Pragmatic Programmer:
Eliminate effects between unrelated things
by designing components that are:
self-contained, independent,
and have a single, well-defined purpose.

Module 5: High Level Design

Reid Holmes & Nick Bradley

Module Topics

- Design principles: Guidance for building evolvable systems.
- Coupling & cohesion: Measures of module connectedness.
- Technical representations: Communicating detailed structural and behavioural information among stakeholders.
- Information Security: Designing systems to mitigate threats from bad actors.
- SOLID: Design principles promoting modular systems.
- Microservices & API design: Exposing features within your system in a deliberate way.

Examinable Skills

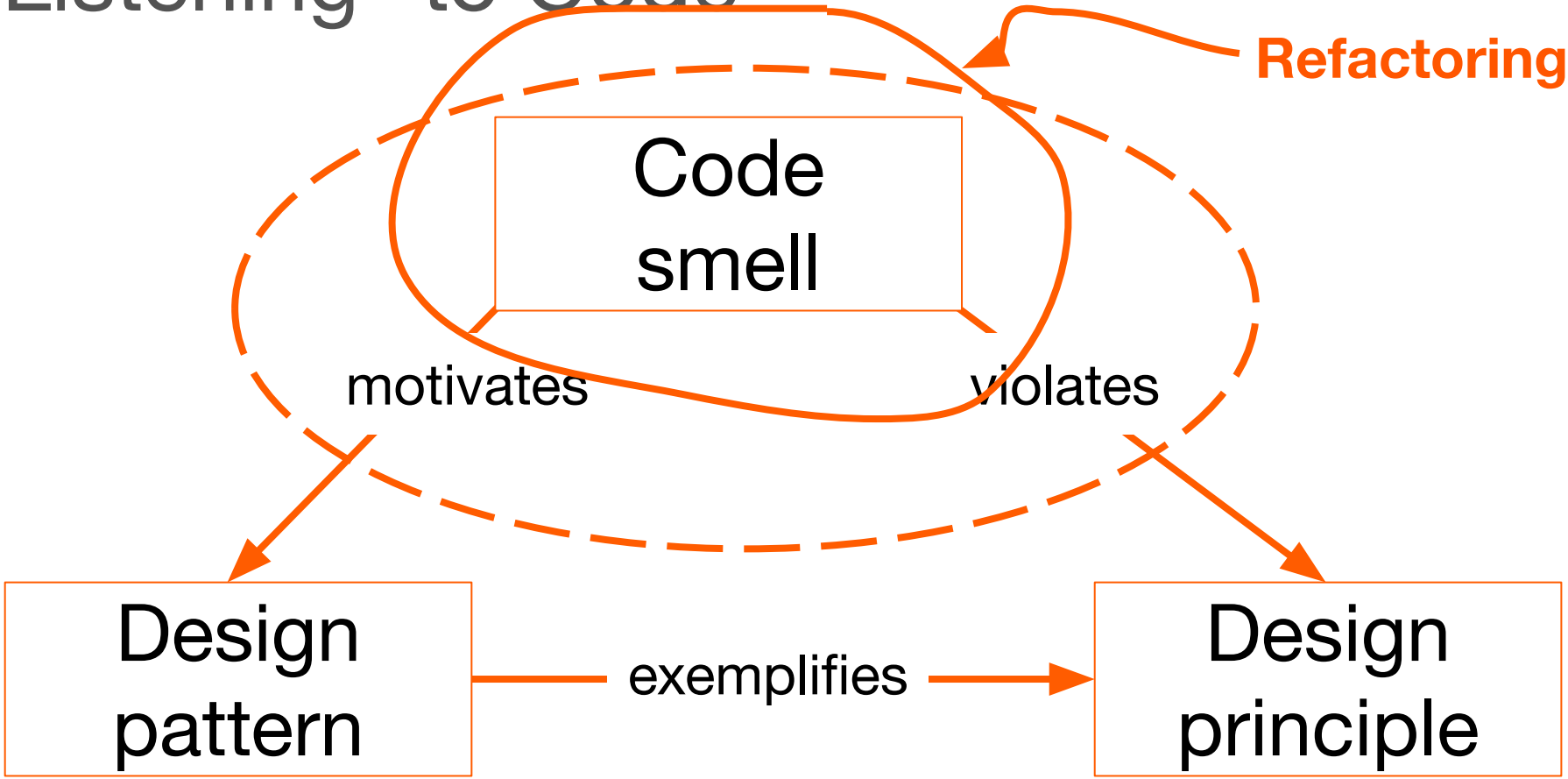
The ability to:

- Understand the core tensions which design principles try to enable.
- Identify design principle violations.
- Identify how code smells arise and how they violate design principles.
- Explain how the presence of a violation of one principle may lead to the violation of other principles (for instance, by violating the interface segregation principle, you're setting yourself up to violate LSP).
- Fix design problems to take a system from violating a principle to no longer violating that principle, usually through refactoring and other code restructuring.
- You do NOT need to remember each kind of coupling/cohesion. Remember the names of the best and worst one of each, and be able to look at code and distinguish between better and worse coupling/cohesion.

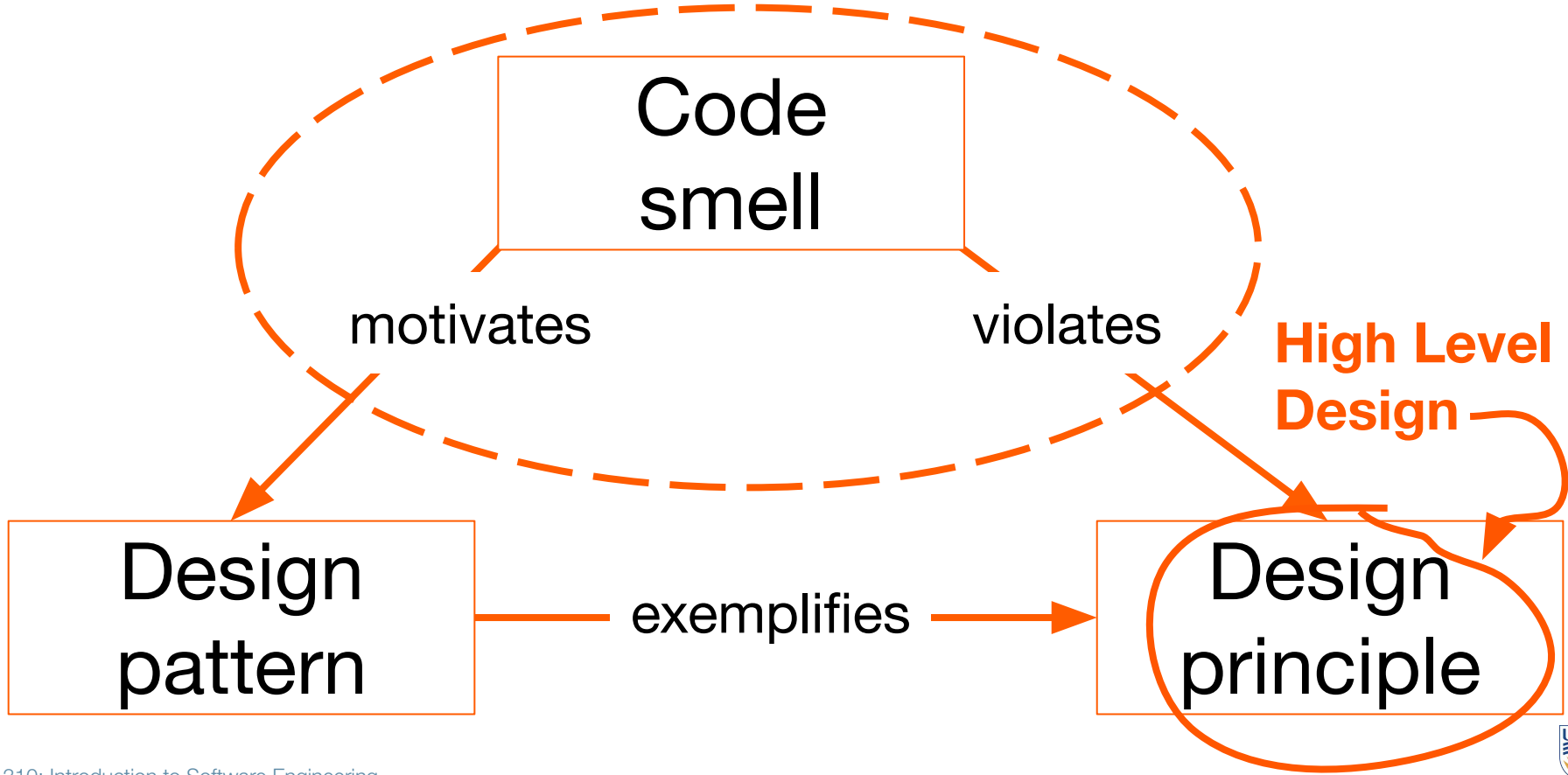
Design Principles

Topics

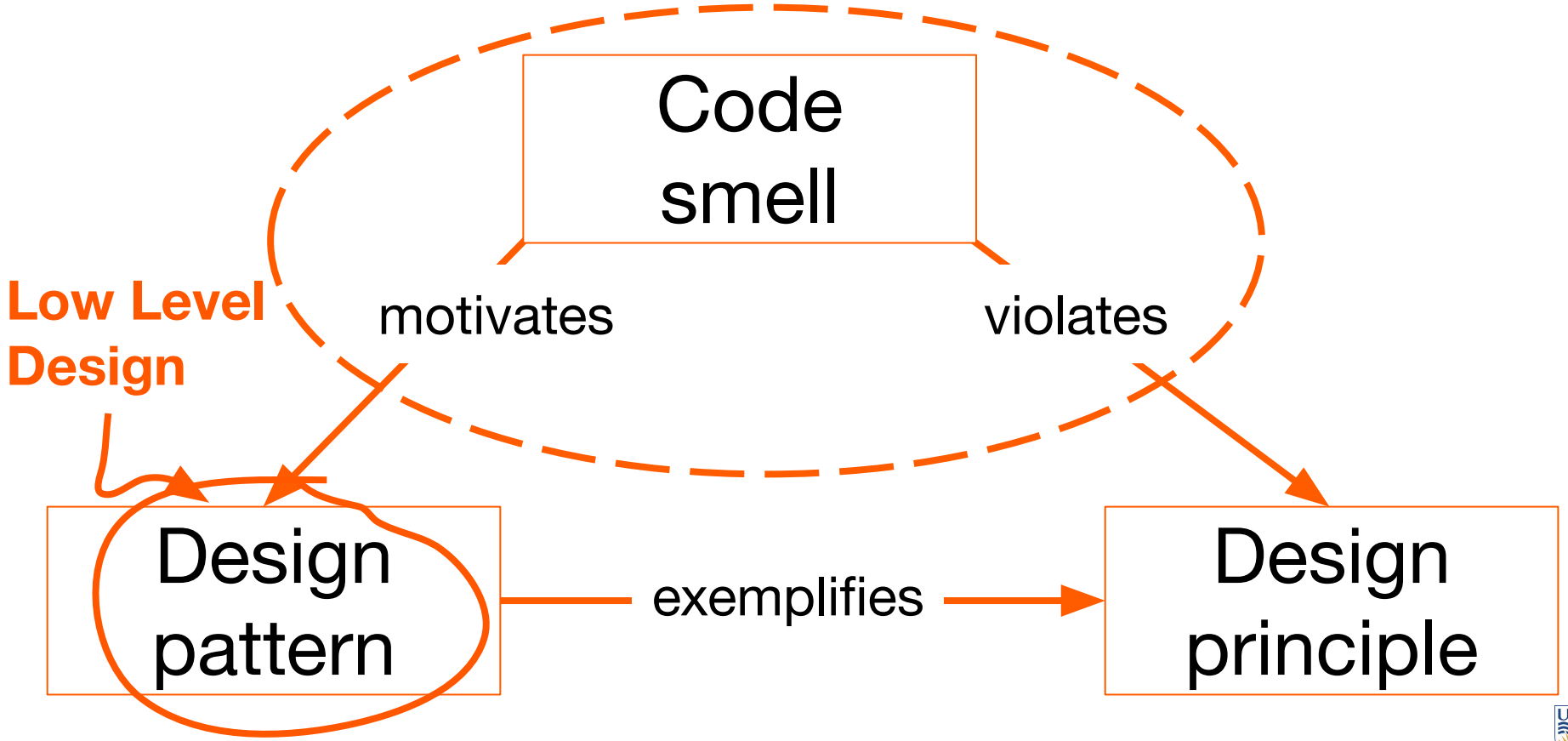
“Listening” to Code



“Listening” to Code



“Listening” to Code



Source: [*Gamma et all, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995*]

