures

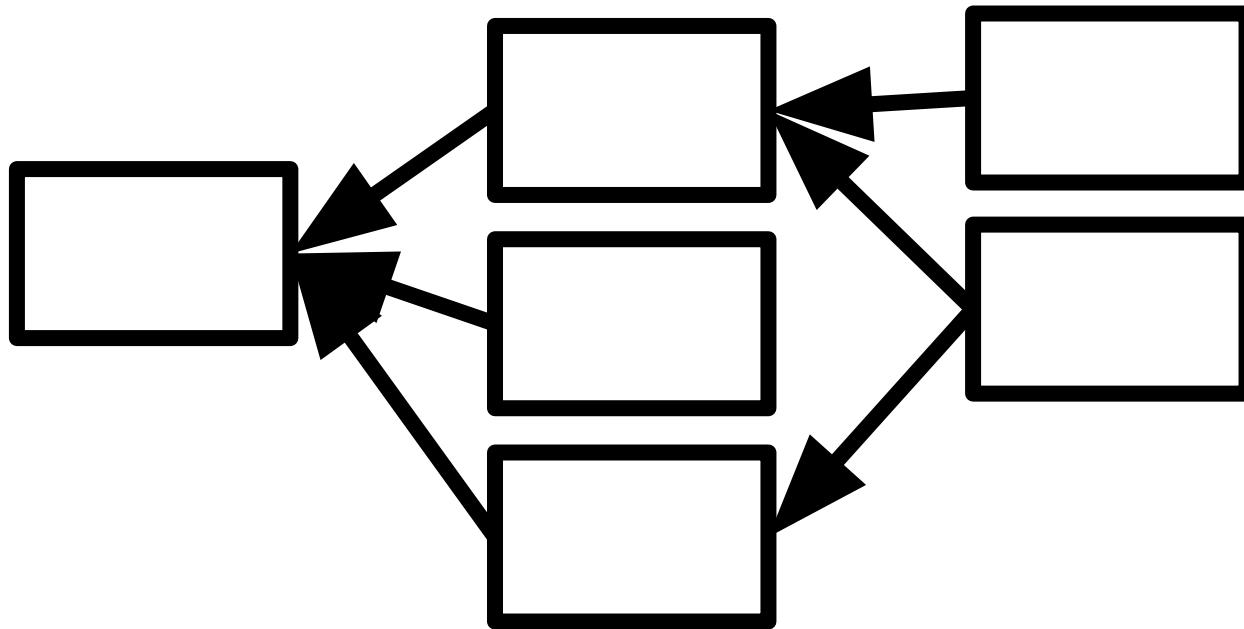
Christian Kästner Vincent Hellendoorn



Where we are

	<i>Small scale:</i> One/few objects	<i>Mid scale:</i> Many objects	<i>Large scale:</i> Subsystems
<i>Design for understanding change/ext. reuse robustness ...</i>	Subtype Polymorphism ✓ Information Hiding, Contracts ✓ Immutability ✓ Types Unit Testing ✓	Domain Analysis ✓ Inheritance & Del. ✓ 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

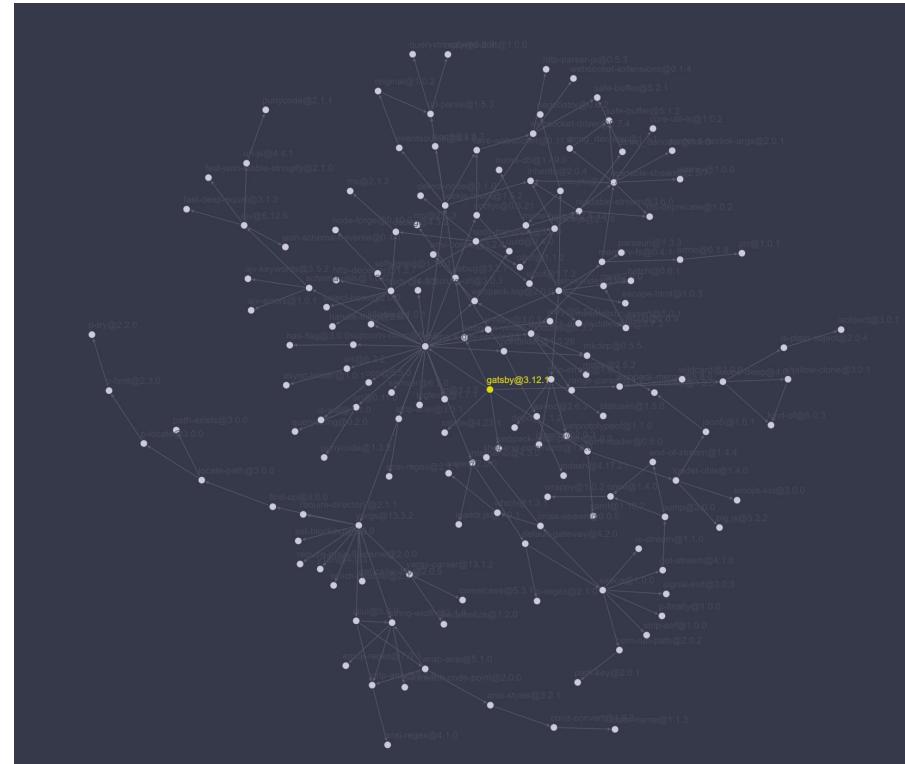
Software Supply Chains / Software Ecosystems



Recall: Modern Software Engineering

Nobody wants to write a million lines of code.

- Instead, you use libraries
 - E.g., import Android => +12M LOC
 - You don't write most of the code you use
 - And why would you want to?
- And your libraries use libraries
 - Et cetera
 - <https://npm.anvaka.com/#/view/2d/gatsby>



Search open source packages, frameworks and tools...

Search

Libraries.io monitors **6,216,328** open source packages across **32** different package managers, so you don't have to. [Find out more](#)

Discover new software

Search 6.22M packages by [license](#), [language](#) or [keyword](#), or explore new, trending or popular packages.

Explore

Monitor your dependencies

Stay up to date with notifications of updates, license incompatibilities or deleted dependencies.

Login

Maintain your OSS project

Understand your users and make informed decisions about features with usage and version data.

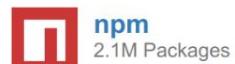
Login

Use Libraries.io data

Use Libraries.io data in your applications, services or research. Use our [API](#) to stay up to date.

Documentation

Supported Package Managers



npm

2.1M Packages



Go

444K Packages



Maven

443K Packages



PyPI

405K Packages



Packagist

1/214/514



NuGet



Rubygems



CocoaPods

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

Module Systems

Foundation for distributing and reusing libraries

Packaging code (binary / source)

Linking against code in a module without knowing
internals

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

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

```
<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

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 }
```

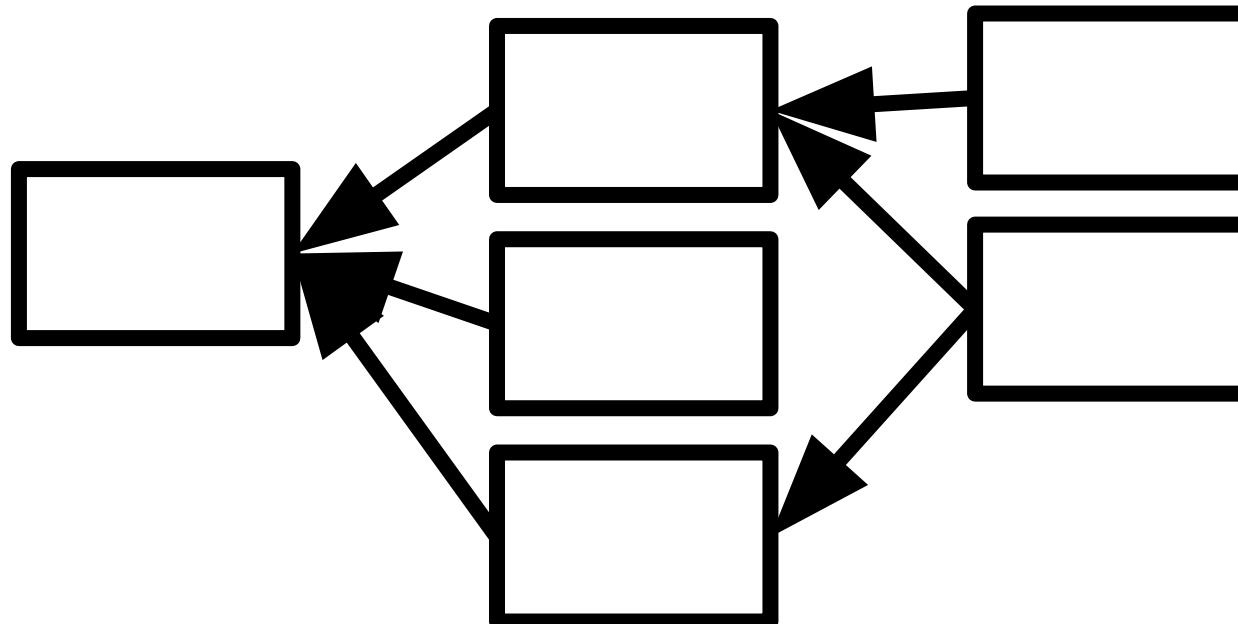
JavaScript Modules and Packages

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

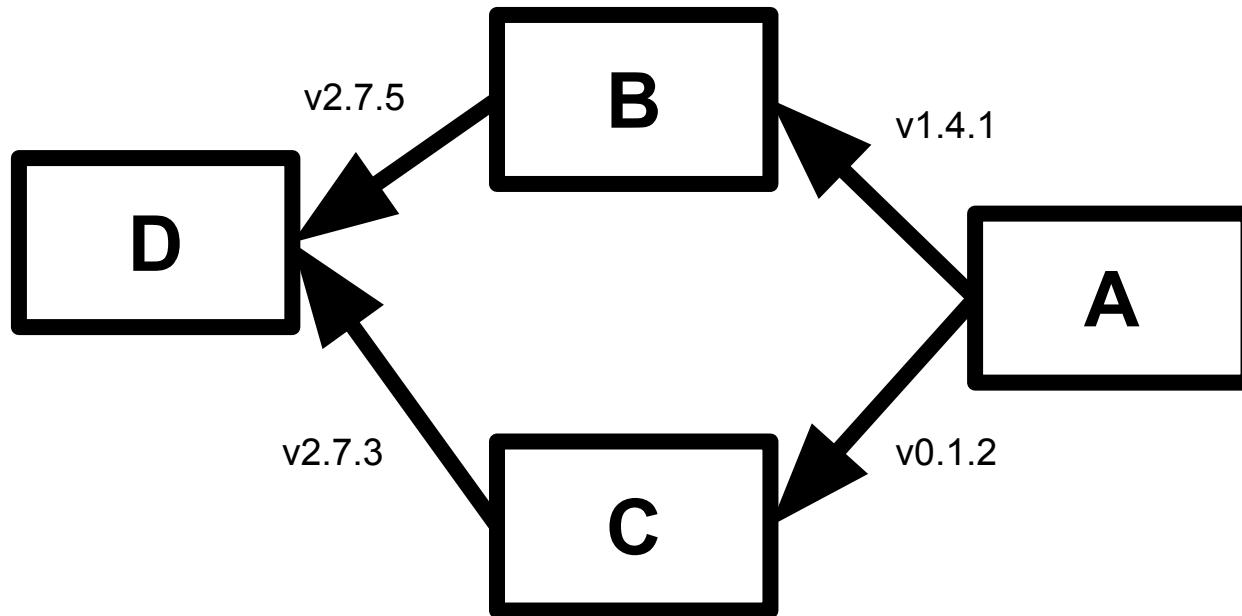
Dependency Graphs



Acyclic

Versioned dependency edges

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

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 receive updates.
npm [WARN] deprecated chokidar@2.1.8: Chokidar 2 will break on node v14+. Upgrade to chokidar 3 with 15x less dependencies.
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 URLSearchParams 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 use @rollup/plugin-babel.
npm [WARN] deprecated fsevents@1.2.13: fsevents 1 will break on node v14+ and could be using insecure binaries. Upgrade to fsevents 2.
npm [WARN] deprecated uuid@3.4.0: Please upgrade to version 7 or higher. Older versions may use Math.random() in certain 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 URLSearchParams API instead.
npm [WARN] deprecated sane@4.1.0: some dependency vulnerabilities fixed, support for node < 10 dropped, and newer ECMAScript 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

EQUIFAX

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?

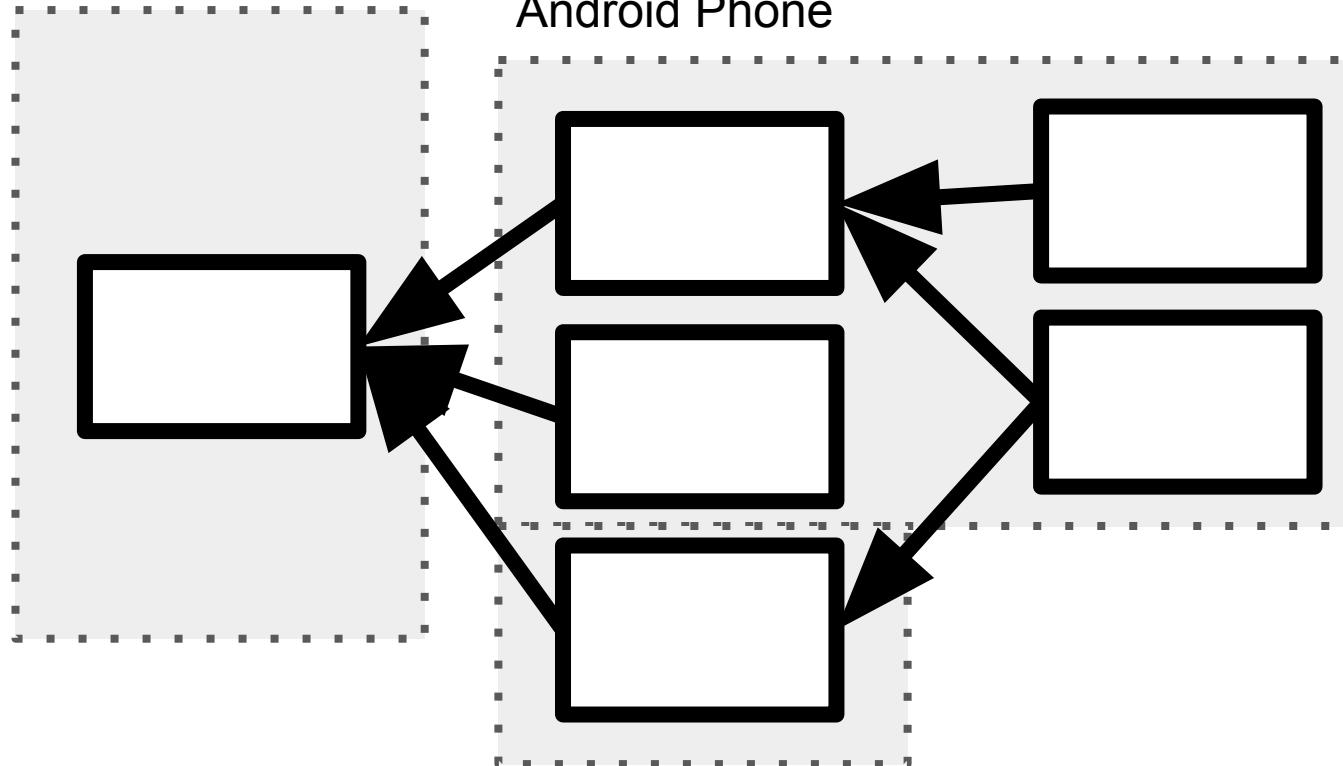
Who funds open source?

Commercial dependencies? Commercial support?

Distributed Modules

Database Server

Android Phone



Credit card server

Distributed Systems

Remote procedure calls instead of function calls

Typically REST API to URL

Benefits? Drawbacks?

Distributed System Benefits

Scalability

Very strong encapsulation (only APIs public)

Computation beyond local resources

Independent deployment, operations, and evolution

Also multiple containers on single system

Pay per transaction / storage / use

Distributed System Problems

Distributed system problem!

All kinds of new problem scenarios: Unavailable services, delayed responses, missing responses, out of order responses, network segmentation, clock problems, unannounced changes of service behavior

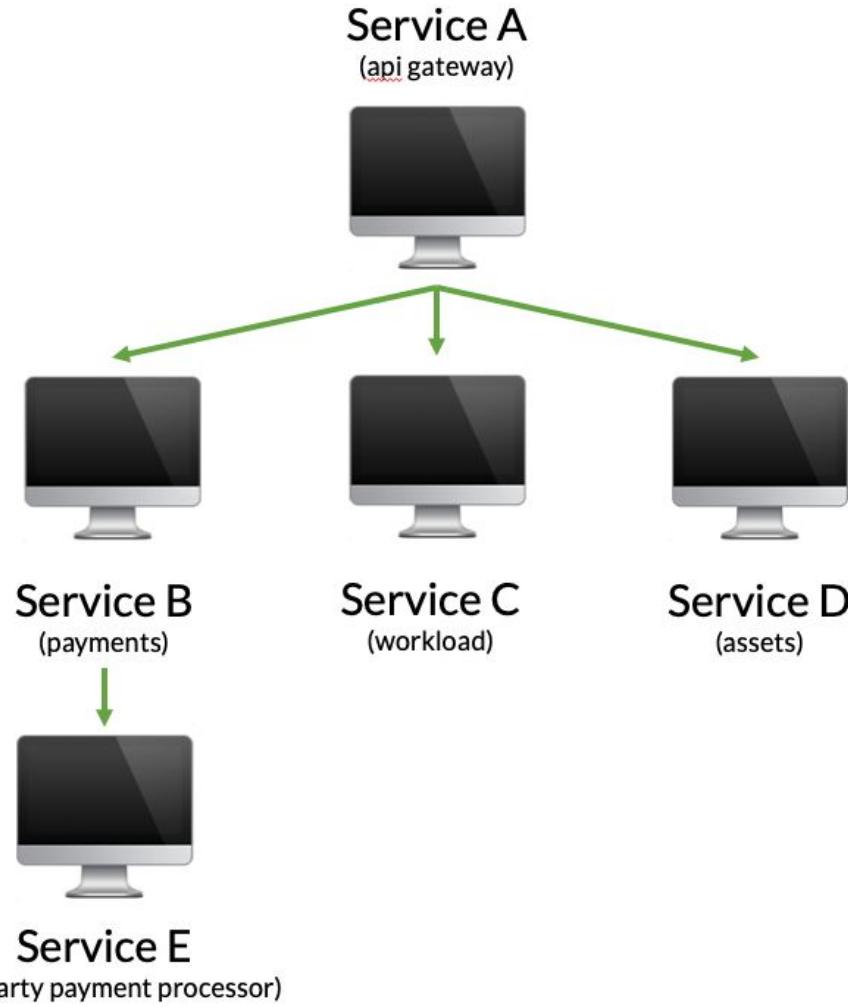
Distributed systems is hard! Do not underestimate! Build on existing abstractions!

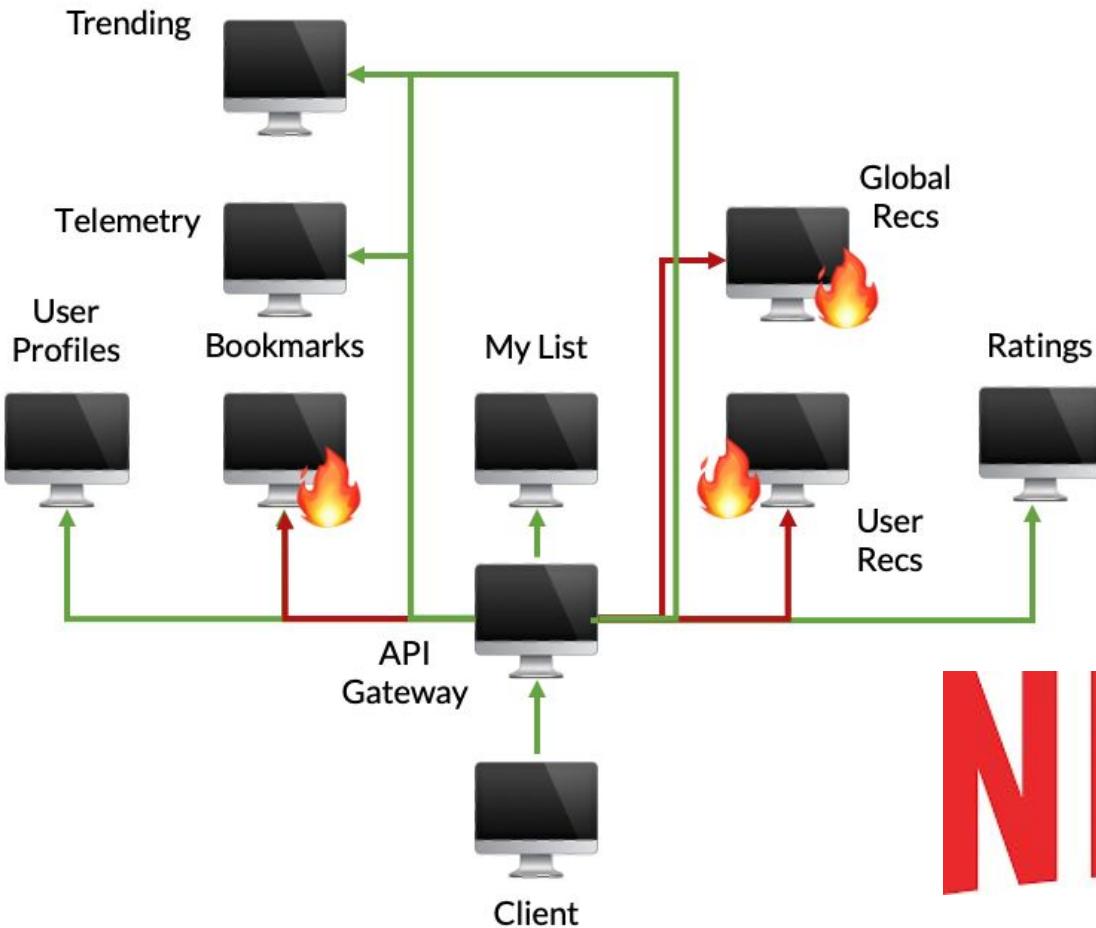
Shifting complexities to the network

Fallacies of distributed computing by Peter Deutsch

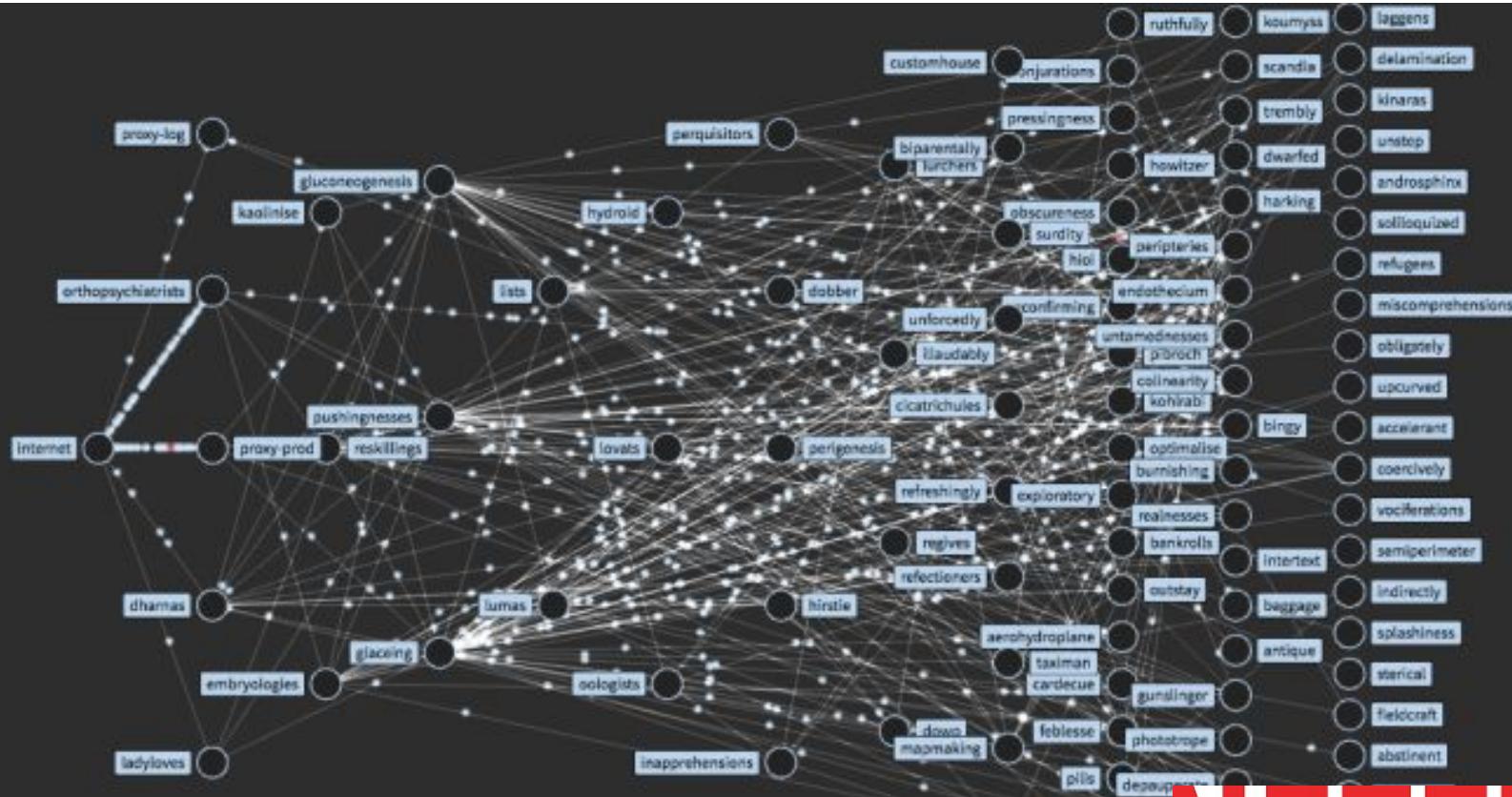
1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.







NETFLIX

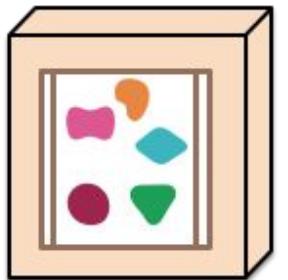
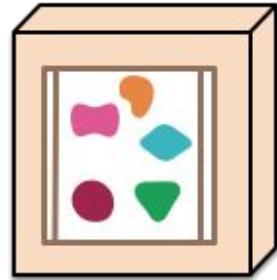


NETFLIX

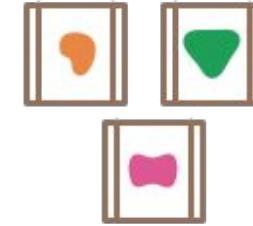
A monolithic application puts all its functionality into a single process...



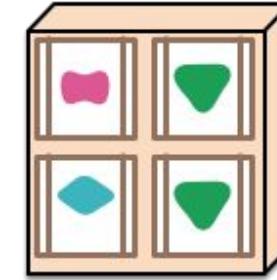
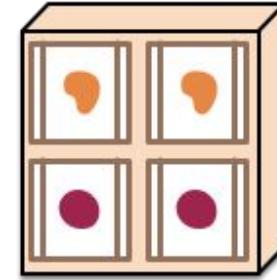
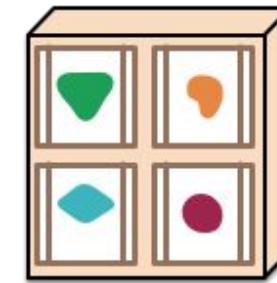
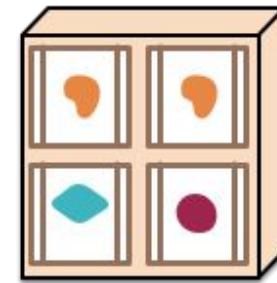
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...



... and scales by distributing these services across servers, replicating as needed.



