Principles of Software Construction: Objects, Design, and Concurrency

Distributed Systems – Events Everywhere!

Claire Le Goues

Vincent Hellendoorn

Administrative

- HW5a due next Monday
  - Start on part b as soon as you can

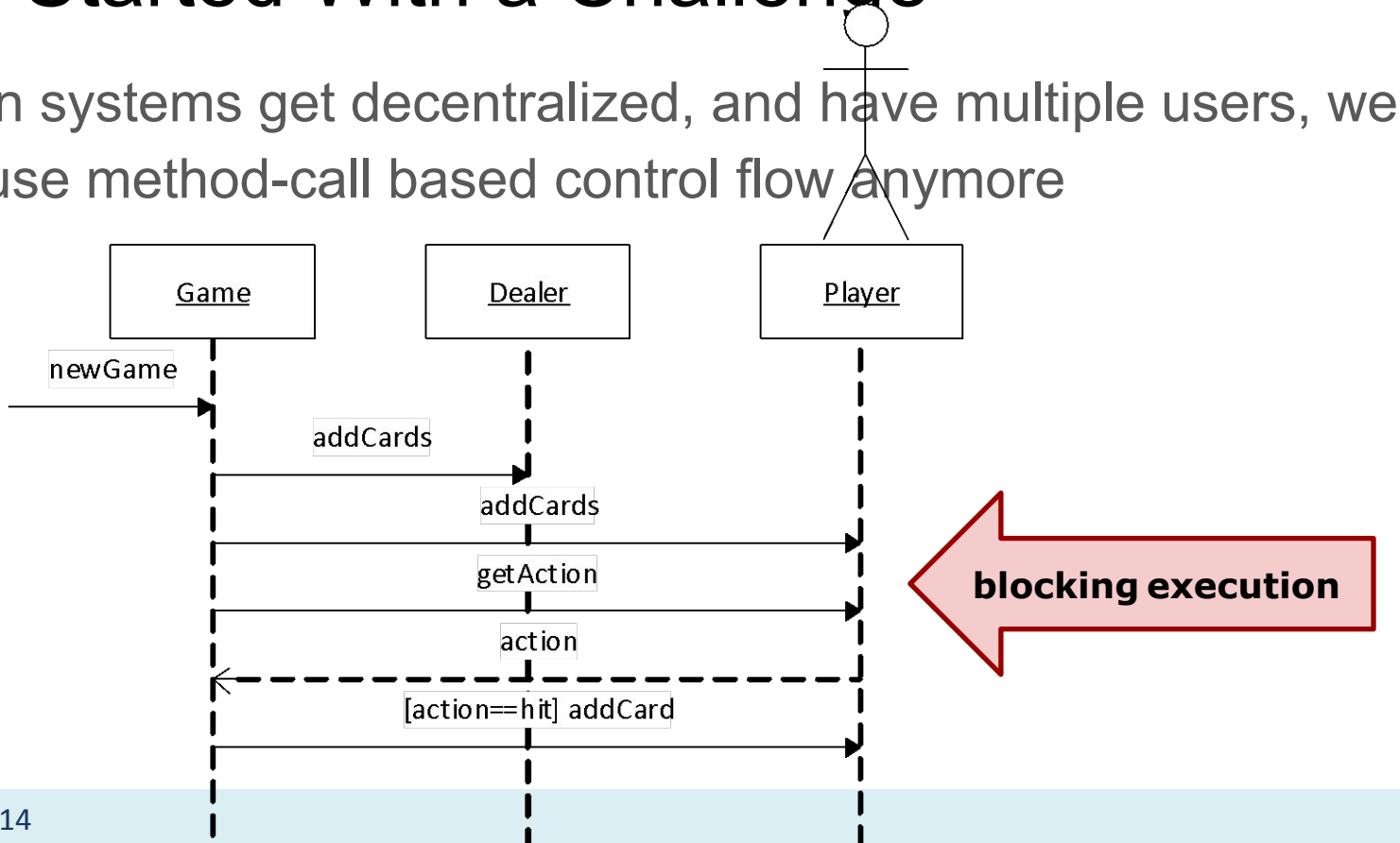
Outline

- Recapping Concurrency
- Distributed Systems
  - Revisiting Event-Based Programming
- Reactive Programming
  - If time permits

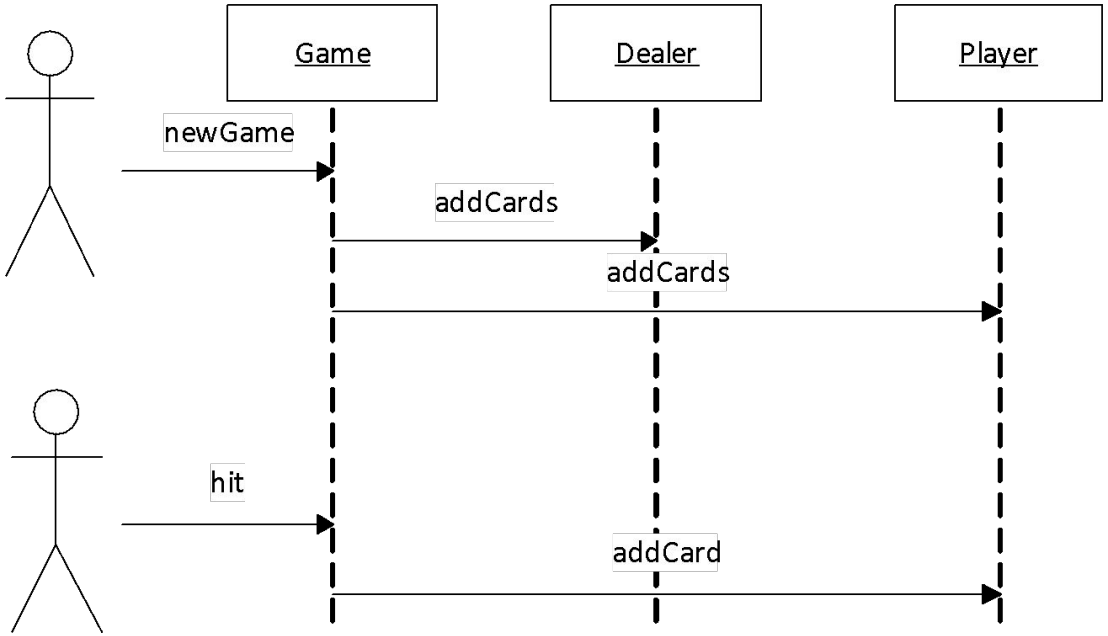
Recapping Concurrency

We Started With a Challenge

When systems get decentralized, and have multiple users, we can't just use method-call based control flow anymore