- Program an Interface not an Implementation
- Favor Composition Versus Inheritance
- Find what varies and encapsulate it

Source: [*R. Martin, "Agile Software Development, Principles, Patterns, and Practices", Prentice-Hall, 2002*]

- Dependency-Inversion Principle
- Liskov Substitution Principle
- Open-Closed Principle
- Interface-Segregation Principle
- Reuse/Release Equivalency Principle
- Common Closure Principle
- Common Reuse Principle
- Acyclic Dependencies Principle
- Stable Dependencies Principle
- Stable Abstraction Principle

Software Design Principles

Source: [*Larman, "Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development", Prentice-Hall, 20*

- Design principles are codified in the GRASP Pattern
- GRASP (Pattern of General Principles in Assigning Responsabilities)
- Assign a responsibility to the information expert
- Assign a responsability so that coupling remains low
- Assign a responsability so that cohesion remains
- Assign responsabilities using polymorphic operatio
- Assign a highly cohesive set of responsabilities to
- Don't talk to strangers (Law of Demeter)

Source: [*Parnas, "On the Criteria To Be Used in Decomp*

- Information Hiding
- Modularity

Source: [*Hunt, Thomas, "The Pragmatic Programmer: F*

- DRY – Don't Repeat yourself
- Make it easy to reuse
- Design for Orthogonality
- Eliminate effects between unrelated things
- Program close to the problem domain
- Minimize Coupling between Modules
- Design Using Services
- Always Design for Concurrency
- Abstractions Live Longer than details

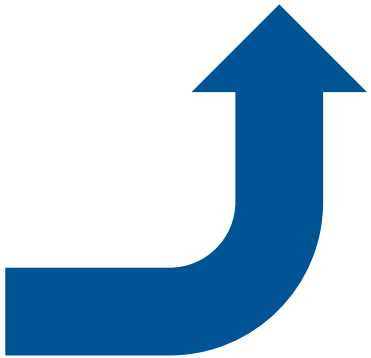
Source: [*Lieberherr,Holland, "Assuring Good Style for Object-Oriented Programs", IEEE Software, September 1989*]

- Law of Demeter

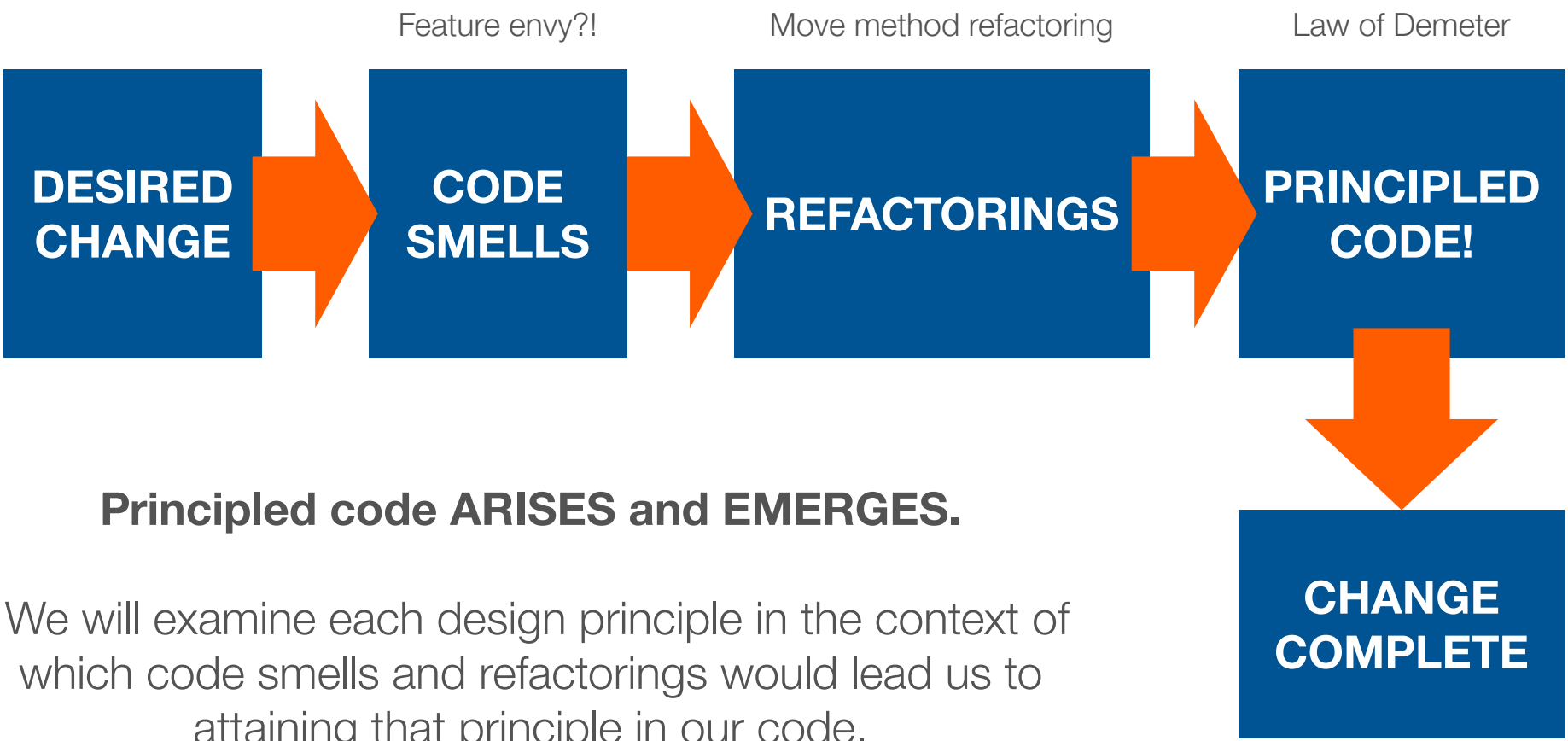
Pragmatic Programmer:
Eliminate effects between unrelated things
by designing components that are:
self-contained, independent,
and have a single, well-defined purpose.

Pragmatic Programmer:
Eliminate effects between unrelated things
by designing components that are:
self-contained, independent,
and have a single, well-defined purpose.

This was the same thing we noted when talking about refactoring!



Or:
1) Localize changes.
2) Eliminate change collisions.



Principled code ARISES and EMERGES.

We will examine each design principle in the context of which code smells and refactorings would lead us to attaining that principle in our code.

Why Principled Code?

- More flexible - affording change, add new features.
- Be understood.
- Be testable.
- Be maintainable - bug free (or easy to find bugs).

Why Principled Code?

Flexible

- Should handle change (like new features).

Understandable

- Code is shared.

Maintainable

- Endure over time (easily find and fix bugs).

Sound familiar? Think back to Broken Code definition.

Design Guidance

Abstraction (high level concept at the root of all that follows)

- Manage complexity by focusing on key aspects for given task and stakeholder.
- Both too much abstraction or too little abstraction inhibit understanding.
- Useful for discussing viewpoints that makes sense for individual stakeholders.

Abstraction (high level concept at the root of all that follows)

- Manage complexity by focusing on key aspects for given task and stakeholder.
- Both too much abstraction or too little abstraction inhibit understanding.
- Useful for discussing viewpoints that makes sense for individual stakeholders.

Decomposition (mechanism for ideating about abstractions)

- Mechanism for breaking down complex description into more manageable pieces.
- Goal is to make common tasks simple while not prohibiting exceptional tasks.
- Can be done both top down or bottom up (or both).

Abstraction (high level concept at the root of all that follows)

- Manage complexity by focusing on key aspects for given task and stakeholder.
- Both too much abstraction or too little abstraction inhibit understanding.
- Useful for discussing viewpoints that makes sense for individual stakeholders.

Decomposition (mechanism for ideating about abstractions)

- Mechanism for breaking down complex description into more manageable pieces.
- Goal is to make common tasks simple while not prohibiting exceptional tasks.
- Can be done both top down or bottom up (or both).

Information hiding (how programs leverage abstraction)

- Hides implementation details from high level interfaces.
- Separate that which varies from that which stays the same.
- Separate names from implementations (aka API from impl).

Abstraction (high level concept at the root of all that follows)

- Manage complexity by focusing on key aspects for given task and stakeholder.
- Both too much abstraction or too little abstraction inhibit understanding.
- Useful for discussing viewpoints that makes sense for individual stakeholders.

Decomposition (mechanism for ideating about abstractions)

- Mechanism for breaking down complex description into more manageable pieces.
- Goal is to make common tasks simple while not prohibiting exceptional tasks.
- Can be done both top down or bottom up (or both).

Information hiding (how programs leverage abstraction)

- Hides implementation details from high level interfaces.
- Separate that which varies from that which stays the same.
- Separate names from implementations (aka API from impl).

Encapsulation (language approach for implementing information hiding)

- Mechanism for implementing abstractions in a program.
- Usually through the use of language interfaces.
- Captures data and behaviour and separates these from their implementation.

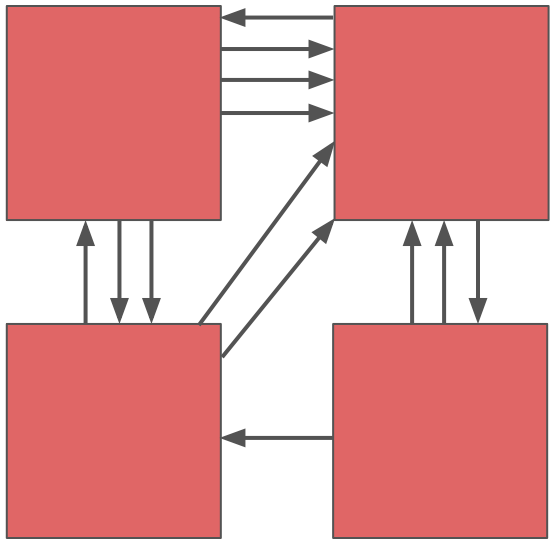
coupling & cohesion

Topics

Coupling Intuition

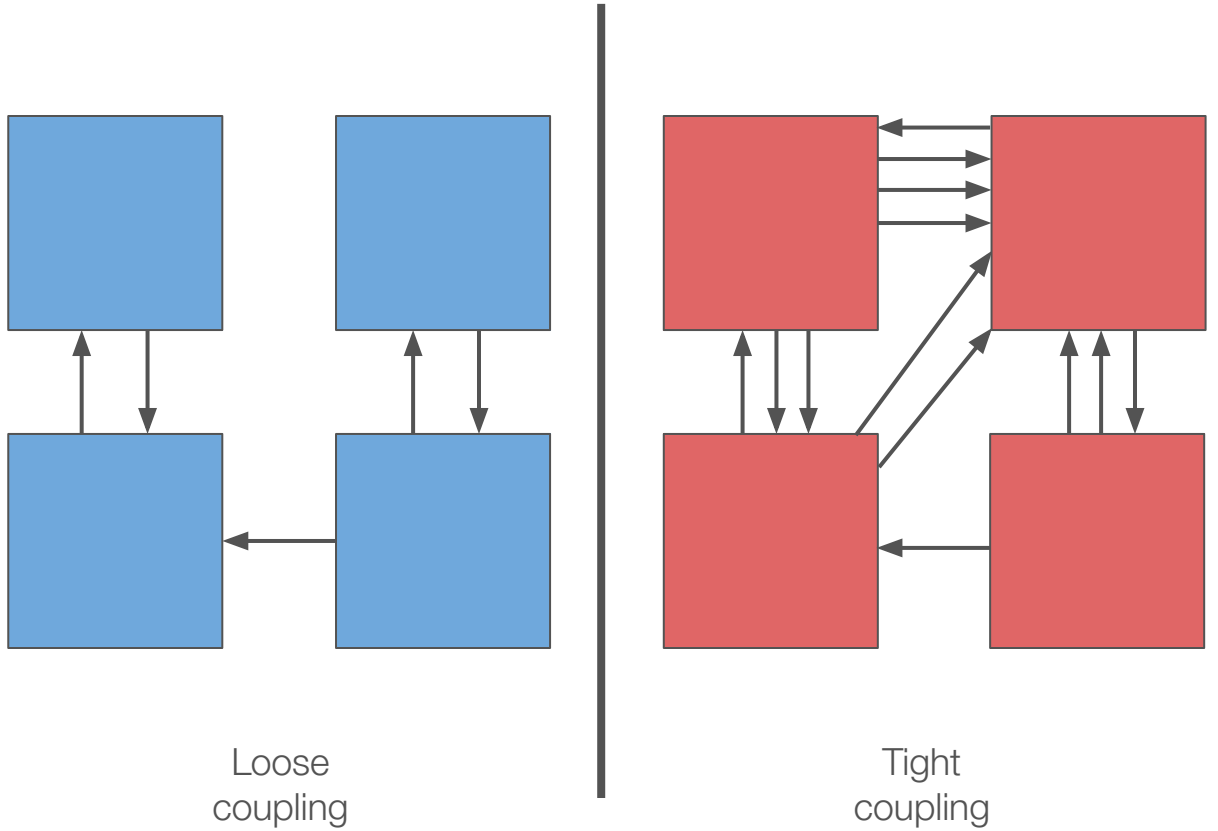
- **Coupling** is the relationship between modules:
  - If we make a change in one module, what will the impact be on other modules within the system?
  - Modules should not depend on each other's internals.
  - Easy to couple modules in a way that violates architecture.