Microservices

Building applications as suite of small and easy to replace services

- fine grained, one functionality per service
- (sometimes 3-5 classes)
- composable
- easy to develop, test, and understand
- fast (re)start, fault isolation

Modelled around business domain

Interplay of different systems and languages, no commitment to technology stack

Easily deployable and replicable

Embrace automation, embrace faults

Highly observable

Technical Considerations

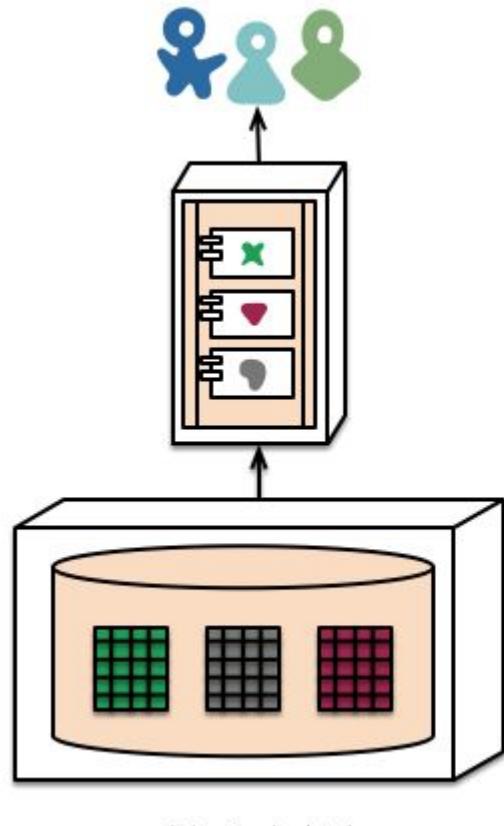
REST APIs, again

Independent development and deployment

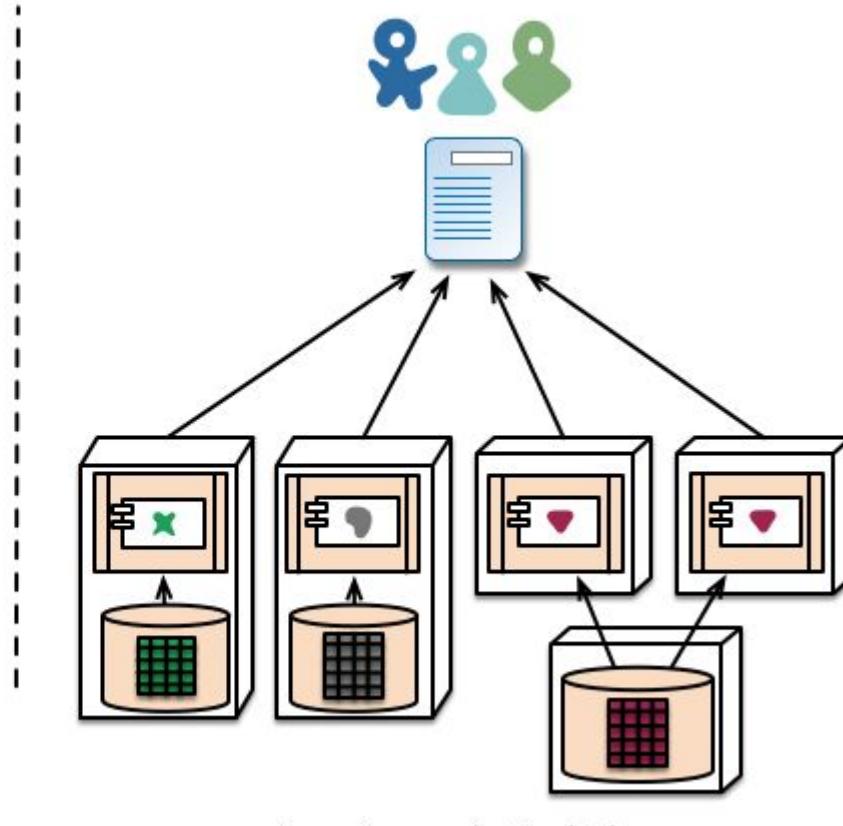
Self-contained services (e.g., each with own database)

- multiple instances behind load-balancer

Streamline deployment



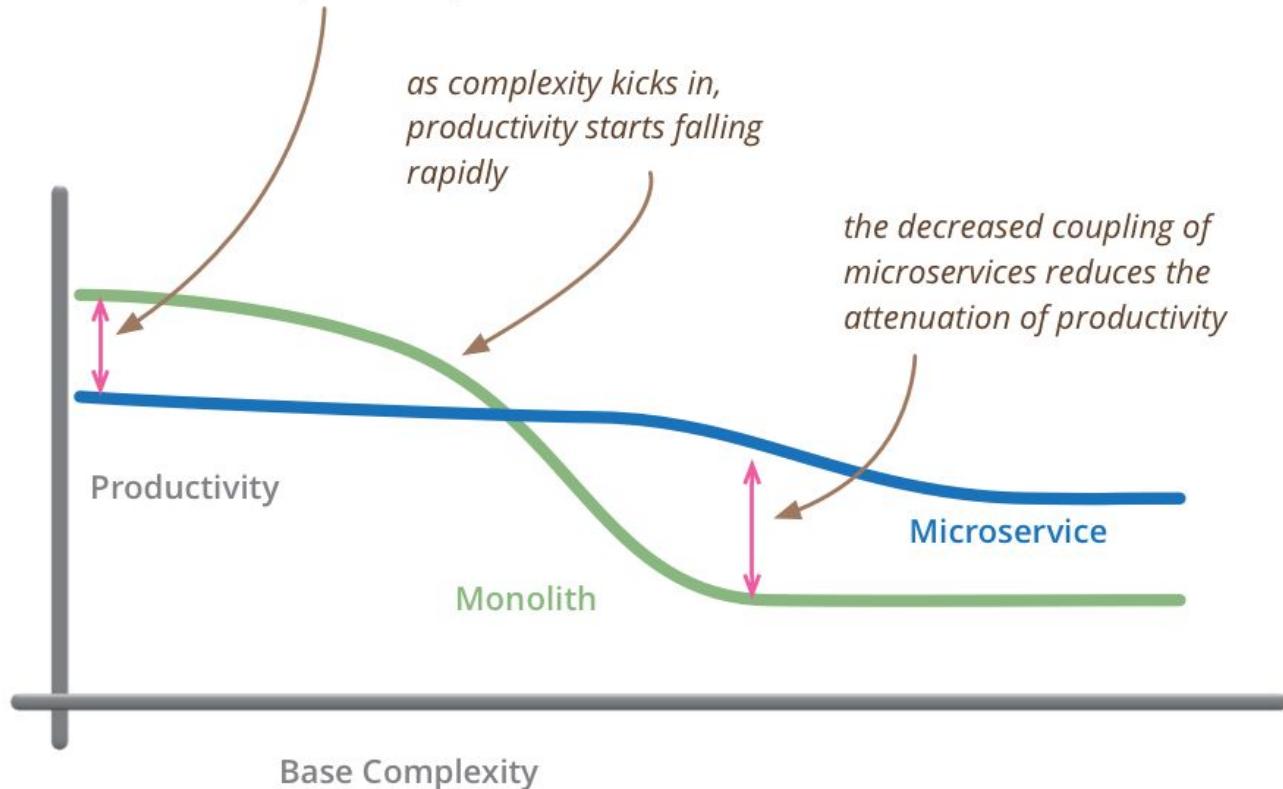
monolith - single database



microservices - application databases

Overhead

for less-complex systems, the extra baggage required to manage microservices reduces productivity



but remember the skill of the team will outweigh any monolith/microservice choice

Excursion: Testing in Distributed Systems

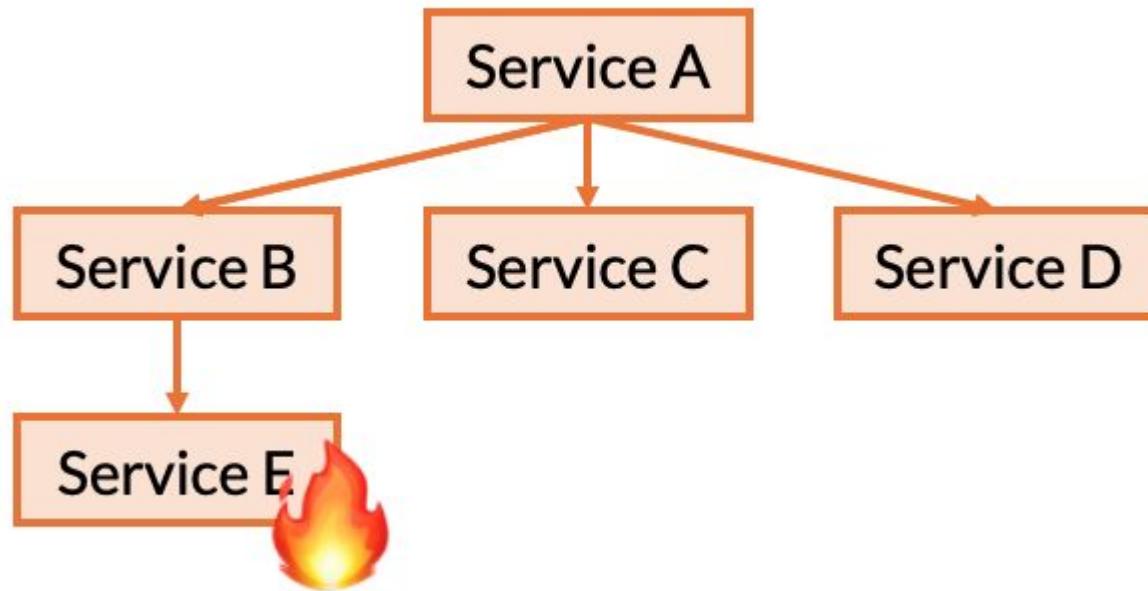
REST API Calls and Testing

Test happy path

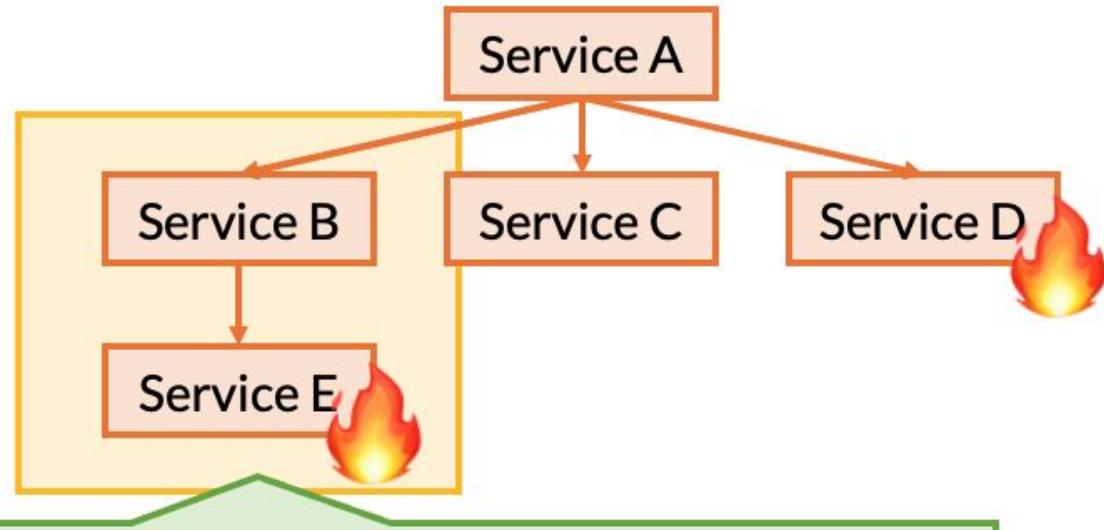
Test also error behavior!

- Correct timeout handling? Correct retry when connection down?
- Invalid response detected?
- Graceful degradation?

Need to understand possible error behavior first



Handle Errors Locally



Service encapsulation hides failure Service E behind Service B such that it is not observable by Service A. *(execution either the same as Service B, C success and D failure combo or Service C success and B and D failure combo, depending on B.)*

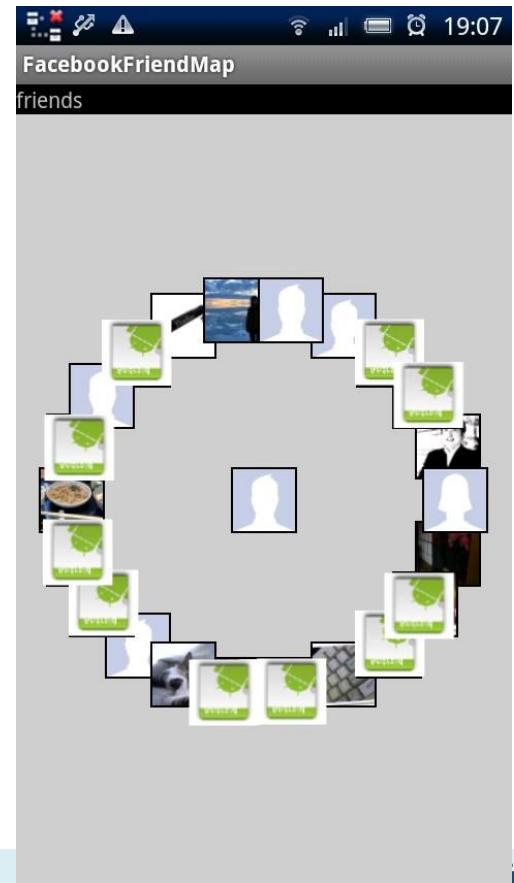
How to test?

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

Return of the Test Doubles!

Recall: Facebook Example

- 3rd party Facebook apps
- Android user interface
- Backend uses Facebook data



Testing in real environments



```
void buttonClicked() {
    render(getFriends());
}

List<Friend> getFriends() {
    Connection c = http.getConnection();
    FacebookAPI api = new FacebookAPI(c);
    List<Node> persons = api.getFriends("john");
    for (Node person1 : persons) {
        ...
    }
    return result;
}
```

Eliminating Android dependency



```
@Test void testGetFriends() {  
    assert getFriends() == ...;  
}  
  
List<Friend> getFriends() {  
    Connection c = http.getConnection();  
    FacebookAPI api = new FacebookAPI(c);  
    List<Node> persons = api.getFriends("john");  
    for (Node person1 : persons) {  
        ...  
    }  
    return result;
```

Eliminating the Remote Service Dependency



```
@Test void testGetFriends() {  
    assert getFriends() == ...;  
}  
  
List<Friend> getFriends() {  
    Connection c = http.getConnection();  
    FacebookAPI api = new FacebookAPI(c);  
    List<Node> persons = api.getFriends("john");  
    for (Node person1 : persons) {  
        ...  
    }  
    return result;  
}
```

The code shows a test method `testGetFriends` and its implementation. The implementation uses a `Connection` object `c` to get a `FacebookAPI` instance via `new FacebookAPI(c)`. This line is highlighted with a red rounded rectangle. An arrow points from a callout box containing the text "Replace by Double" to this highlighted line.

Replace by Double

Introducing a Double (Stub)



```
@Test void testGetFriends() {  
    assert getFriends() == ...;  
}  
List<Friend> getFriends() {  
    Connection c = http.getConnection();  
    FacebookInterface api = new FacebookStub(c);  
    List<Node> person  
    for (Node person1  
        for (Node per  
            ...  
    }  
}
```

```
class FacebookStub implements FacebookInterface {  
    void connect() {}  
    List<Node> getFriends(String name) {  
        if ("john".equals(name)) {  
            List<Node> result=new List();  
            result.add(...);  
        }  
    }  
}
```

Fault injection



- Mocks can emulate failures such as timeouts
- Allows you to verify the robustness of system

```
class FacebookSlowStub implements FacebookInterface {  
    void connect() {}  
    int counter = 0;  
    List<Node> getFriends(String name) {  
        Thread.sleep(4000);  
        if ("john".equals(name)) {  
            List<Node> result=new List();  
            result.add(...);  
        } else {  
            List<Node> result=new List();  
            result.add(...);  
        }  
        return result;  
    }  
}
```

Fault injection

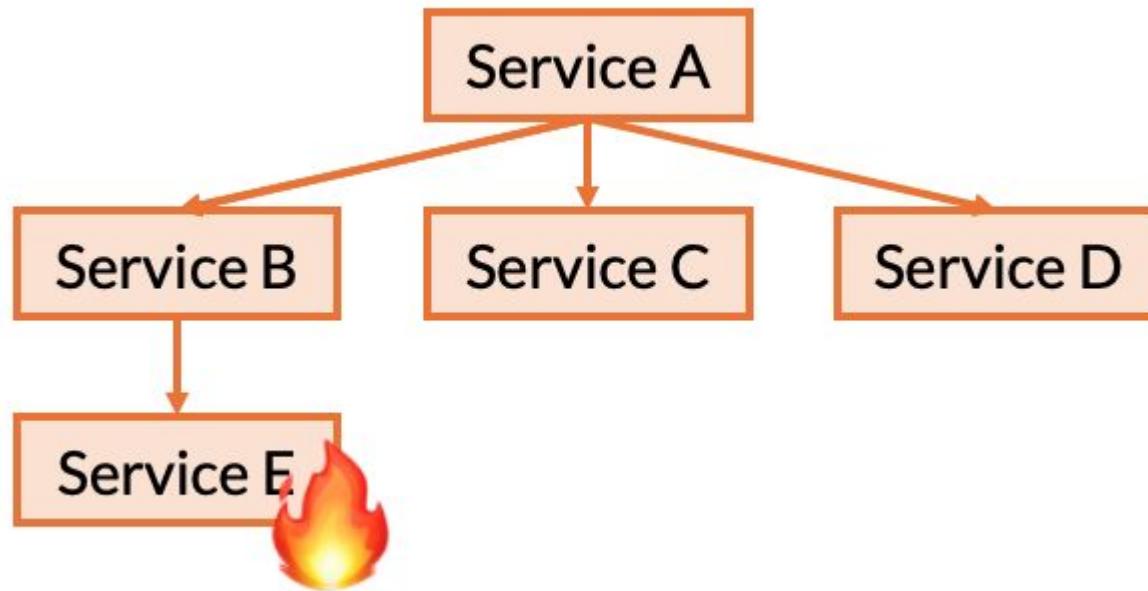


```
class FacebookErrorStub implements FacebookInterface {  
    void connect() {}  
    int counter = 0;  
    List<Node> getFriends(String name) {  
        counter++;  
        if (counter % 3 == 0)  
            throw new SocketException("Network is unreachable");  
        if ("john".equals(name)) {  
            List<Node> result=new List();  
            result.add(...);  
            return result;
```

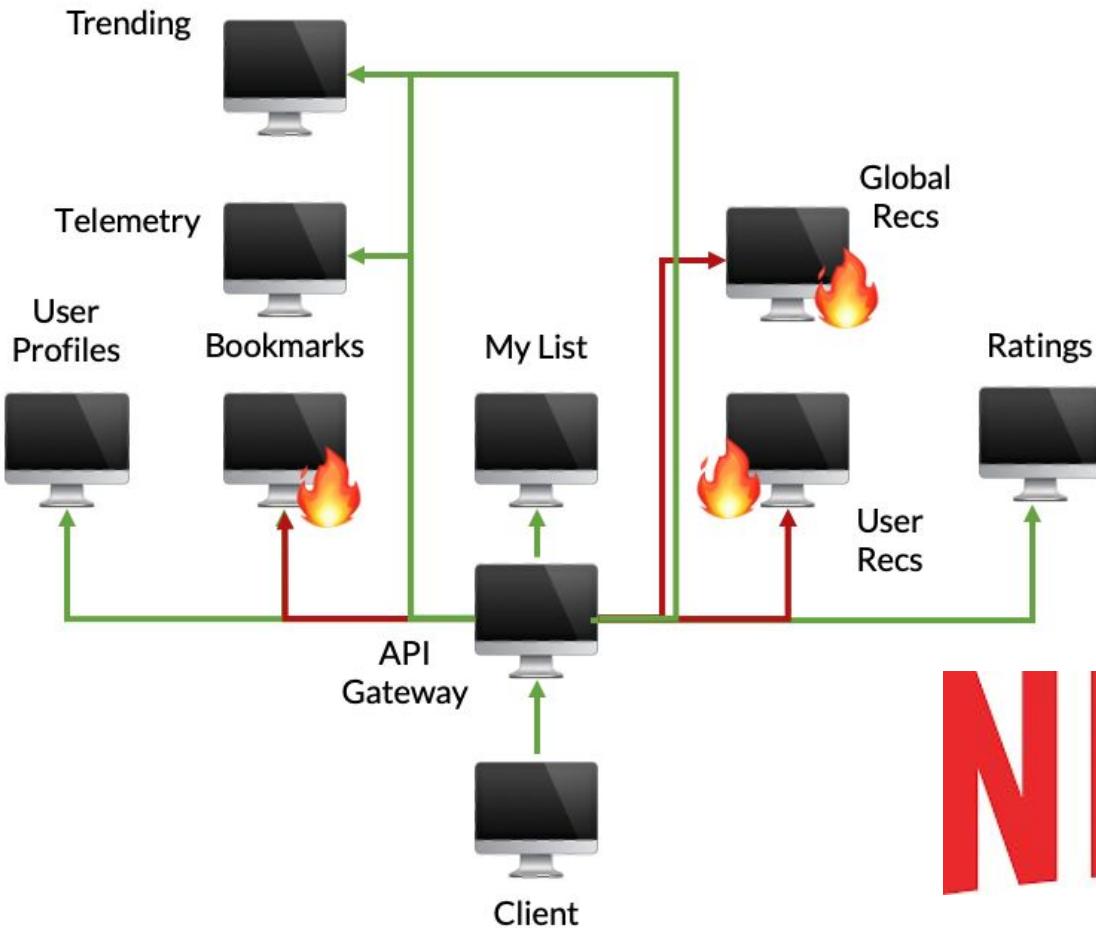
Chaos Engineering

Experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production





Distributed Event-Based System



NETFLIX

Options

Manage access

Repository roles

Security & analysis

Branches

Webhooks

Notifications

Integrations

Deploy keys

Autolink references

Actions

Environments

Secrets

Pages

Webhooks / Add webhook

We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, etc). More information can be found in [our developer documentation](#).

Payload URL *

https://example.com/postreceive

Content type

application/x-www-form-urlencoded

Secret

Which events would you like to trigger this webhook?

- Just the push event.
- Send me **everything**.
- Let me select individual events.

Active

We will deliver event details when this hook is triggered.

Add webhook

Push vs Pull: RPC vs Callbacks

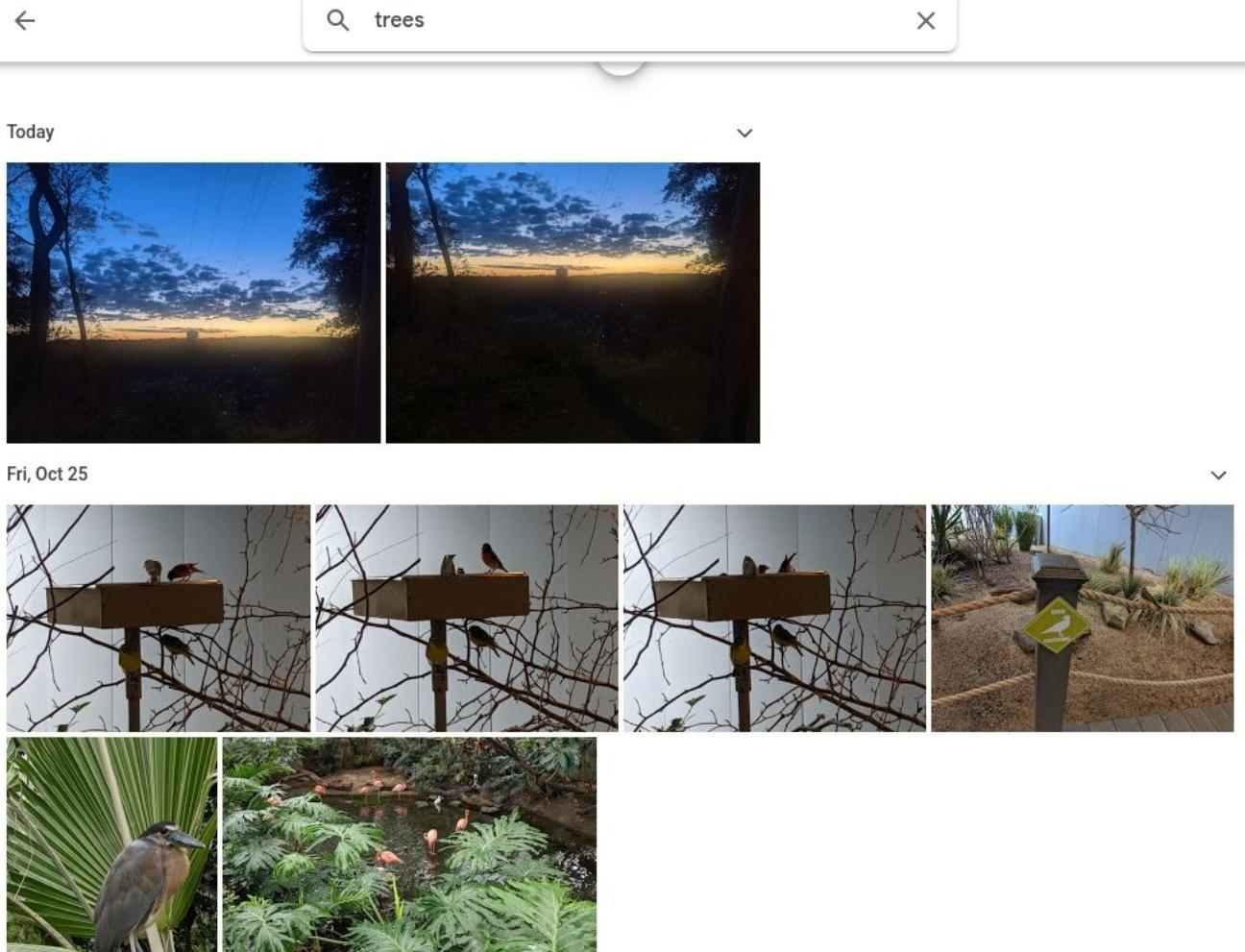
Both libraries and frameworks possible with RPC

- Netflix: Gateway calls and orchestrates services (pull; Strategy Pattern)
- GitHub WebHooks: GitHub pushes events to custom URL (Observer Pattern)

Reactive Programming and Event/Stream Processing

Stream processing: Distributed system design based on event queuing and processing

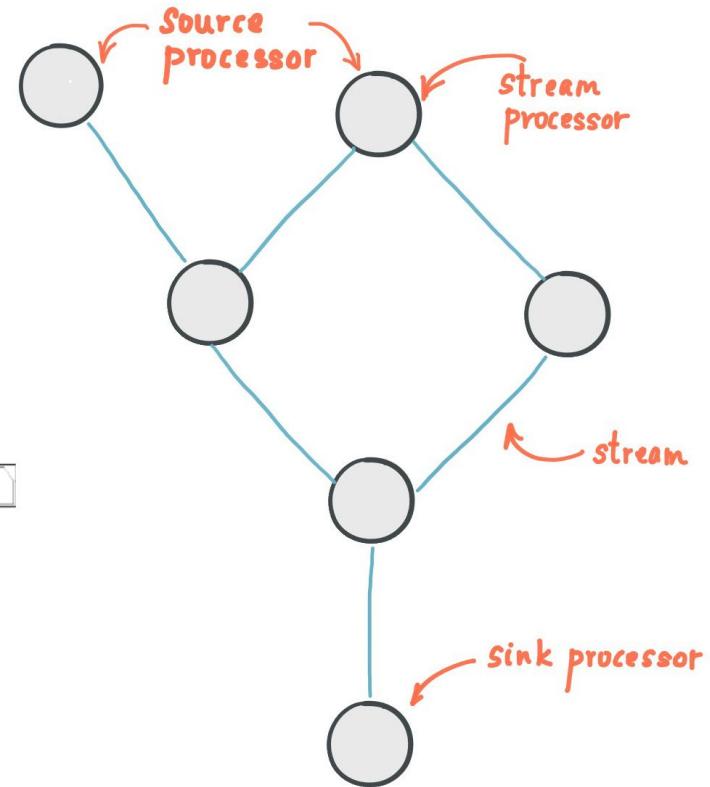
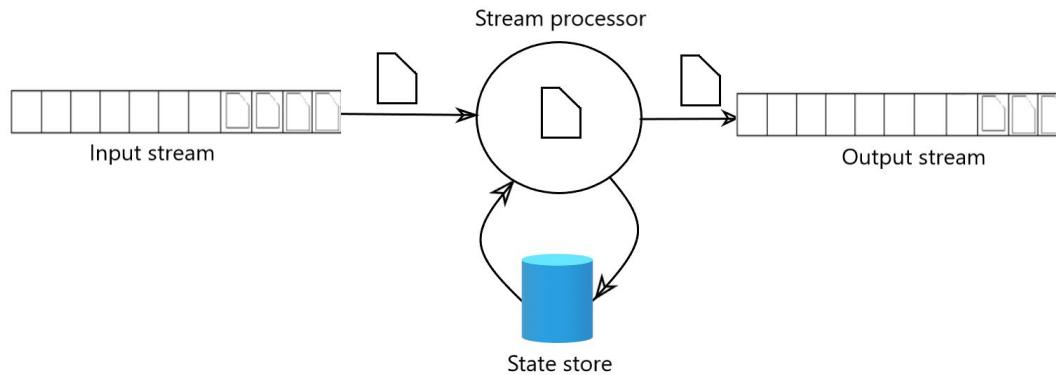
Example: Tagging of many images Indexing for search



Recall: RxJava

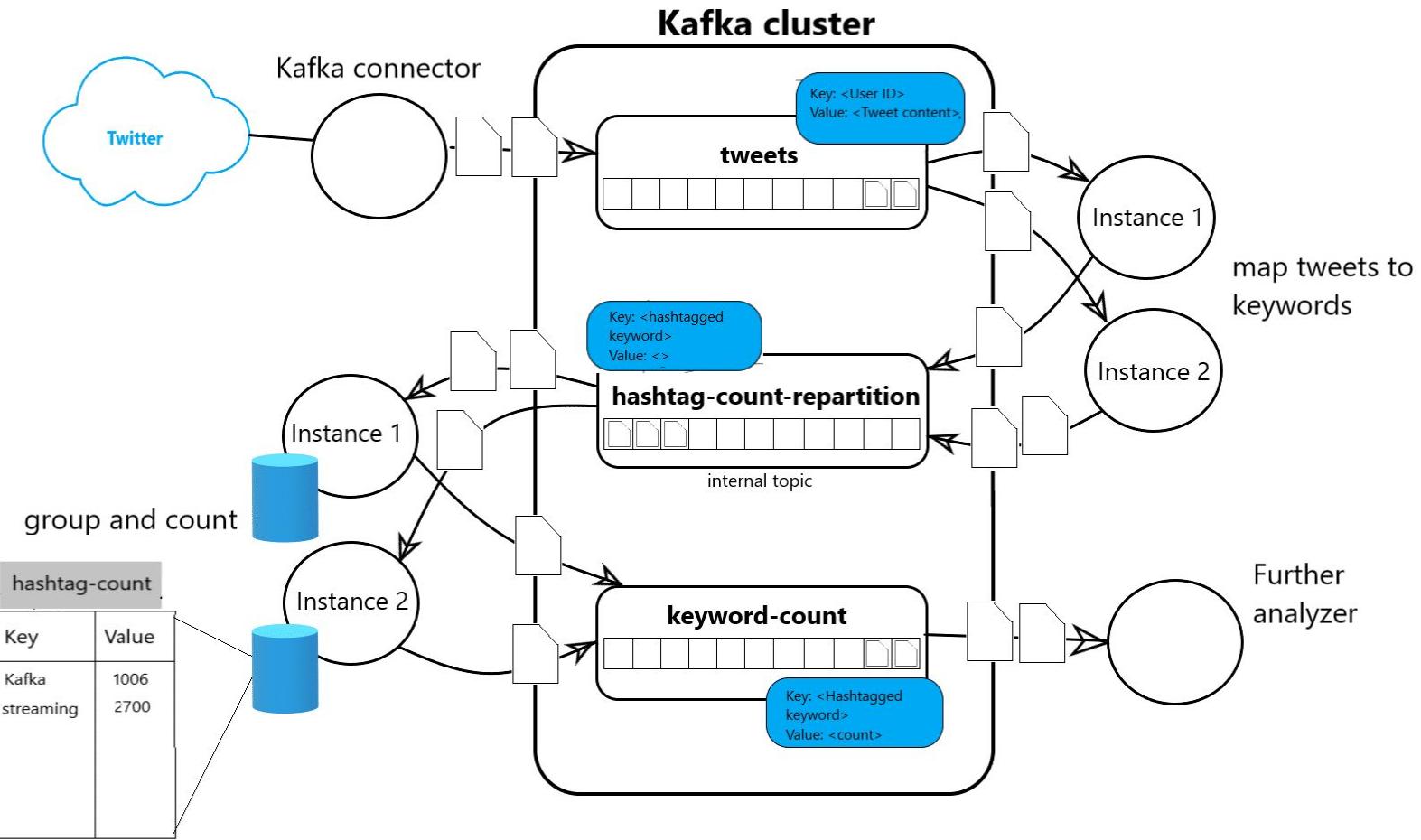
```
PublishSubject<Integer> x = PublishSubject.create();
PublishSubject<Integer> y = PublishSubject.create();
Observable<Integer> z = Observable.combineLatest(x, y,
(a,b)->a+b);
z.subscribe(System.out::println);
x.onNext(3);
y.onNext(5);
x.onNext(5);
```

Apache Kafka



PROCESSOR TOPOLOGY

<https://www.novatec-gmbh.de/en/blog/kafka-101-series-part-2-stream-processing-and-kafka-streams-api/>



```
final String topic = "topicName";
final Consumer<String, String> consumer = new KafkaConsumer<>();
consumer.subscribe(Arrays.asList(topic));

try {
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(100);
        for (ConsumerRecord<String, String> record : records) {
            String key = record.key();
            String value = record.value();
            // process data
        }
    }
} finally {
    consumer.close();
}
```

Kafka Consumer Code Example

Summary

Heavy reliance on dependencies

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

Modularly organize systems at scale

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

Testing with Stubs and Chaos Engineering

Principles of Software Construction: Objects, Design, and Concurrency

A Quick Tour of all 23 GoF Design Patterns

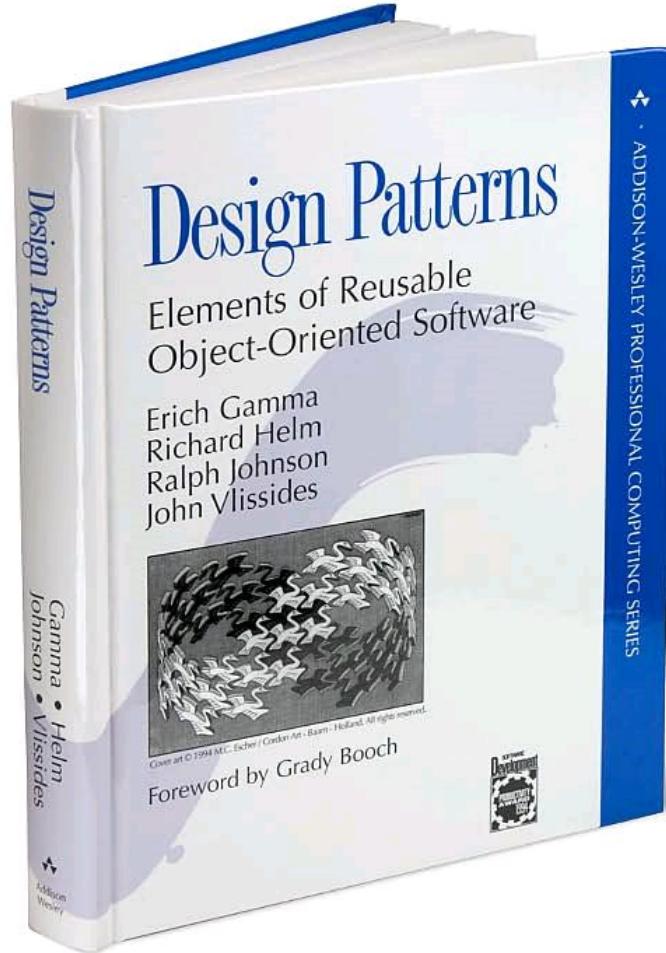
Christian Kästner Vincent Hellendoorn



Where we are

	<i>Small scale:</i> One/few objects	<i>Mid scale:</i> Many objects	<i>Large scale:</i> Subsystems
<i>Design for understanding change/ext. reuse robustness ...</i>	Subtype Polymorphism ✓ Information Hiding, Contracts ✓ Immutability ✓ Types Unit Testing ✓	Domain Analysis ✓ Inheritance & Del. ✓ 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

- Published 1994
- 23 Patterns
- Widely known



Warmup: Scenario

You are developing a mobile application for cities where users can report potholes and similar problems (with photos) and city crews can investigate, prioritize, and address reports.