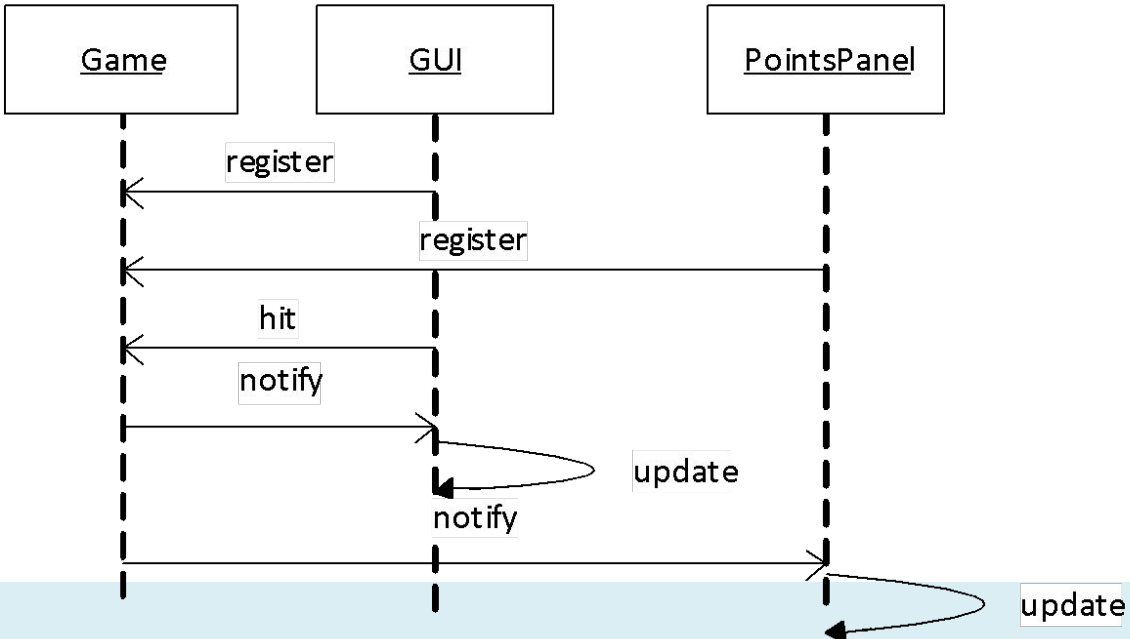


Instead, We Use Event-Based Programming



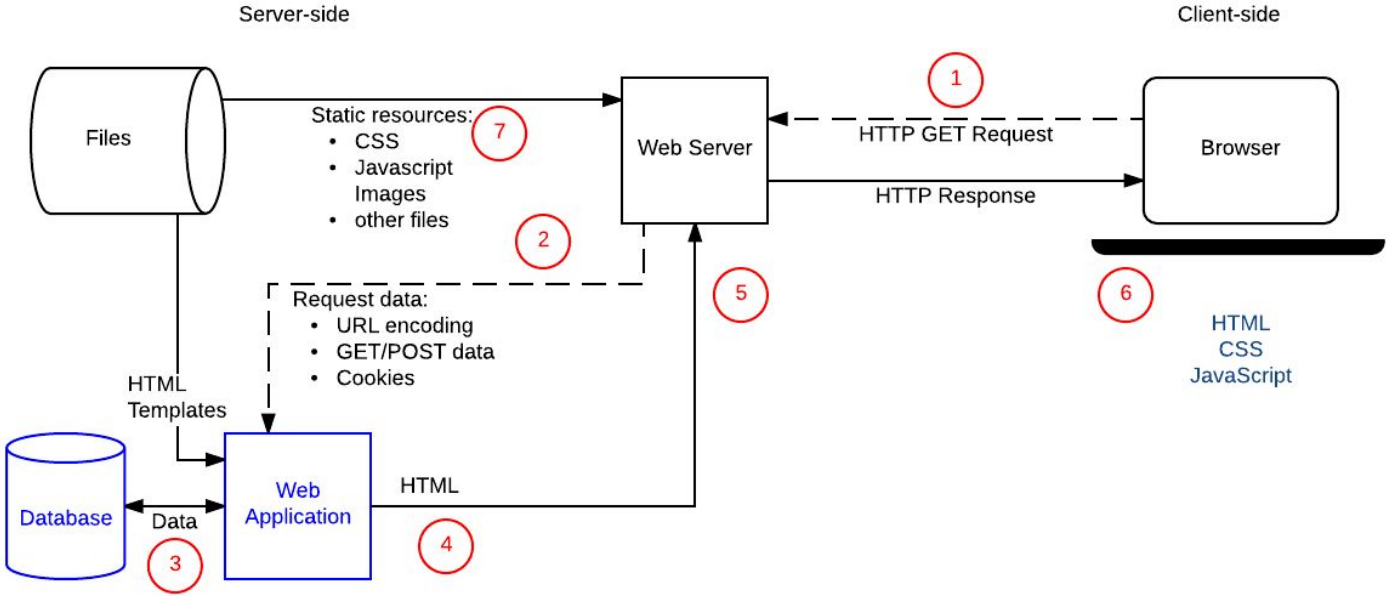
We Separated Out the GUI from the Game

And realized that we needed to *decouple* them to keep the design clean and extensible. Hence the observer pattern



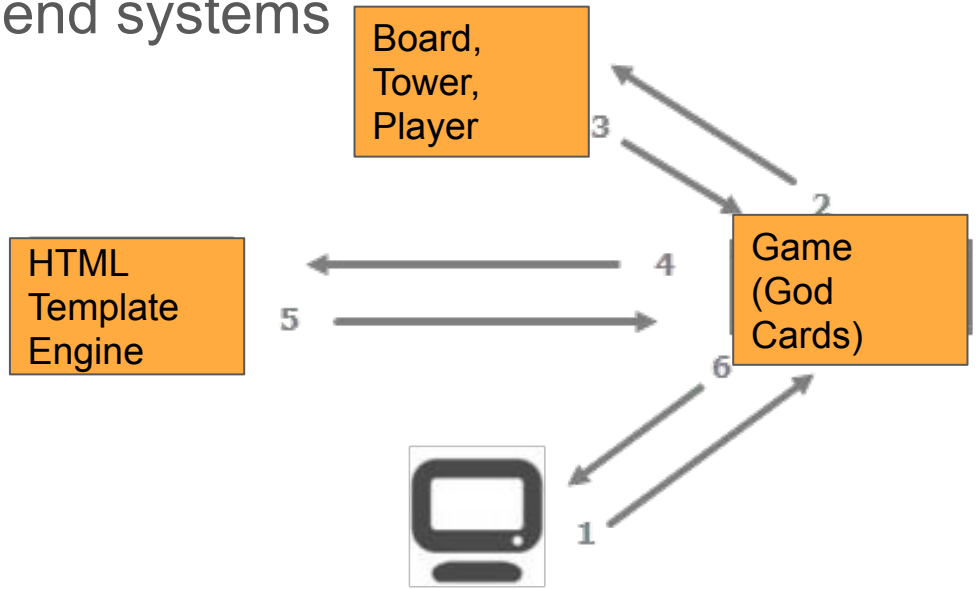
Web-Apps Decouple the Client & Server

The web server can do more work, provide security



First Architectural Pattern: Model-View-Controller

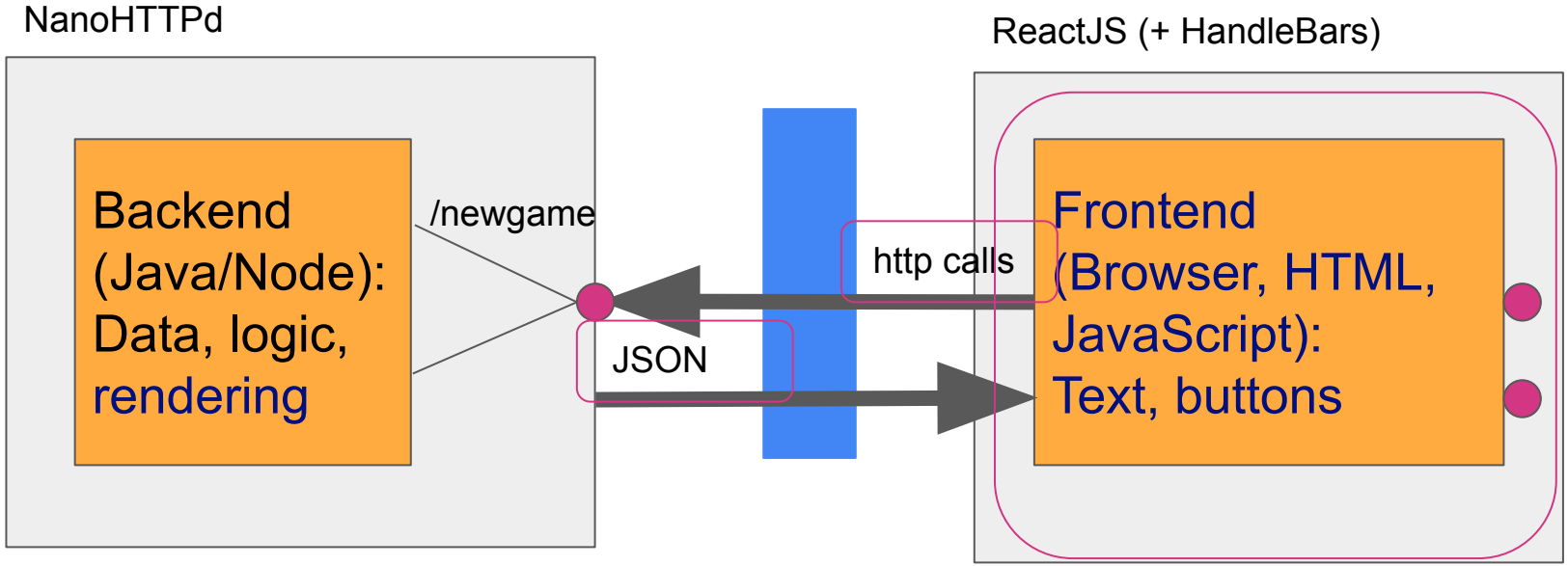
Provides a high-level strategy for designing large, front-end/back-end systems



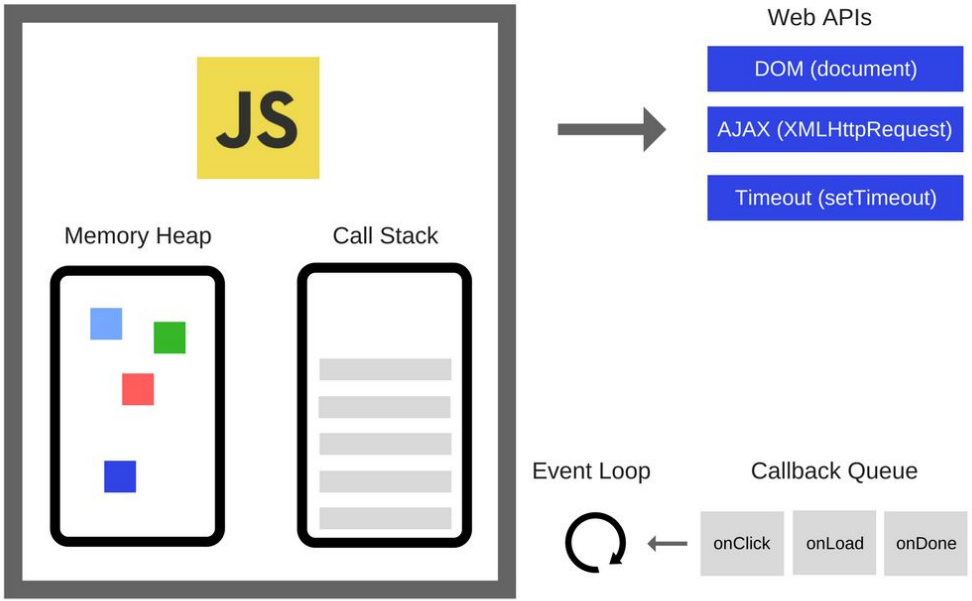
https://overiq.com/django-1-10/mvc-pattern-and-django/