Coupling

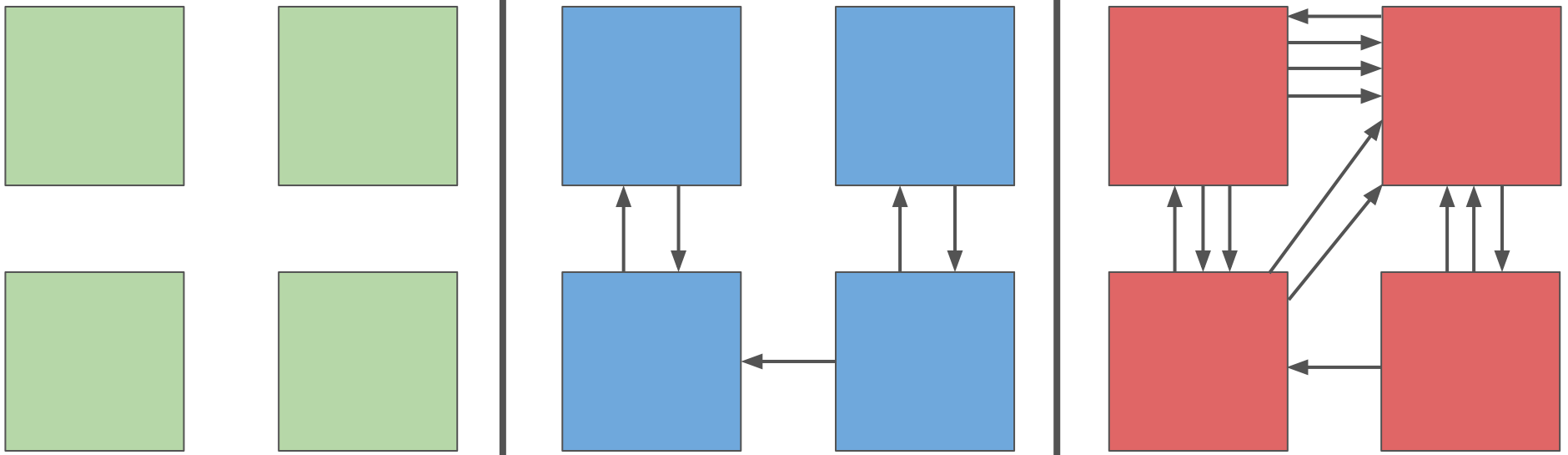


Tight coupling

Coupling



Coupling

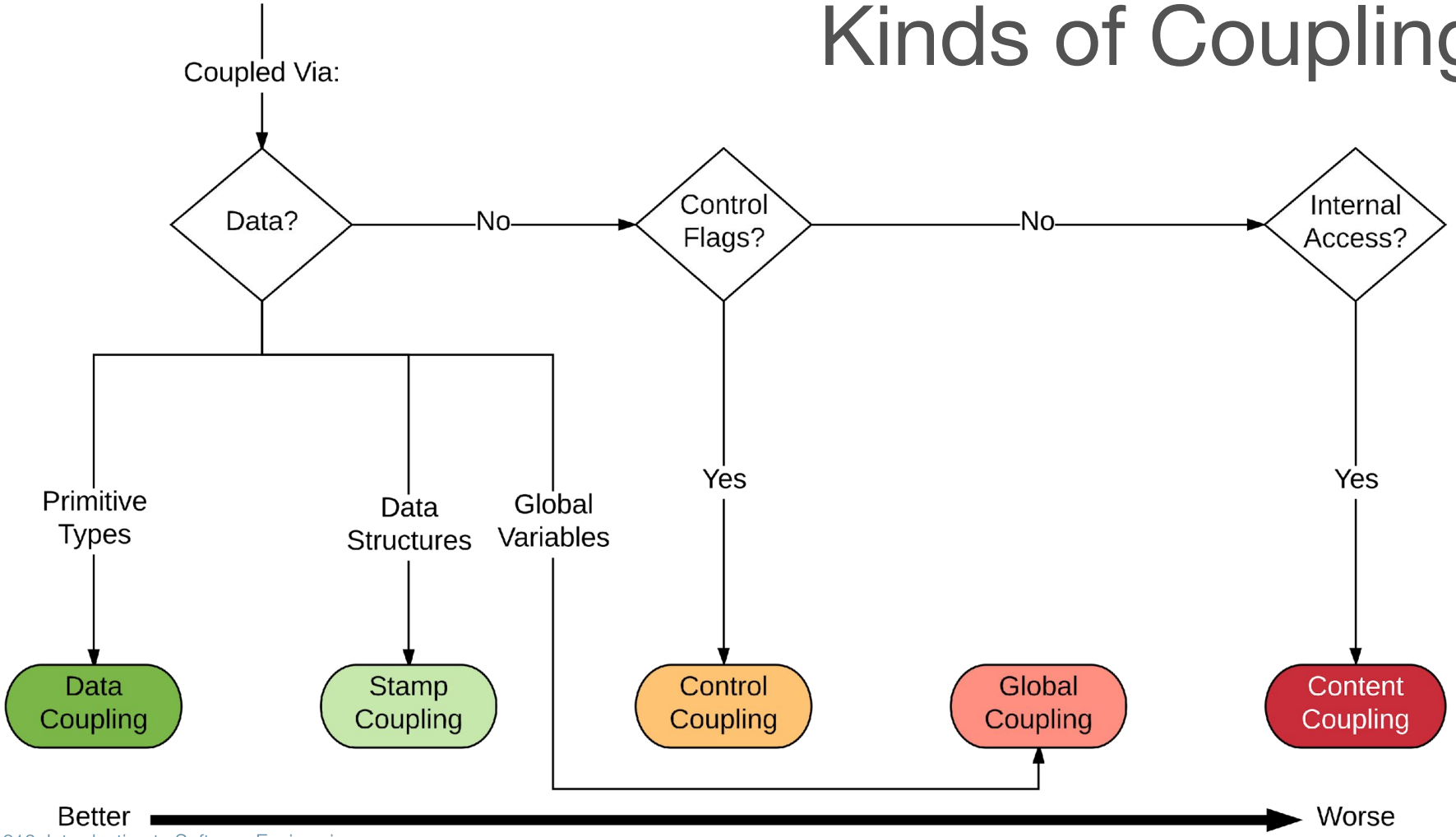


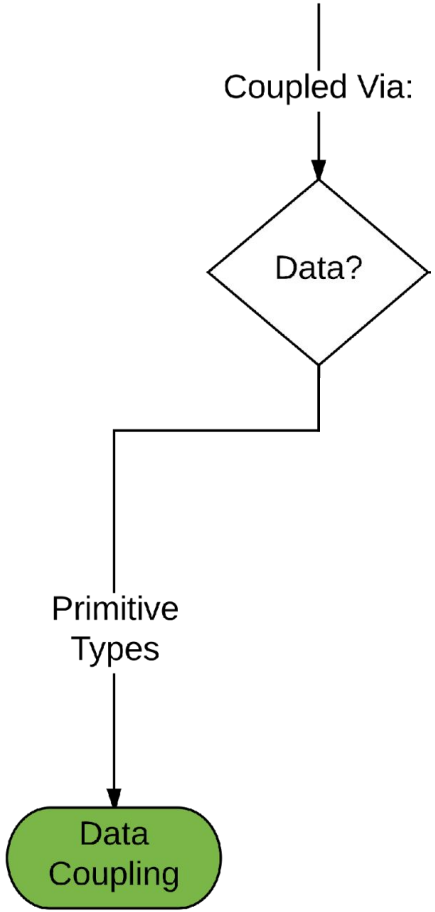
No coupling

Loose coupling

Tight coupling

Kinds of Coupling

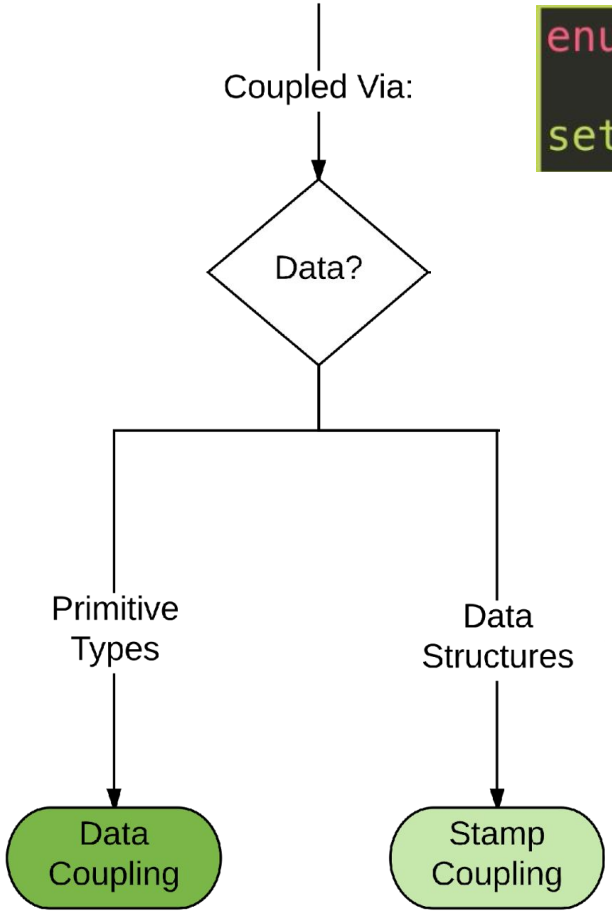





```
setSize(width: number, height: number) { .. }
```

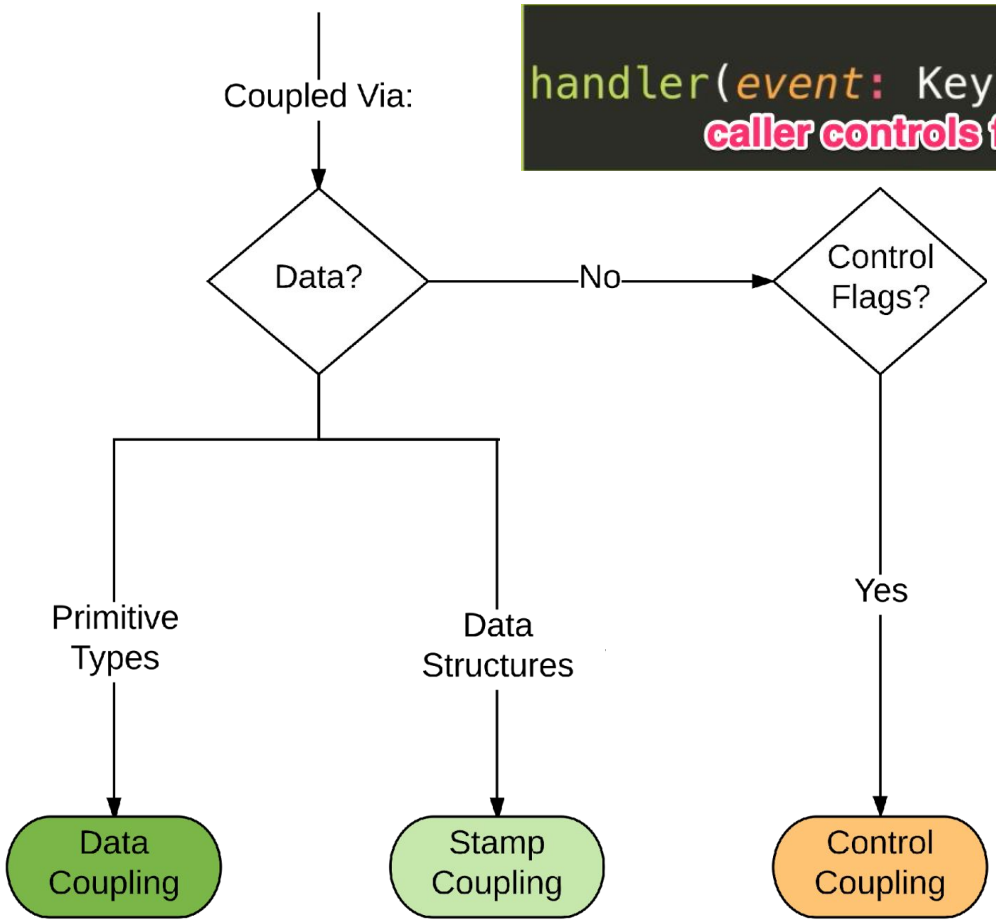
classes sharing data through primitive parameters