Design problem 1: You want to create monthly reports. However different cities want this report slightly differently, with different text on top and sorted in different ways. You want to vary text and sorting in different ways.

Our course so far

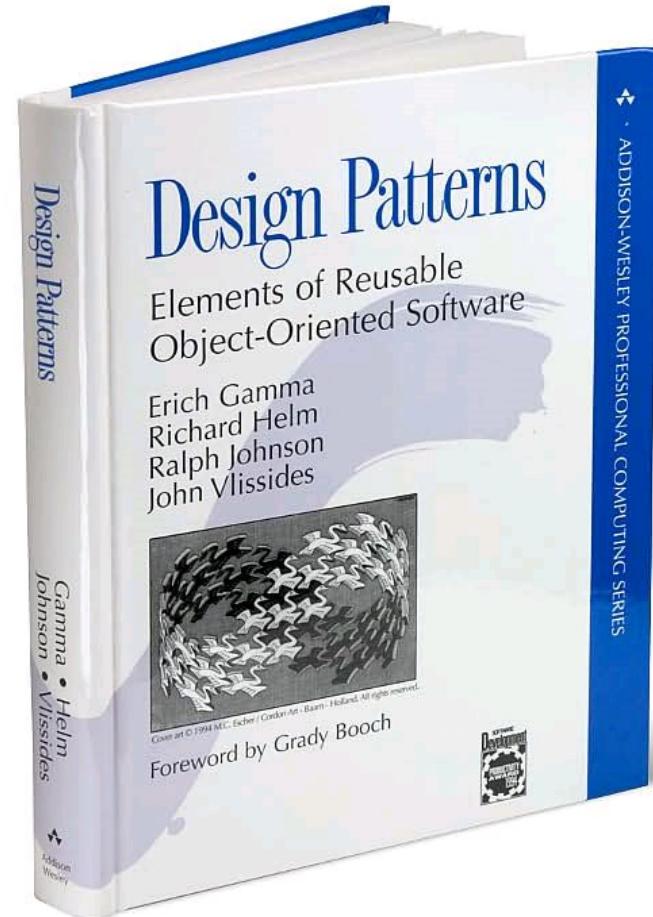
- Composite
 - Strategy
 - Template Method
 - Iterator
 - Proxy
 - Adapter
 - Decorator
 - Observer
 - Factory Method
- Not in the book:
- Model view controller
 - Promise
 - Generator
 - Module (JS)

Why?

- Seminal and canonical list of well-known patterns
- Not all patterns are commonly used
- Does not cover all popular patterns
- At least know where to look up when somebody mentions the “Bridge pattern”

Grouping Patterns

- I. Creational Patterns
- II. Structural Patterns
- III. Behavioral Patterns



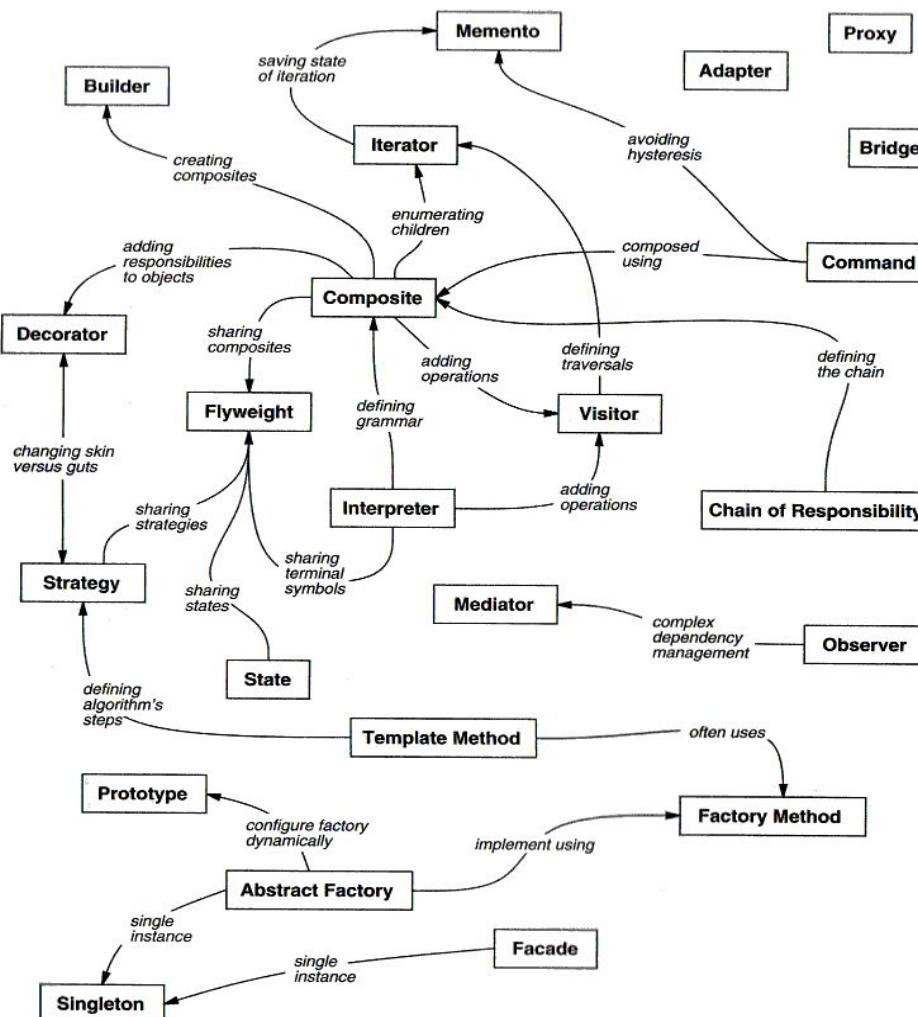


Figure 1.1: Design pattern relationships.

Pattern Name

- **Intent** – the aim of this pattern
- **Use case** – a motivating example
- **Key types** – the types that define pattern
 - Italic type name indicates abstract class; typically this is an interface when the pattern is used in Java
- **Examples**

Illustration

- **Code sample, diagram, or drawing**
 - Time constraints make it impossible to include illustrations from some patterns

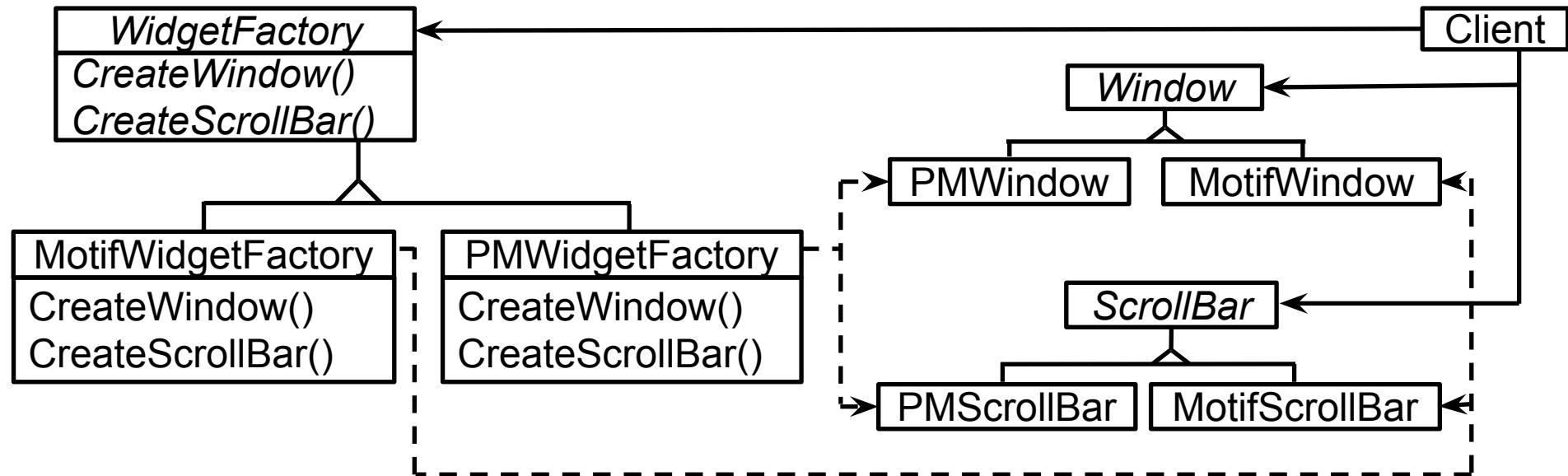
I. Creational Patterns

1. Abstract factory
2. Builder
3. Factory method
4. Prototype
5. Singleton

1. Abstract Factory

- Intent – allow creation of families of related objects independent of implementation
- Use case – look-and-feel in a GUI toolkit
 - Each L&F has its own windows, scrollbars, etc.
- Key types – *Factory* with methods to create each family member, *Products*
- Not common in JDK / JavaScript

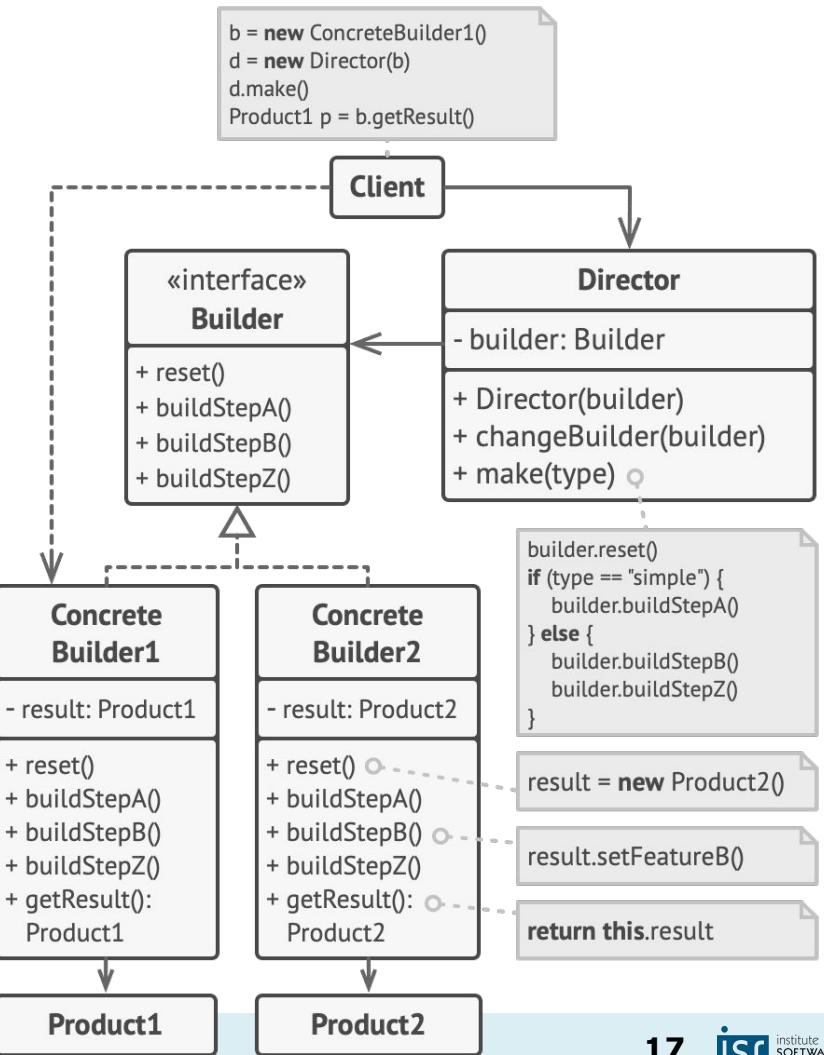
Abstract Factory Illustration



Builder Pattern

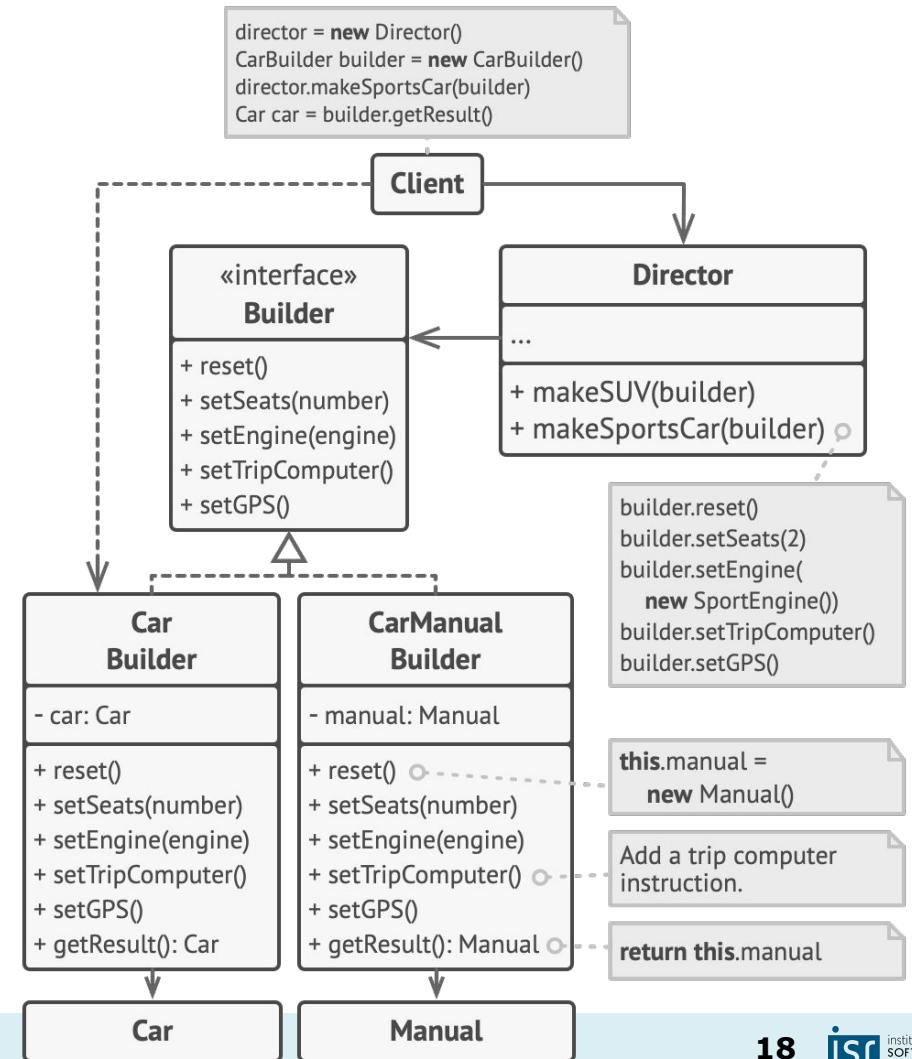
- Intent – separate construction of complex object from representation **so same creation process can create different representations**
- use case – converting rich text to various formats
- types – *Builder, ConcreteBuilders, Director, Products*
- `StringBuilder` (Java), `DirectoryBuilder` (HW2)

Gof4 Builder Illustration



<https://refactoring.guru/design-patterns/builder>

Builder Example



Builder Code Example

```
NutritionFacts twoLiterDietCoke = new NutritionFacts.Builder(  
    "Diet Coke", 240, 8).sodium(1).build();  
public class NutritionFacts {  
    public static class Builder {  
        public Builder(String name, int servingSize,  
                      int servingsPerContainer) { ... }  
        public Builder totalFat(int val) { totalFat = val; }  
        public Builder saturatedFat(int val) { satFat = val; }  
        public Builder transFat(int val) { transFat = val; }  
        public Builder cholesterol(int val) { cholesterol = val; }  
        ... // 15 more setters  
        public NutritionFacts build() {  
            return new NutritionFacts(this);  
        }  
    }  
    private NutritionFacts(Builder builder) { ... }  
}
```

Builder Discussion

- Emulates named parameters in languages that don't support them
- Emulates 2^n constructors or factories with n builder methods, by allowing them to be combined freely
- Cost is an intermediate (Builder) object

Recall: Factory Method Pattern

- Intent – abstract creational method that lets subclasses decide which class to instantiate
- Use case – creating documents in a framework
- Key types – *Creator*, which contains abstract method to create an instance
- Java: `Iterable.iterator()`
- Related *Static Factory pattern* is very common
 - Technically not a GoF pattern, but close enough, e.g. `Integer.valueOf(int)`

Factory Method Illustration

```
public interface Iterable<E> {  
    public abstract Iterator<E> iterator();  
}  
  
public class ArrayList<E> implements List<E> {  
    public Iterator<E> iterator() { ... }  
    ...  
}  
  
public class HashSet<E> implements Set<E> {  
    public Iterator<E> iterator() { ... }  
    ...  
}
```

Static Factory Method Example

```
public DatabaseConnection {  
    private DatabaseConnection(String address) { ... }  
    public static DatabaseConnection create  
        (String address) {  
        //optional caching or checking...  
        return new DatabaseConnection(address);  
    }  
}  
  
c = new DatabaseConnection("localhost");  
c = DatabaseConnection.create("localhost");
```

Prototype Pattern

- Intent – create an object by cloning another and tweaking as necessary
- Use case – writing a music score editor in a graphical editor framework
- Key types – *Prototype*
- Java: `Cloneable`, but avoid (except on arrays)
- JavaScript: Builtin language feature

Singleton Pattern

- Intent – ensuring a class has only one instance
- Use case – GoF say **print queue, file system, company in an accounting system**
 - Compelling uses are rare but they do exist
- Key types – Singleton
- Java: `java.lang.Runtime.getRuntime()`,
`java.util.Collections.emptyList()`

Singleton Illustration

```
public class Elvis {  
    private static final Elvis ELVIS = new Elvis();  
    public static Elvis getInstance() { return ELVIS; }  
    private Elvis() { }  
    ...  
}
```

```
const elvis = { ... }  
function getElvis() {  
  
export { getElvis }
```

Singleton Discussion

Singleton = global variable

No flexibility for change or extension

Tends to be overused

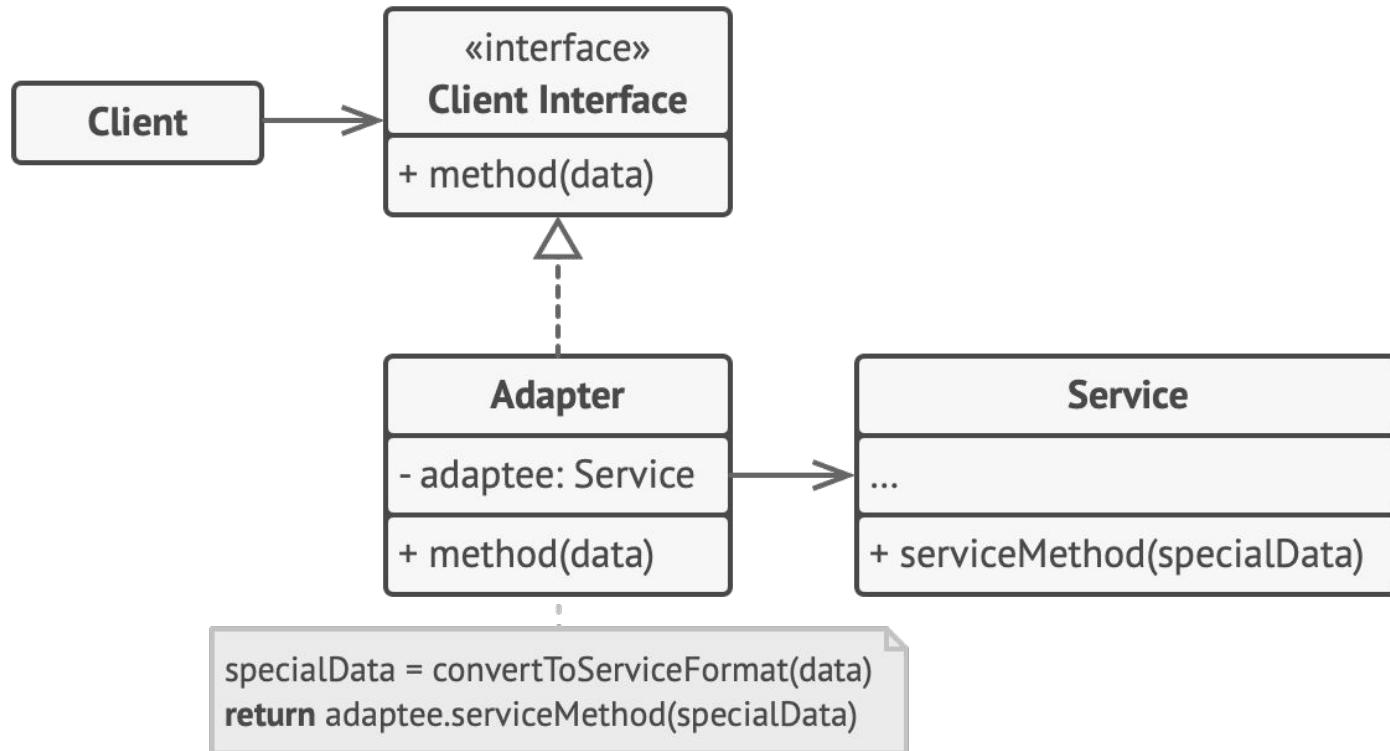
II. Structural Patterns

1. Adapter
2. Bridge
3. Composite
4. Decorator
5. Façade
6. Flyweight
7. Proxy

Recall: Adapter Pattern

- Intent – convert interface of a class into one that another class requires, allowing interoperability
- Use case – numerous, e.g., arrays vs. collections
- Key types – Target, Adaptee, Adapter
- JDK – `Arrays.asList(T[])`

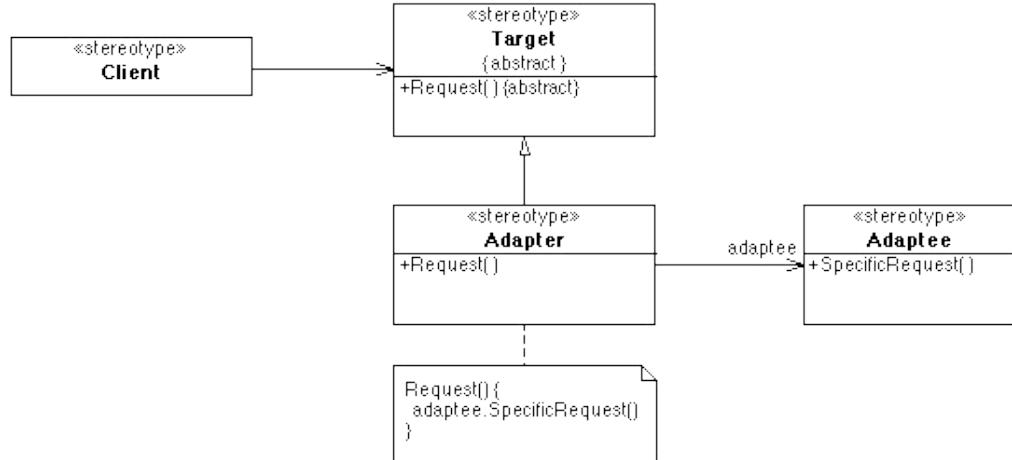
Recall: The *Adapter* Design Pattern



Recall: The *Adapter* Design Pattern

Applicability

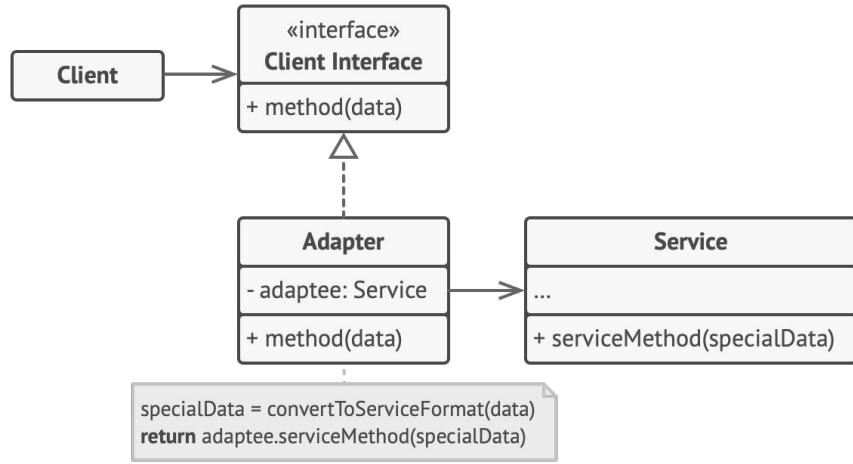
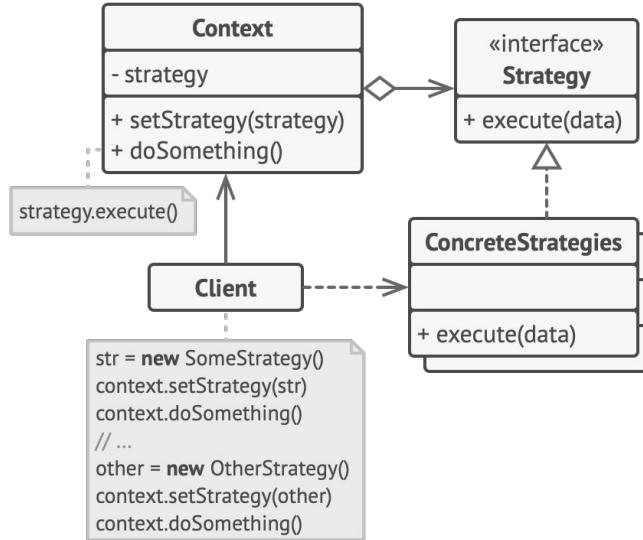
- You want to use an existing class, and its interface does not match the one you need
- You want to create a reusable class that cooperates with unrelated classes that don't necessarily have compatible interfaces
- You need to use several subclasses, but it's impractical to adapt their interface by subclassing each one



Consequences

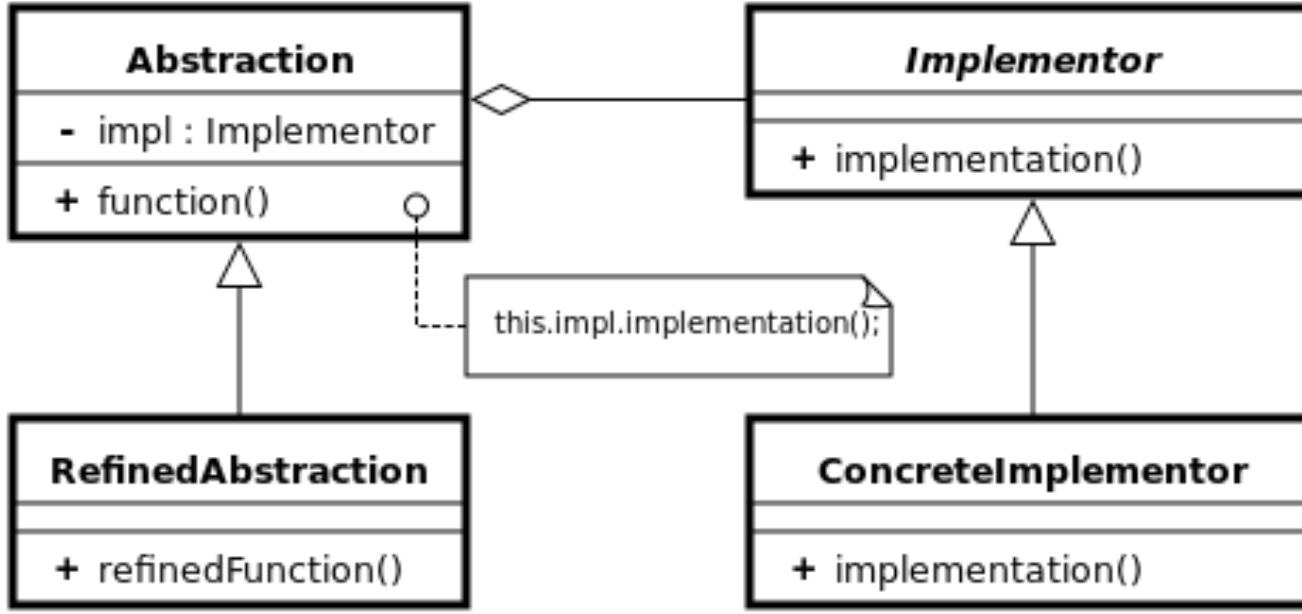
- Exposes the functionality of an object in another form
- Unifies the interfaces of multiple incompatible adaptee objects
- Lets a single adapter work with multiple adaptees in a hierarchy
- -> **Low coupling, high cohesion**

Adapter vs Strategy?



2. Bridge

- Intent – decouple an abstraction from its implementation so they can vary independently
- Use case – portable windowing toolkit
- Key types – Abstraction, *Implementor*
- Java: JDBC, Java Cryptography Extension (JCE), Java Naming & Directory Interface (JNDI)



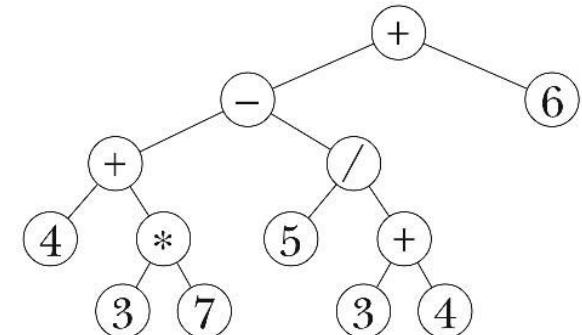
Bridge is very similar to Adapter: In Bridge you define both the abstract interface and the underlying implementation; you do not adapt to some legacy or third-party code. The goal is to swap out implementations, not to ensure compatibility after the fact. In addition, abstraction and implementation can vary independently.