We Studied An Example Setup in Recitation

And realized that we have to start thinking about *concurrency*, to handle asynchronous communication



We Talked About a Relative Simple Case: The JavaScript Runtime



Engine plus:

- Web APIs — provided by browsers, like the DOM, AJAX, setTimeout and more.
- Event loop
- Callback queue

With A Special Discussion on Callbacks

By far the most common way to express and manage asynchronicity in JavaScript programs.

```
function task(message) {
    // emulate time consuming task
    let n = 10000000000;
    while (n > 0){
        n--;
    }
    console.log(message);
}

console.log('Start script...');
setTimeout(() => {
    task('Download a file.');
}, 1000);
console.log('Done!');
```

Remember “Callback Hell” (and how we solved it)?

```
const makeBurger = nextStep => {
  getBeef(function (beef) {
    cookBeef(beef, function (cookedBeef) {
      getBuns(function (buns) {
        putBeefBetweenBuns(buns, beef,
          function(burger) {
            nextStep(burger)
        })
      })
    })
  })
}

// Make and serve the burger
makeBurger(function (burger) => {
  serve(burger)
})
```

```
let bunPromise = getBuns();
let cookedBeefPromise = getBeef()
    .then(beef => cookBeef(beef));
// Resolve both promises in parallel
Promise.all([bunPromise, cookedBeefPromise])
    .then(([buns, beef]) =>
          putBeefBetweenBuns(buns, beef))
    .then(burger => serve(burger))
```

Then We Went Deep Into Concurrency

Focusing mostly on Java

```
public interface Runnable {  // java.lang.Runnable
    public void run();
}

public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);  // Number of threads;

    Runnable greeter = () -> System.out.println("Hi!");
    for (int i = 0; i < n; i++) {
        new Thread(greeter).start();
    }
}
```

Then We Went Deep Into Concurrency

We covered:

- Hazards: Safetly, liveness, performance
- Atomicity, and language features that offer it
- Thread confinement & Immutability
  - Rules for ensuring these properties; design benefits
- Synchronization
  - How to enable it, and the risk of deadlocks
- Keywords and primitives
  - volatile for visibility, wait, notify for guarded suspension

Where Does That Get Us?

Design for

understanding

change/ext.

reuse

robustness

...

Where Does That Get Us?

Design for

understanding

change/ext.

reuse

robustness

...

Small scale:
One/few objects

Subtype Polymorphism ✓

Information Hiding, Contracts ✓

Immutability ✓

Types ✓

Unit Testing ✓

Where Does That Get Us?

Design for	*Small scale:* One/few objects	*Mid scale:* Many objects
understanding	Subtype Polymorphism ✓	Domain Analysis ✓
change/ext. reuse	Information Hiding, Contracts ✓	Inheritance & Del. ✓
robustness	Immutability ✓	Responsibility Assignment, Design Patterns, Antipattern ✓
...	Types ✓	Promises/ Reactive P. ✓
	Unit Testing ✓	Integration Testing ✓

Where Does That Get Us?

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

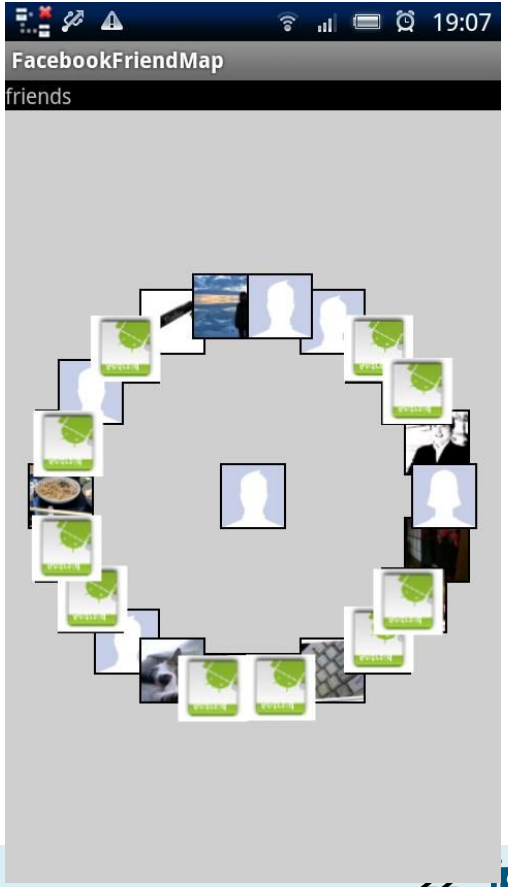
Where Does That Get Us?

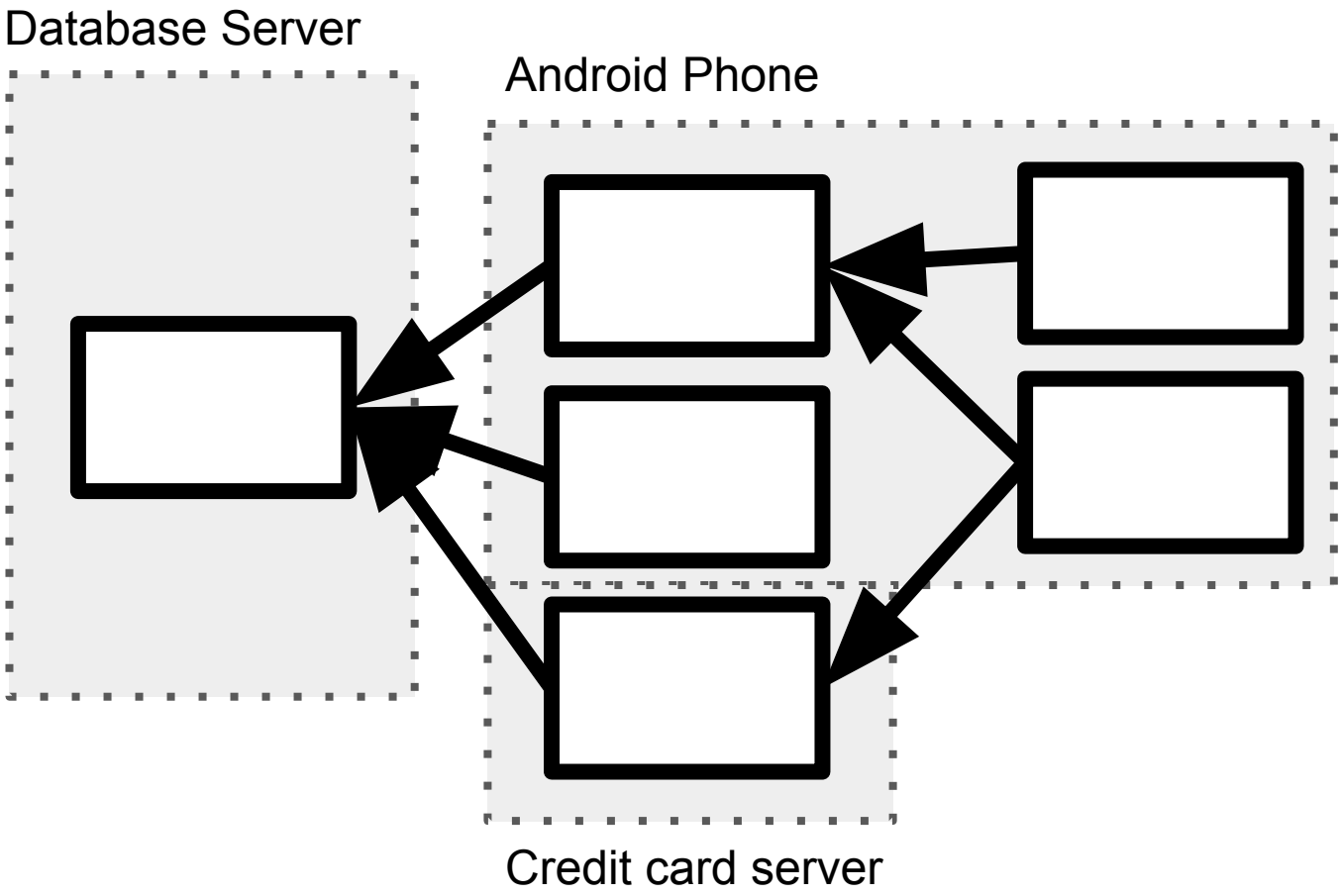
Design for	*Small scale:* One/few objects	*Mid scale:* Many objects	*Large scale:* Subsystems
understanding	Subtype Polymorphism ✓	Domain Analysis ✓	GUI vs Core ✓
change/ext.	Information Hiding, Contracts ✓	Inheritance & Del. ✓	Frameworks and Libraries, APIs
reuse	Immutability ✓	Responsibility Assignment,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

Modern software is dominated by systems composed of [components, APIs, modules], developed by completely different people, communicating over a network!