Better → Worse




```
enum SizeState {small = 1, big = 2};
                        classes sharing data structures
setState(state: SizeState) {..}
```

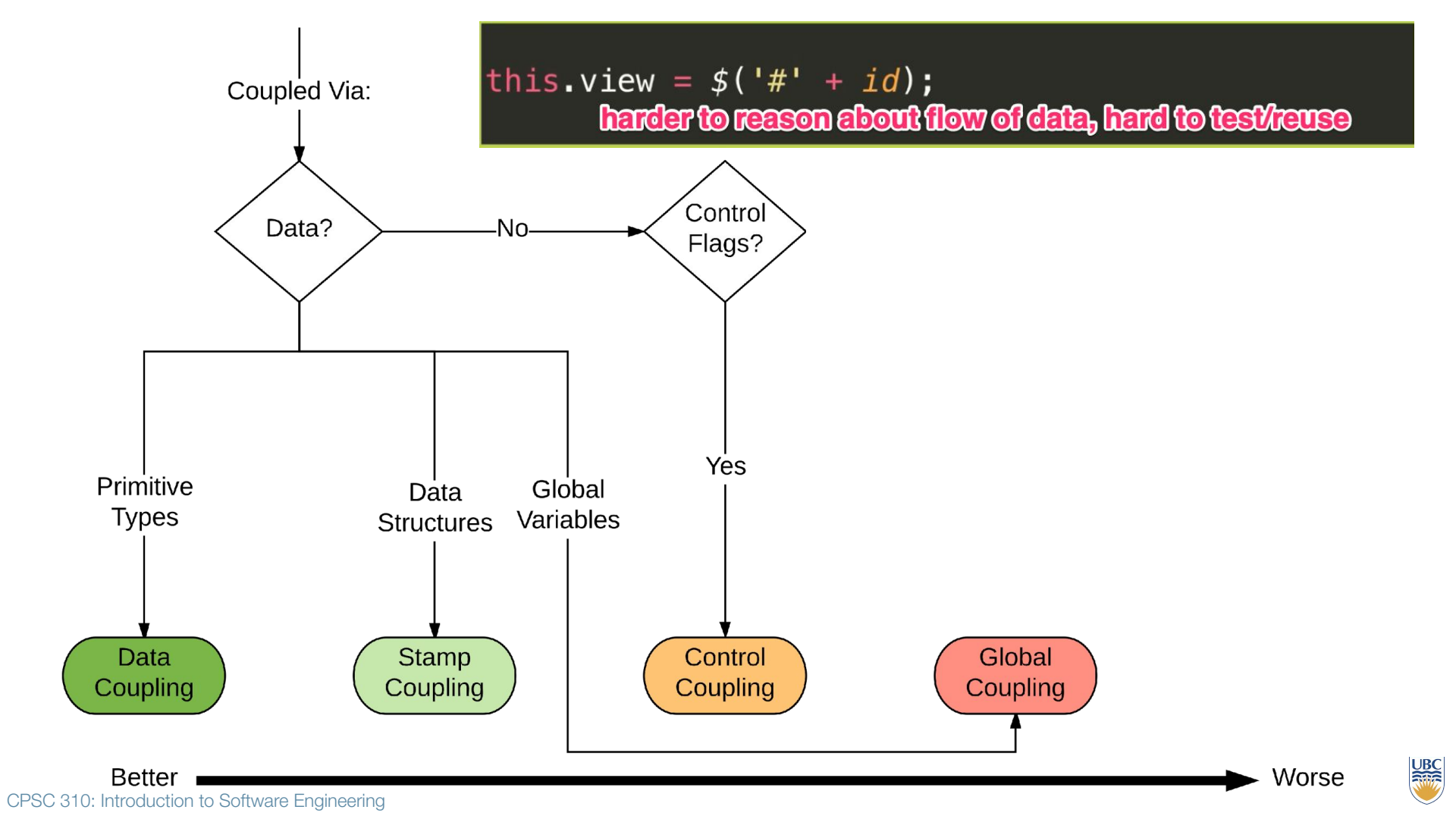
Better → Worse

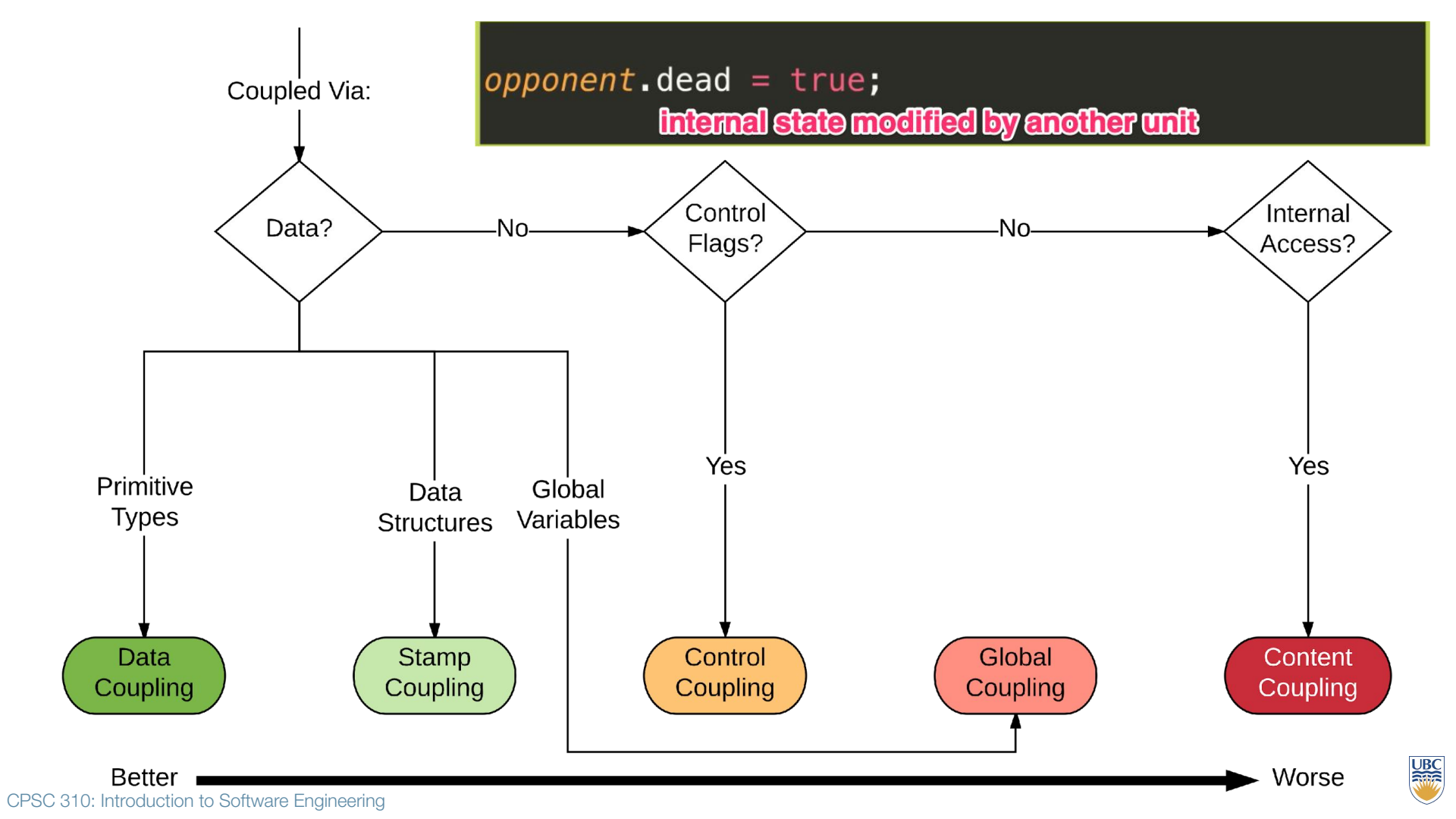



```
handler(event: KeyboardEvent, status: boolean)
```

caller controls flow within another unit

Better → Worse

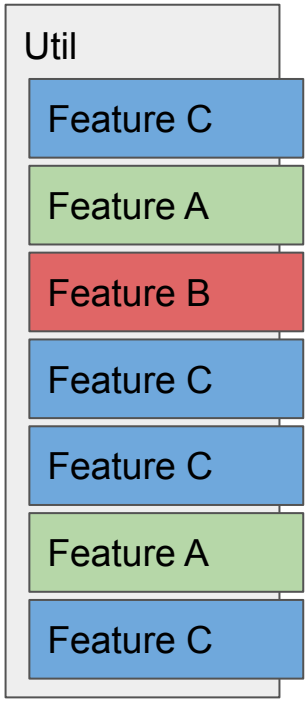

this.view = $('#' + id);
harder to reason about flow of data, hard to test/reuse
Coupled Via:
Data?
No
Control Flags?
Primitive Types
Data Structures
Global Variables
Yes
Data Coupling
Stamp Coupling
Control Coupling
Global Coupling
Better
Worse


Coupled Via:
opponent.dead = true;
internal state modified by another unit
Data?
No
Control Flags?
No
Internal Access?
Primitive Types
Data Structures
Global Variables
Yes
Yes
Data Coupling
Stamp Coupling
Control Coupling
Global Coupling
Content Coupling
Better
Worse

Cohesion Intuition

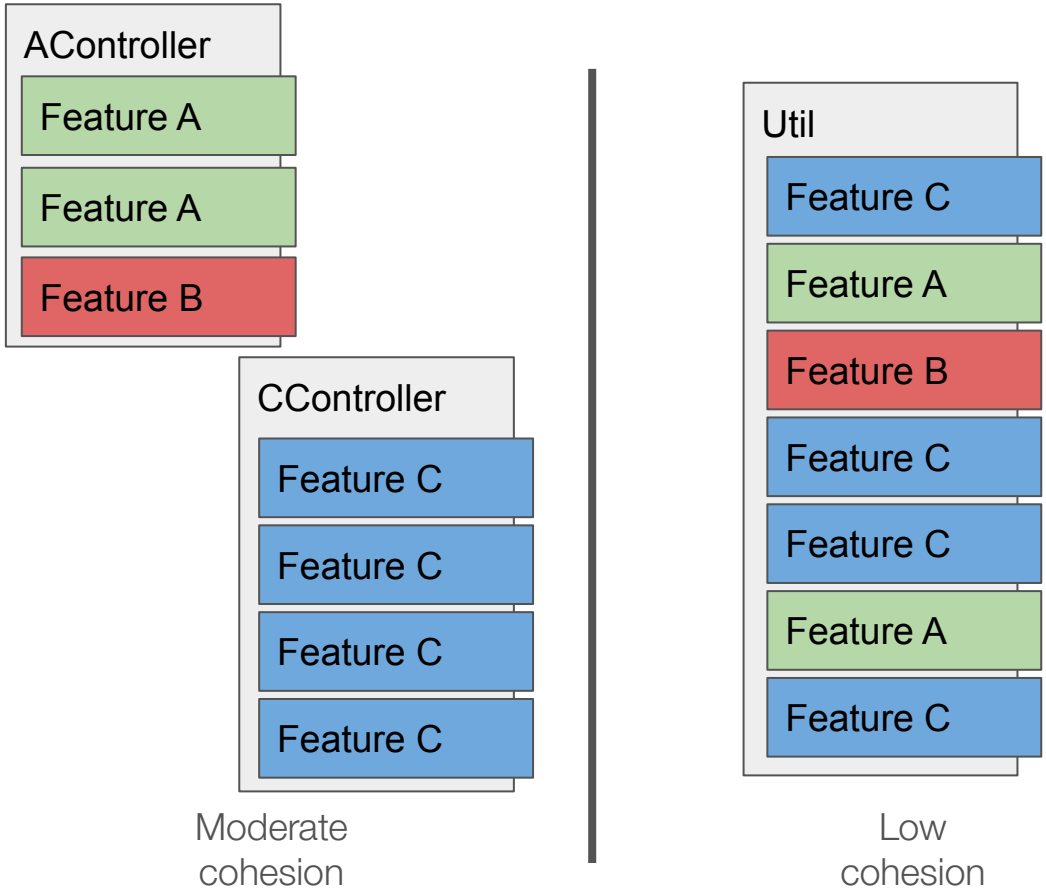
- **Cohesion** is the relationship between elements within a module:
  - Modules should only contain functions that *belong together*.
  - Often observable by the data *modified* by the functions.
  - Commonly violated over time as code evolves.