Recall: Composite Pattern

- Intent – compose objects into tree structures. **Let clients treat primitives & compositions uniformly.**
- Use case – GUI toolkit (widgets and containers)
- Key type – *Component* that represents both primitives and their containers
- Java: `javax.swing.JComponent`

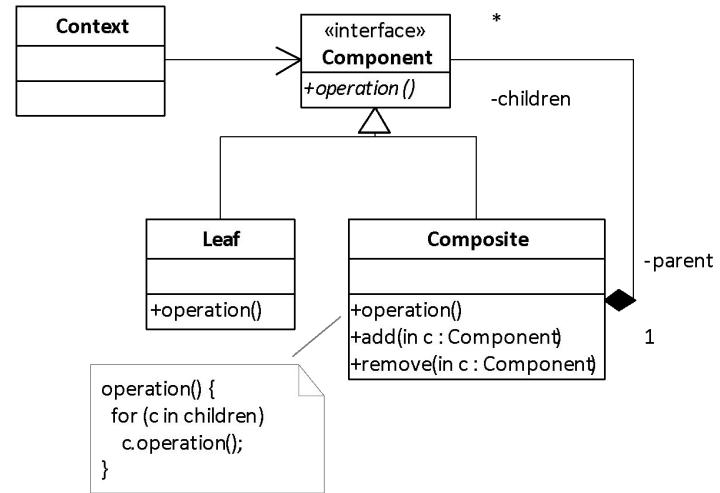
Composite Illustration

```
public interface Expression {  
    double eval();      // Returns value  
    String toString(); // Returns infix expression string  
}  
  
public class UnaryOperationExpression implements Expression {  
    public UnaryOperationExpression(  
        UnaryOperator operator, Expression operand);  
}  
  
public class BinaryOperationExpression implements Expression {  
    public BinaryOperationExpression(BinaryOperator operator,  
        Expression operand1, Expression operand2);  
}  
  
public class NumberExpression implements Expression {  
    public NumberExpression(double number);  
}
```



The Composite Design Pattern

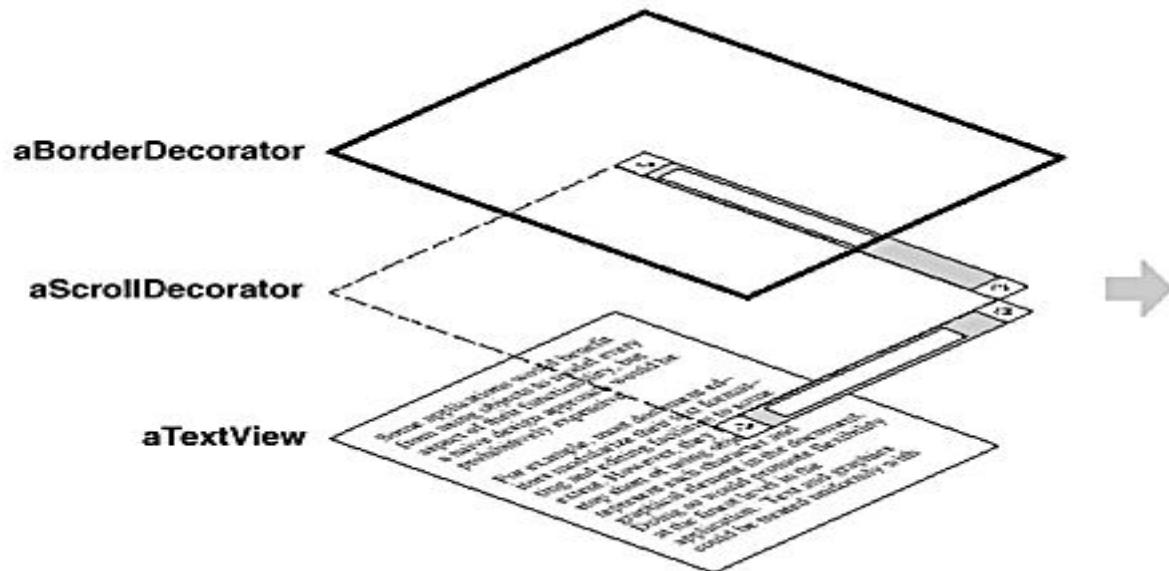
- Applicability
 - You want to represent part-whole hierarchies of objects
 - You want to be able to ignore the difference between compositions of objects and individual objects
- Consequences
 - Makes the client simple, since it can treat objects and composites uniformly
 - Makes it easy to add new kinds of components
 - Can make the design overly general
 - Operations may not make sense on every class
 - Composites may contain only certain components



Recall: Decorator Pattern

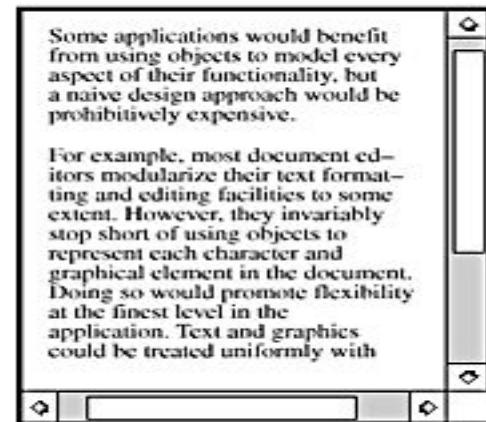
- Intent – attach features to an object dynamically
- Use case – attaching borders in a GUI toolkit
- Key types – *Component*, implement by decorator and decorated
- Java: Collections (e.g., Synchronized wrappers),
`java.io` streams, Swing components

Decorator Illustration



Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.

For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with



Decorator

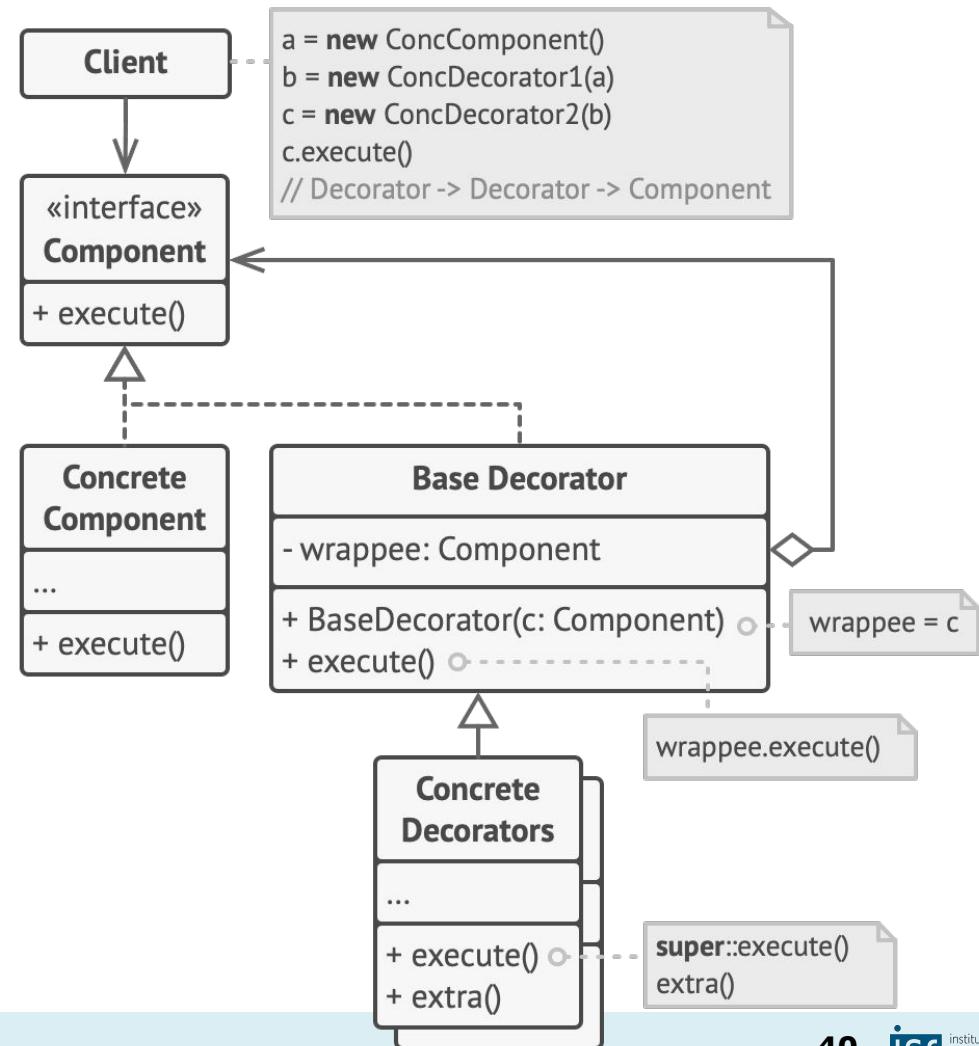
One or multiple base classes
(same interface)

Any number of decorators

Adding decorators at runtime

Base decorator provides
default forwarding logic

No open recursion



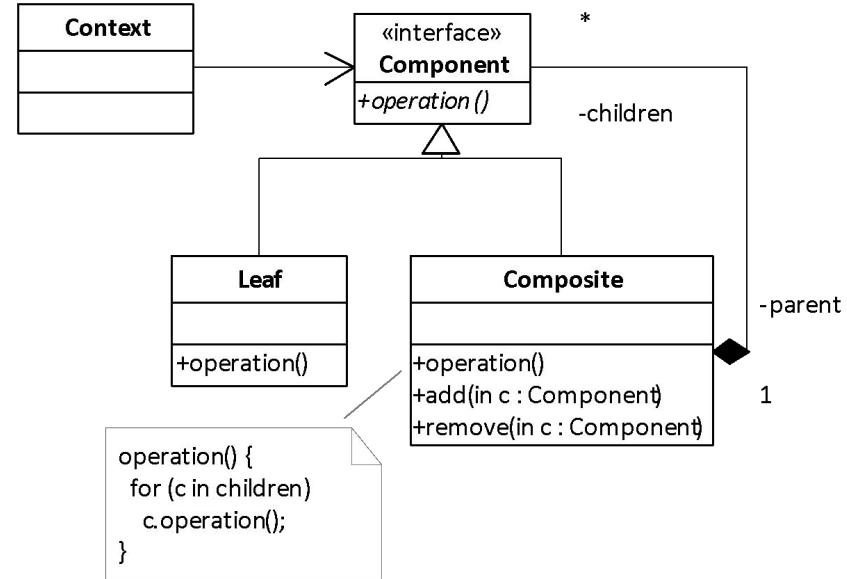
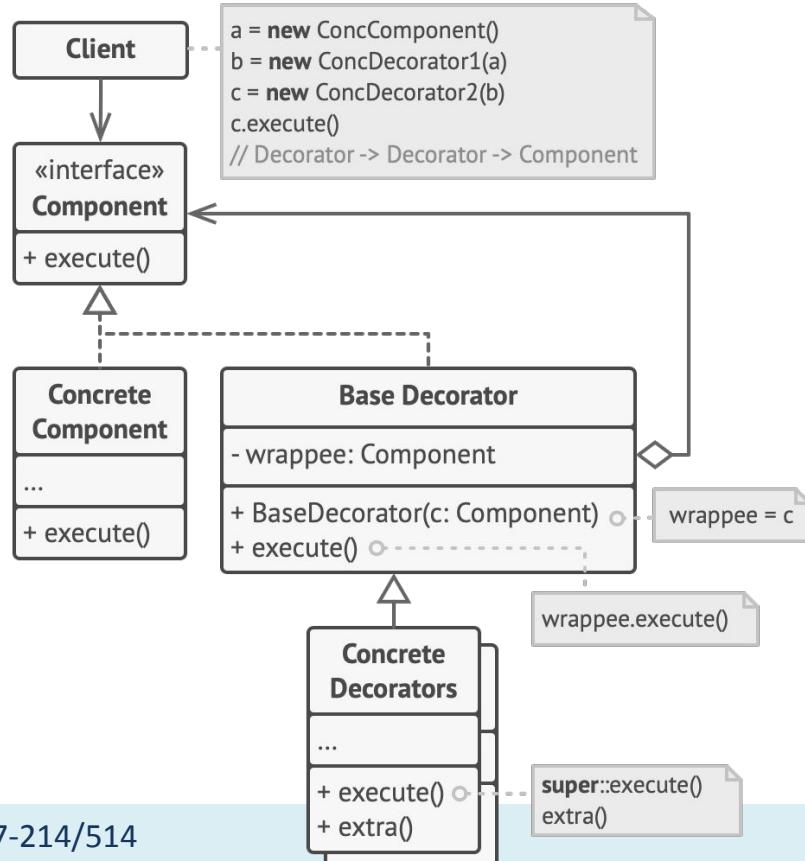
```
interface GameLogic {
    isValidMove(w, x, y)
    move(w, x, y)
}

class BasicGameLogic implements GameLogic {
    constructor(board) { ... }
    isValidMove(w, x, y) { ... }
    move(w, x, y) { ... }
}

class AbstractGodCardDecorator implements GameLogic {
    readonly gl: GameLogic
    constructor(gameLogic) { this.gl = gameLogic }
    isValidMove(w, x, y} { return this.gl.isValidMove(w, x, y) }
    move(w, x, y} { return this.gl.move(w, x, y) }
}

class PanDecorator extends AbstractGodCardDecorator implements GameLogic {
    move(w, x, y} { /* this.gl.move(w, x, y) + checkWinner */ }
}
```

Decorator vs Composite?



Decorator vs Strategy?

```
interface GameLogic {  
    isValidMove(w, x, y)  
    move(w, x, y)  
}  
  
class BasicGameLogic  
    implements GameLogic { ... }  
  
class AbstractGodCardDecorator  
    implements GameLogic { ... }  
  
class PanDecorator  
    extends AbstractGodCardDecorator  
    implements GameLogic { ... }
```

```
interface GameLogic {  
    isValidMove(w, x, y)  
    move(w, x, y)  
}  
  
class BasicGameLogic  
    implements GameLogic {  
    constructor(board) { ... }  
    isValidMove(w, x, y) { ... }  
    move(w, x, y) { ... }  
}  
  
class PanDecorator  
    extends BasicGameLogic {  
    move(w, x, y} { /* super.move(w,  
x, y) + checkWinner */ }  
}
```

Design Problem

You are developing a mobile application for cities where users can report potholes and similar problems (with photos) and city crews can investigate, prioritize, and address reports.

Design problem 2: City workers after inspecting a problem can mark the problem as high priority, as delegated, or in several other ways. Markers change how issues are shown (e.g., in reports).

Design Problem

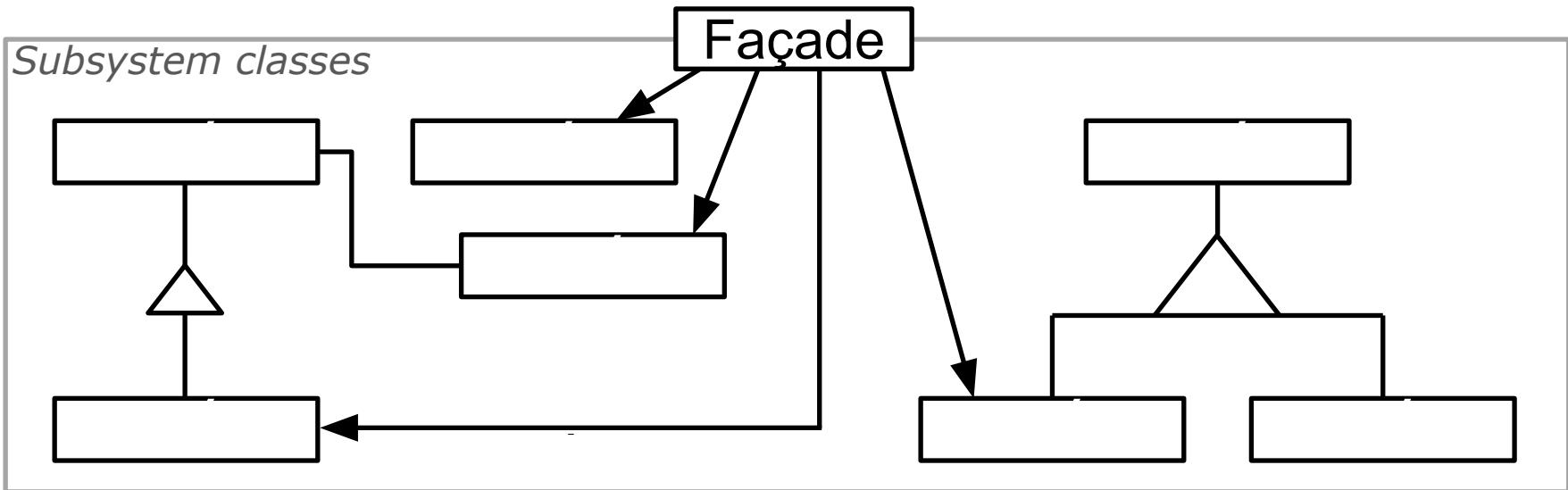
You are developing a mobile application for cities where users can report potholes and similar problems (with photos) and city crews can investigate, prioritize, and address reports.

Design problem 3: You want to group problems that are related into a problem group with a new name, and those might be grouped again, but still count them directly. Those groups should still show up in reports and all scheduling activities.

Façade Pattern

- Intent – provide a simple unified interface to a set of interfaces in a subsystem
 - GoF allow for variants where the complex underpinnings are exposed and hidden
- Use case – any complex system; GoF use compiler
- Key types – Façade (the simple unified interface)
- JDK – `java.util.concurrent.Executors`

Façade Illustration



```
class SantoriniController {  
    newGame() { ... }  
    isValidMove(w, x, y) { ... }  
    move(w, x, y) { ... }  
    getWinner() { ... }  
}
```

Discussion

Facade vs Controller Heuristic

Same idea

Facade for subsystem, controller for use case

Facade vs Singleton

Facade sometimes a global variable

Typically little design for change/extension

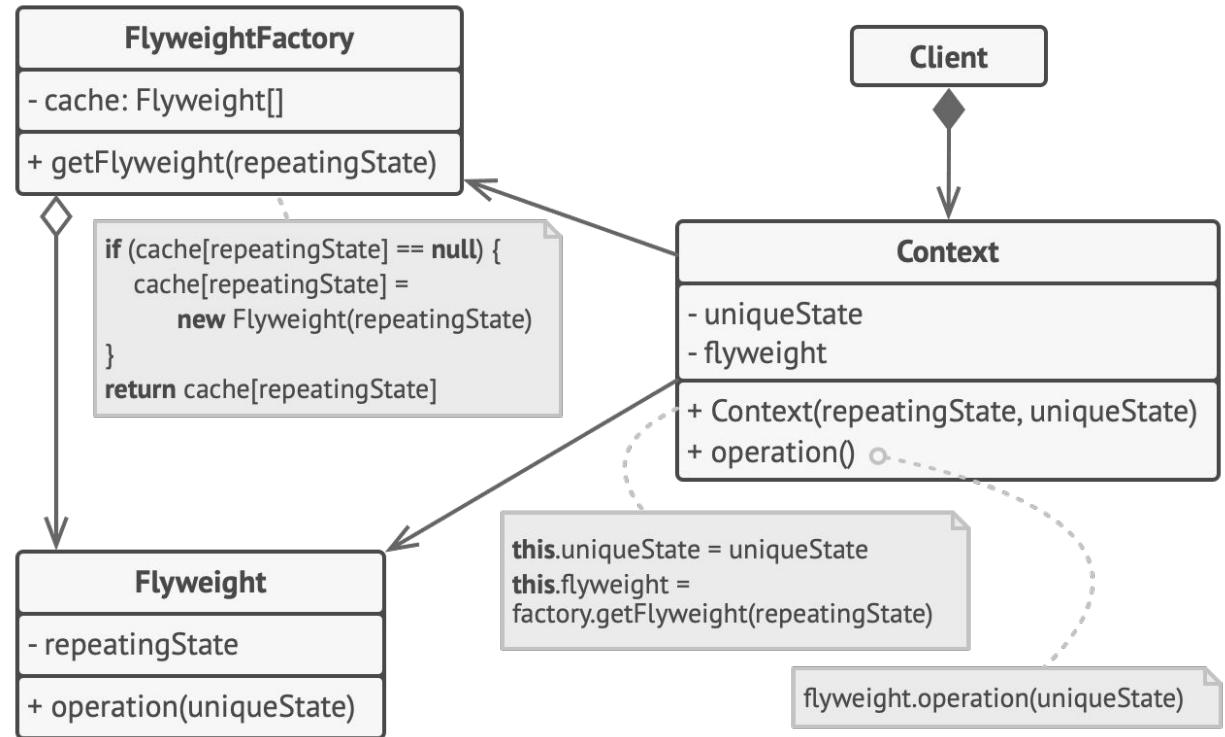
Flyweight Pattern

- Intent – use sharing to support large numbers of fine-grained objects efficiently
- Use case – characters in a document
- Key types – Flyweight (instance-controlled!)
 - Some state can be *extrinsic* to reduce number of instances
- Java: String literals (JVM feature), Integer
- “Hash Consing” in functional programming

Flyweight

Key idea: Avoid copies of structurally equal objects, reuse object

Requires immutable objects and factory with caching

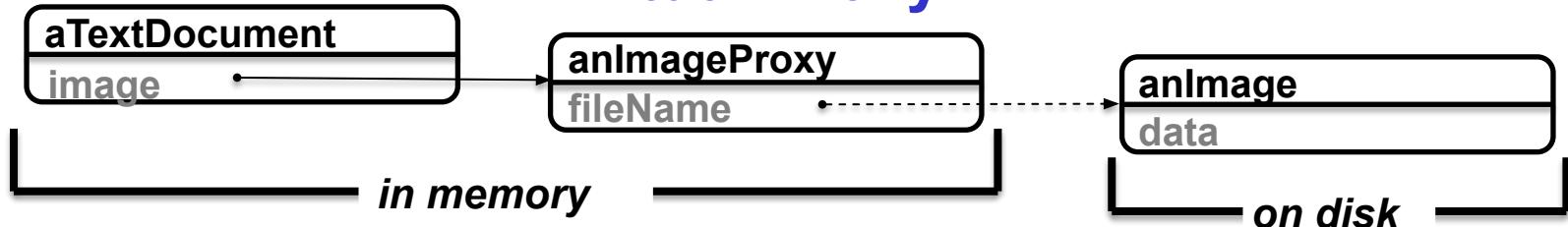


Proxy Pattern

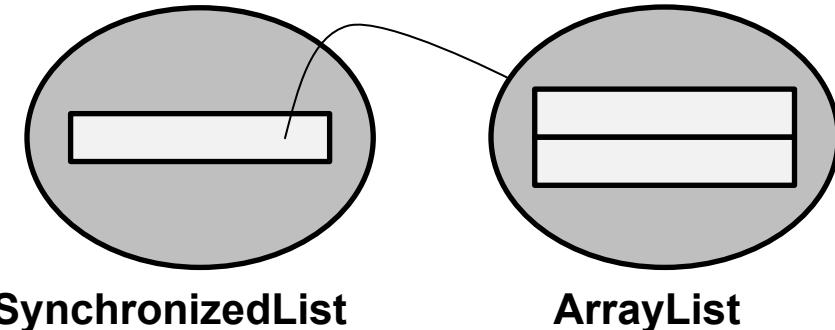
- Intent – surrogate for another object
- Use case – delay loading of images till needed
- Key types – *Subject*, Proxy, RealSubject
- Gof mention several flavors
 - virtual proxy – stand-in that instantiates lazily
 - remote proxy – local representative for remote obj
 - protection proxy – denies some ops to some users
 - smart reference – does locking or ref. counting, e.g.
- JDK – RMI, collections wrappers

Proxy Illustrations

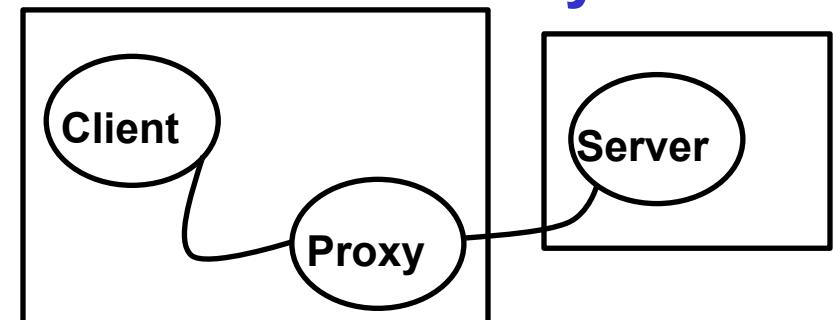
Virtual Proxy



Smart Reference

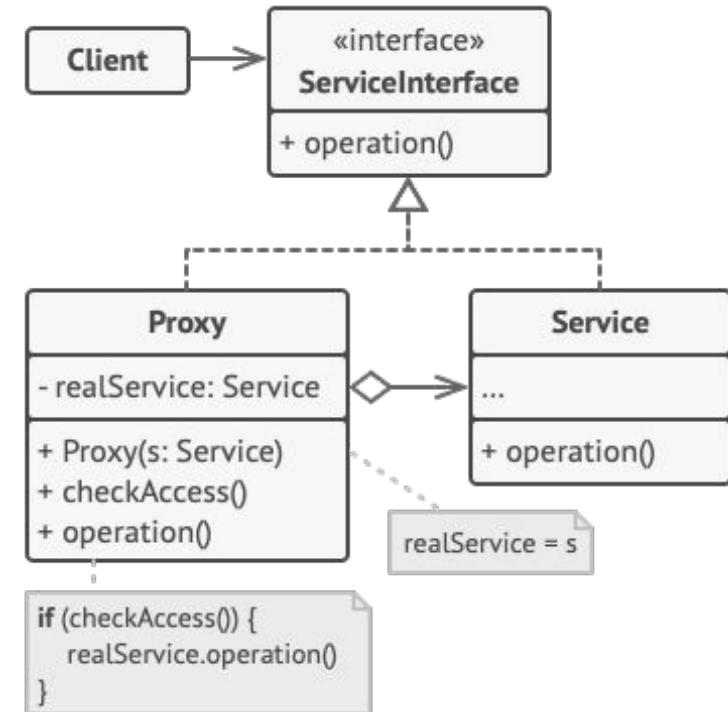


Remote Proxy

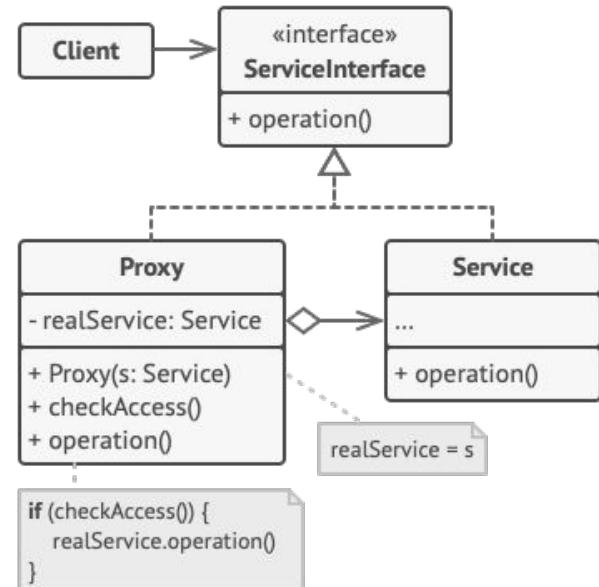
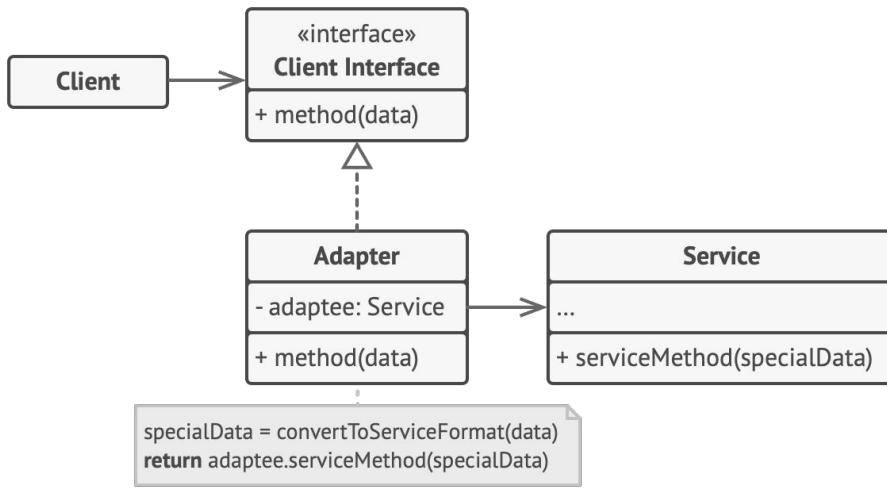


Proxy Design Pattern

- Local representative for remote object
 - Create expensive obj on-demand
 - Control access to an object
- Hides extra “work” from client
 - Add extra error handling, caching
 - Uses *indirection*



Proxy vs Adapter?



Design Problem

You are developing a mobile application for cities where users can report potholes and similar problems (with photos) and city crews can investigate, prioritize, and address reports.

Design problem 4: Some problems point to large pictures stored in another database and you do not want to keep them in memory, but load them only when needed.

Design Problem

You are developing a mobile application for cities where users can report potholes and similar problems (with photos) and city crews can investigate, prioritize, and address reports.

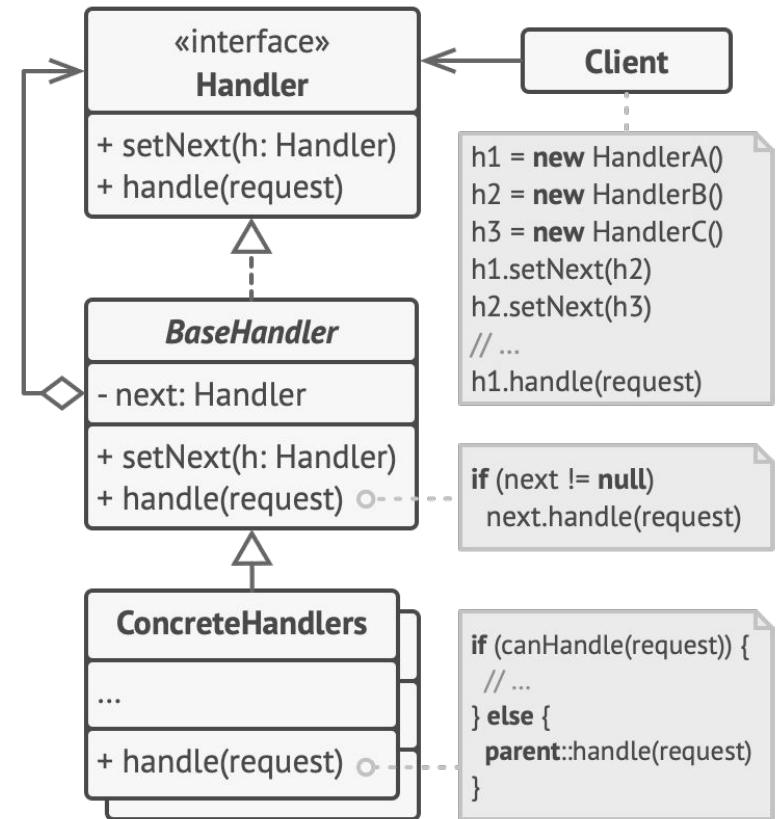
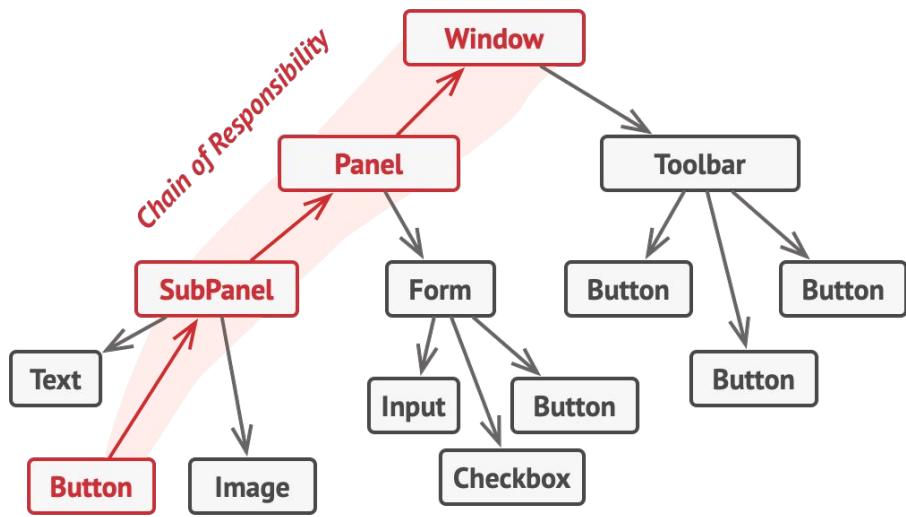
Design problem 5: The county has a different system that records potholes in a different format. You want to include them in your reports regardless.