For example

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




Database Server
Android Phone
Credit card server

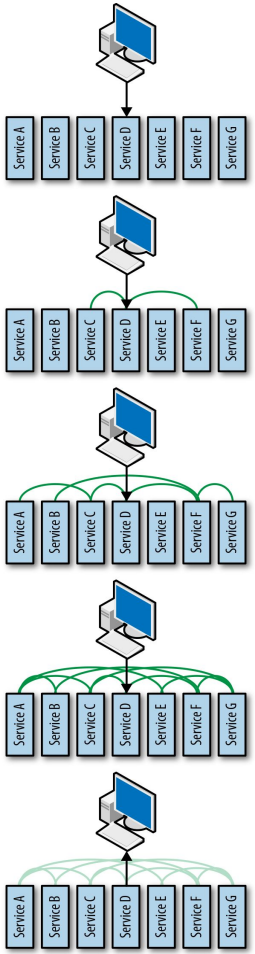
What is a distributed system?

- 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 …
- 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

Drawbacks?

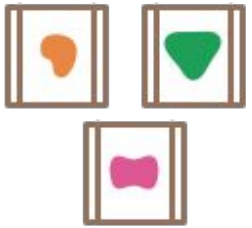
“A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.”

-- Leslie Lamport

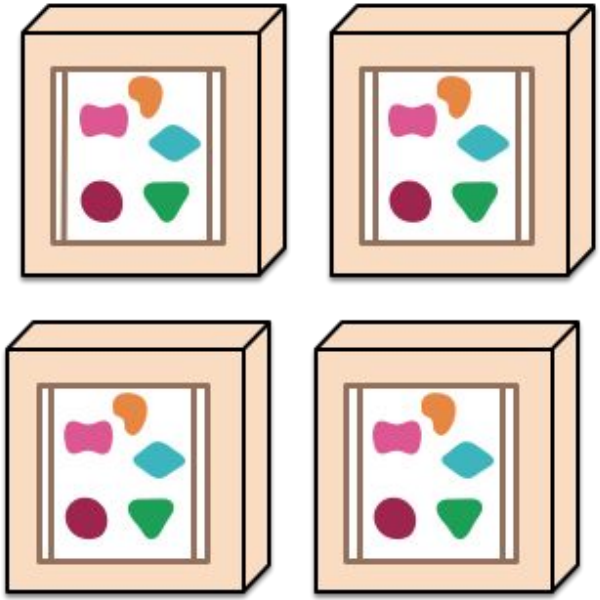

Microservices
Microservices Everywhere

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

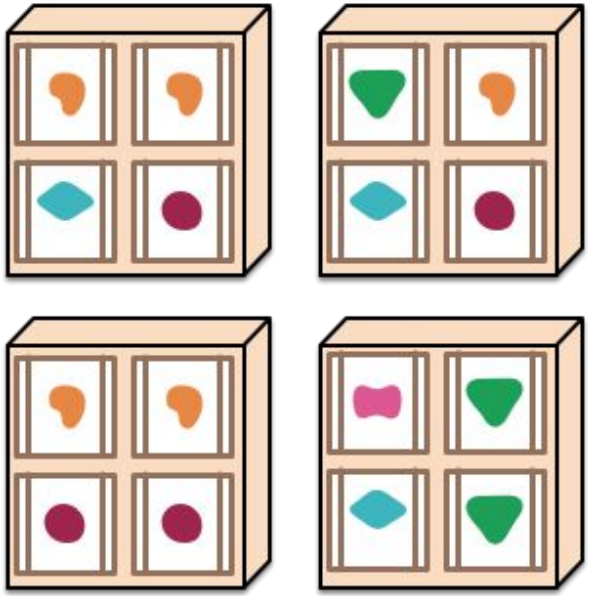
A microservices architecture puts each element of functionality into a separate service...



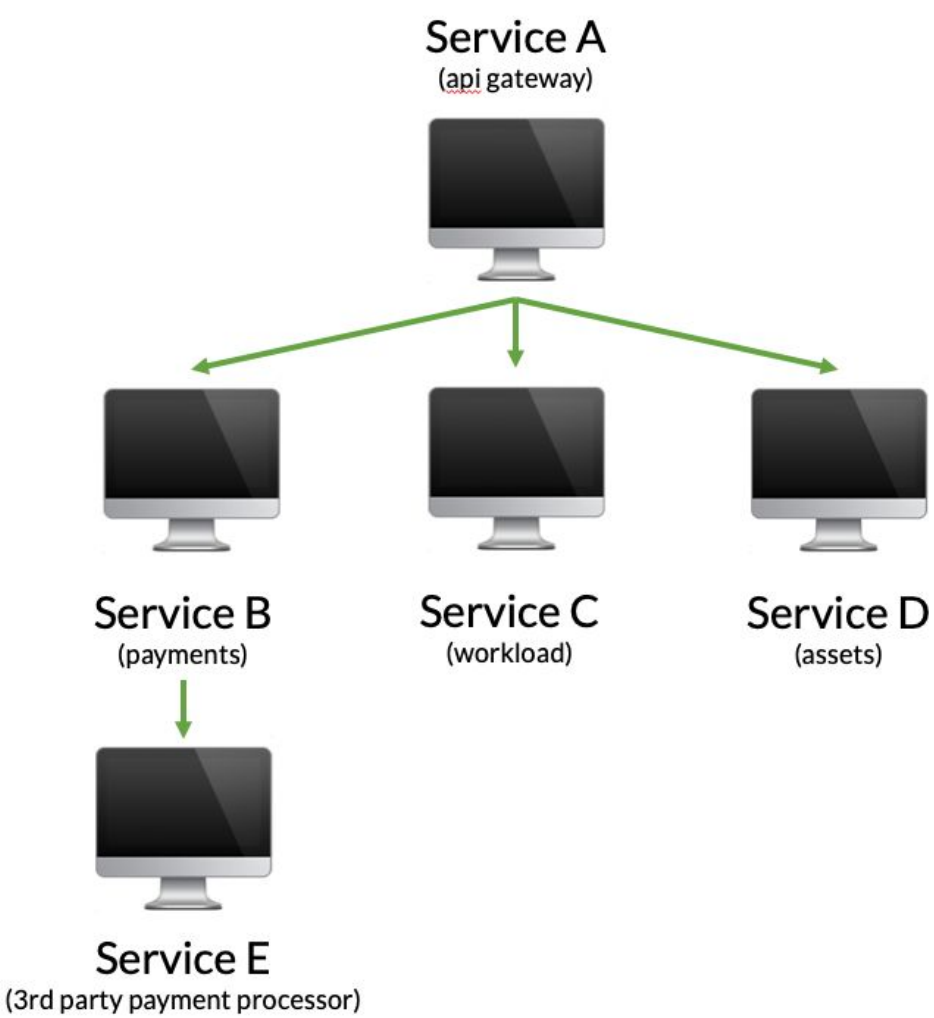
... and scales by replicating the monolith on multiple servers