Cohesion

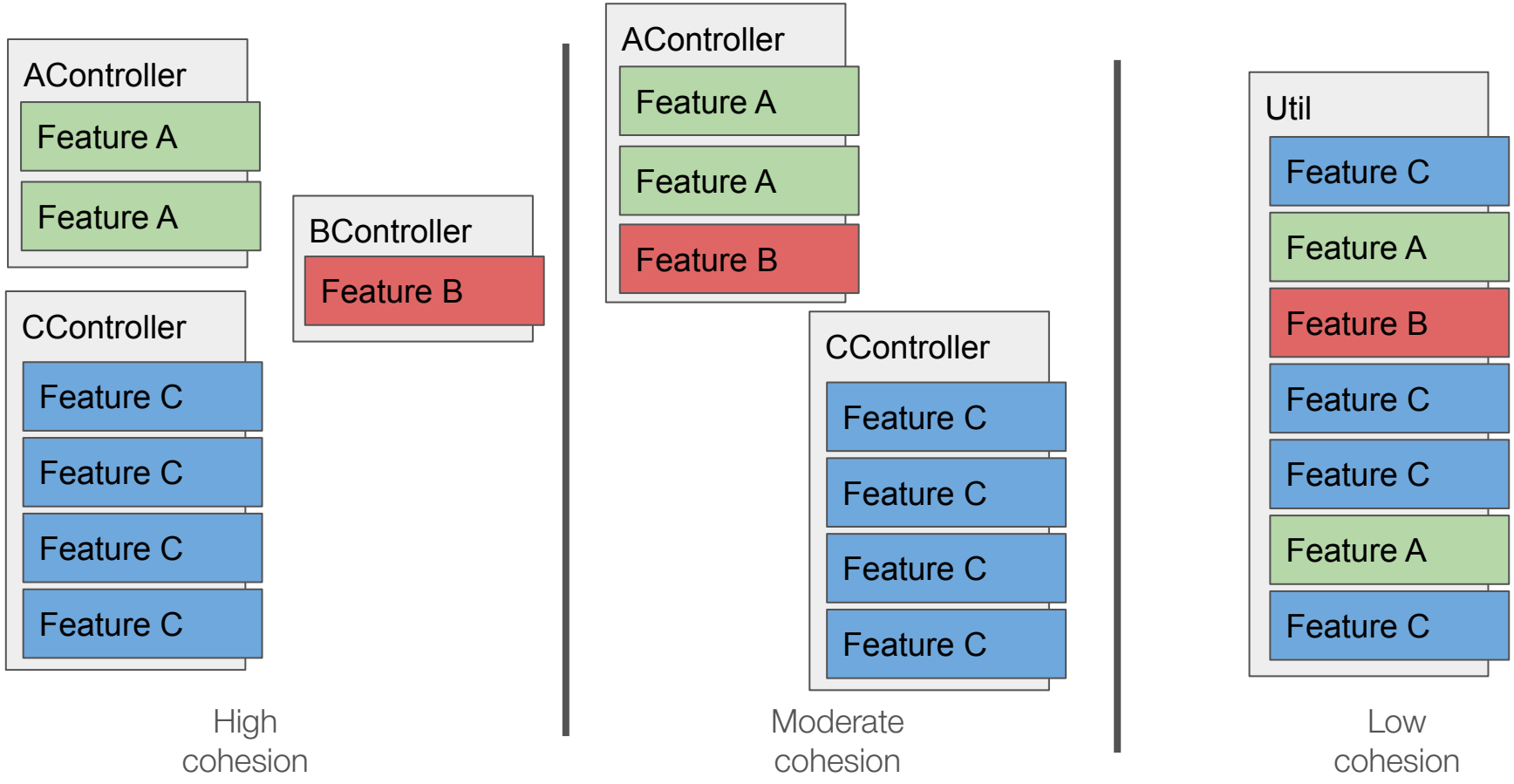


Low cohesion

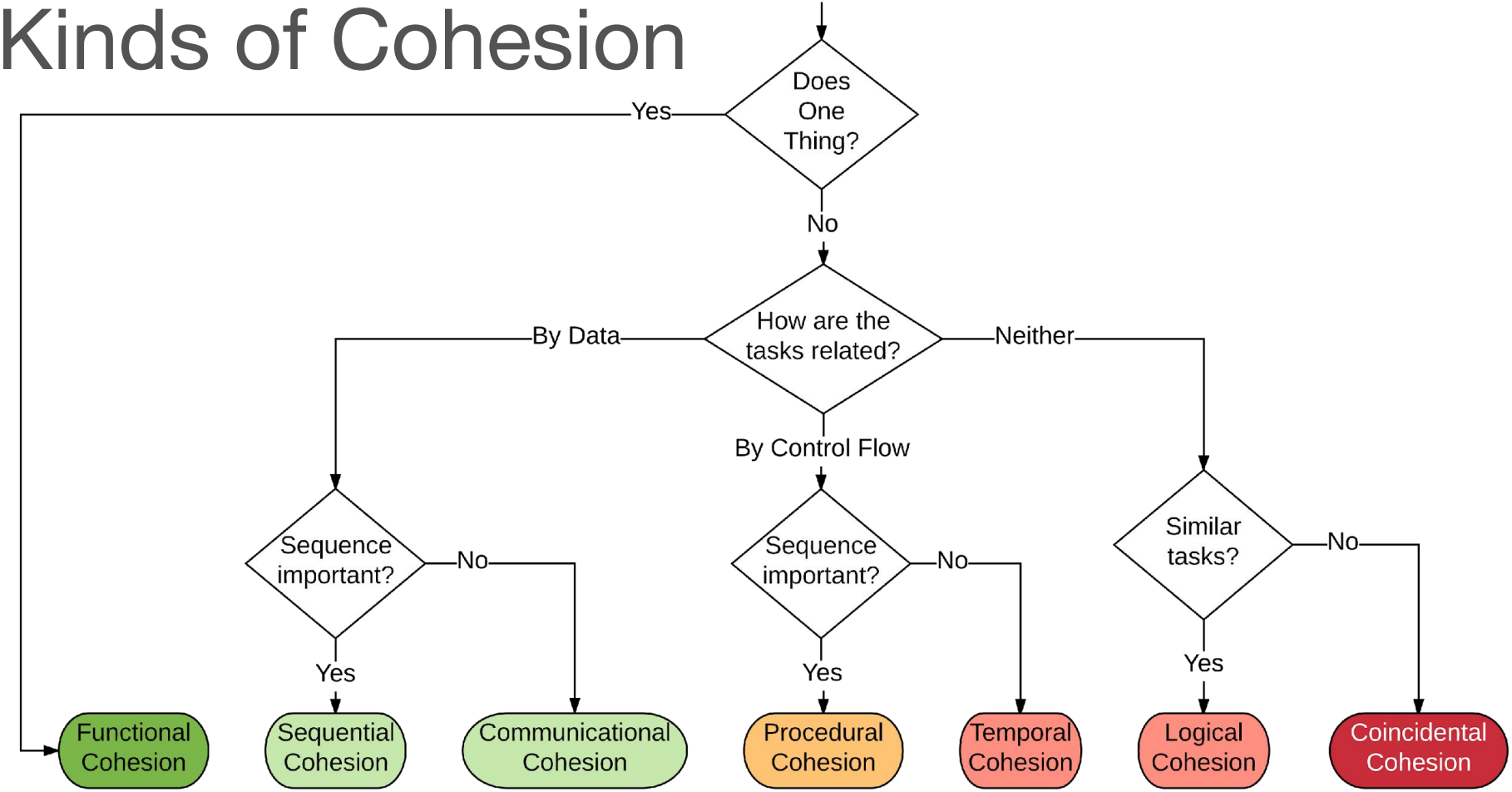
Cohesion

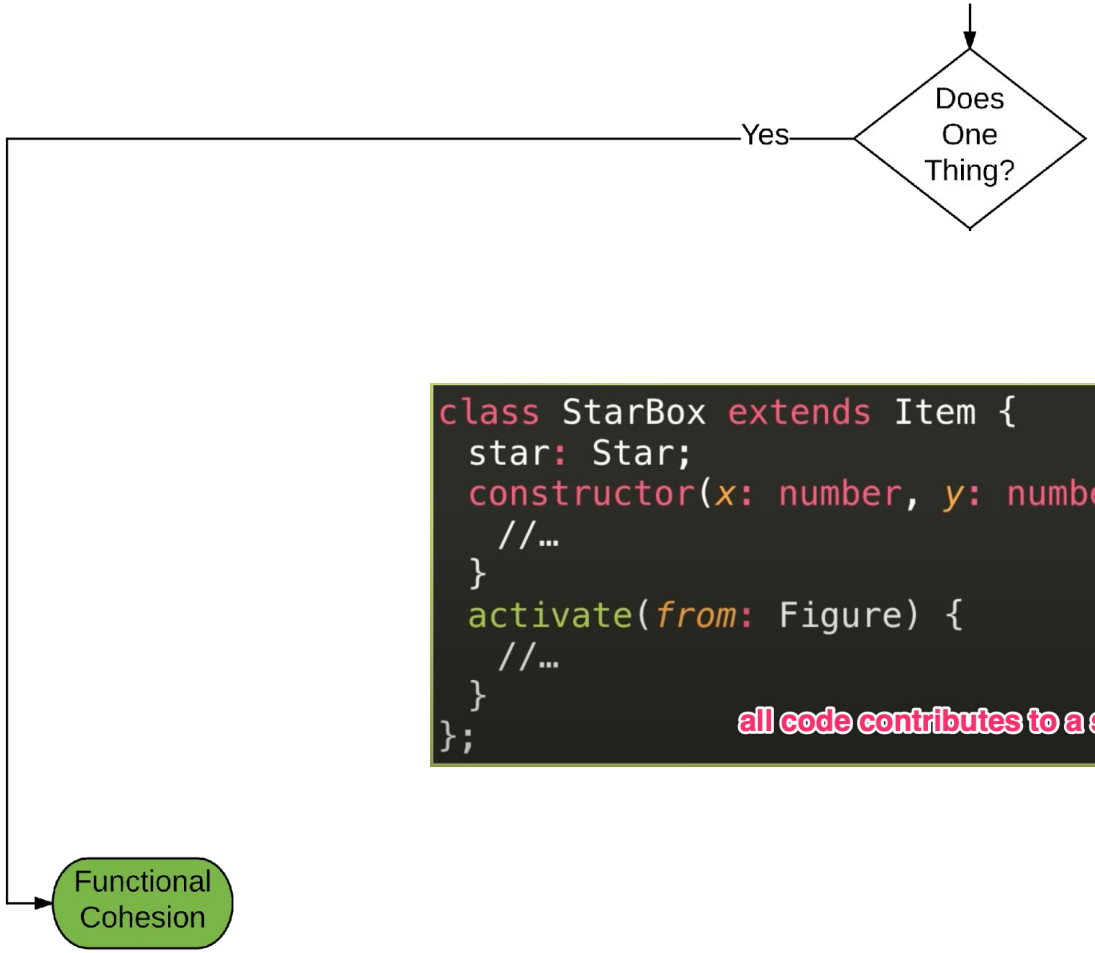

AController
Feature A
Feature A
Feature B
CController
Feature C
Feature C
Feature C
Feature C
Moderate cohesion
Util
Feature C
Feature A
Feature B
Feature C
Feature C
Feature A
Feature C
Low cohesion