III. Behavioral Patterns

1. Chain of Responsibility
2. Command
3. Interpreter
4. Iterator
5. Mediator
6. Memento
7. Observer
8. State
9. Strategy
10. Template method
11. Visitor

Chain of Responsibility Pattern

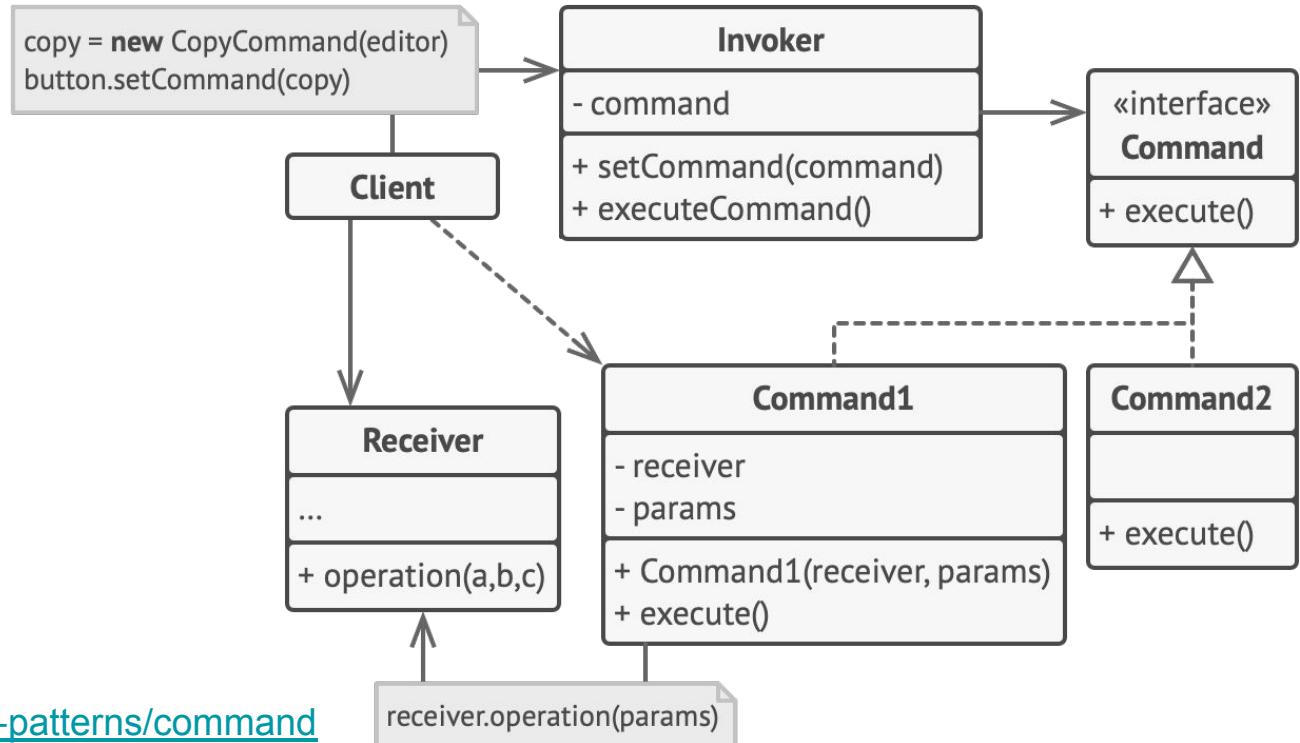
- Intent – avoid coupling sender to receiver by passing request along until someone handles it
- Use case – context-sensitive help facility
- Key types – *RequestHandler*
- JDK – ClassLoader, Properties
- Exception handling could be considered a form of Chain of Responsibility pattern



Command Pattern

- Intent – encapsulate a request as an object, letting you parameterize one action with another, queue or log requests, etc.
- Use case – menu tree
- Key type – *Command* (Runnable)
- JDK – Common! Executor framework, etc. -- see higher order function
- Is it Command pattern if you run it repeatedly? If it takes an argument?
Returns a val?

Command Pattern



<https://refactoring.guru/design-patterns/command>

Command Illustration

```
class ClickAction {  
    constructor(name) { this.name = name }  
    execute() { /* ... update based on click event */ }  
}  
  
let c = new ClickAction("Restart Game")  
getElementById("menu").addEventListener("click", c.execute)  
getElementById("btn").addEventListener("click", c.execute)  
setTimeout(c.execute, 2000)
```

Object (or function) represents an action, execution deferred, arguments possibly configured early.
Can be reused in multiple places. Can be queued, logged, ...

Interpreter Pattern

- Intent – given a language, define class hierarchy for parse tree, recursive method to interpret it
- Use case – regular expression matching
- Key types – *Expression*, *NonterminalExpression*, *TerminalExpression*
- JDK – no uses I'm aware of
- Necessarily uses Composite pattern!

Iterator Pattern

- Intent – provide a way to access elements of a collection without exposing representation
- Use case – collections
- Key types – *Iterable*, *Iterator*
 - But GoF discuss internal iteration, too
- Java and JavaScript: collections, for-each statement ..

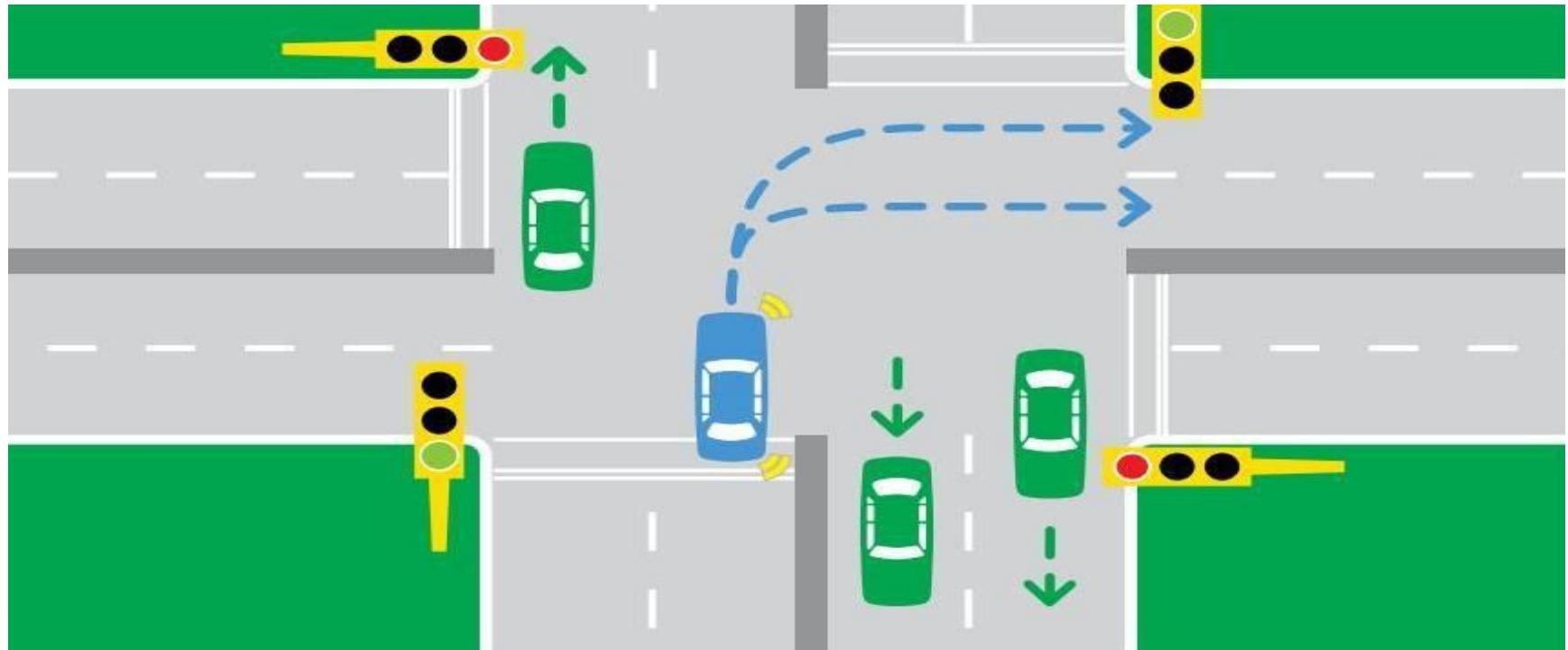
Iterator Illustration

```
public interface Iterable<E> {
    public abstract Iterator<E> iterator();
}
public class ArrayList<E> implements List<E> {
    public Iterator<E> iterator() { ... }
    ...
}
public class HashSet<E> implements Set<E> {
    public Iterator<E> iterator() { ... }
    ...
}
Collection<String> c = ...;
for (String s : c) // Creates an Iterator appropriate to c
    System.out.println(s);
```

Mediator Pattern

- Intent – define an object that encapsulates how a set of objects interact, to reduce coupling.
 - $\mathcal{O}(n)$ couplings instead of $\mathcal{O}(n^2)$
- Use case – dialog box where change in one component affects behavior of others
- Key types – Mediator, Components
- JDK – Unclear

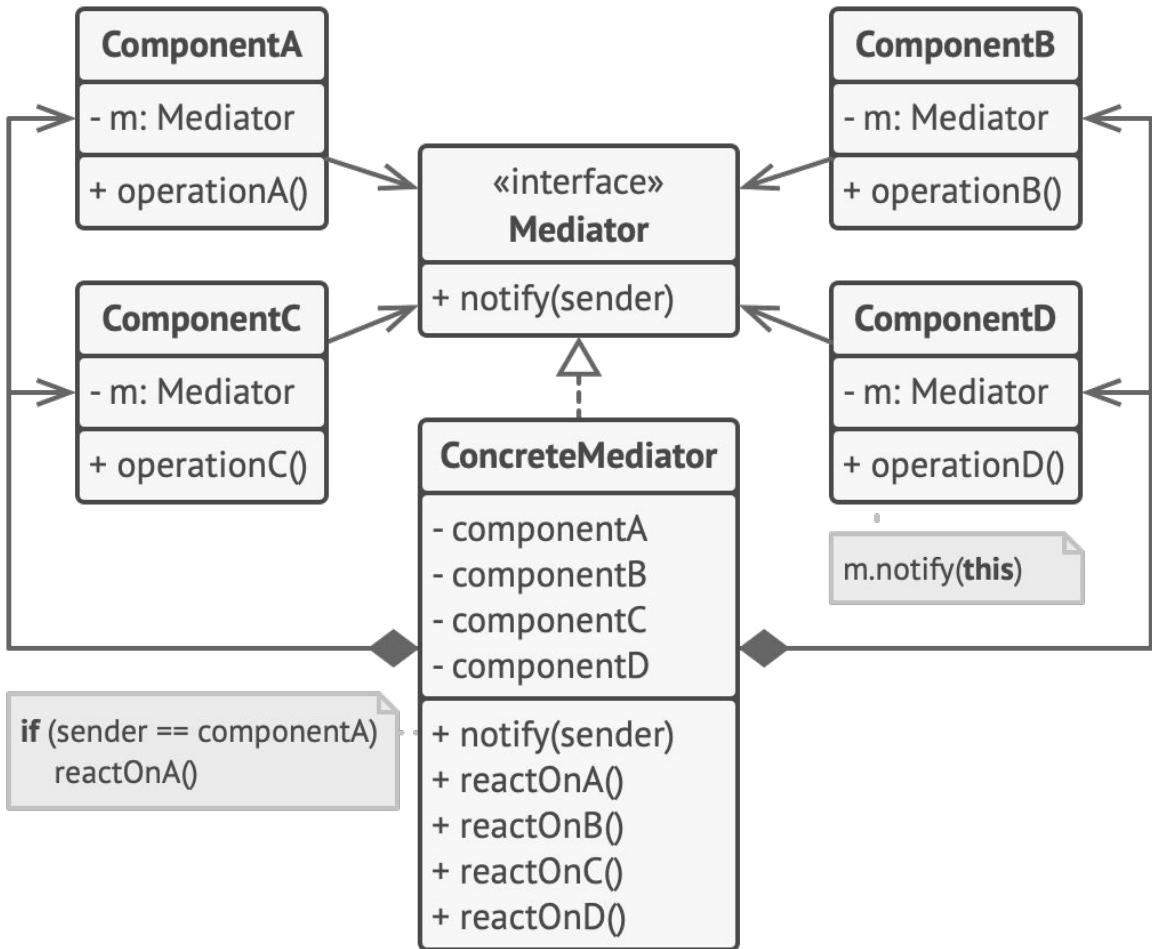
Mediator Illustration



Single responsibility
at mediator

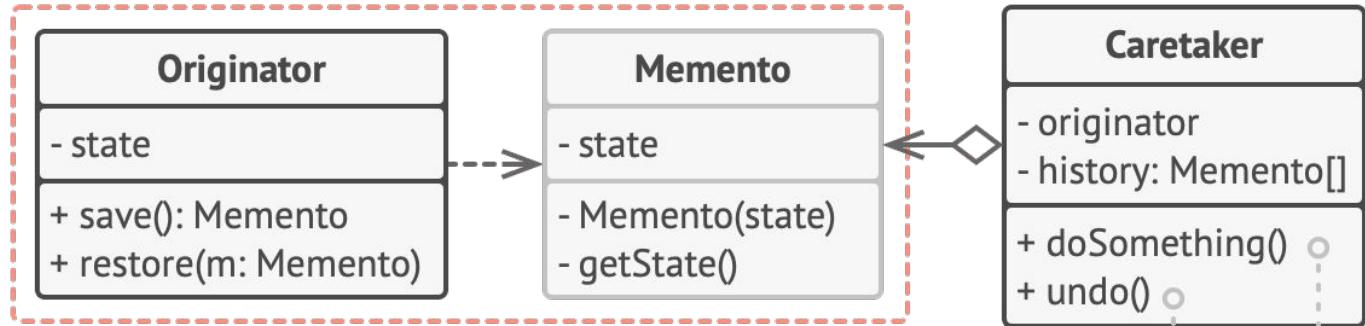
Coupling to single
component

God object?



Memento Pattern

- Intent – without violating encapsulation, allow client to capture an object's state, and restore
- Use case – undo stack for operations that aren't easily undone, e.g., line-art editor
- Key type – Memento (opaque state object)
- JDK – none that I'm aware of (*not* serialization)



Record snapshots of state

Avoid access to internal state

Allows undo

Consider using immutable
objects to begin with

```
m = history.pop()
originator.restore(m)
```

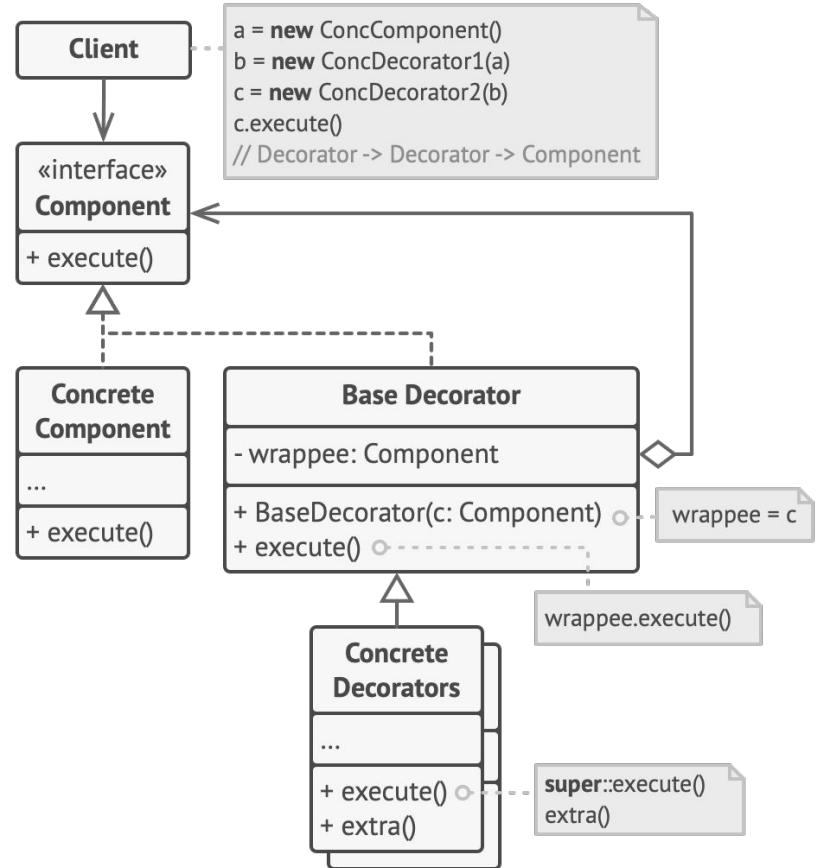
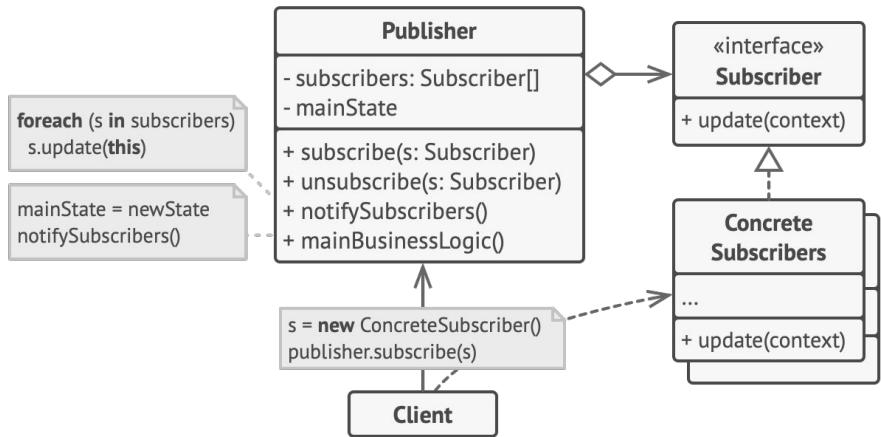
```
m = originator.save()
history.push(m)
// originator.change()
```

Observer Pattern

- Intent – let objects observe the behavior of other objects so they can stay in sync
- Use case – multiple views of a data object in a GUI
- Key types – *Subject* (“Observable”), *Observer*
 - GoF are agnostic on many details!
- Examples: All GUIs

Observer vs. Promise

Observer vs. Decorator



Observer vs Generator

Design Problem

You are developing a mobile application for cities where users can report potholes and similar problems (with photos) and city crews can investigate, prioritize, and address reports.

Design problem 6: Every time a report is resolved, one of multiple actions should be taken (email, text message, ...). The action is selected by the person creating the report.

Design Problem

You are developing a mobile application for cities where users can report potholes and similar problems (with photos) and city crews can investigate, prioritize, and address reports.

Design problem 7: Every time a report is resolved, multiple follow-up actions should be performed. Results should be added to a database, an email should be sent, a supervisor should be informed, etc. More actions might be added later.

Observer Characteristics

Inversion of control, remove direct dependency, reduce coupling

Listen to events, multiple events

Multiple observers possible

Add and remove observers at runtime

Push model/event-based programming: Observable pushes events to observer

State Pattern

- Intent – allow an object to alter its behavior when internal state changes. “Object will appear to change class.”
- Use case – TCP Connection (which is stateful)
- Key type – *State* (Object delegates to state!)
- JDK: none that I’m aware of, but easy to use

State Example

Without the pattern:

```
class Connection {  
    boolean isOpen = false;  
    void open() {  
        if (isOpen) throw new InvalidConnectionException();  
        ...//open connection  
        isOpen=true;  
    }  
    void close() {  
        if (!isOpen) throw new InvalidConnectionException();  
        ...//close connection  
        isOpen=false;  
    }  
}
```

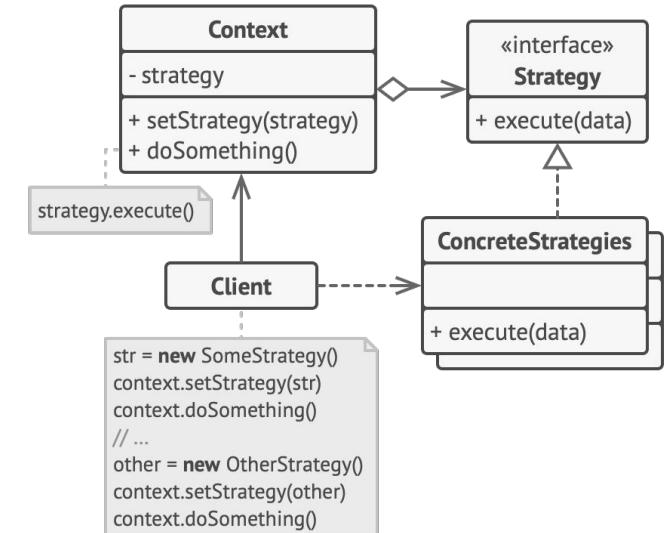
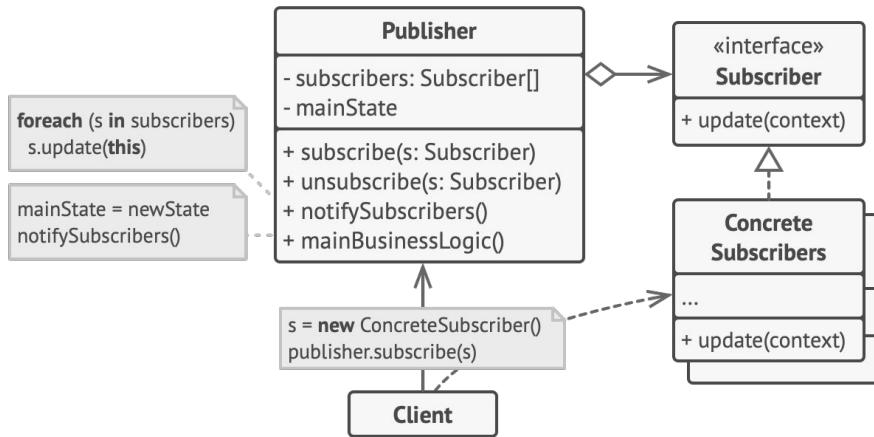
With the pattern:

```
class Connection {  
    private State state = new Closed();  
    public void setState(State s) { ... }  
    void open() { state.open(this); }  
    ...  
}  
  
interface State {  
    void open(Connection c);  
    void close(Connection c);  
}  
  
class Open implements State {  
    void open(Connection c) { throw new InvalidConnectionException(); }  
    void close(Connection c) {  
        //...close connection  
        c.setState(new Closed());  
    }  
}  
  
class Closed impl. State { ... }
```

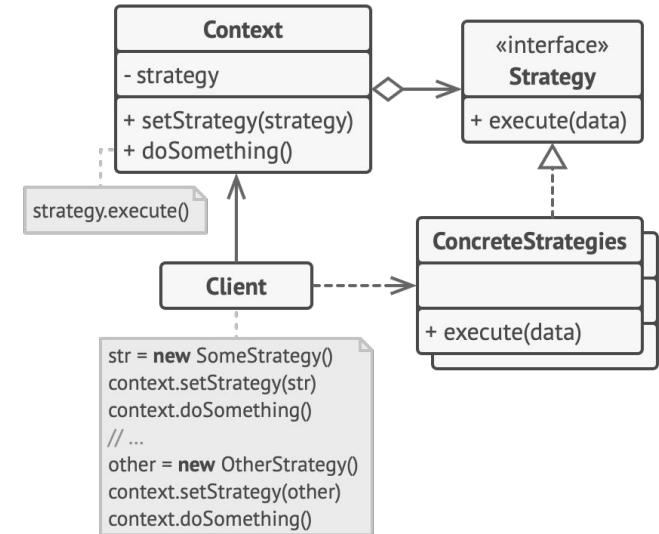
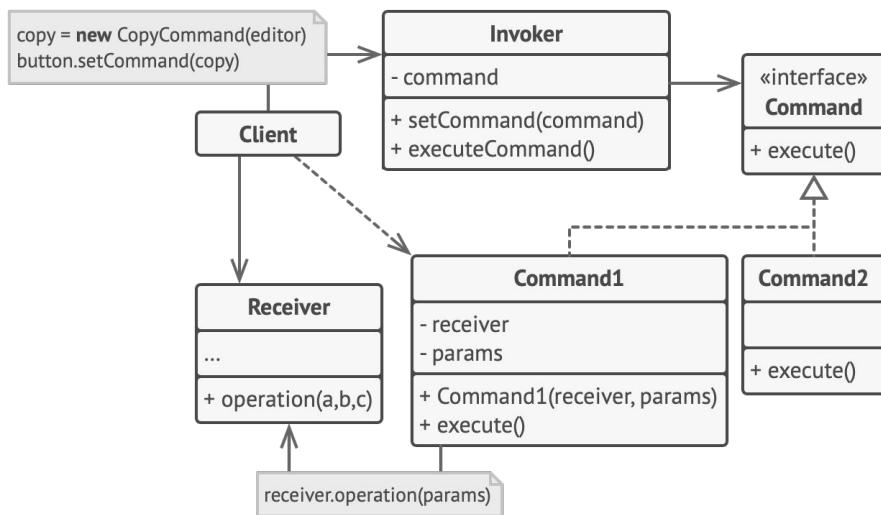
Strategy Pattern

- Intent – represent a behavior that parameterizes an algorithm for behavior or performance
- Use case – line-breaking for text compositing
- Key types – *Strategy*
- JDK – Comparator

Observer vs. Strategy



Command vs. Strategy

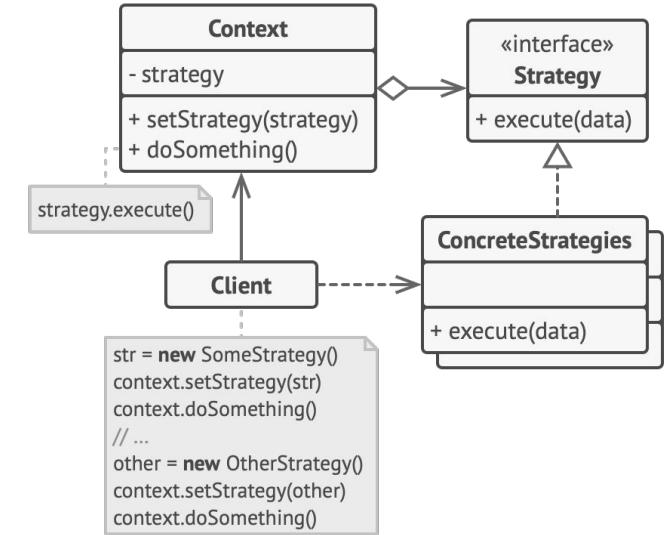
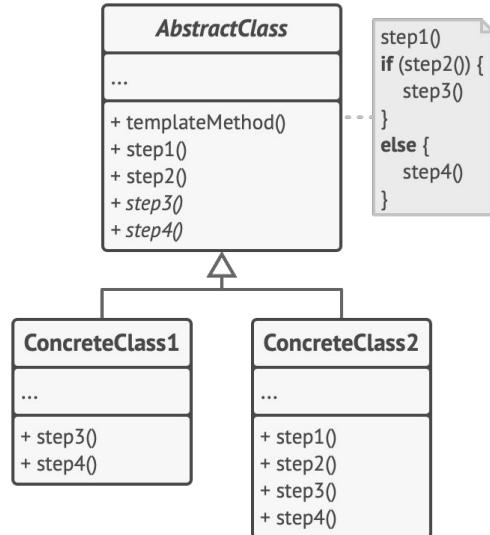


Very similar structure, but different intentions: Command is reusable, delayed function; strategy configures part of algorithm

Template Method Pattern

- Intent – define skeleton of an algorithm or data structure, deferring some decisions to subclasses
- Use case – application framework that lets plugins implement all operations on documents
- Key types – *AbstractClass*, *ConcreteClass*
- JDK – skeletal collection impls (e.g., *AbstractList*)

Strategy vs Template Method



Strategy vs Template Method

Delegation vs inheritance; context $\sim\sim$ template method

Template method: Single variation point, configured with constructor call

Strategy: Possibly multiple variation points in context, configured during constructor or dynamically later

Visitor Pattern

- Intent – represent an operation to be performed on elements of an object structure (e.g., a parse tree). Visitor lets you define a new operation without modifying the type hierarchy.
- Use case – type-checking, pretty-printing, etc.
- Key types – *Visitor*, *ConcreteVisitors*, all the element types that get visited
- JDK – *SimpleFileVisitor*; *AnnotationValueVisitor*; very common in compilers

Visitor Pattern Discussion

Double dispatch

Add new operations (like Command pattern)

Iterate over object structure (like Iterator pattern)

Provide object-specific visit methods to avoid dynamic type lookup

Most commonly used in context of compilers and other operations on trees

Different versions exist

Bonus: Other Design Principles

Where we are

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

SOLID Principles

Single-responsibility principle: Every class should have only one responsibility
-- *cohesion; low coupling; information expert*

The Open-closed principle: "Software entities ... should be open for extension, but closed for modification." -- *encapsulation*

Liskov substitution principle: Program against interface, even with subclassing

Interface segregation principle: Prefer specific small interfaces; multiple interfaces per object okay; cohesion

Dependency inversion principle: "Depend upon abstractions, [not] concretions." -- *prefer interfaces over class types; dynamic dispatch*

Other Common Principles

DRY Principle: Don't Repeat Yourself

KISS Principle: Keep It Simple, Stupid

YAGNI Principle: You Aren't Gonna Need It

Principle of Least Astonishment

Boy Scout Rule: Leave the Code Cleaner than you Found it

Summary

- Now you know all the Gang of Four patterns
- Definitions can be vague
- Coverage is incomplete
- But they're extremely valuable
 - They gave us a vocabulary
 - And a way of thinking about software
- Look for patterns as you read and write software
 - GoF, non-GoF, and undiscovered

Principles of Software Construction: Objects, Design, and Concurrency

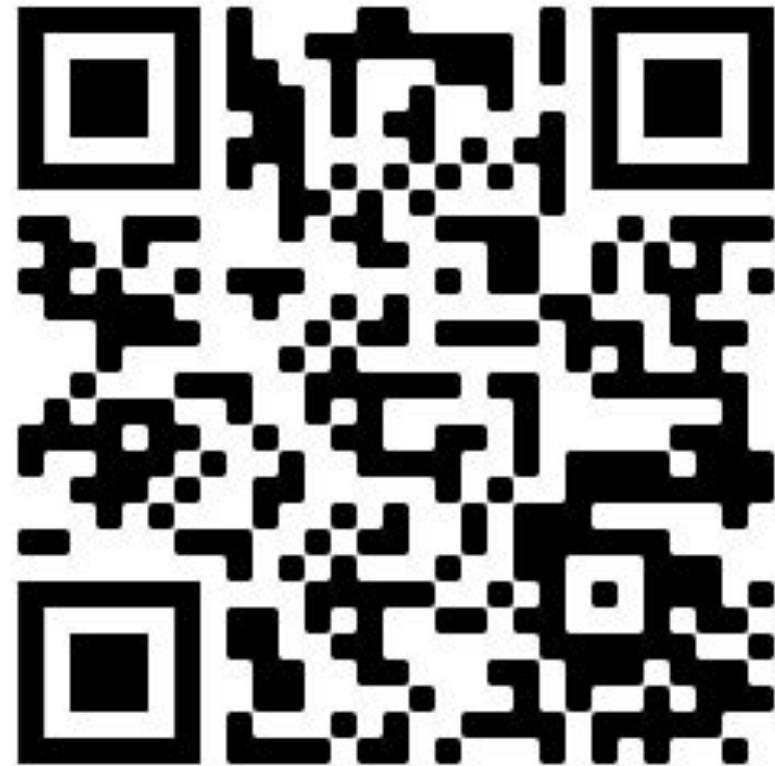
{Static & Dynamic} x {Typing & Analysis}

Christian Kästner Vincent Hellendoorn



Quiz

<https://rb.gy/ql0x0m>



How Do You Find Bugs?

```
private static int getValue(Integer i) {  
    return i.intValue();  
}
```

How Do You Find Bugs?

- Run it?

```
public class Fails {  
    public static void main(String[] args) {  
        getValue( i: null);  
    }  
  
    private static int getValue(Integer i) {  
        return i.intValue();  
    }  
}
```

```
Exception in thread "main" java.lang.NullPointerException Create breakpoint : Cannot invoke "java.lang.Integer.intValue()" because "i" is null  
at misc.Fails.getValue(Fails.java:9)  
at misc.Fails.main(Fails.java:5)
```

How Else Do You Find Bugs?

```
public class Fails {
    public static void main(String[] args) {
        getValue( i: null);
    }

    private static int getValue(Integer i) {
        return i.intValue();
    }
}
```

Static Analysis

IntelliJ can look at this code and say:

```
public static void main(String[] args) {
    getValue( i: null);
}

private static int getValue(Integer i) {
    return i.intValue();
}
```

Passing 'null' argument to parameter annotated as @NotNull

Static Analysis

How?

```
public static void main(String[] args) {
    getValue( i: null);
}

private static int getValue(Integer i) {
    return i.intValue();
}
```

Passing 'null' argument to parameter annotated as @NotNull

Static Analysis

How?