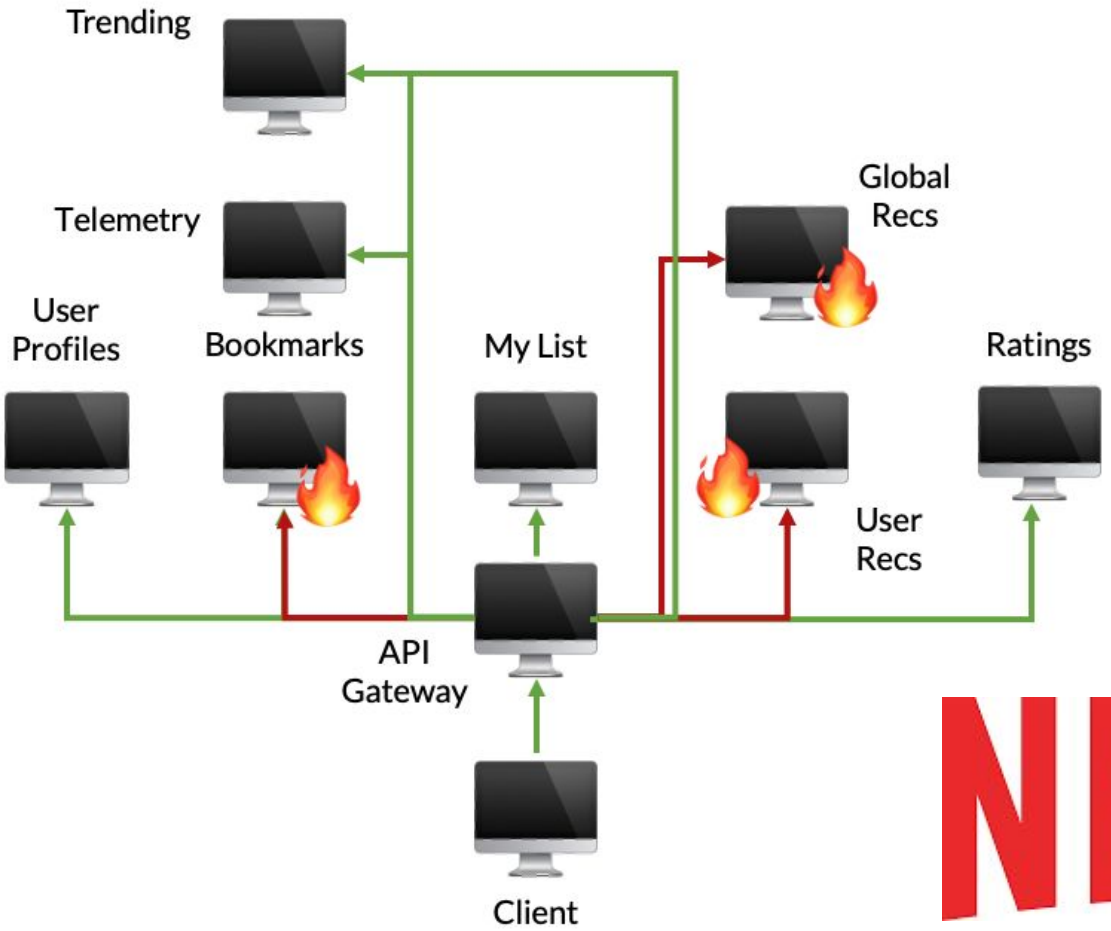
... and scales by distributing these services across servers, replicating as needed.



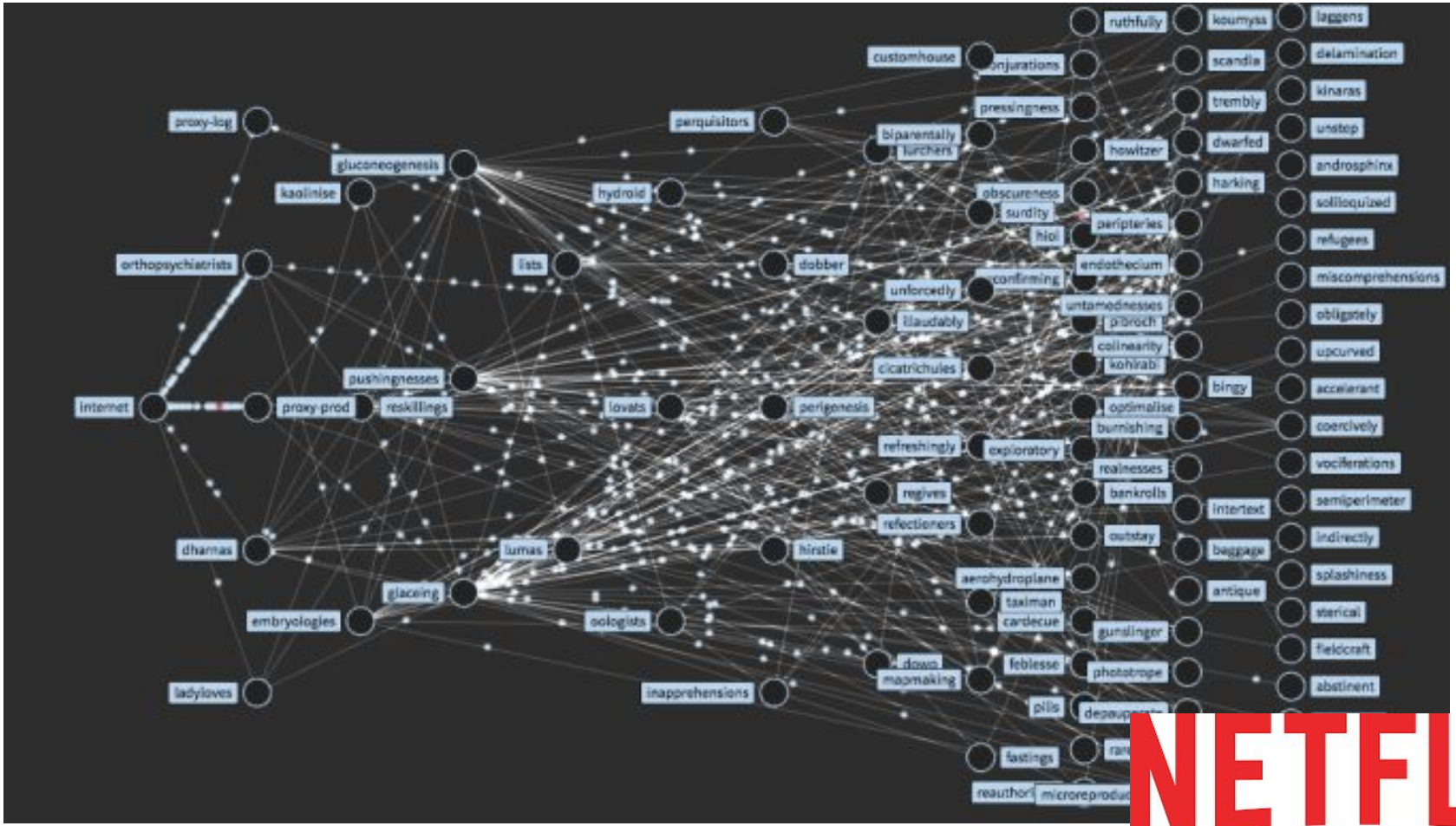
source: http://martinfowler.com/articles/microservices.html



http://christophermeiklejohn.com/filibuster/2021/10/14/filibuster-4.html


Trending
Telemetry
Global Recs
User Profiles
Bookmarks
My List
Ratings
User Recs
API Gateway
Client