Cohesion


AController
Feature A
Feature A
CController
Feature C
Feature C
Feature C
Feature C
BController
Feature B
AController
Feature A
Feature A
Feature B
CController
Feature C
Feature C
Feature C
Feature C
Util
Feature C
Feature A
Feature B
Feature C
Feature C
Feature A
Feature C
High cohesion
Moderate cohesion
Low cohesion

Kinds of Cohesion

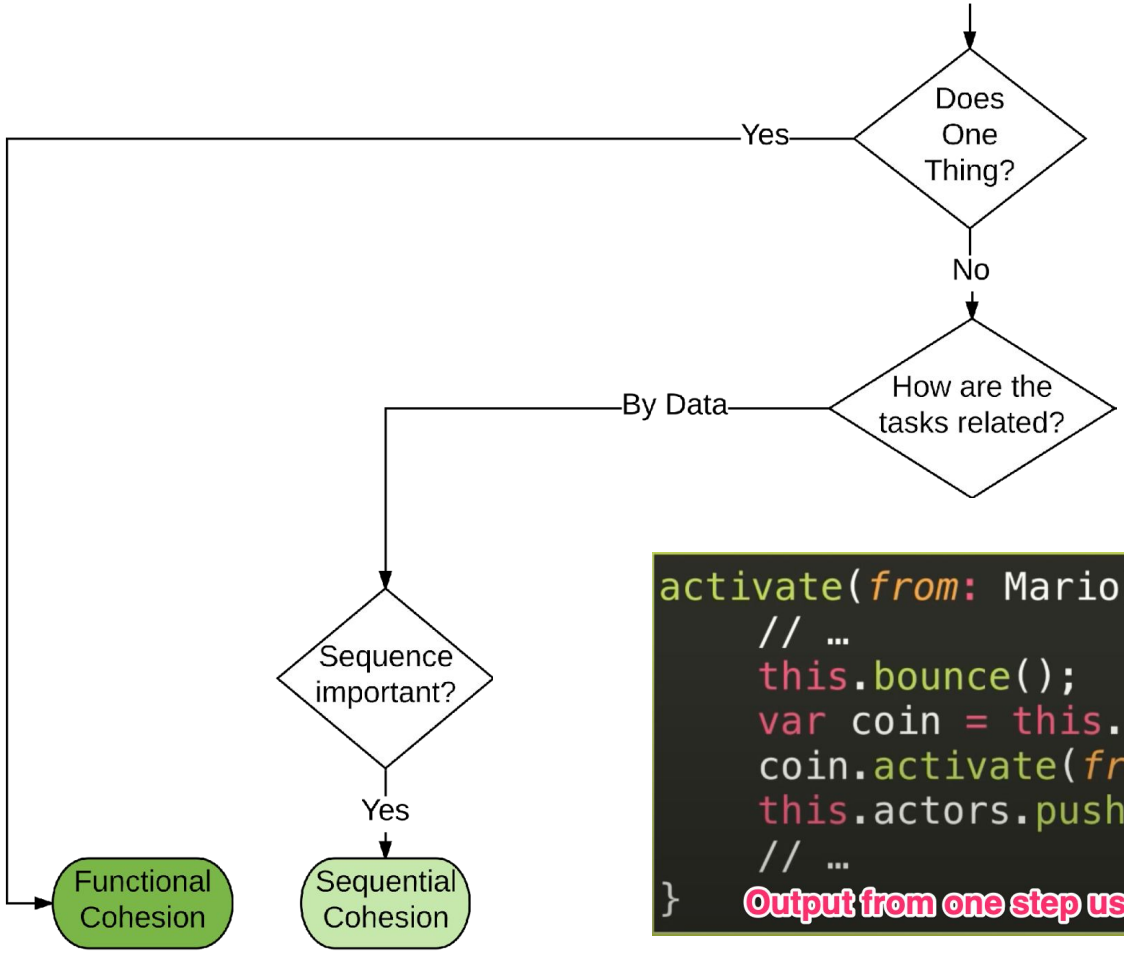





```
class StarBox extends Item {
  star: Star;
  constructor(x: number, y: number, level: Level) {
    //…
  }
  activate(from: Figure) {
    //…
  }
};
```

all code contributes to a single feature or task

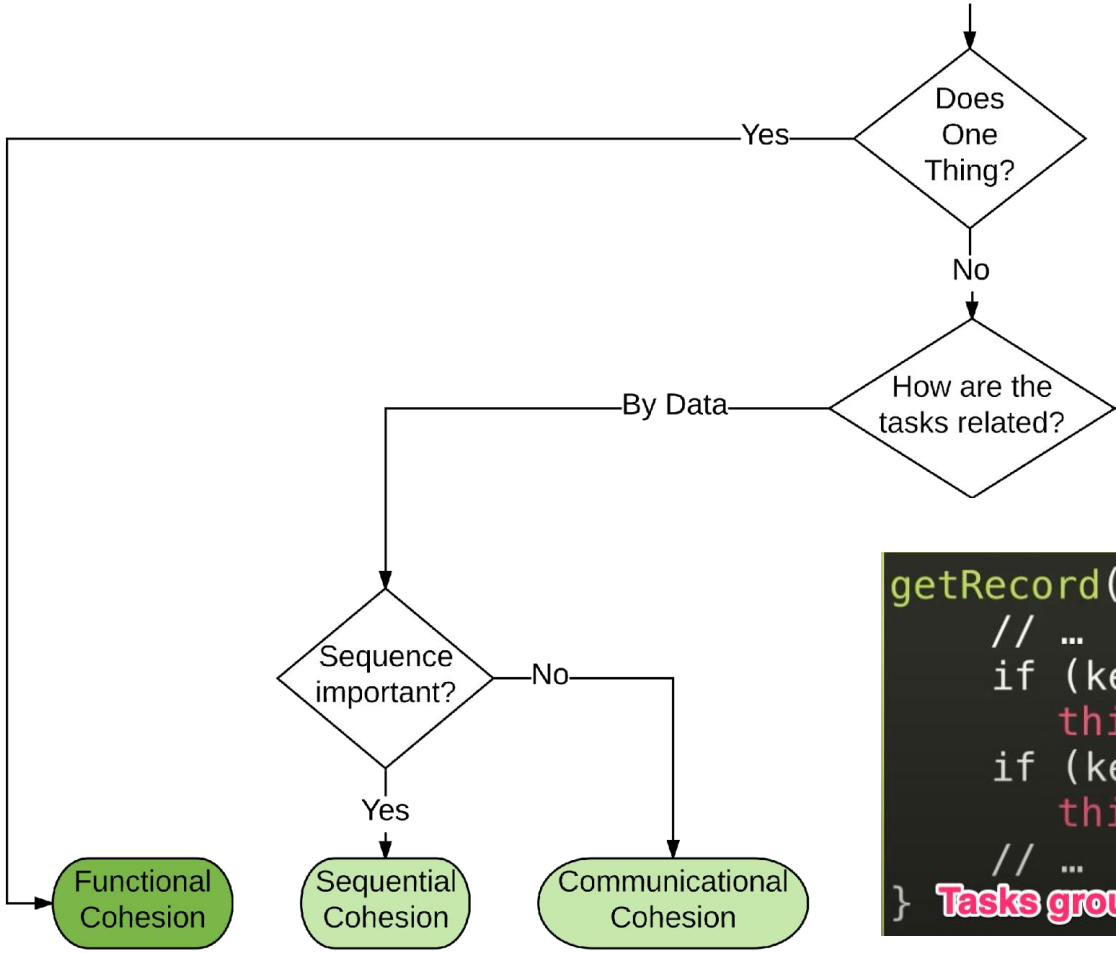
Better → Worse




```
activate(from: Mario) { // from CoinBox
    // …
    this.bounce();
    var coin = this.items.pop();
    coin.activate(from);
    this.actors.push(coin);
    // …
}
```

Output from one step used as input in next step

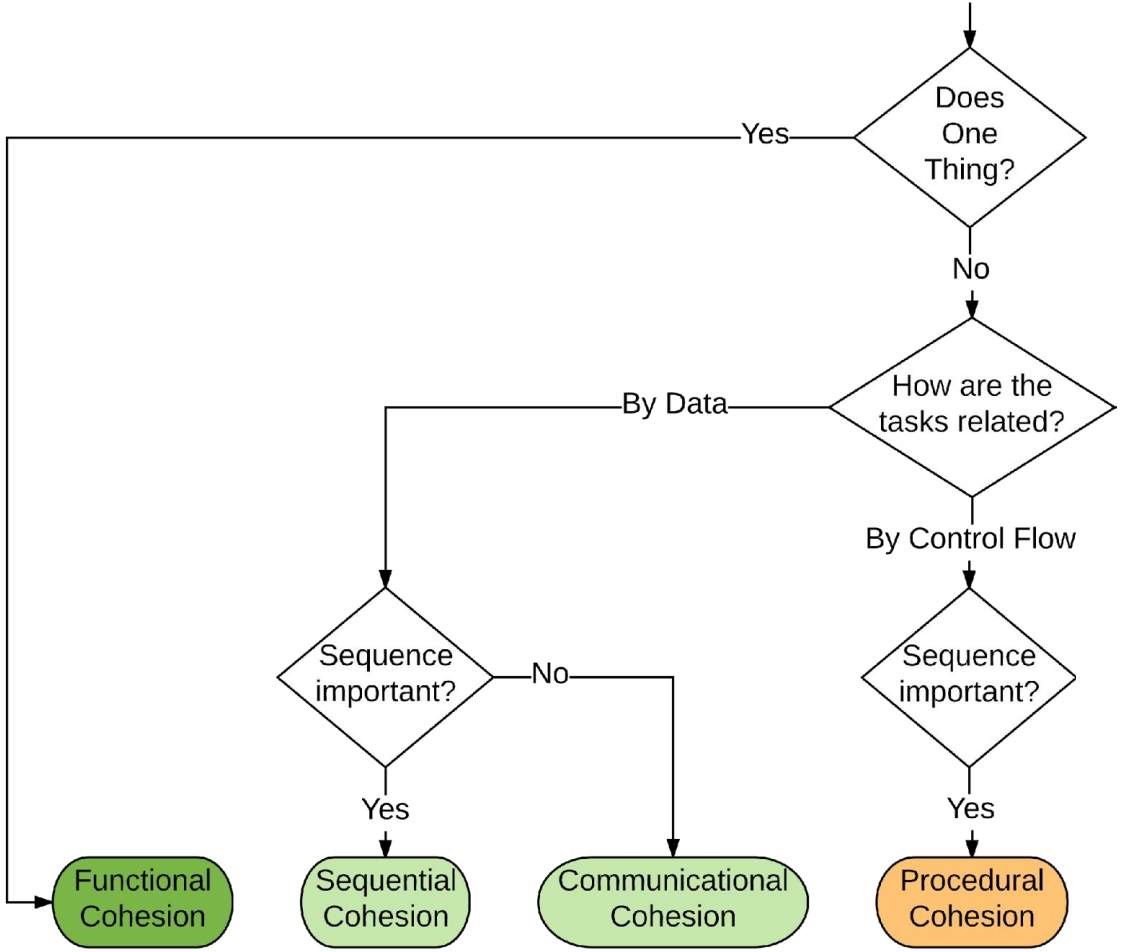
Better → Worse




```
getRecord(key: string) { // synthetic
    // …
    if (key === 'bar')
      this.translateBar(this.data);
    if (key === 'baz')
      this.translateBaz(this.data);
    // …
}
```