- We know at *compile time* where `getValue` gets routed to
- `getValue` calls a method on `i`
- `i` can be `null`

```
public static void main(String[] args) {
    getValue( i: null);
}

private static int getValue(Integer i) {
    return i.intValue();
}
```

Passing 'null' argument to parameter annotated as @NotNull

Static Analysis

How about JS?

fails.js

```
function getValue(x) {  
    return x.valueOf();  
}
```

Static Analysis

Run it: ✓

JS fails.js > ...

```
1  function getValue(x) {  
2      return x.valueOf();  
3  }  
4  
5  console.log(getValue("32"));  
6  console.log(getValue(null));
```

PROBLEMS

3

OUTPUT

TERMINAL

DEBUG CONSO

return x.valueOf();
^

TypeError: Cannot read property 'valueOf' of null

Static Analysis

Why no warning?

```
function getValue(x) {  
    return x.valueOf();  
}  
  
console.log(getValue("32"));  
console.log(getValue(null));
```

Another Java vs JS Example

```
class Foo {  
    constructor(x) {  
        this.x = x;  
    }  
}
```

```
function bar(foo) {  
    return foo.x;  
}
```

```
var foo = new Foo(3);  
console.log(bar(foo));  
console.log(bar(3));
```

```
class Foo {  
    int x;  
    Foo(int x) {  
        this.x = x;  
    }  
}
```

```
public static void main(String[] args) {  
    Foo foo = new Foo( 3 );  
    bar(foo);  
    bar( foo: 3 );  
}
```

```
private static void bar(Foo foo) {  
    System.out.println(foo.x);  
}
```

Static vs. Dynamic Typing

- The more knowledge we inject in the code, the more bugs we can catch at compile time
 - Types, nullity annotations, invariants
- At compile-time:
 - Dynamically typed languages assume nothing
 - Types exist only for *values*
 - Static typing is not completely precise either
 - Objects have declared types and run-time types
 - Different “strength” type systems

Static vs. Dynamic Typing

- The more knowledge we inject in the code, the more bugs we can catch at compile time
 - Types, nullity annotations, invariants
- Is it worth it?
 - Dynamic typing can severely limit inference
 - But... static types are a lot of work

Static vs. Dynamic Typing

- The more knowledge we inject in the code, the more bugs we can catch at compile time
 - Types, nullity annotations, invariants
- Is it worth it?
 - Dynamic typing can severely limit inference
 - But... static types are a lot of work

Do Static Type Systems Improve the Maintainability of Software Systems? An Empirical Study

Sebastian Kleinschmager,
Stefan Hanenberg

University of Duisburg-Essen
Essen, Germany

sebastian.kleinschmager@stud.uni-due.de
stefan.hanenberg@icb.uni-due.de

Romain Robbes,
Éric Tanter

Computer Science Dept (DCC)
University of Chile, Chile
rrobbes@dcc.uchile.cl
etanter@dcc.uchile.cl

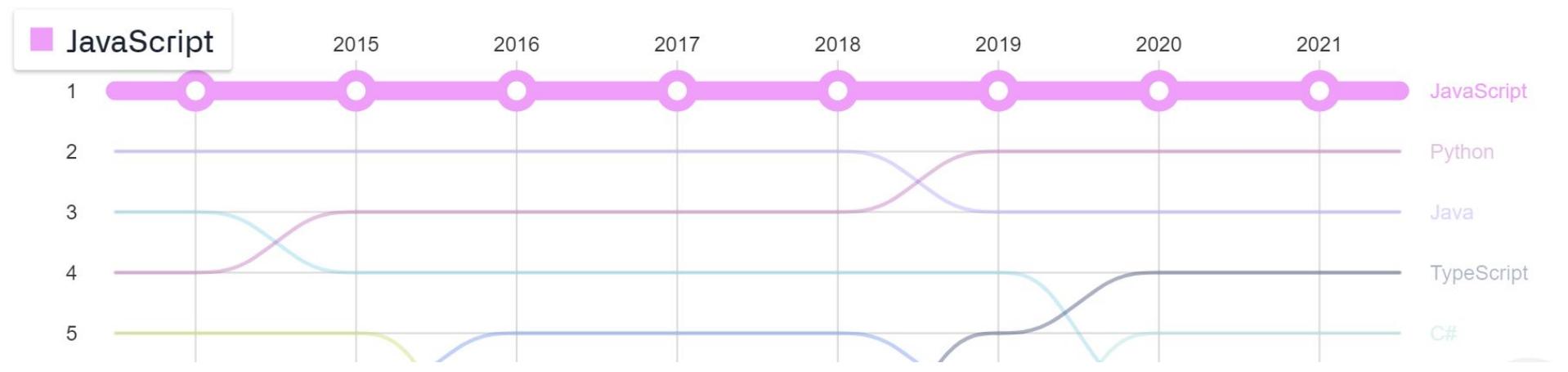
Andreas Stefik

Department of Computer Science
Southern Illinois University Edwardsville
Edwardsville, IL

Static vs. Dynamic Typing

Okay, but:

Top languages over the years



False Dichotomy?

Yes, but:

Top languages over the years



Partial Types

- Low effort, some utility
 - Static types exist and are checked at compile-time
 - Dynamic types are used at run-time
 - So annotations get ignored!
 - Type checker can be shallow or deep; TS is shallow

To Type or Not to Type: Quantifying Detectable Bugs in JavaScript

Zheng Gao
University College London
London, UK
z.gao.12@ucl.ac.uk

Christian Bird
Microsoft Research
Redmond, USA
cbird@microsoft.com

Earl T. Barr
University College London
London, UK
e.barr@ucl.ac.uk

*Abstract—*JavaScript is growing explosively and is now used in large mature projects even outside the web domain. JavaScript is also a dynamically typed language, so it is natural to invest in static type...

Types in TypeScript

```
function getValue(x: number) {  
    return x.valueOf();  
}
```

Argument of type 'null' is not assignable to parameter of type 'number'. ts(2345)

[View Problem](#) No quick fixes available

```
console.log(getValue(null));
```

Types in TypeScript

```
function getValue(x: number | null) {  
    return x.valueOf();  
}  
  
Object is possibly 'null'. ts(2531)  
(parameter) x: number | null  
  
View Problem No quick fixes available  
  
console.log(getValue(null));
```

Step Back

- Why do we care about types so much?

Step Back

- Why do we care about types so much?
 - We care about *common mistakes*
 - Type errors happen to be very common
 - What else is common?

Step Back

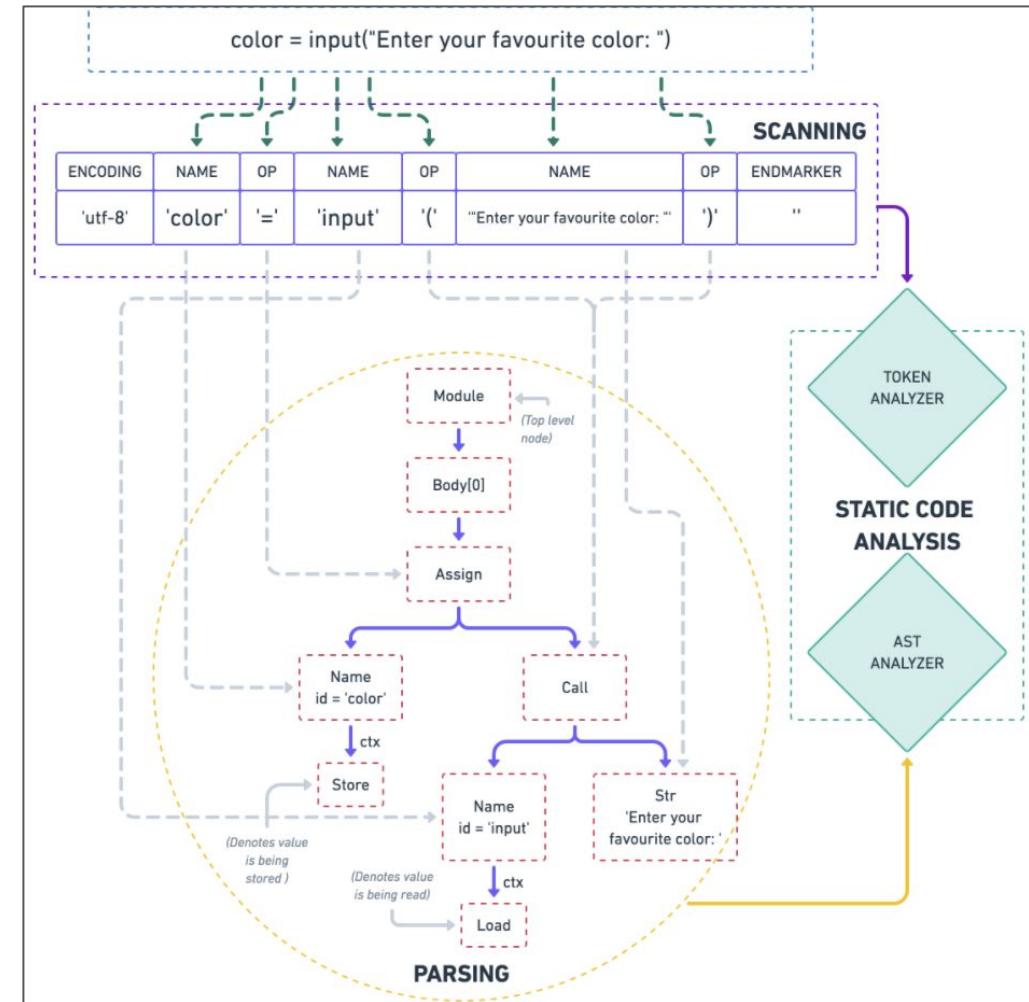
- Why do we care about types so much?
 - We care about *common mistakes*
 - Type errors happen to be very common
 - What else is common?
 - Nullity errors
 - Missing imports
 - ```
public void forward(String sender) {
 if (sender == "me") {
 sendSelf();
 } else if (sender == "other") {
```

# Static Analysis

- Detect real or plausible bugs based on code patterns
  - Plausible: look for risk-prone areas
    - Deeply nested loops
    - Overly general types (e.g., ‘any’ in TS)
    - Dead code/unused variables
    - Any other places we often make mistakes?

# Static Analysis

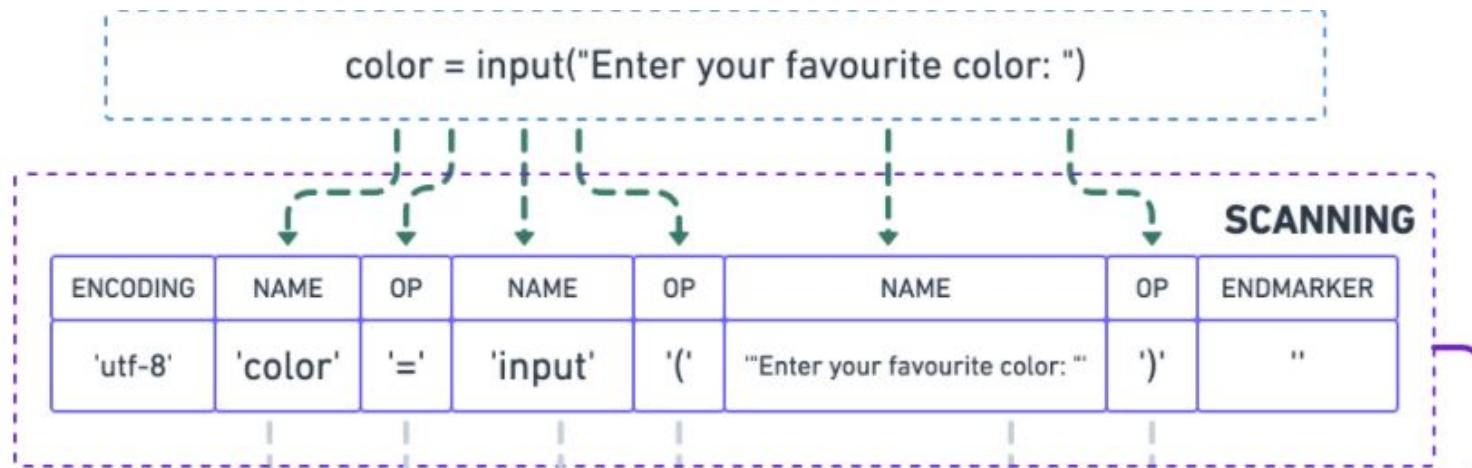
- How?
  - Program analysis + Vocabulary of patterns



<https://deepsource.io/blog/introduction-static-code-analysis/>

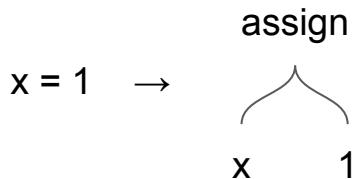
# Static Analysis

- Step 1: Tokenization
  - Tokens are like the words of software
  - *Lexical* categories, incl. punctuation, identifiers, operators, strings

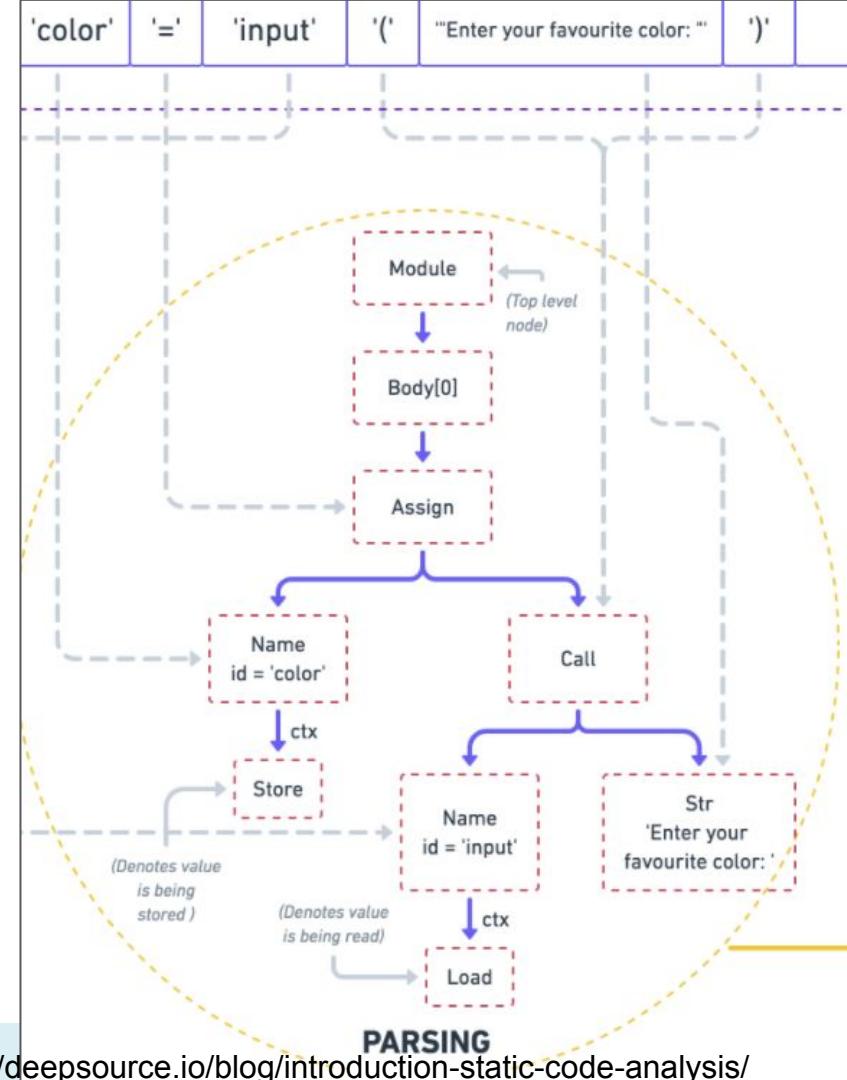


# Static Analysis

- Step 2: Parsing
  - To the compiler/interpreter, software is a tree
  - Root node is file/module
  - Leaves mainly identifiers, literals
  - Internal nodes capture **structure**

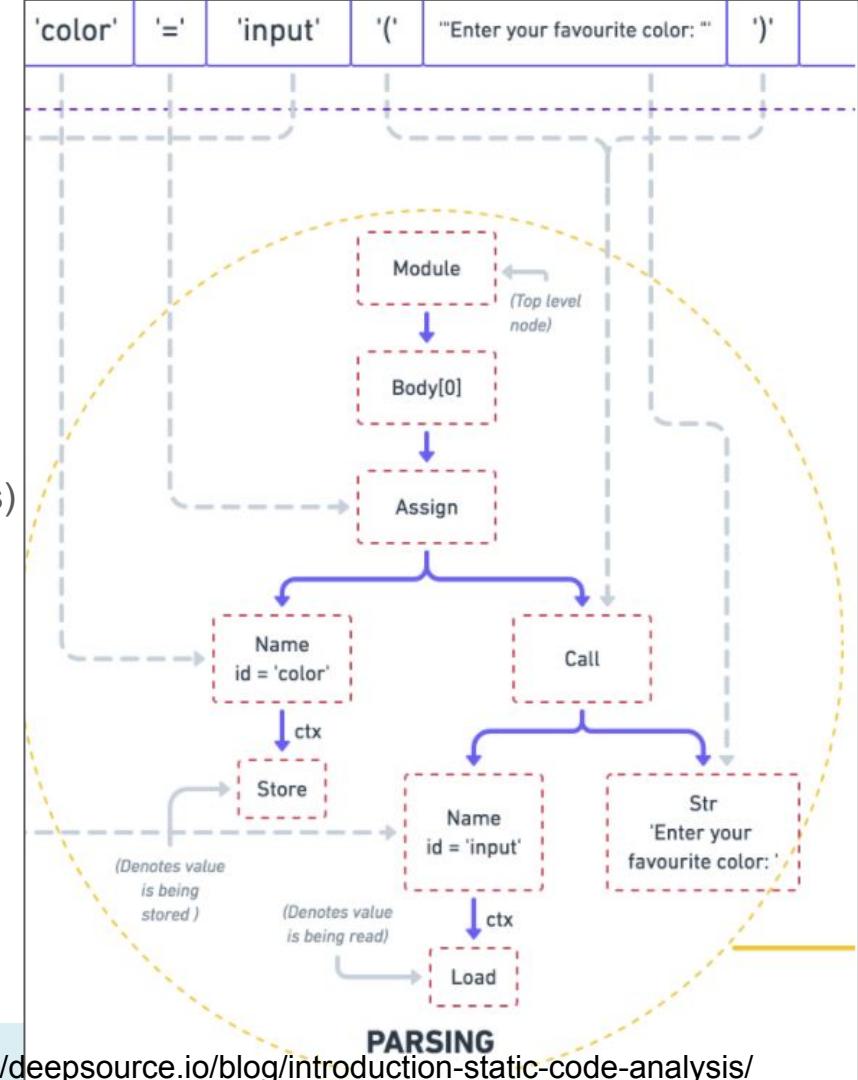


Consider checking out: <https://ast.carlosroso.com/>



# Static Analysis

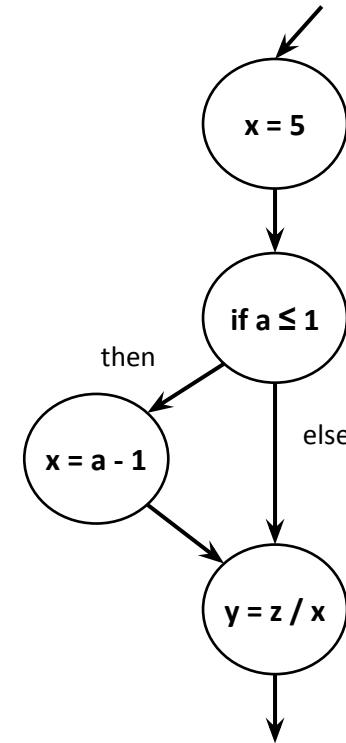
- Step 2: Parsing
  - What does this get us?
  - Rich structure
    - Syntactic types (variables, method calls)
    - Dead code, deep nesting
  - A lot of type resolution
    - What vars are stored, loaded
    - Not complete!
    - Need to *build* to understand imports



# Static Analysis

- Step 2b: Advanced Analysis
  - The compiler doesn't stop at parsing
  - Familiar?

```
public boolean div(int a, int z) {
 int x = 5;
 if (a <= 1) {
 x = a - 1;
 }
 return z / x;
}
```



# Static Analysis

- Step 2b: Advanced Analysis
  - The compiler doesn't stop at parsing
  - There is a lot more down this rabbit hole
    - Control/data-flow, abstract interpretation, (dynamic) symbolic execution,
  - Consider a Programming Languages or Compilers course

# Static Analysis

- Step 3: register analyzers
  - At the core: walk the tree

```
class ListDefinitionChecker(BaseChecker):
 msg = "usage of 'list()' detected, use '[]' instead"

 def visit_Call(self, node):
 name = getattr(node.func, "id", None)
 if name and name == list.__name__ and not node.args:
 self.violations.append((self.filename, node.lineno, self.msg))
```

# Static Analysis

- Step 3: register analyzers
  - At the core: walk the tree
  - Sometimes more complex

```
class UnusedImportChecker(BaseChecker):
 def __init__(self):
 self.import_map = defaultdict(set)
 self.name_map = defaultdict(set)

 def _add_imports(self, node):
 for import_name in node.names:
 # Store only top-level module name ("os.path" -> "os").
 # We can't easily detect when "os.path" is used.
 name = import_name.name.partition(".")[0]
 self.import_map[self.filename].add((name, node.lineno))

 def visit_Import(self, node):
 self._add_imports(node)

 def visit_ImportFrom(self, node):
 self._add_imports(node)

 def visit_Name(self, node):
 # We only add those nodes for which a value is being read from.
 if isinstance(node.ctx, ast.Load):
 self.name_map[self.filename].add(node.id)
```

# Static Analysis

- Compared to Linters:
  - Linters mainly enforce style -- comments, quotes, idioms
    - This also requires static analysis! Just nothing particularly fancy
  - Some overlap; good conventions help avoid bugs

# Static Analysis

- Compared to Parsers:
  - Parsers check for syntactic correctness
    - Can catch bugs as well, e.g. missing “;”
  - Parsing is often a key step in static analysis
    - Hard to do right with just text/regexes.
  - Parsing is a platform for further analyses
    - control-flow, data-flow

# So... Static Analysis for Everything?

- Can we find every bug?

- No! Rice's Theorem

"Any nontrivial property about the language recognized by a Turing machine is undecidable." -- Henry Gordon Rice, 1953

- Every static analysis is necessarily incomplete or unsound or undecidable (or multiple of these)

# So... Static Analysis for Everything?

- Can we find every bug?
- Can we guarantee correctness?

# So... Static Analysis for Everything?

- Can we find every bug?
- Can we guarantee correctness?
  - Yes, but... much less useful

```
public class Fails {
 public static void main(String[] args) {
 getValue(i: null);
 }

 private static int getValue(Integer i) {
 return i.intValue();
 }
}
```

# Soundness & Precision

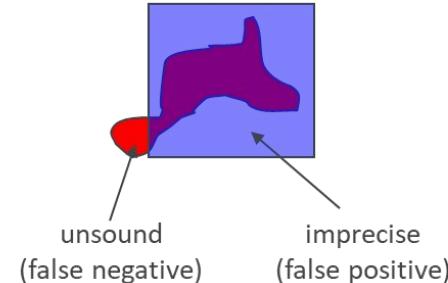
- Since we can't perfectly analyze behavior statically
  - We may miss things by being cautious (unsound; false negative)
  - We might identify non-problems (imprecision, false positive)



Program state covered in actual execution



Program state covered by abstract execution with analysis



# The Social Side

- How to deploy tools that are neither sound nor complete?

# Static Analysis at Google

- Centered around FindBugs (succeeded by SpotBugs)
  - Essentially, a huge collection of risky patterns on Java bytecode
  - Annotated with five levels of concern

CONTRIBUTED ARTICLES

## Lessons from Building Static Analysis Tools at Google

By Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, Ciera Jaspan  
Communications of the ACM, April 2018, Vol. 61 No. 4, Pages 58-66  
10.1145/3188720  
[Comments](#)

VIEW AS:

SHARE:

Software bugs cost developers and software companies billions of dollars every year.



# Static Analysis at Google

- Three experiments in the early 2000s:
  1. **A dashboard:** run FindBugs overnight, report results in a centralized location  
*Failed because:* dashboard is outside the developer's workflow

# Static Analysis at Google

- Three experiments in the early 2000s:
  1. **A dashboard:** run FindBugs overnight, report results in a centralized location  
*Failed because:* dashboard is outside the developer's workflow
  2. **Recurring FixIt events:** company-wide one-week effort to fix warnings  
*Failed because:* actually fixed some bugs, but FindBugs is too imprecise (44% of issues were “bugs”, but only 16% mattered)

# Static Analysis at Google

- Three experiments in the early 2000s:
  1. **A dashboard:** run FindBugs overnight, report results in a centralized location  
*Failed because:* dashboard is outside the developer's workflow
  2. **Recurring FixIt events:** company-wide one-week effort to fix warnings  
*Failed because:* actually fixed some bugs, but FindBugs is too imprecise (44% of issues were “bugs”, but only 16% mattered)
  3. **Add to Code Review:** run on every change, allow toggling warnings  
*Failed because:* too imprecise; suppressing FPs made it inconsistent

# Static Analysis at Google

Okay so then what?

- What went wrong / what do we need?

# Static Analysis at Google

Okay so then what?

- What went wrong / what do we need?
  1. Precision is key -- developers lose faith in inaccurate tools
  2. Provide timely warnings -- in-IDE or rapidly on builds
    - a. Checkers are way more useful during coding
  3. Make a platform -- allow adding useful checks

# Static Analysis at Google

Specifically:

- At compile-time:
  - Perfectly Precise
    - **No** false-positives; never halt a build incorrectly
  - Simple
  - Actionable
    - Ideally to the point of auto-fix suggestions

# Static Analysis at Google

Specifically:

- At review time: TriCoder
  - 90%+ precise
    - If it drops below, checker gets disabled! Onus on checker authors to fix
  - Actionable, but may require some work
  - Improve correctness or code quality
  - Some compile-time checks moved to review-time!
- Ran 50K times per day -- in 2018

# TriCoder

```
package com.google.devtools.staticanalysis;

public class Test {
```

▼ Lint Missing a Javadoc comment.

Java

1:02 AM, Aug 21

[Please fix](#)

[Not useful](#)

```
 public boolean foo() {
 return getString() == "foo".toString();
```

▼ ErrorProne String comparison using reference equality instead of value equality

StringEquality

1:03 AM, Aug 21

(see <http://code.google.com/p/error-prone/wiki/StringEquality>)

[Please fix](#)

```
//depot/google3/java/com/google/devtools/staticanalysis/Test.java
```

```
package com.google.devtools.staticanalysis;
```

```
public class Test {
 public boolean foo() {
 return getString() == "foo".toString();
 }
```

```
 public String getString() {
 return new String("foo");
 }
}
```

```
package com.google.devtools.staticanalysis;
```

```
import java.util.Objects;
```

```
public class Test {
 public boolean foo() {
 return Objects.equals(getString(), "foo".toString());
 }
```

```
 public String getString() {
 return new String("foo");
 }
}
```

# Static Analysis at Google

- The gist: Many simple precise checks
  - What else could one do?

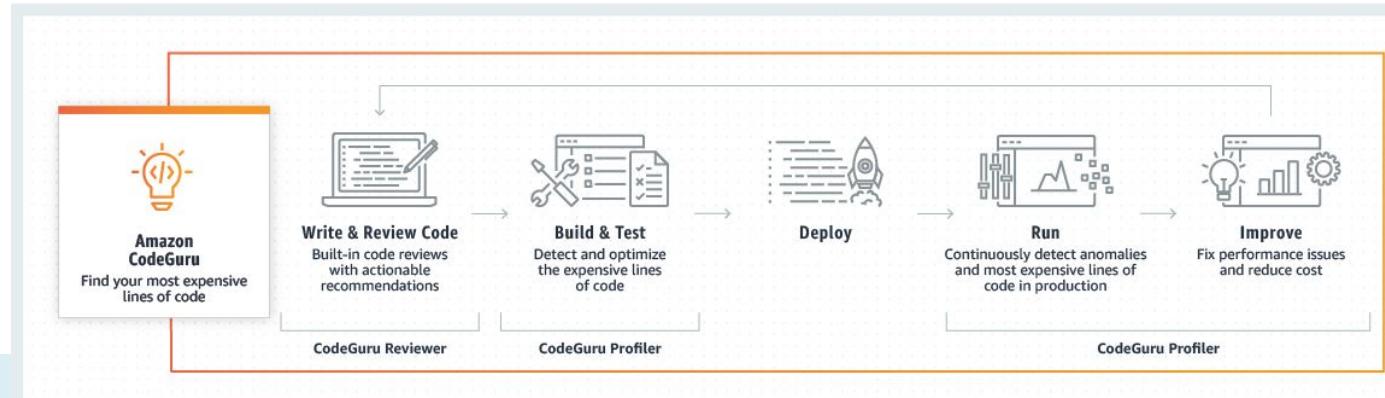
# Static Analysis at Google

- The gist: Many simple precise checks
  - What else could one do?
- Infer at Facebook
  - Built around separation logic; geared heavily towards tracking resources
    - Null-pointer dereferences, resource leaks, unintended data access
  - Google claims this won't (easily) scale to their multi-billion line mono-repo



# Static Analysis at Google

- The gist: Many simple precise checks
  - What else could one do?
- Use AI?
  - Rule-mining from previous reviews
    - Detects typical vulnerabilities, bad patterns
  - Mostly fairly simple ML (details limited)



# Static Analysis at Google

- The gist: Many simple precise checks
  - What else could one do?
- Use AI?
  - Microsoft's IntelliSense in VSCode
  - Mostly refactorings, code completions
  - Trained on large volumes of code

# Static Analysis at Google

- The gist: Many simple precise checks
  - What else could one do?
- Use AI?
  - Shameless plug: AI is rapidly learning to program. If this interests you, come do research with me :)



# Summary

- We all constantly make mistakes
  - Static analysis captures common issues
  - Choose suitable abstractions; consider trade-offs
    - E.g., dynamic vs. static typing; sound vs. precise
- At big-tech-scale, automated checks are key
  - Help normalize coding standards
  - Even rare bugs are common at scale
  - But: social factors are very important

# HW6 Feedback

<https://rb.gy/itzmja>



# Principles of Software Construction: Objects, Design, and Concurrency

## DevOps

Christian Kästner   Vincent Hellendoorn



17-214/514



# Topics

From CI to CD

Containers

Configuration management

Monitoring

Feature flags, testing in production

# Where we are

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

# Recall: Continuous Integration

**travis**

Home Stats Blog Docs

miles

Fork me on GitHub

**rails/rails**

Ruby on Rails

11762 2563

Recent My Repositories

**diasporg/diaspora #209**  
Duration: 19 min 26 sec, Finished: 9 minutes ago

**rubinius/rubinius #815**  
Duration: 16 min 28 sec, Finished: about an hour ago

**robgleeson/ed #31**  
Duration: 4 min 33 sec, Finished: about an hour ago

**niku/frange #4**  
Duration: 51 sec, Finished: about 2 hours ago

**tedsuo/raraaa #48**  
Duration: 1 min, Finished: about 2 hours ago

**holman/play 6 #84**  
Duration: 4 min 49 sec, Finished: about 2 hours ago

**crcn/sift.js #35**  
Duration: 41 sec, Finished: about 2 hours ago

**BonzaProject/Bonza #19**  
Duration: 40 sec, Finished: about 2 hours ago

**Current Build History**

Build 1995 Commit f3e079e (master)  
 Finished about 6 hours ago Compare b5927b8...f3e079e  
 Duration 1 hr 33 min 32 sec Author Vijay Dev  
 Message Merge pull request #4248 from andrew/2012 Updated copyright notices for 2012

**Build Matrix**

| Job    | Duration      | Finished          | Rvm   | Env                   |
|--------|---------------|-------------------|-------|-----------------------|
| 1995.1 | 19 min 5 sec  | about 6 hours ago | 1.9.3 | GEM=railties          |
| 1995.2 | 12 min 38 sec | about 6 hours ago | 1.9.3 | GEM=ap,am,amo,ares,as |
| 1995.3 | 16 min 57 sec | about 6 hours ago | 1.9.3 | GEM=ar:mysql          |
| 1995.4 | 12 min 55 sec | about 6 hours ago | 1.9.3 | GEM=ar:mysql2         |
| 1995.5 | 12 min 34 sec | about 6 hours ago | 1.9.3 | GEM=ar:sqlite3        |
| 1995.6 | 19 min 23 sec | about 6 hours ago | 1.9.3 | GEM=ar:postgresql     |

**Workers**

- erlang.worker.travis-ci.org
- nodejs1.worker.travis-ci.org
- php1.worker.travis-ci.org
- rails1.worker.travis-ci.org
- rails2.worker.travis-ci.org
- ruby1.worker.travis-ci.org
- ruby2.worker.travis-ci.org
- ruby3.worker.travis-ci.org
- spree.worker.travis-ci.org

**Queue: Common**  
No jobs

**Queue: NodeJs**  
No jobs

**Queue: Php**  
No jobs

**Queue: Rails**  
No jobs

**Queue: Erlang**  
No jobs

**Queue: Spree**  
No jobs

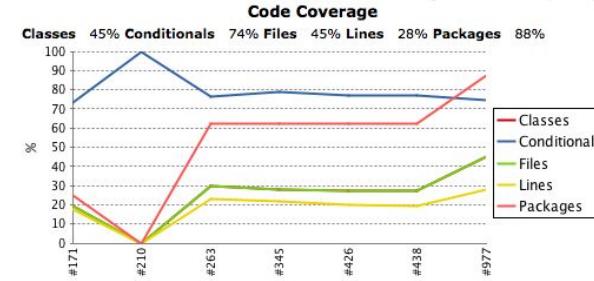
- [Back to Dashboard](#)
- [Status](#)
- [Changes](#)
- [Workspace](#)
- [Build Now](#)
- [Delete Project](#)
- [Configure](#)
- [Set Next Build Number](#)
- [Duplicate Code](#)
- [Coverage Report](#)
- [SLOCCount](#)
- [Git Polling Log](#)

## Project Stop-tabac dev

CI build

[edit description](#)  
[Disable Project](#)

- [!\[\]\(04a9d1ca2fd6373ec1bd7524fdd9aebe\_img.jpg\) Coverage Report](#)
- [!\[\]\(7bf184123b1b86d325d73cfc1c8a1420\_img.jpg\) Workspace](#)
- [!\[\]\(c75116e691bad00875b41418e9fffc1b\_img.jpg\) Recent Changes](#)
- [!\[\]\(8b133b12e7b11df3c9410fcc9b7fddc9\_img.jpg\) Latest Test Result \(no failures\)](#)



### Build History (trend)

- [!\[\]\(24e79c029131d23a574d721c7f2b7ac9\_img.jpg\) #977 Aug 27, 2012 4:37:27 PM](#)
- [!\[\]\(d8f6d0bfed519b84a67c72fb2c7dd6dc\_img.jpg\) #438 Jun 28, 2012 8:47:42 AM](#)
- [!\[\]\(ed4e14458d15584c30779ca21f237dd6\_img.jpg\) #426 Jun 26, 2012 1:39:39 PM](#)
- [!\[\]\(2eae9a7c7e6642b3b2248b983b739110\_img.jpg\) #345 Jun 19, 2012 9:02:20 AM](#)
- [!\[\]\(5f25f417fba43e1b1bf9cda92118a624\_img.jpg\) #263 Jun 6, 2012 9:14:42 PM](#)
- [!\[\]\(4d93bd45a46d1450e6a69cdb5ca92941\_img.jpg\) #210 May 31, 2012 8:42:29 AM](#)
- [!\[\]\(f664f5d0325f95fcaa5af352c52e98e2\_img.jpg\) #171 May 23, 2012 9:58:18 PM](#)
- [!\[\]\(c36e39b39b85d0b17cac7f643917492c\_img.jpg\) #90 May 15, 2012 11:49:41 AM](#)

 RSS for all  RSS for failures



# Continuous Integration

- Automation
  - Ensures absence of obvious build issues and configuration issues (e.g., dependencies all checked in)
  - Ensures tests are executed
  - May encourage more tests
  - Can run checks on different platforms
- 
- What can all be automated?