NETFLIX



NETFLIX

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

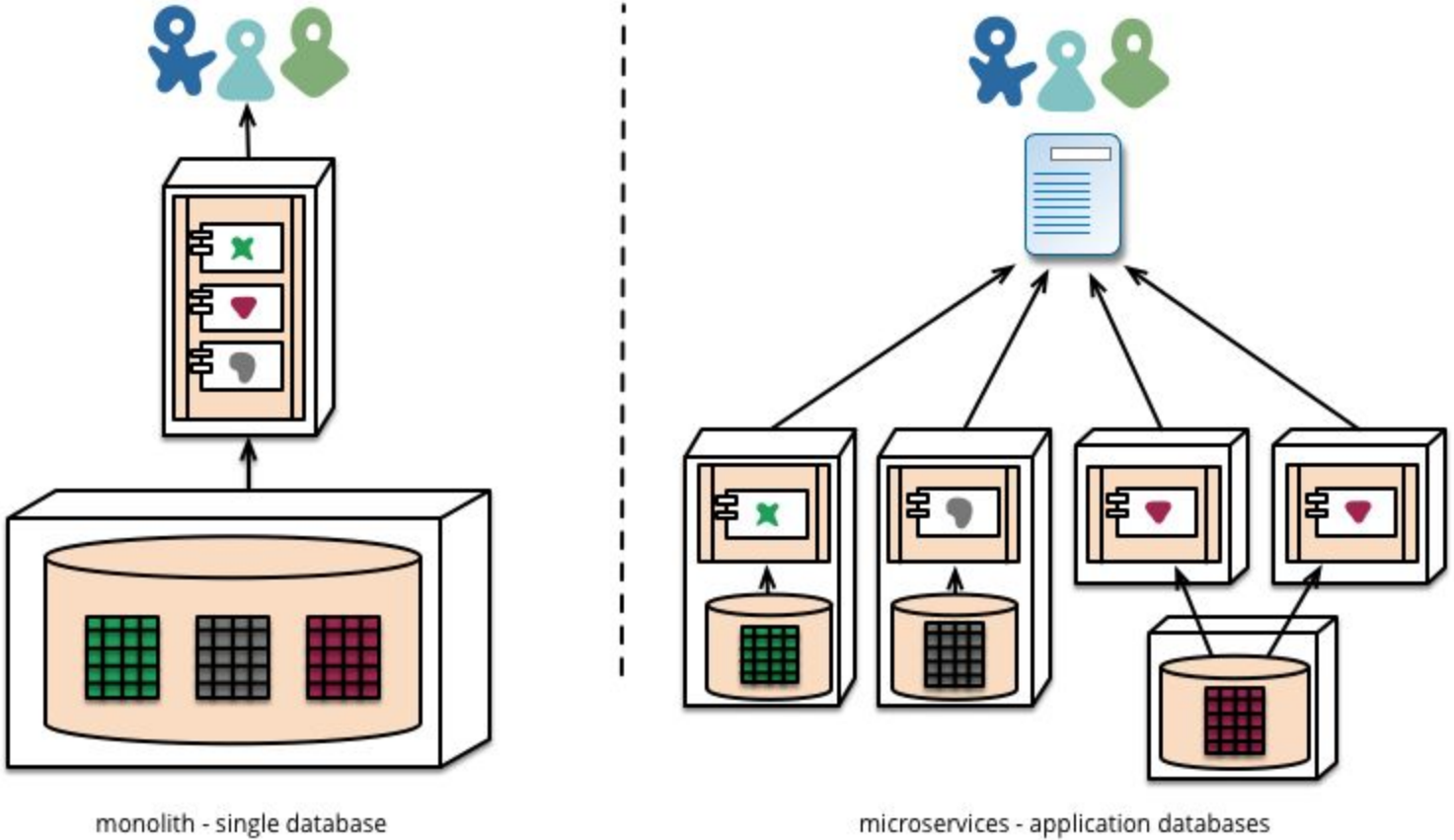
REST APIs

Independent development and deployment

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

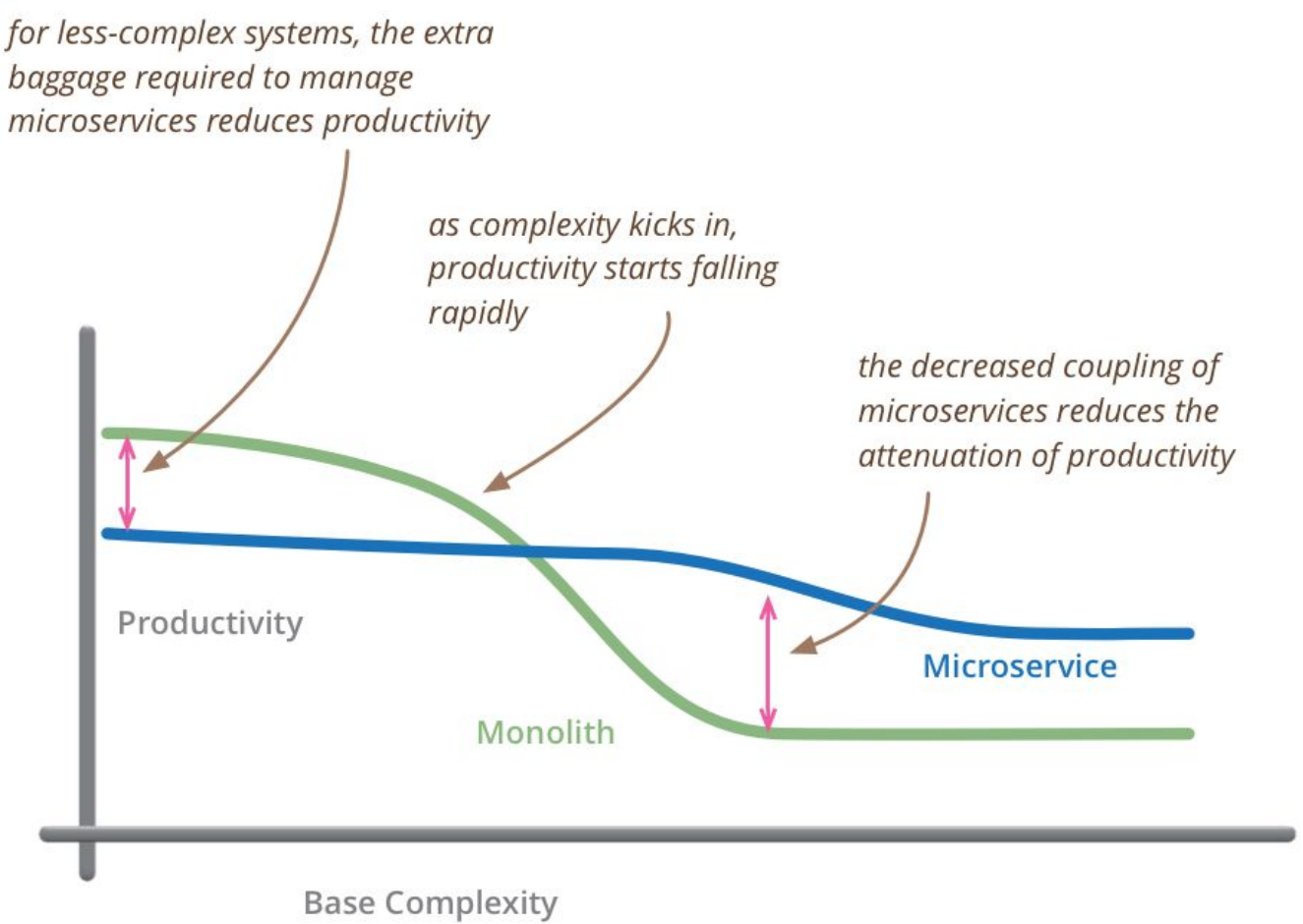
- multiple instances behind load-balancer

Streamline deployment



source: http://martinfowler.com/articles/microservices.html

Overhead



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

Software Architecture vs Design Patterns

Design patterns: Composition and interaction of objects

Architectural pattern: System-level structures, subsystems

Architecture often has focus on system qualities as performance, scalability, robustness, security

Typical architectural patterns/styles: client server, microservice, event-based, pipe and filter

This introduces new challenges when 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? How can we make a robust system in light of this?
  - How can we test considering all the different components and dependencies?
  - What if the system is too big to test?

This introduces new challenges when 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? How can we make a robust system in light of this?**
  - How can we test considering all the different components and dependencies?
  - What if the system is too big to test?

What will you do if

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

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

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);
  }
```

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);
  }
```

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

What will you do if

- An API your data plugin uses is temporarily down?
  - Or returns a surprising error code
- Consider: retry
  - Have a plan
  - Have a back-up plan

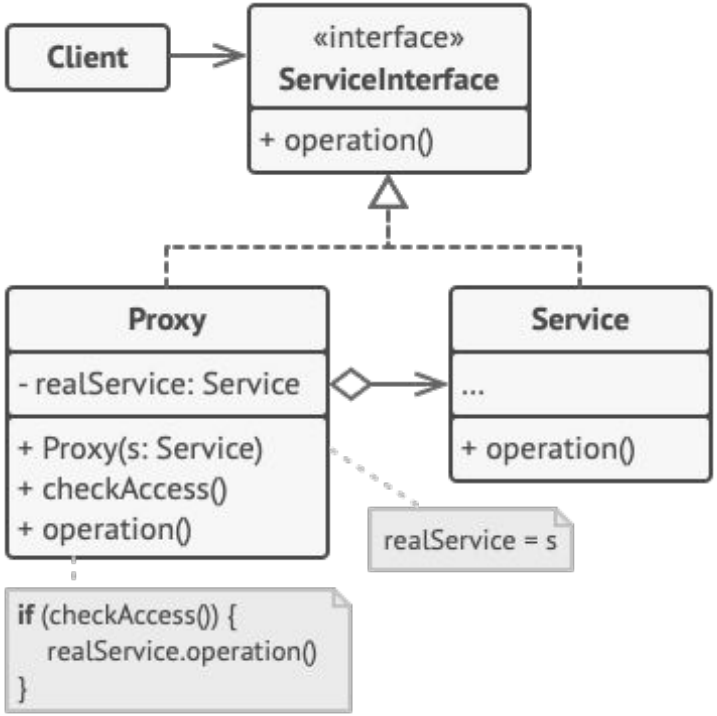
https://blog.cloudflare.com/october-2021-facebook-outage/

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*



https://refactoring.guru/design-patterns/proxy

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

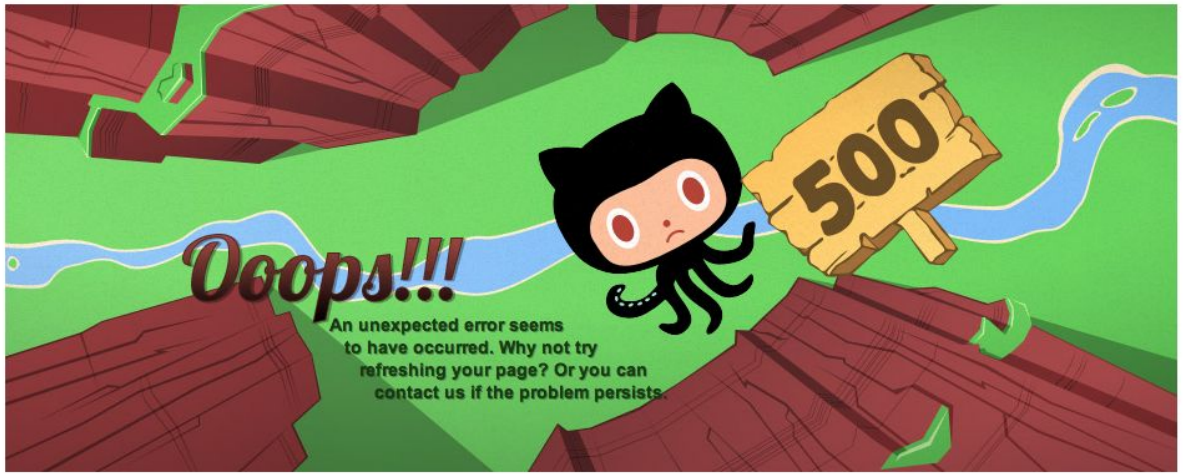
- We need a fallback plan
  - Can't just `e.printStackTrace()`
  - What *can* we do?
- In case of failure, redirect
  - If at all plausible, hand work over to proxy
    - Local data(set), fallback service
  - If not, recruit clean-up service
    - Proces, display errors