Tasks grouped in method only because of data

Better → Worse

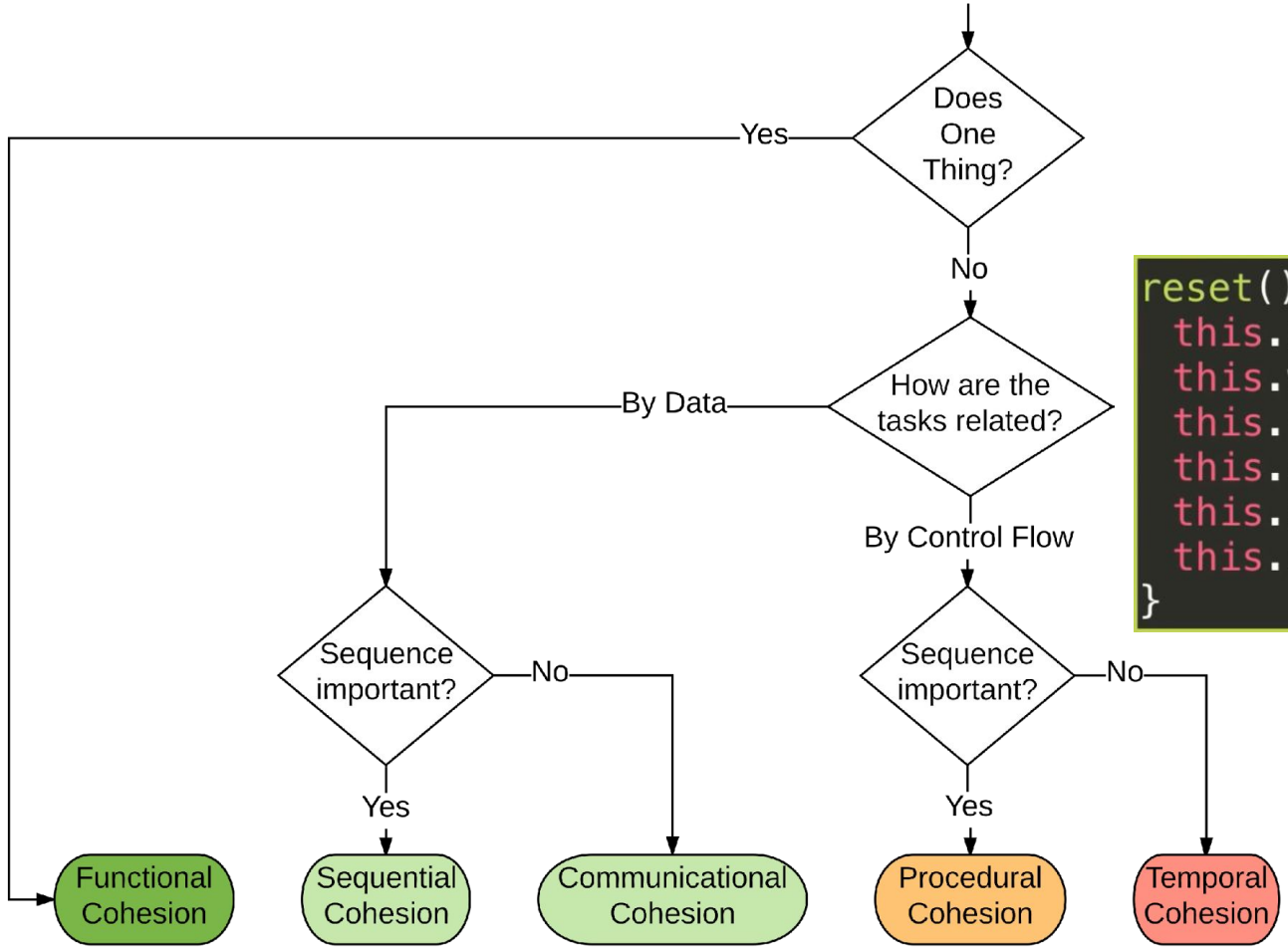


follows specific execution sequence...

```
victory() { // from Mario
  this.level.playMusic('win');
  this.clearFrames();
  this.view.show();
  this.setImage(…);
  this.level.next();
}
```

...driven by control flow

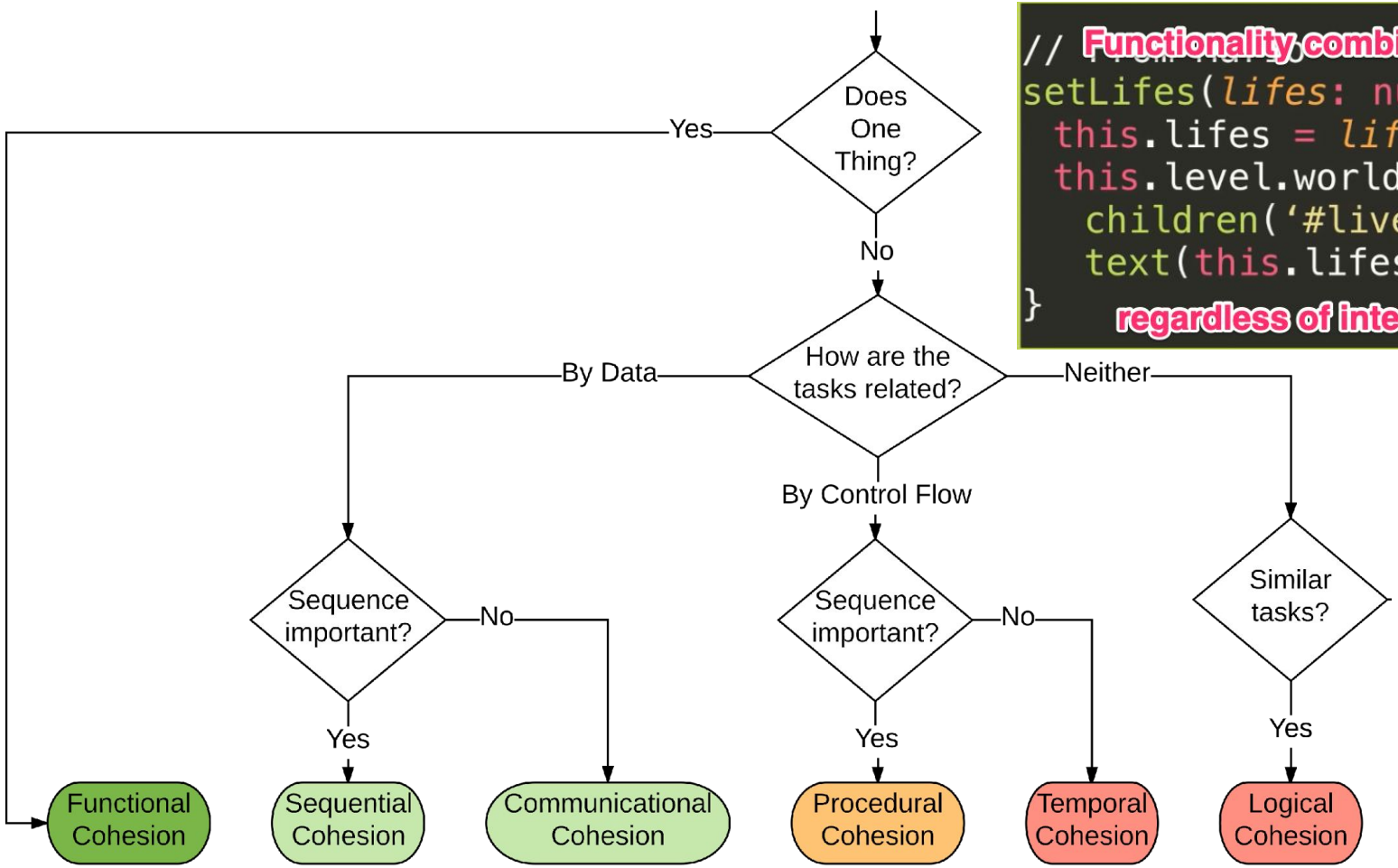
Better → Worse




```
reset() { // from Level
  this.active = false;
  this.world.empty();
  this.figures = [];
  this.obstacles = [];
  this.items = [];
  this.decorations = [];
}
```

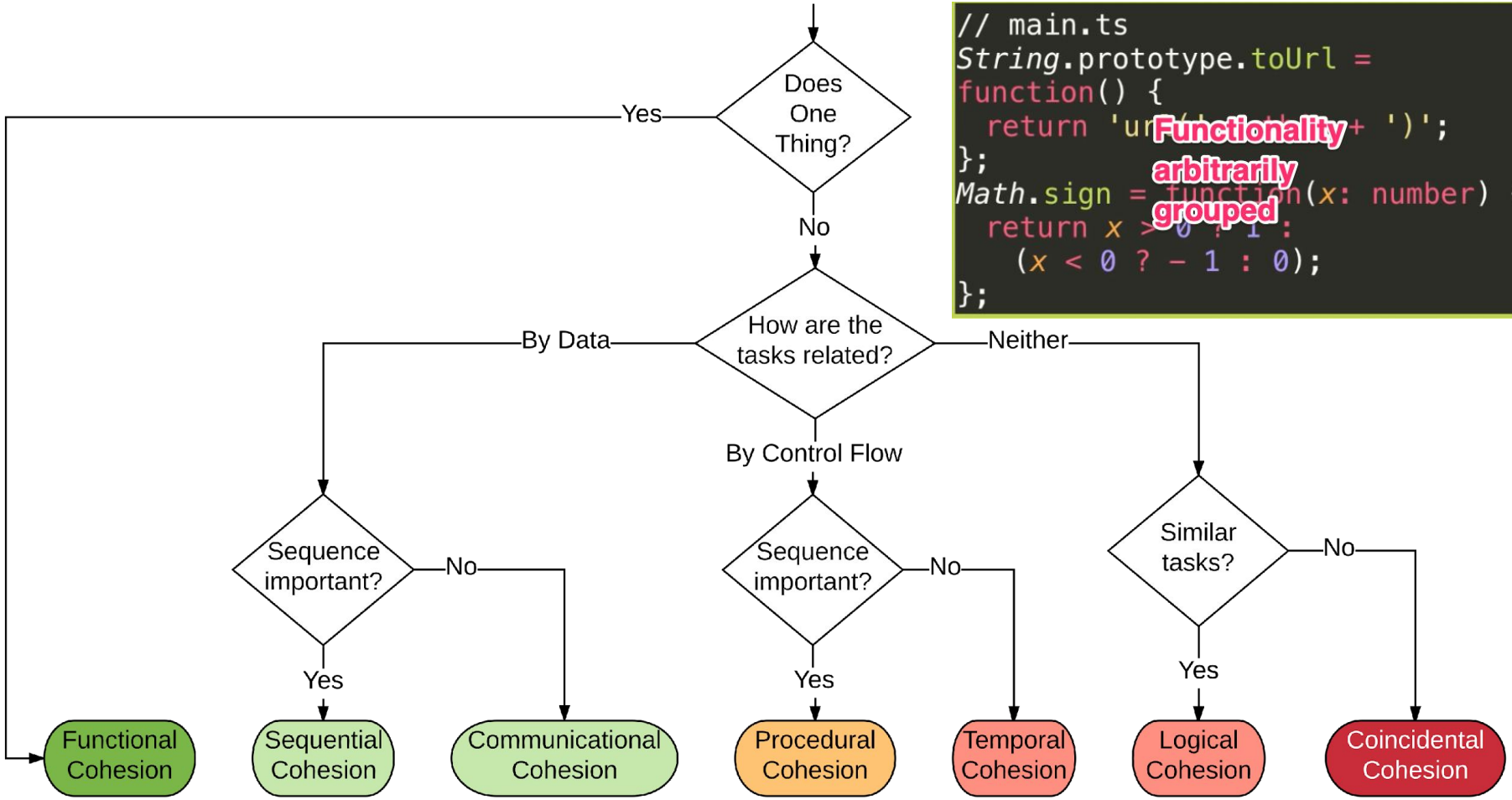
Functionality grouped because of execution timing

Better → Worse




```
// Functionality combined for task,
setLifes(lifes: number) {
  this.lifes = lifes;
  this.level.world.parent().
    children('#liveNumber').
    text(this.lifes);
}
```