# Any repetitive QA work remaining?

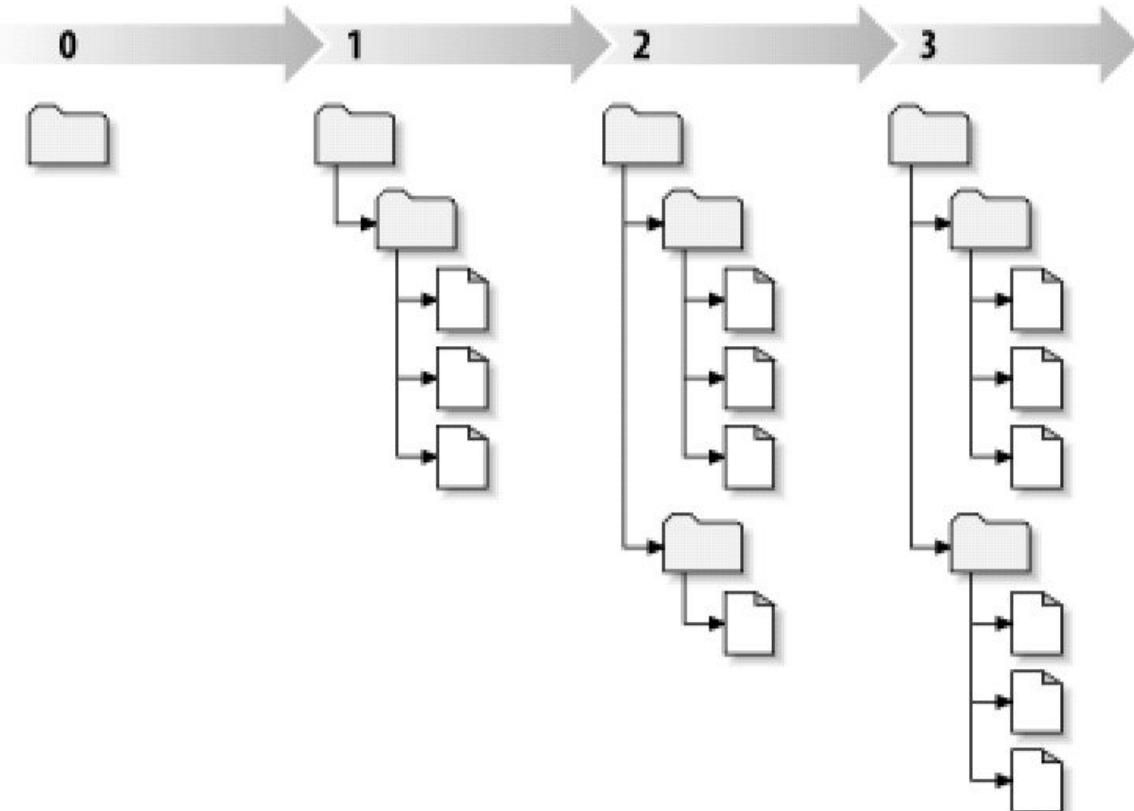
# Releasing Software

# Semantic Versioning for Releases

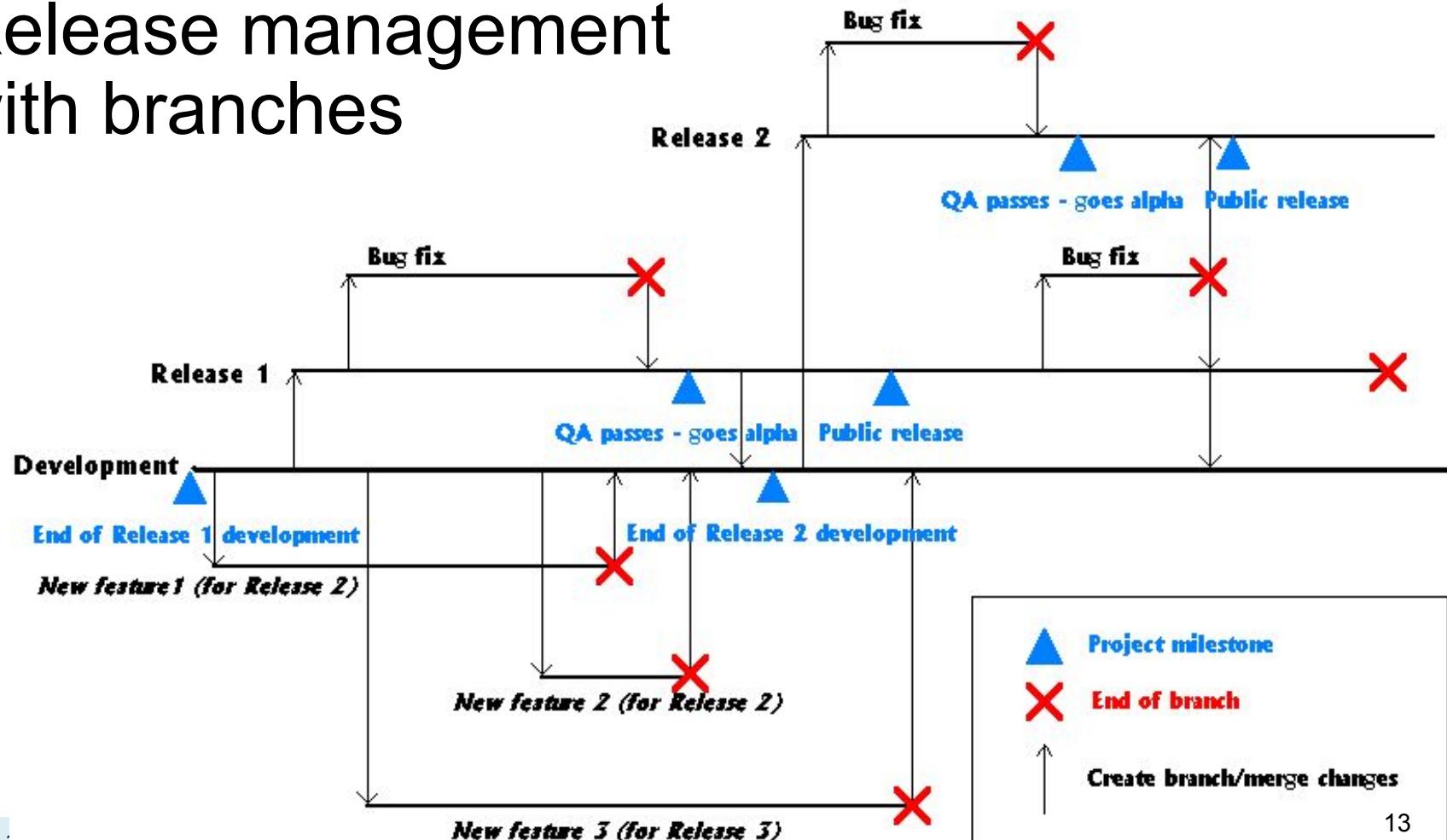
- Given a version number MAJOR.MINOR.PATCH, increment the:
  - MAJOR version when you make incompatible API changes,
  - MINOR version when you add functionality in a backwards-compatible manner, and
  - PATCH version when you make backwards-compatible bug fixes.
- Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

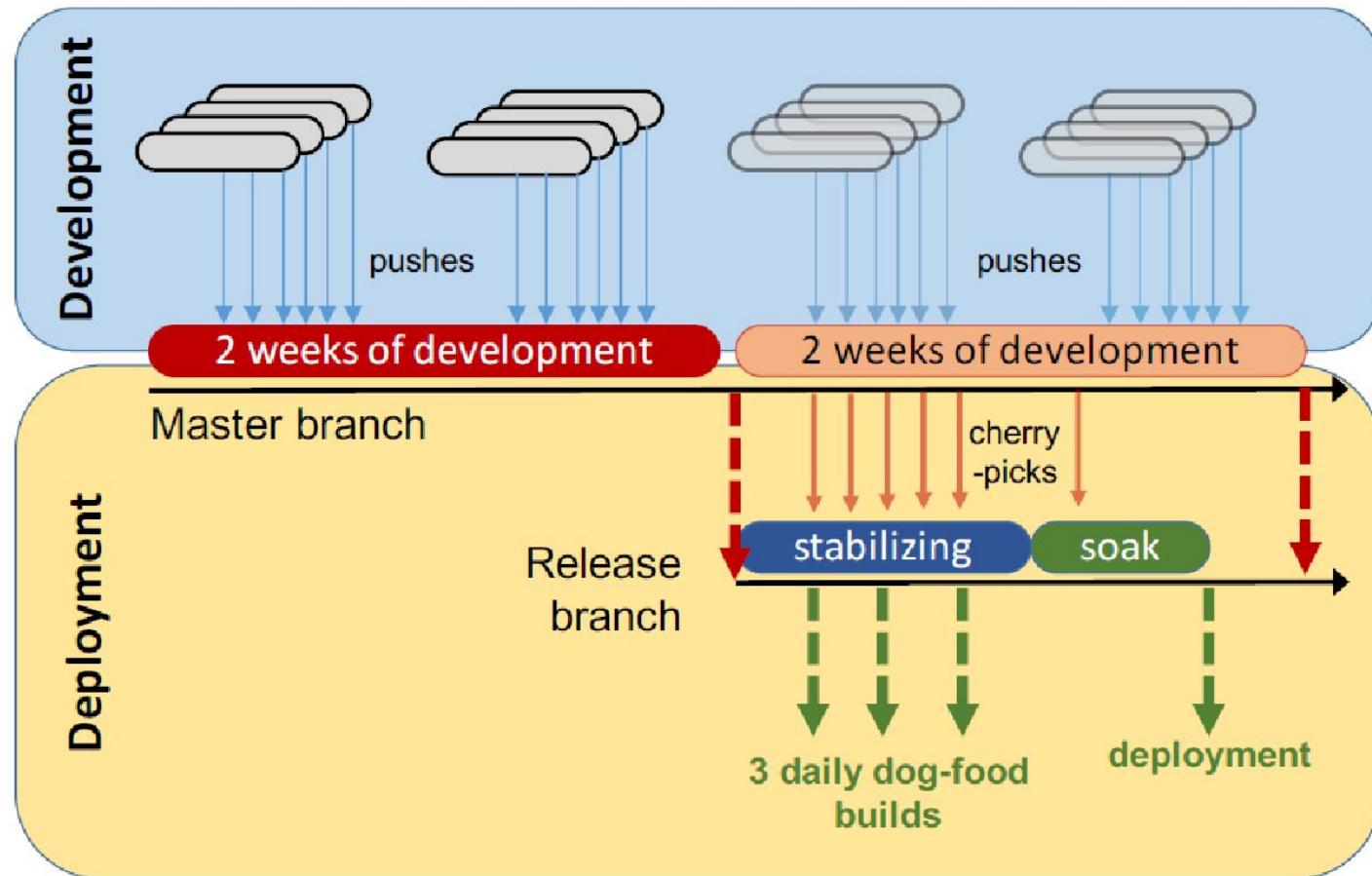
<http://semver.org/>

# Versioning entire projects



# Release management with branches





# Facebook Tests for Mobile Apps

Unit tests (white box)

Static analysis (null pointer warnings, memory leaks, ...)

Build tests (compilation succeeds)

Snapshot tests (screenshot comparison, pixel by pixel)

Integration tests (black box, in simulators)

Performance tests (resource usage)

Capacity and conformance tests (custom)

Further readings: Rossi, Chuck, Elisa Shibley, Shi Su, Kent Beck, Tony Savor, and Michael Stumm. Continuous deployment of mobile software at facebook (showcase). In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 12-23. ACM, 2016.

# Release Challenges for Mobile Apps

Large downloads

Download time at user discretion

Different versions in production

Pull support for old releases?

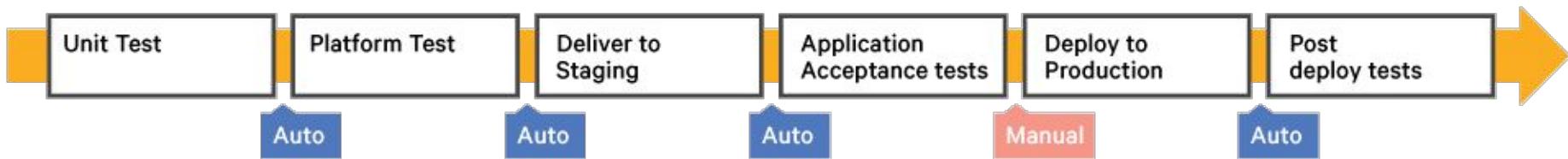
Server side releases silent and quick, consistent

-> App as container, most content + layout from server

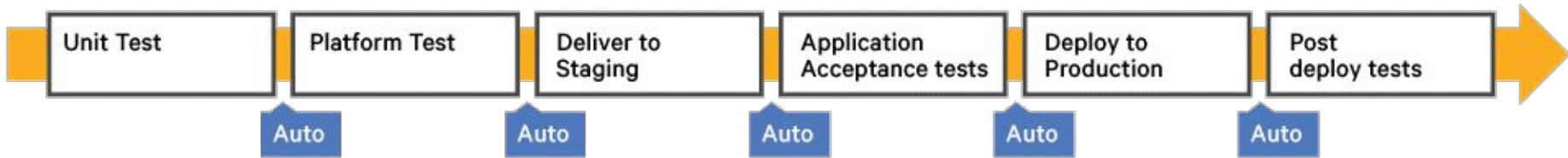
# From Release Date to Continuous Release

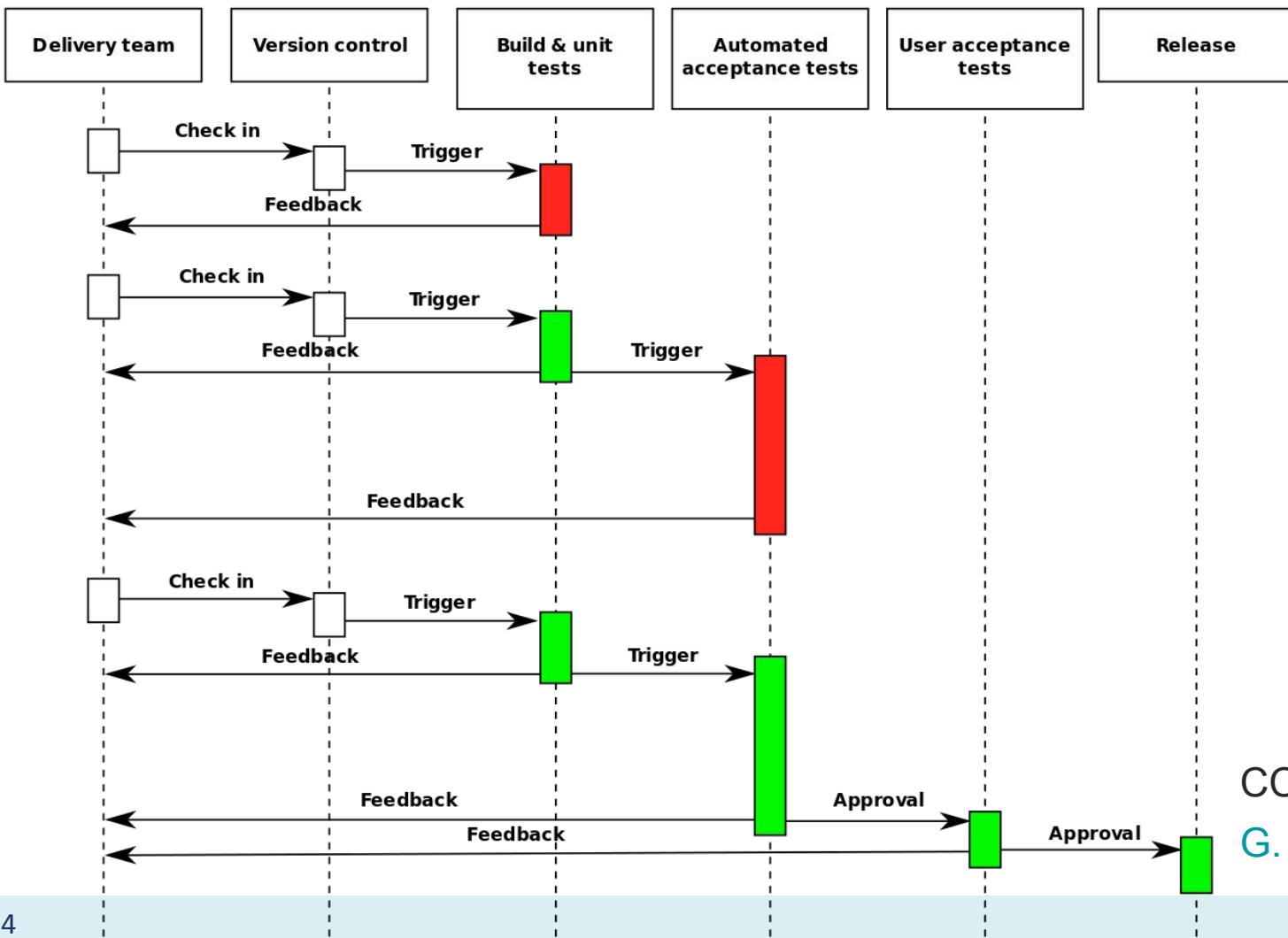
- Traditional View: Boxed Software
  - Working toward fixed release date, QA heavy before release
  - Release and move on
  - Fix post-release defects in next release or through expensive patches
- Frequent releases
  - Incremental updates delivered frequently (weeks, days, ...), e.g. Browsers
  - Automated updates (“patch culture”; “update done? ship it”)
- Hosted software
  - Frequent incremental releases, hot patches, different versions for different customers, customer may not even notice update

## Continuous Delivery



## Continuous Deployment





CC BY-SA 4.0  
G. Détrez

# The Shifting Development-Operations Barrier

# WORKED FINE IN DEV

# OPS PROBLEM NOW

memegenerator.net

# Common Release Problems?

# Common Release Problems (Examples)

- Missing dependencies
- Different compiler versions or library versions
- Different local utilities (e.g. unix grep vs mac grep)
- Database problems
- OS differences
- Too slow in real settings
- Difficult to roll back changes
- Source from many different repositories
- Obscure hardware? Cloud? Enough memory?

# The Dev – Ops Divide

- Coding
  - Testing, static analysis, reviews
  - Continuous integration
  - Bug tracking
  - Running local tests and scalability experiments
  - ...
- Allocating hardware resources
  - Managing OS updates
  - Monitoring performance
  - Monitoring crashes
  - Managing load spikes, ...
  - Tuning database performance
  - Running distributed at scale
  - Rolling back releases
  - ...

QA responsibilities in both roles

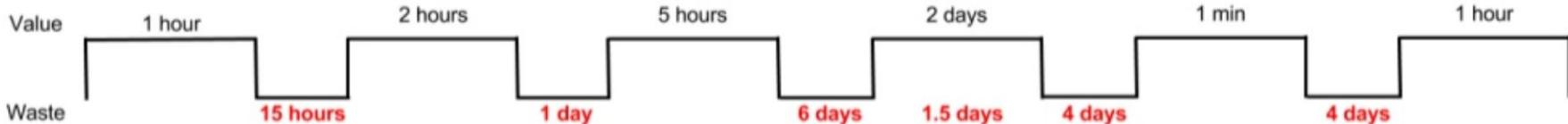
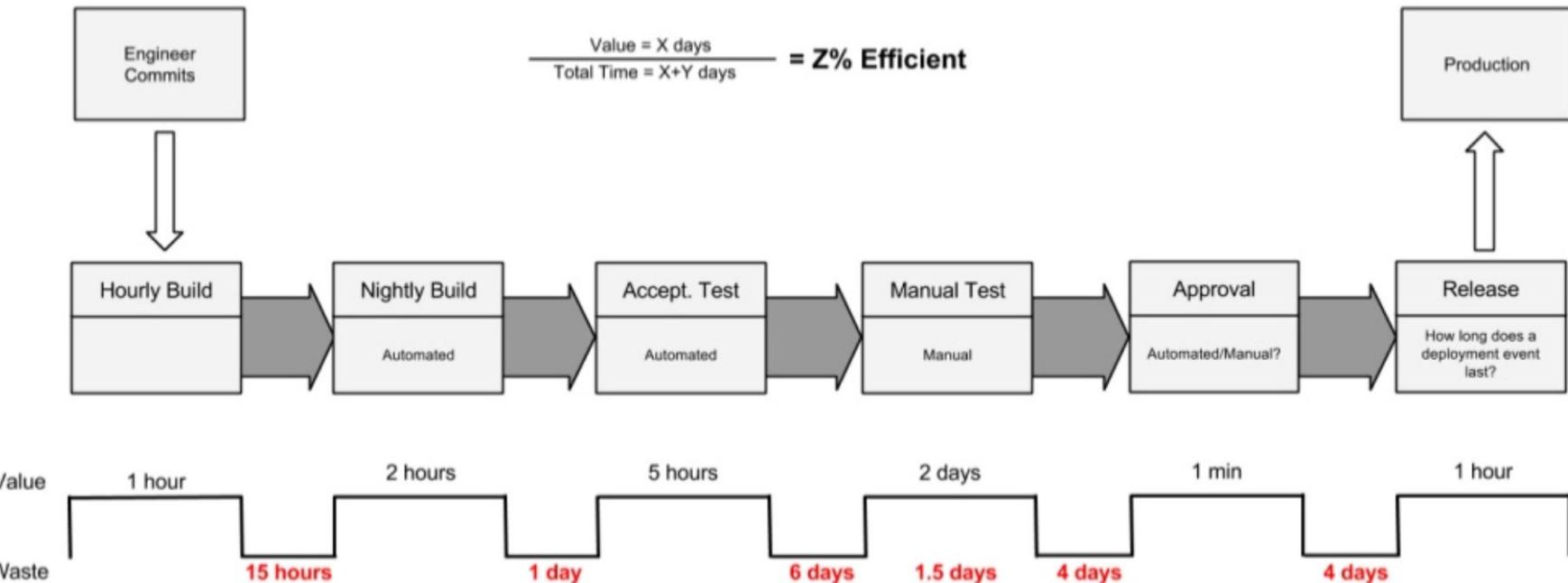
# QA Does not Stop in Dev

# QA Does not Stop in Dev

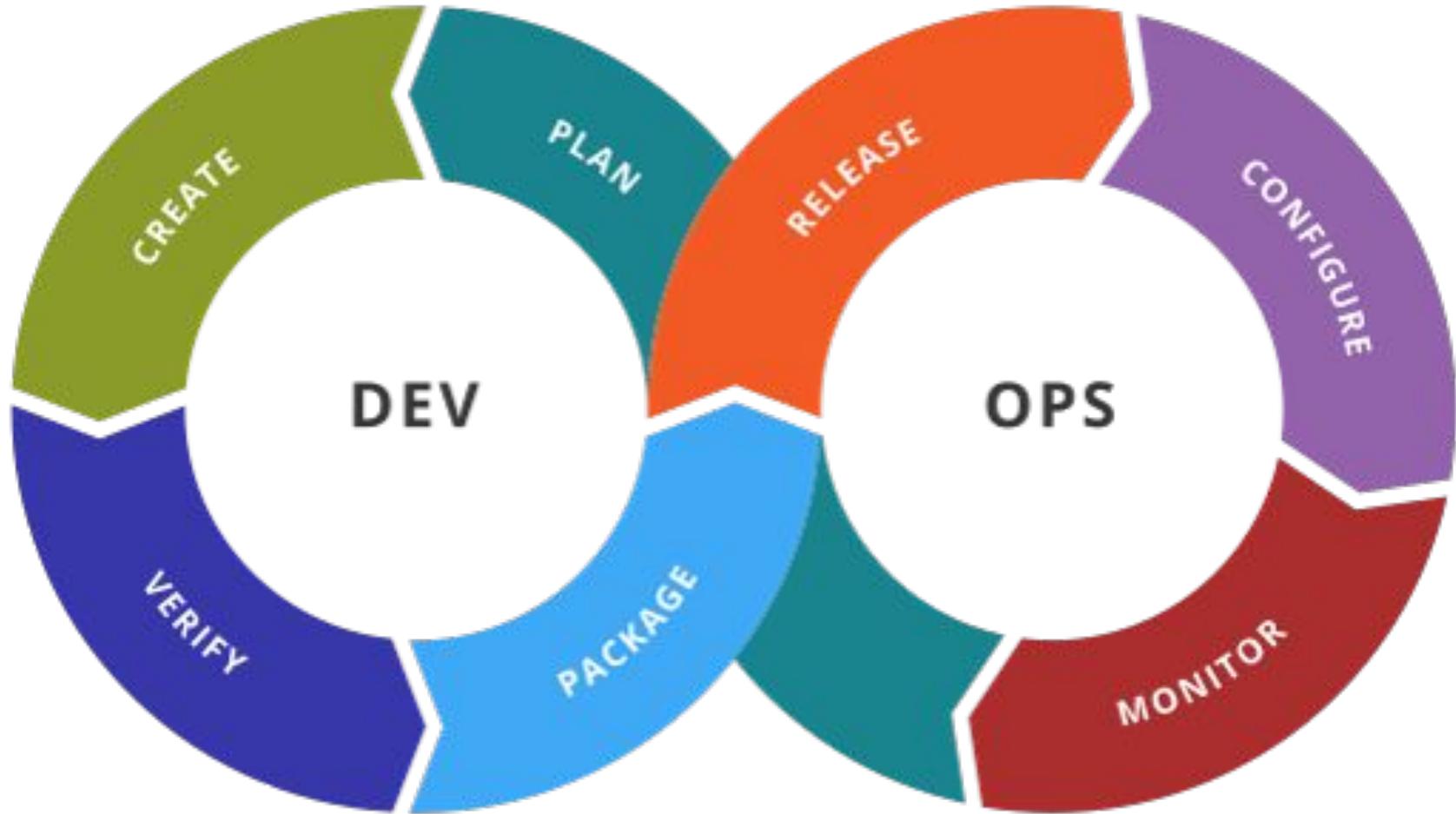
- Ensuring product builds correctly (e.g., reproducible builds)
- Ensuring scalability under real-world loads
- Supporting environment constraints from real systems (hardware, software, OS)
- Efficiency with given infrastructure
- Monitoring (server, database, Dr. Watson, etc)
- Bottlenecks, crash-prone components, ... (possibly thousands of crash reports per day/minute)

# Efficiency of release pipeline

Clip slide



# DevOps



# Key Ideas and Principles

Better coordinate between developers and operations (collaborative)

Key goal: Reduce friction bringing changes from development into production

Considering the entire tool chain into production (holistic)

Documentation and versioning of all dependencies and configurations  
("configuration as code")

Heavy automation, e.g., continuous delivery, monitoring

Small iterations, incremental and continuous releases

Buzz word!

# Common practices

- Code: Version control, dependency management, review
- Build: Continuous integration, independent builds
- Test: Automated test execution on every build
- Package: Deploying binary to repository/staging area
- Release: Change management, deployment, rollback of packages
- Configure: Manage and configure infrastructure, automated
- Monitor: Monitor performance, crashes, ... and possibly automated reaction

# Common Practices

All configurations in version control

Test and deploy in containers

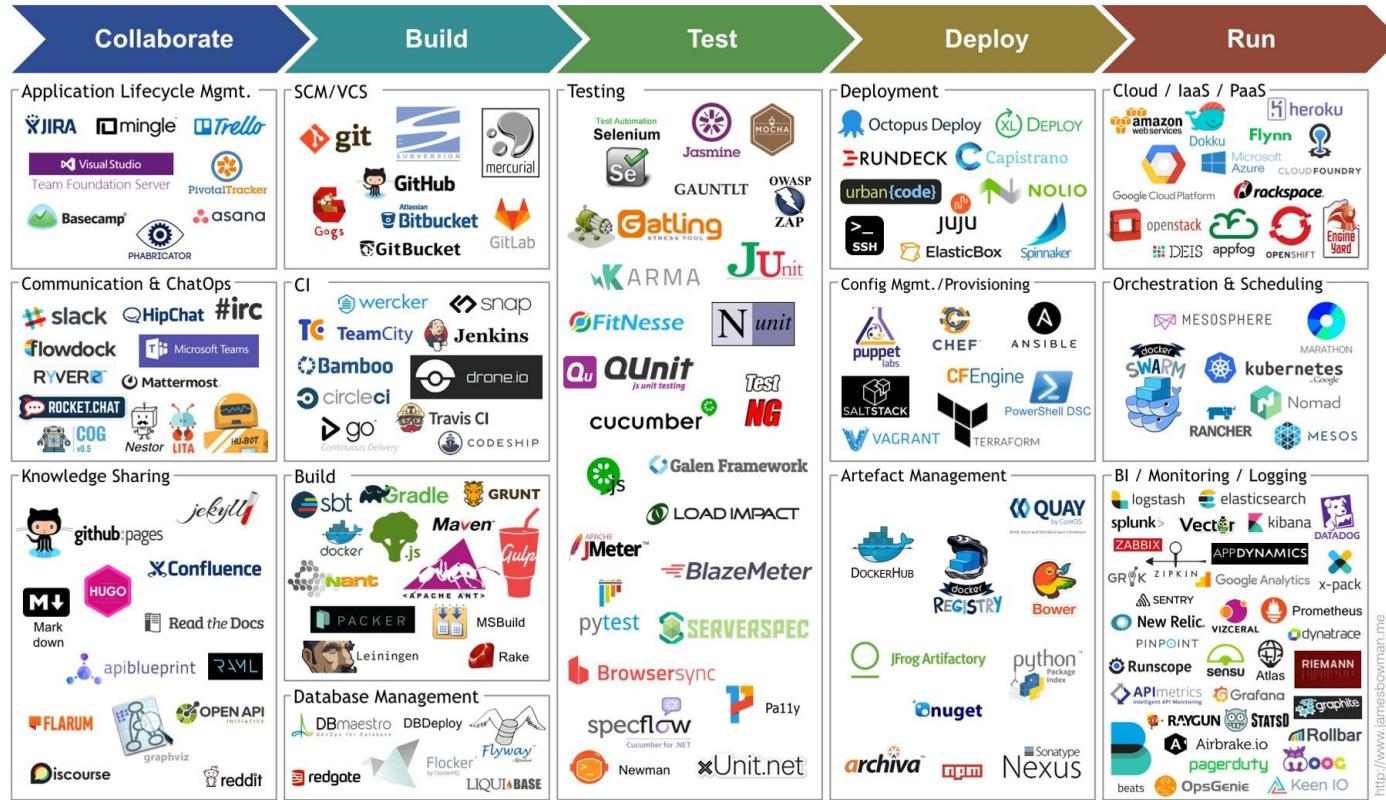
Automated testing, testing, testing, ...