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

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

```
curl https://api.stripe.com/v1/charges \
   -u sk_test_BQokikJOvBiI2HlWgH4olfQ2: \
   -H "Idempotency-Key: AGJ6FJMkGQIpHUTX" \
   -d amount=2000 \
   -d currency=usd \
   -d description="Charge for Brandur" \
   -d customer=cus_A8Z5MHwQS7jUmZ
```

https://stripe.com/blog/idempotency

Testing Distributed Systems

- Challenges:
  - Volatility
    - Users are hard to simulate
    - Real-world effects -- things crashing, delays, indicative use/data.
  - Performance
    - Massive databases? Systems with minutes-long start-up times?
    - Very common in ML
- We will return to this later!
  - We'll return to Quality Assurance in "big" (and distributed) systems in the near future.
  - Key principle: isolation – don't test the entire real system!

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)
  - We will get back to testing as a designer
  - Provide error handling through isolation
- Learn to communicate with, and provide your own, nodes
  - API design
  - Microservices

Reactive Programming

Reactive Programming

Programming strategy or patterns, where programs react to data

Embraces concurrency, focuses on data flows

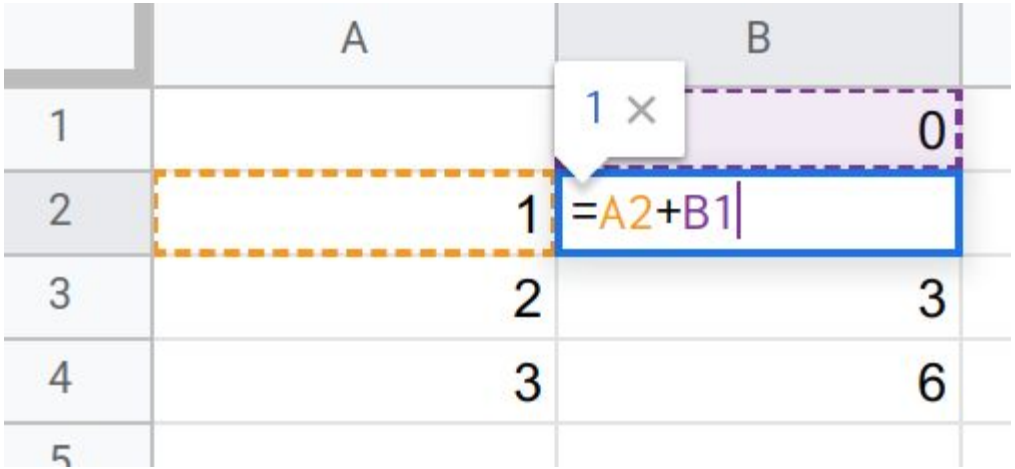
Takes event-based programming to an extreme

Decouples programs around data

Useful analogy: Spreadsheets

Cells contain data or formulas

Formula cells are computed automatically whenever input data changes



Implementing Spreadsheet-Like Computations?

Implementing Spreadsheet-Like Computations?

```
x = 3

y = 5

z = x + y

print(z) // prints 8

x = 5

print(z) // expect 10, prints 8
```

in imperative computations,
no update when inputs change

Implementing Spreadsheet-Like Computations?

```
x = 3

y = 5

z = () => x + y

print(z()) // prints 8

x = 5

print(z()) // prints 10
```

computation performed on demand (pull)
caching possible

Does not easily work in Java, since Java requires variables in closure to be final. Need object with mutable internal state

Implementing Spreadsheet-Like Computations?

```
x = new Cell(3)
y = new Cell(5)
z = new DerivedCell(x, y, (a,b)=>a+b)
print(z.get()) // prints 8
x.set(5)
print(z.get()) // prints 10
```

Cell implements observer pattern, informs observers of changes (push)

DerivedCell listens to changes from Cell, updates internal state on changes, informs own observers of changes

Complications

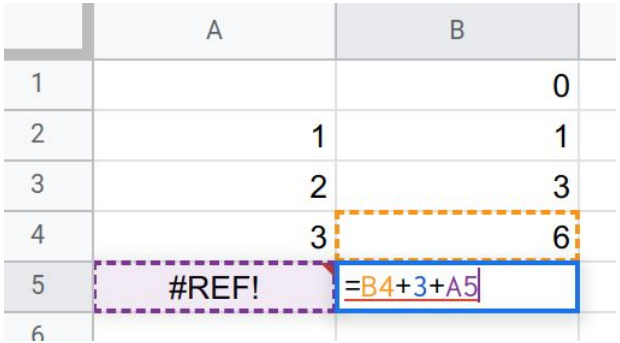
Single change in cell can trigger many computations (push)

- Possibly put in queue, compute asynchronously
- Perform some computations lazily when needed

Cyclic dependencies can result in infinite loops

- Detect, special ways to handle

Observers can hinder garbage collection



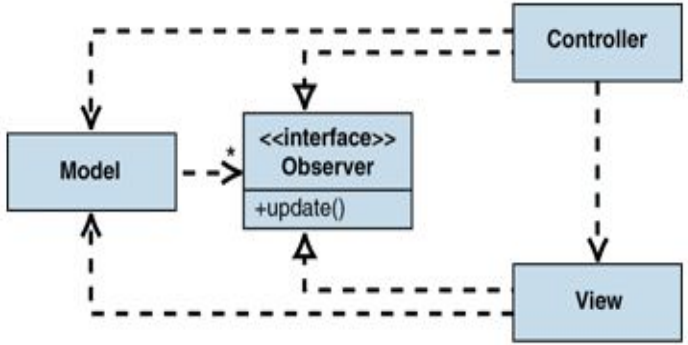
Reactive Programming and GUIs

Store state in observable cells, possibly derived

Have GUI update automatically on state changes

Have buttons perform state changes on cells

Mirrors active model-view-controller pattern (model is observable cell)



From Pull to Push

Instead of expecting clients to look for state (pull)

observers react to state changes with actions (push)

Commonly, observables indicate that something has changed, triggering observers to get updated state (push-pull)

Beyond Spreadsheet Cells

	SINGLE	MULTIPLE
Pull	`Function`	`Iterator`
Push	`Promise`	`Observable`

https://rxjs.dev/guide/observable

Reactive Programming Libraries

RxJava, RxJS, many others

Provide Stream-like interfaces for event handling, with many convenience functions (similar to promises)

Observables typically allow pushing multiple values in sequence

Cells can be implemented by considering only the latest value of observables

Previous Example with 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);
```

Chaining Computations along Data

```
awk '{print $7}' < /var/log/nginx/access.log |
    sort |
    uniq -c |
    sort -r -n |
    head -n 5 > out
```

Multiple programs executed in sequence each read lines and produce lines;
can start reading lines before previous program is finished

Streams / Reactive Programming / Events

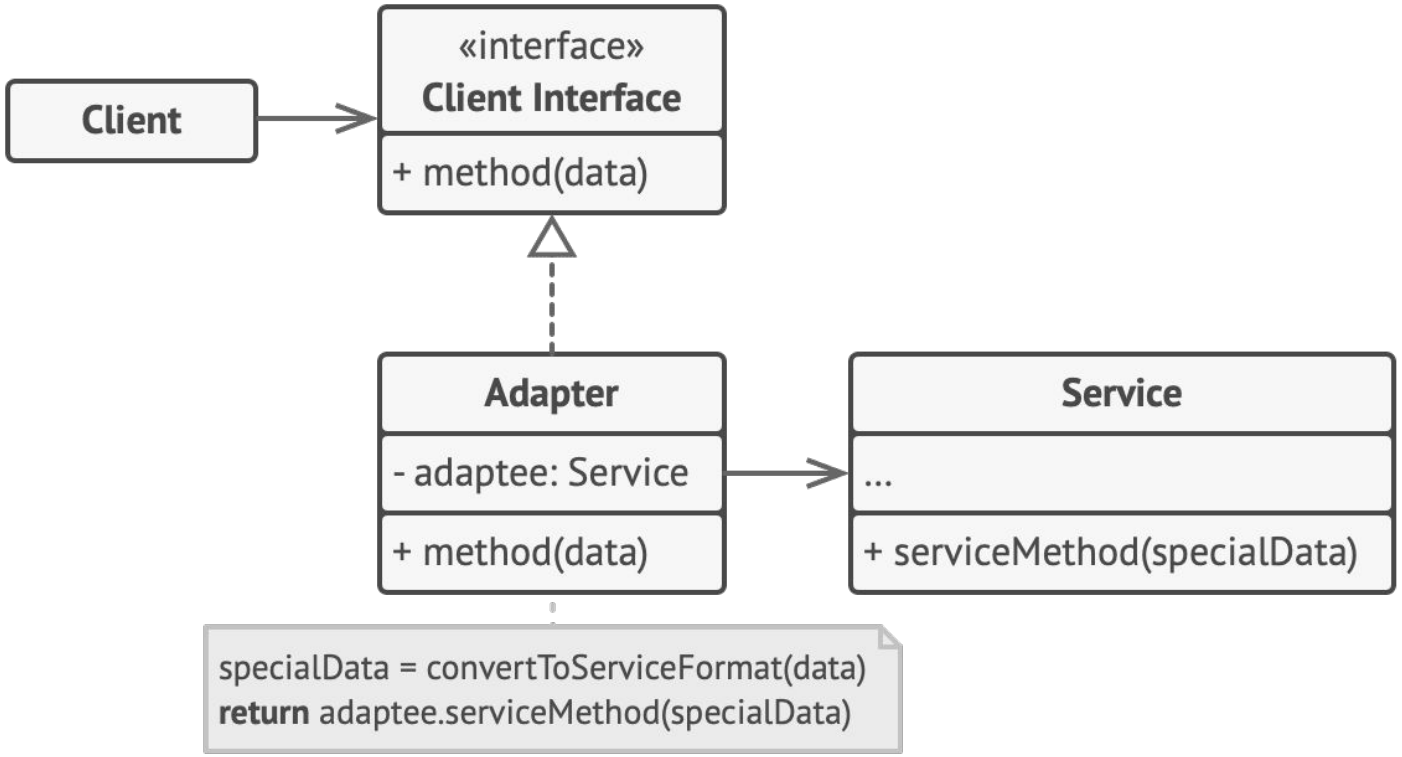
Instead of calling methods in sequence,
set up pipelines for data processing

Let data control the execution

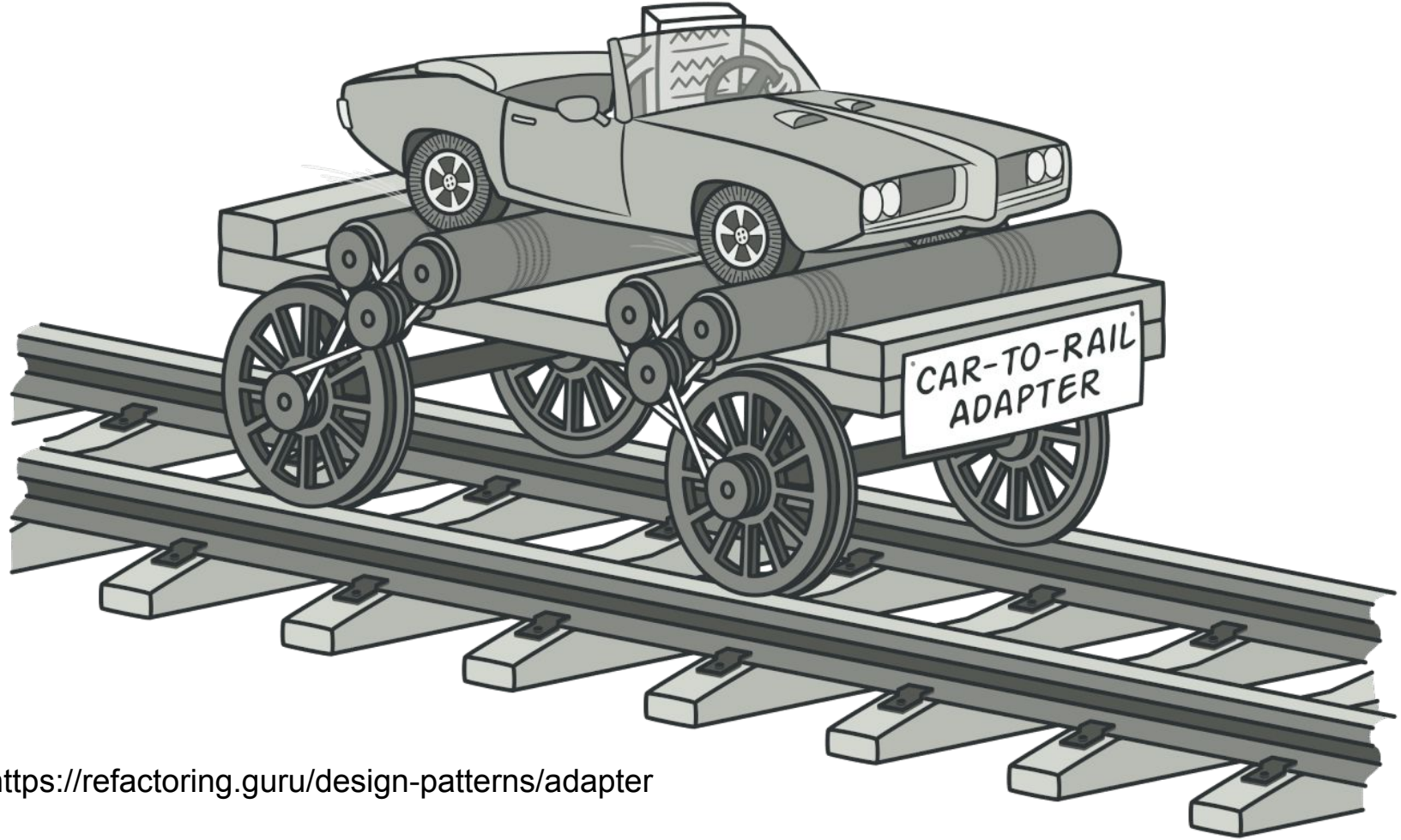
```
var lines = IOHelper.readLinesFromFile(file);
var linesObs = Observable.fromIterable(lines);
linesObs.
       map(Parser::getURLColumn).
       groupBy(...).
       sorted(comparator).
       subscribe(IOHelper.writeToFile(outFile));
```

Aside: The Adapter Pattern

The *Adapter* Design Pattern

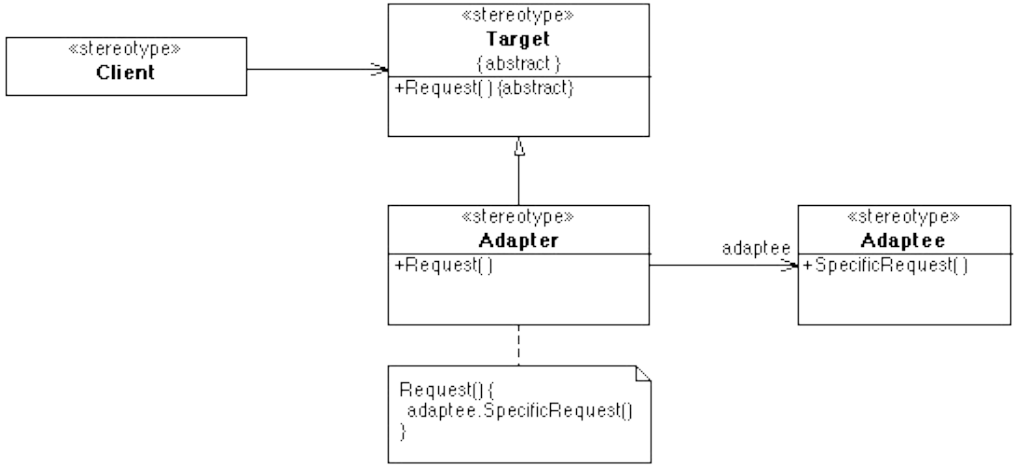


https://refactoring.guru/design-patterns/adapter



https://refactoring.guru/design-patterns/adapter

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**

Adapters for Collections/Streams/Observables

```
var lines = IOHelper.readLinesFromFile(file);
var linesObs = Observable.fromIterable(lines);
linesObs.
       map(Parser::getURLColumn).
       groupBy(...).
       sorted(comparator).
       subscribe(IOHelper.writeToFile(outFile));
```

Any others?

Façade/Controller vs. Adapter

- Motivation
  - Façade: simplify the interface
  - Adapter: match an existing interface
- Adapter: interface is given
  - Not typically true in Façade
- Adapter: polymorphic
  - Dispatch dynamically to multiple implementations
  - Façade: typically choose the implementation statically

Summary

- Most modern systems are distributed
  - Goes beyond basic concurrency; processes vs. threads
  - Start thinking about microservices
  - We began discussing robustness
- Recapped decoupling GUI, promises
- Reactive programming decouples programs along data
  - Observer pattern on steroids
  - New Design Pattern: Adapter