Monitoring, orchestration, and automated actions in practice

Microservice architectures

Release frequently

# Heavy Tooling and Automation



# Heavy tooling and automation -- Examples

Infrastructure as code — Ansible, Terraform, Puppet, Chef

CI/CD — Jenkins, TeamCity, GitLab, Shippable, Bamboo, Azure DevOps

Test automation — Selenium, Cucumber, Apache JMeter

Containerization — Docker, Rocket, Unik

Orchestration — Kubernetes, Swarm, Mesos

Software deployment — Elastic Beanstalk, Octopus, Vamp

Measurement — Datadog, DynaTrace, Kibana, NewRelic, ServiceNow

# DevOps: Tooling Overview

# DevOps Tools

- Containers and virtual machines (Docker, ...)
- Orchestration and configuration (ansible, Puppet, Chef, Kubernetes, ...)
- Sophisticated (custom) pipelines



- Lightweight virtualization
- Sub-second boot time
- Sharable virtual images with full setup incl. configuration settings
- Used in development and deployment
- Separate docker images for separate services (web server, business logic, database, ...)

# Configuration management, Infrastructure as Code

- Scripts to change system configurations (configuration files, install packages, versions, ...); declarative vs imperative
- Usually put under version control

```
- hosts: all
 sudo: yes
 tasks:
 - apt: name={{ item }}
 with_items:
 - ldap-auth-client
 - nscd
 - shell: auth-client-config -t nss -p lac_ldap
 - copy: src=ldap/my_mkhomedir dest=...
 - copy: src=ldap/ldap.conf dest=/etc/ldap.conf
 - shell: pam-auth-update --package
 - shell: /etc/init.d/nscd restart
```

(ansible)

```
$nameservers = ['10.0.2.3']
file { '/etc/resolv.conf':
 ensure => file,
 owner => 'root',
 group => 'root',
 mode => '0644',
 content => template('resolver/r.conf'),
}
```

(Puppet)

# Container Orchestration with Kubernetes

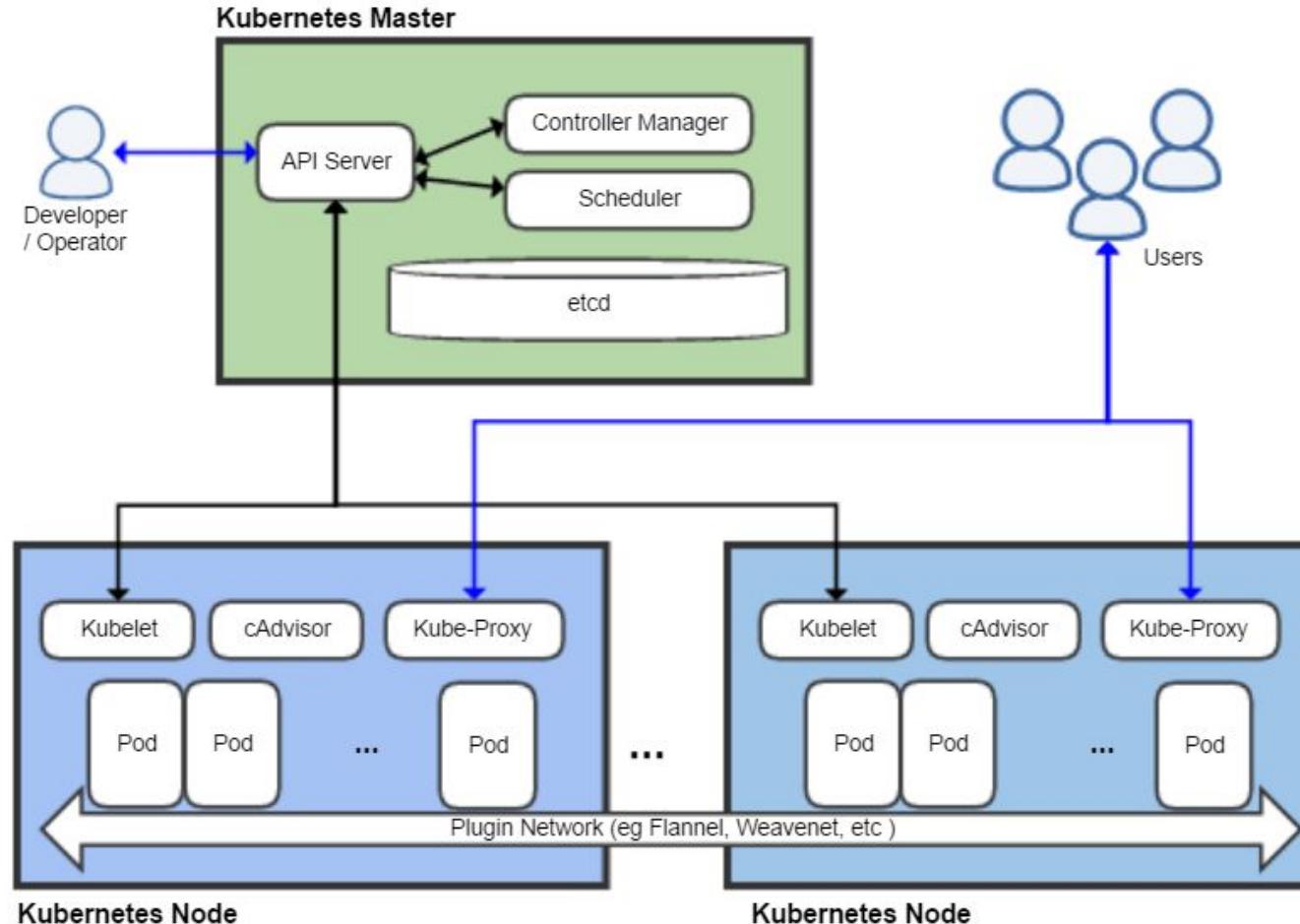
Manages which container to deploy to which machine

Launches and kills containers depending on load

Manage updates and routing

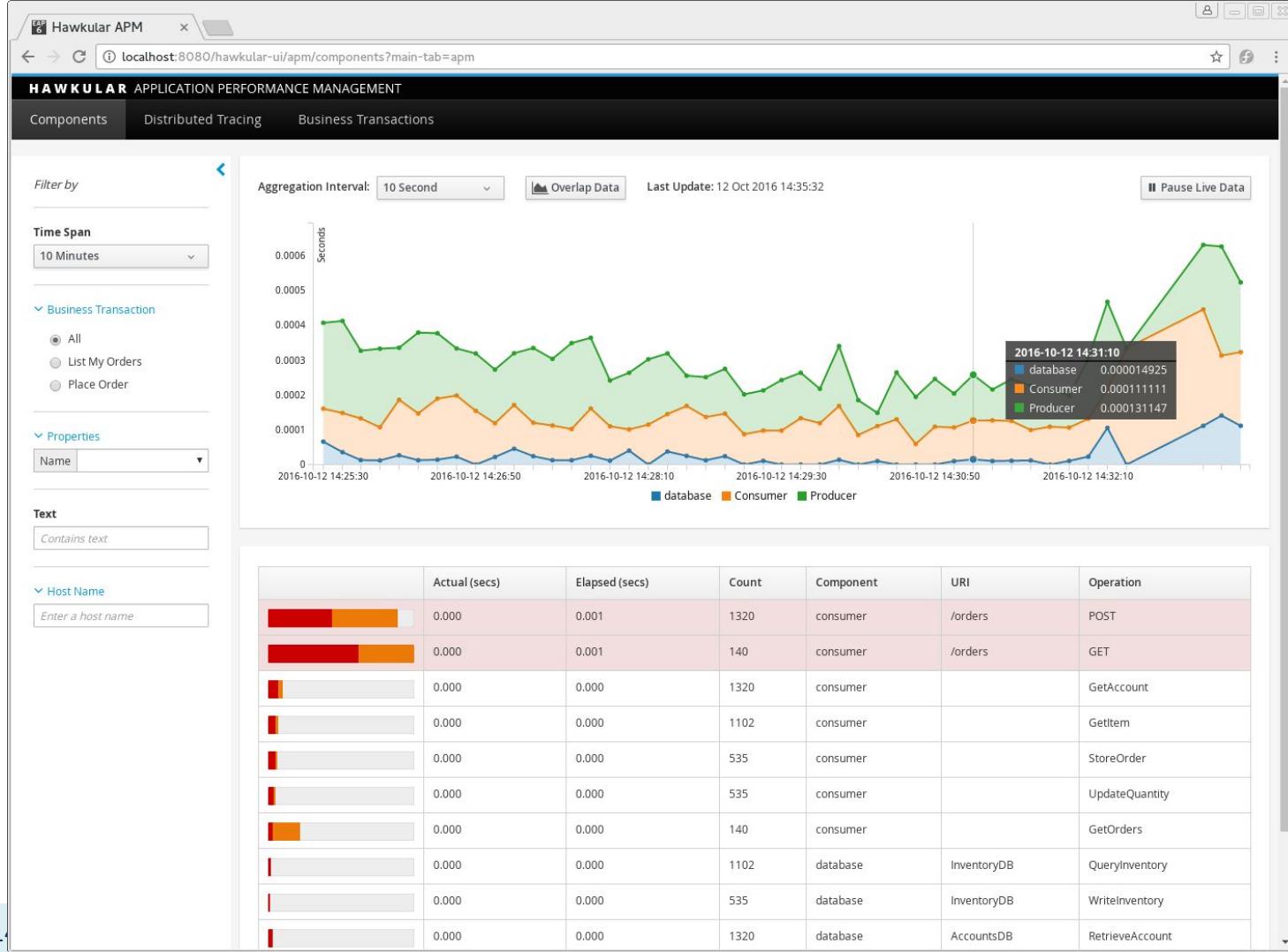
Automated restart, replacement, replication, scaling

Kubernetes master controls many nodes



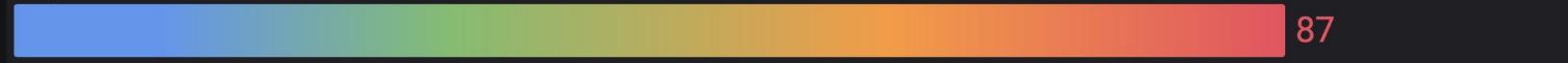
# Monitoring

- Monitor server health
- Monitor service health
- Collect and analyze measures or log files
- Dashboards and triggering automated decisions
  - Many tools, e.g., Grafana as dashboard, Prometheus for metrics, Loki + ElasticSearch for logs
  - Push and pull models





Temperature



A-series



25

B-series



97

E-series



4

C-series



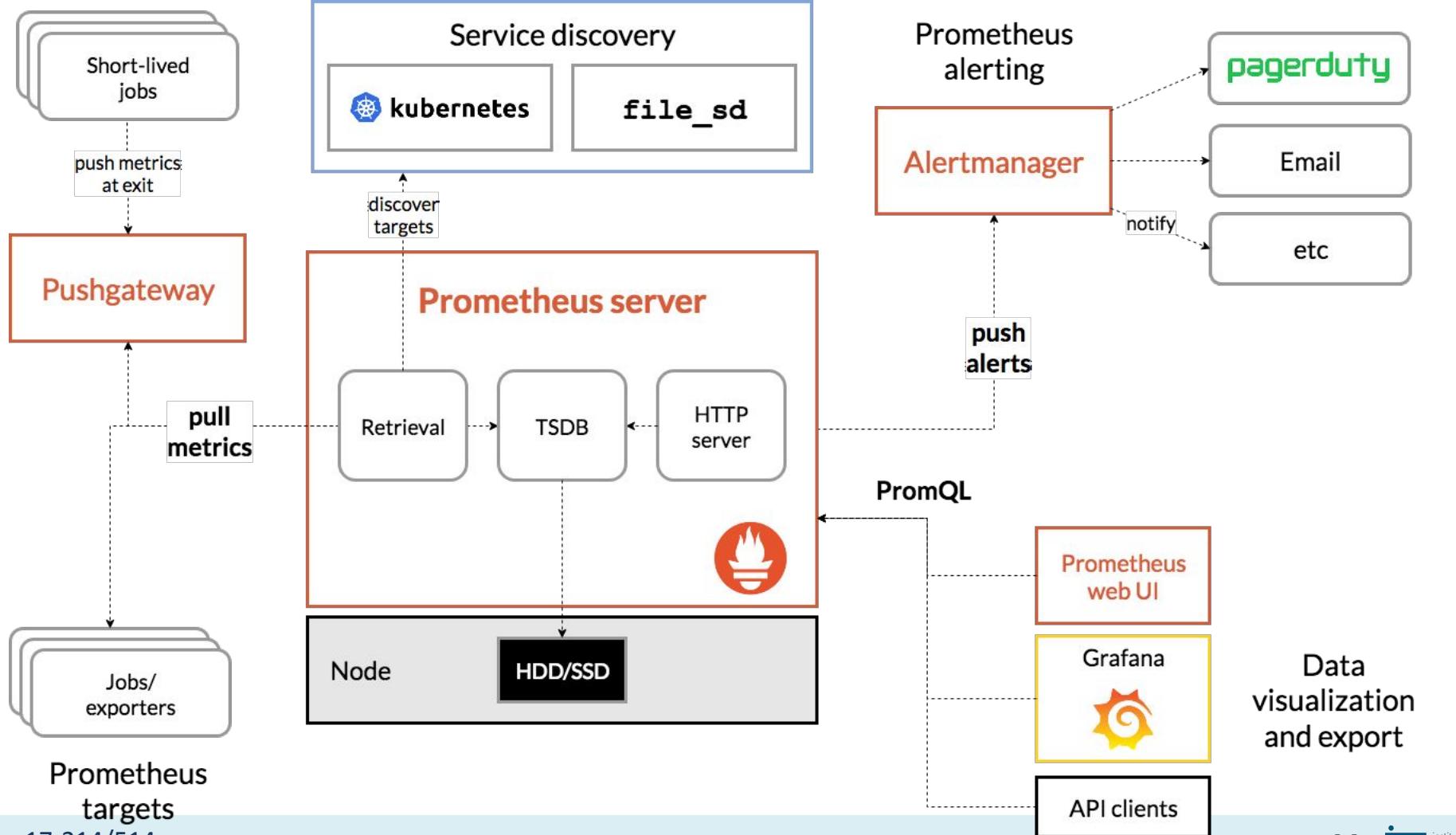
77

### Text panel heading

For markdown syntax help: [commonmark.org/help](http://commonmark.org/help)

# Grafana





# Testing in Production

# Testing in Production



Changelog  
@changelog



"Don't worry, our users will notify us if there's a problem"



10:03 AM · Jun 8, 2019



2.2K



12



Share this Tweet

[Tweet your reply](#)

# Chaos Experiments





Microsoft  
**Windows 95**  
Final Beta Release



# Crash Telemetry



# A/B Testing

Original: 2.3%



The original landing page for Groove features a large image of a smiling man in a plaid shirt. The headline reads "SaaS & eCommerce Customer Support." Below it, a quote from "Griffin, Customer Champion at Allocacoo" says, "Managing customer support requests in Groove is so easy. Way better than trying to use Gmail or a more complicated help desk." A testimonial bar below the quote states, "97% of customers recommend Groove." At the bottom, there's a green "Learn More" button and four navigation links: "How it works", "What you get", "What it costs", and "How we're different".

You'll be up and running in less than a minute.

Long Form: 4.3%



The long-form landing page for Groove has a larger headline: "Everything you need to deliver awesome, personal support to every customer." It includes a testimonial from Allan: "Assign support emails to the right people, feel confident that customers are being followed up with and always know what's going on." Below the testimonial is a video player showing Allan speaking. To the right, there's a sidebar with a "WHAT YOU'LL DISCOVER ON THIS PAGE" section containing five bullet points: "Three reasons pricing teams choose Groove", "How Groove makes your whole team more productive", "Delivering a personal support experience every time", "Take a screencast tour", and "A personal note from our CEO". The footer includes logos for BuySellAds, iStatusPage, and MetalAd.

# WHAT IF...?

... we had plenty of subjects for experiments

... we could randomly assign subjects to treatment and control group without them knowing

... we could analyze small individual changes and keep everything else constant

- ▶ Ideal conditions for controlled experiments

# Experiment Size

With enough subjects (users), we can run many many experiments

Even very small experiments become feasible

Toward causal inference



# IMPLEMENTING A/B TESTING

Implement alternative versions of the system

- using feature flags (decisions in implementation)
- separate deployments (decision in router/load balancer)

Map users to treatment group

- Randomly from distribution
- Static user - group mapping
- Online service (e.g., [launchdarkly](#), [split](#))

Monitor outcomes per group

- Telemetry, sales, time on site, server load, crash rate

# FEATURE FLAGS

Boolean options

Good practices: tracked explicitly, documented, keep them localized and independent

External mapping of flags to customers

- who should see what configuration
- e.g., 1% of users sees one\_click\_checkout, but always the same users; or 50% of beta-users and 90% of developers and 0.1% of all users

```
if (features.enabled(userId, "one_click_checkout")) {
 // new one click checkout function
} else {
 // old checkout functionality
}
```

```
def isEnabled(user): Boolean = (hash(user.id) % 100) < 10
```

▼ Treatments ⓘ | 2 treatments, if Split is killed serve the default treatment of "off"

| Treatment | Default | Description                                         |
|-----------|---------|-----------------------------------------------------|
| on        |         | The new version of registration process is enabled. |
| off       |         | The old version of registration process is enabled. |

[+ Add treatment](#) | Learn more about multivariate treatments.

▼ Whitelist ⓘ | 0 user(s) or segments individually targeted.

[+ Add whitelist](#)

▼ Traffic Allocation ⓘ | 100% of user included in Split rules evaluation below.

Total Traffic Allocation: 100 % total User in Split

▼ Targeting Rules ⓘ | 2 rules created for targeting.

```

graph TD
 R1[if user is in segment qa] --> S1[Then serve on]
 R2[else if user is in segment beta_testers] --> S2[Then serve percentage]
 S2 --> P1[50 on]
 S2 --> P2[50 off]

```

[+ Add rule](#)

▼ Default Rule ⓘ | Serve treatment of "off".

serve off

# Comparing Outcomes

## Group A

base game

2158 Users

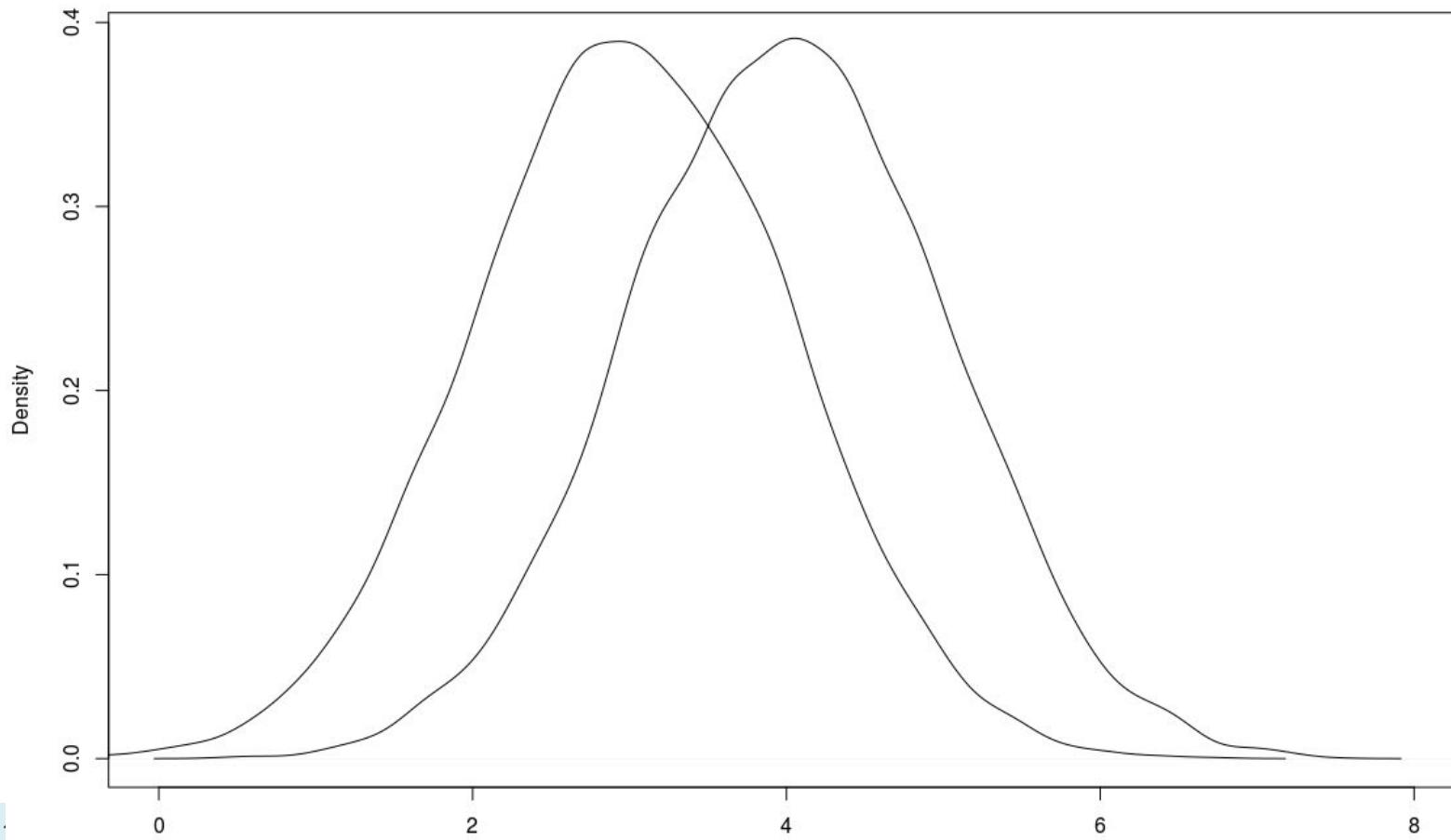
average 18:13 min time  
on site

## Group B

game with extra god  
cards

10 Users

average 20:24 min time  
on site



# Experiment Created

[Edit](#)[Remove](#)[Delete](#)

The percentage of visitors who clicked on a tracked element.

✓ Test it out is beating Original Page by +25.4%.

| Variations    | Statistics             |                       |                                                                                   |                         |             |
|---------------|------------------------|-----------------------|-----------------------------------------------------------------------------------|-------------------------|-------------|
| Experiment    | Conversions / Visitors | Conversion Rate       | Baseline                                                                          | Chance to beat Baseline | Improvement |
| Test it out   | 462 / 3,568            | 12.9% ( $\pm 1.1\%$ ) |  | ✓ 100.0%                | +25.4%      |
| Give it a try | 440 / 3,479            | 12.6% ( $\pm 1.1\%$ ) |  | ✓ 99.9%                 | +22.5%      |
| Try it out    | 395 / 3,504            | 11.3% ( $\pm 1.0\%$ ) |  | 90.2%                   | +9.2%       |
| Original Page | 378 / 3,662            | 10.3% ( $\pm 1.0\%$ ) |  | ✓                       | ---         |



# The Morality Of A/B Testing

Josh Constine @joshconstine / 11:50 PM EDT • June 29, 2014

 Comment



We don't use the "real" Facebook. Or Twitter. Or Google, Yahoo, or LinkedIn. We are almost all part of experiments they quietly run to see if different versions with little changes make us use more, visit more, click more, or buy more. By signing up for these services, we technically give consent to be treated like guinea pigs.

But this weekend, Facebook stirred up [controversy](#) because one of its data science researchers published the results of an experiment on 689,003 users to see if showing them more positive or negative sentiment posts in the News Feed would affect their happiness levels as deduced by what they posted. The impact of this experiment on manipulating emotions was tiny, but it

# Canary Releases



foto Javier Baño

# Canary Releases

Testing releases in production

Incrementally deploy a new release to users, not all at once

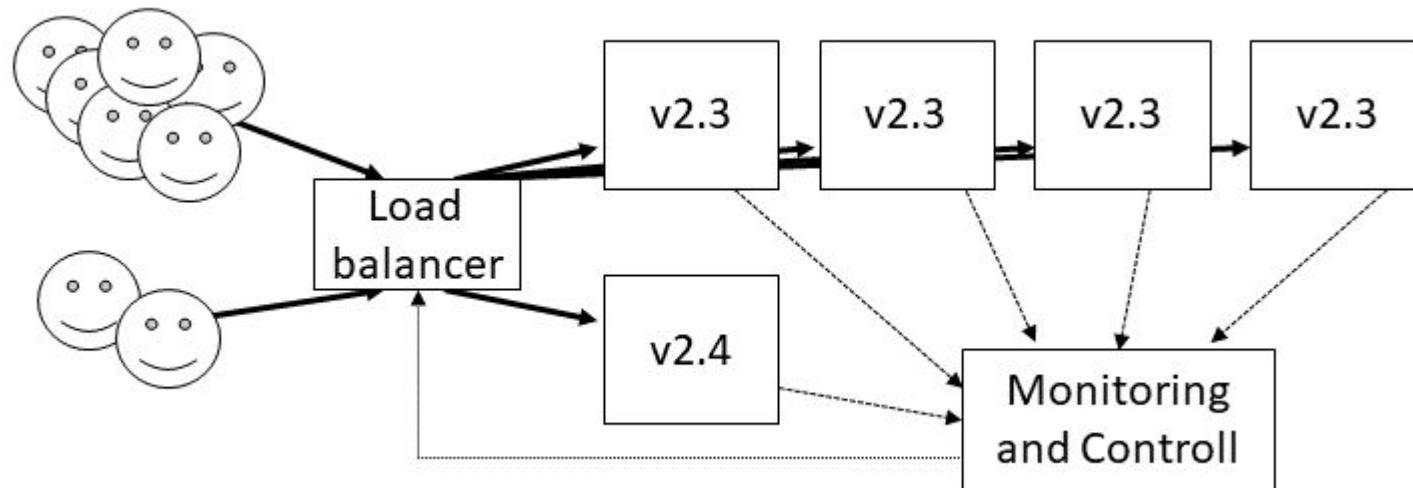
Monitor difference in outcomes (e.g., crash rates, performance, user engagement)

Automatically roll back bad releases

Technically similar to A/B testing

Telemetry essential

# Canary Releases



# Canary Releases at Facebook

Phase 0: Automated unit tests

Phase 1: Release to Facebook employees

Phase 2: Release to subset of production machines

Phase 3: Release to full cluster

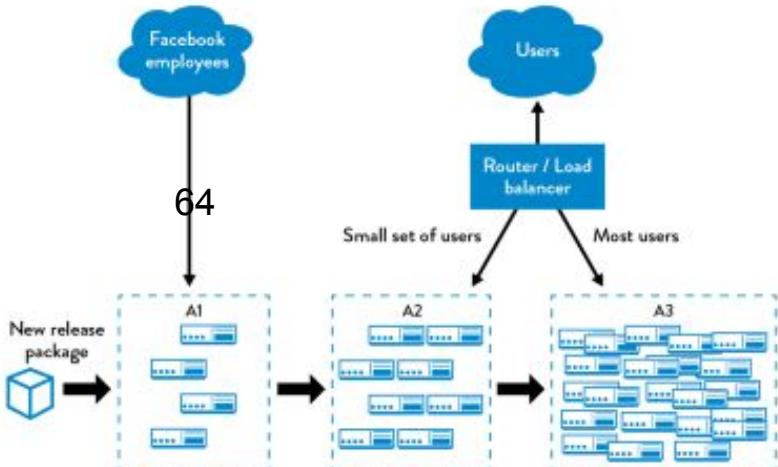
Phase 4: Commit to master, rollout everywhere

Monitored metrics: server load, crashes, click-through rate

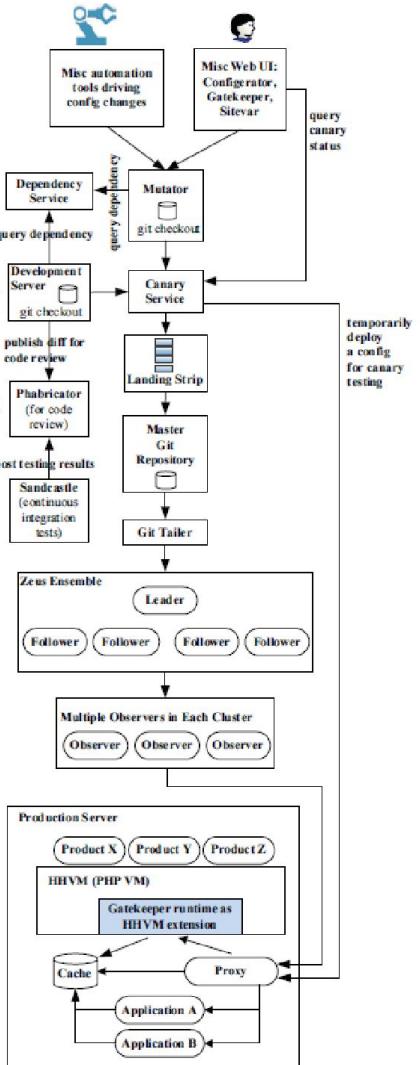
Further readings: Tang, Chunqiang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. [Holistic configuration management at Facebook](#). In Proceedings of the 25th Symposium on Operating Systems Principles, pp. 328-343. ACM, 2015. and Rossi, Chuck, Elisa Shibley, Shi Su, Kent Beck, Tony Savor, and Michael Stumm. [Continuous deployment of mobile software at facebook \(showcase\)](#). In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 12-23. ACM, 2016.

# Real DevOps Pipelines are Complex

- Incremental rollout, reconfiguring routers
- Canary testing
- Automatic rolling back changes



Chunqiang Tang,  
Thawan Kooburat,  
Pradeep  
Venkatachalam, Akshay  
Chander, Zhe Wen,  
Aravind Narayanan,  
Patrick Dowell, and  
Robert Karl. [Holistic Configuration Management at Facebook](#). Proc. of SOSP: 328–343 (2015).



# Chaos Experiments



# Two more things

# TAing in Spring 2022?

Enjoyed content of this class?

Practicing critiquing other designs?

Thinking through design problems with other students?

If interested, talk to us or apply directly at

<https://www.ugrad.cs.cmu.edu/ta/S22/> (select 17214)

Course feedback  
please:

<https://bit.ly/214testing>



# Summary

Increasing automation of tests and deployments

Containers and configuration management tools help with automation, deployment, and rollbacks

Monitoring becomes important

Many new opportunities for testing in production (feature flags are common)