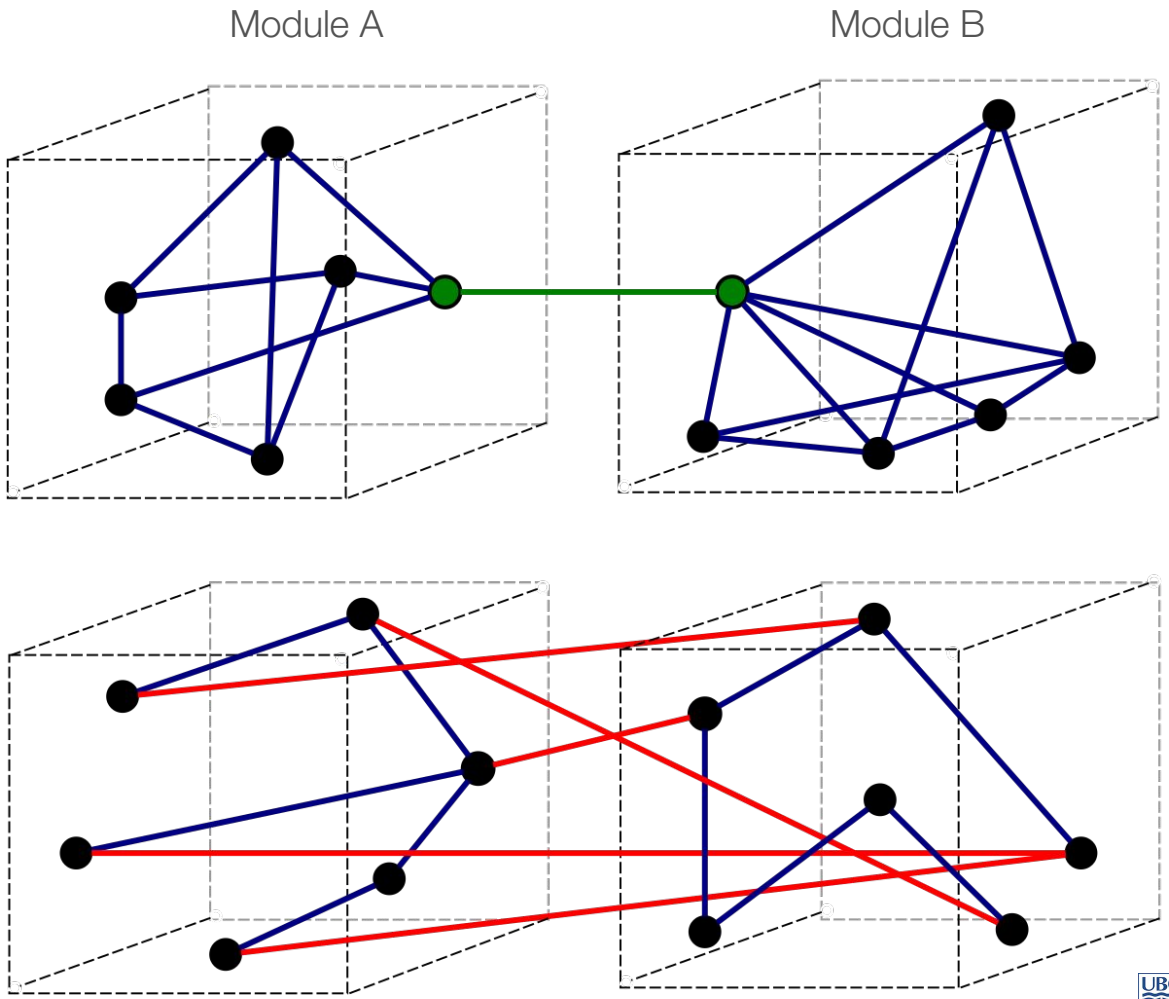
regardless of intent




```
// main.ts
String.prototype.toUrl =
function() {
  return 'ur(' + this + ')';
};
Math.sign = function(x: number) {
  return x > 0 ? 1 :
    (x < 0 ? -1 : 0);
};
```

Functionality arbitrarily grouped

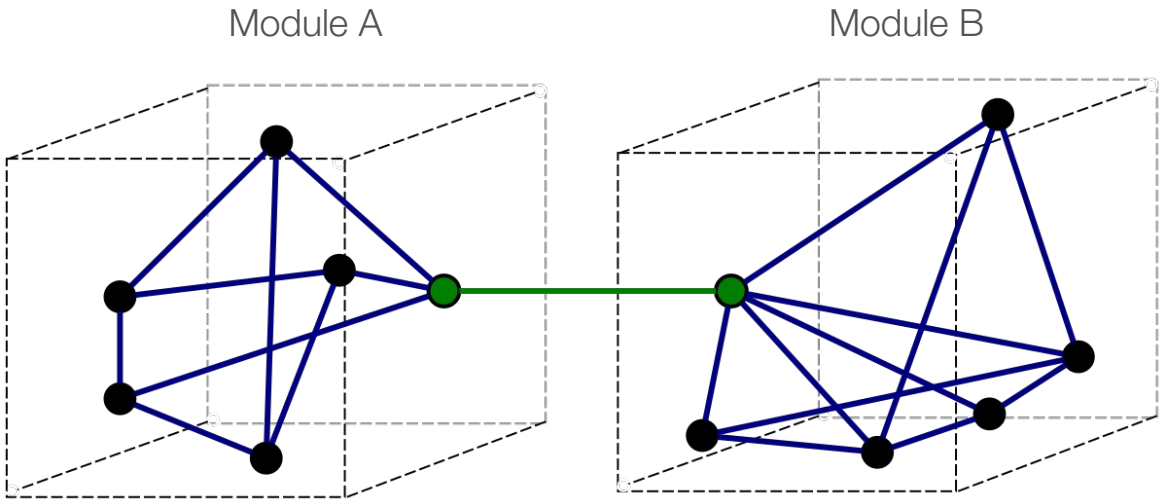
Balancing Coupling and Cohesion



[https://www.linkedin.com/pulse/software-engineering-principle-coupling-cohesion-mohammad-quanit]

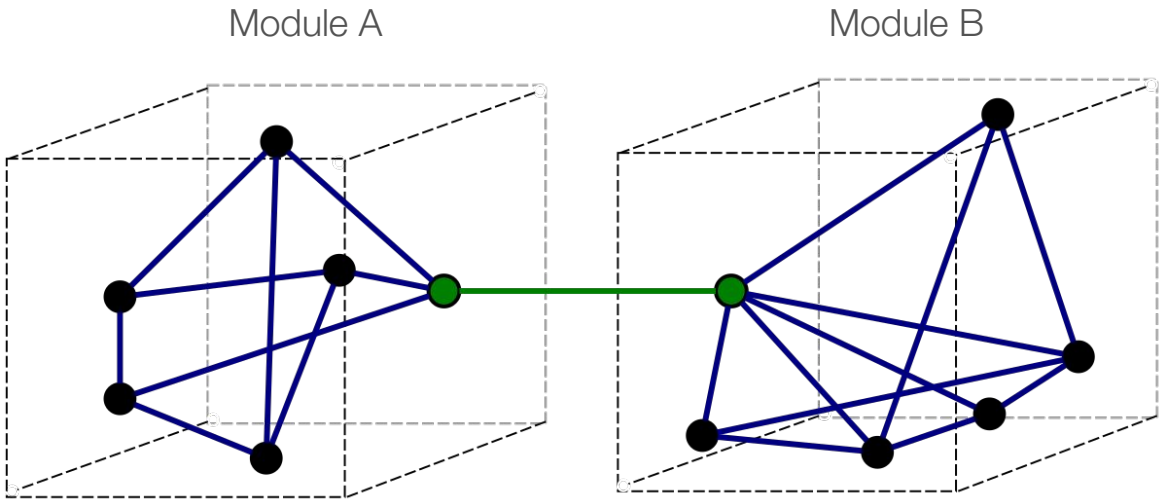
▼Coupling ▲Cohesion

- Lots of within-module collaboration.
- Controlled between-module collaboration.



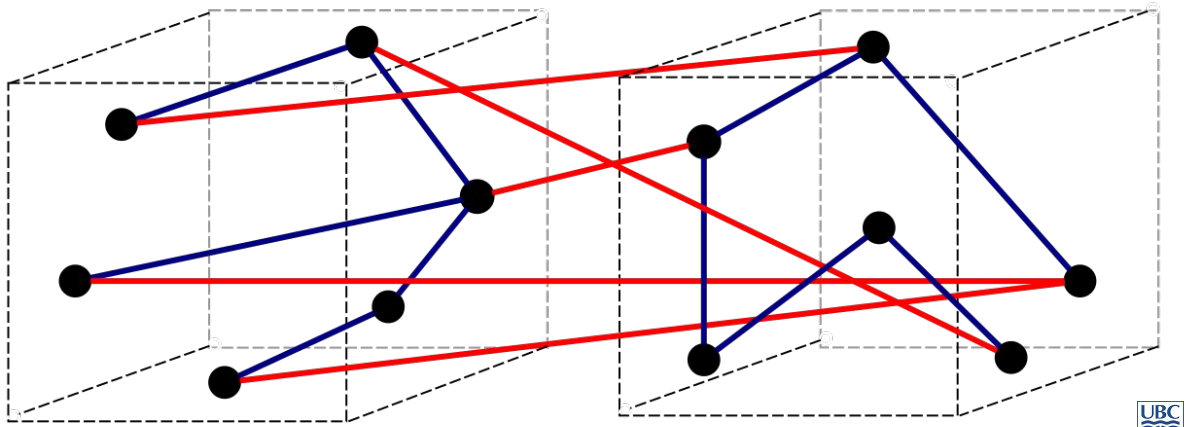
▼Coupling ▲Cohesion

- Lots of within-module collaboration.
- Controlled between-module collaboration.



▲Coupling ▼Cohesion

- Little within-module collaboration.
- Common between-module collaboration.




coupling


cohesion

Technical Representations

Topics

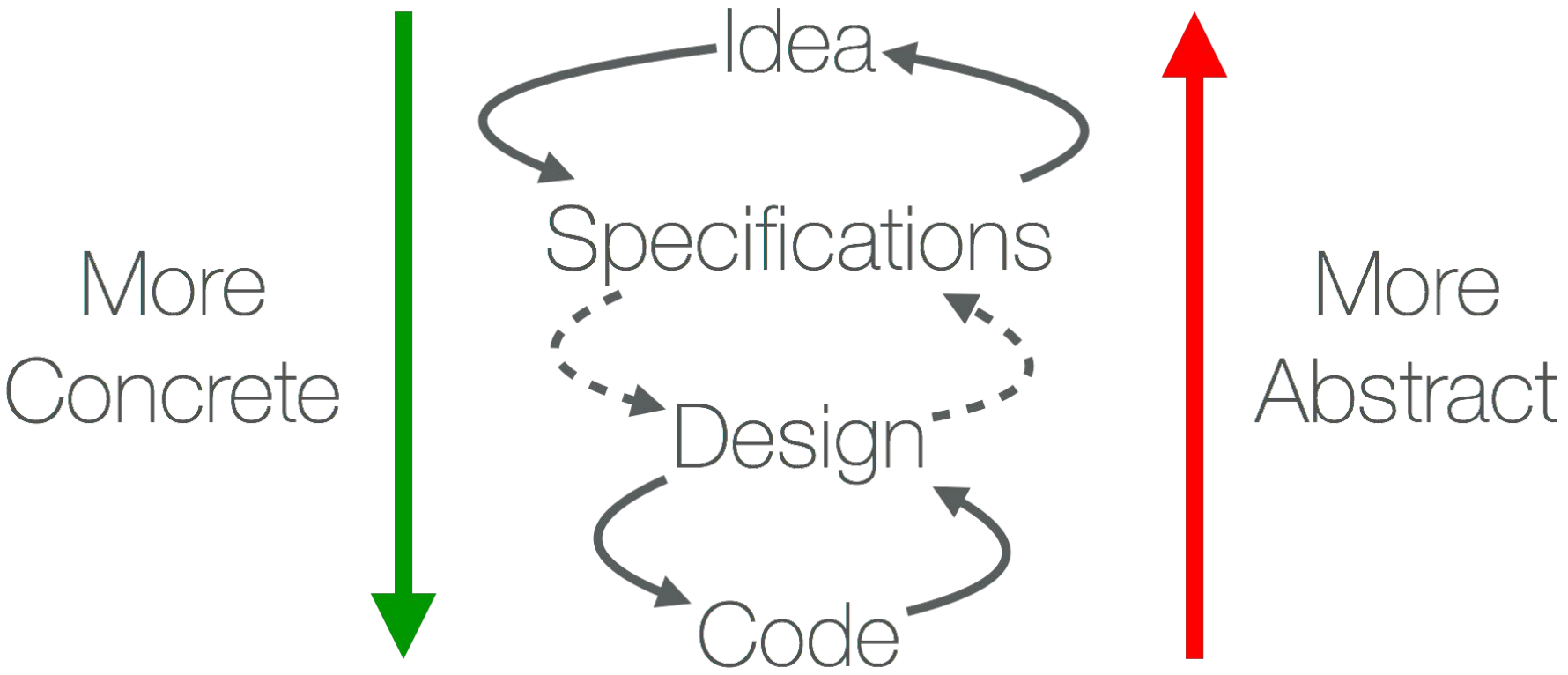
Examinable Skills

- Understand why technical representations are important.
- Be able to select and justify the appropriate technical representation for the task at hand.
- Be able to translate a technical description to a variety of different representations.
- Know how to derive the limitations of a design from one or more technical representations.

Architectural & Design Representations

- Abstractions that are fundamentally about **facilitating technical communication** between project stakeholders.
- An opaque abstraction or one abstracting the wrong detail has no value as it will not be adequately **understood.**
- Properties of representations:
  - **Ambiguity**: Open to more than one interpretation?
  - **Accuracy**: Correct within tolerances.
  - **Precision**: Consistent but not necessarily correct.

Moving Between Representations

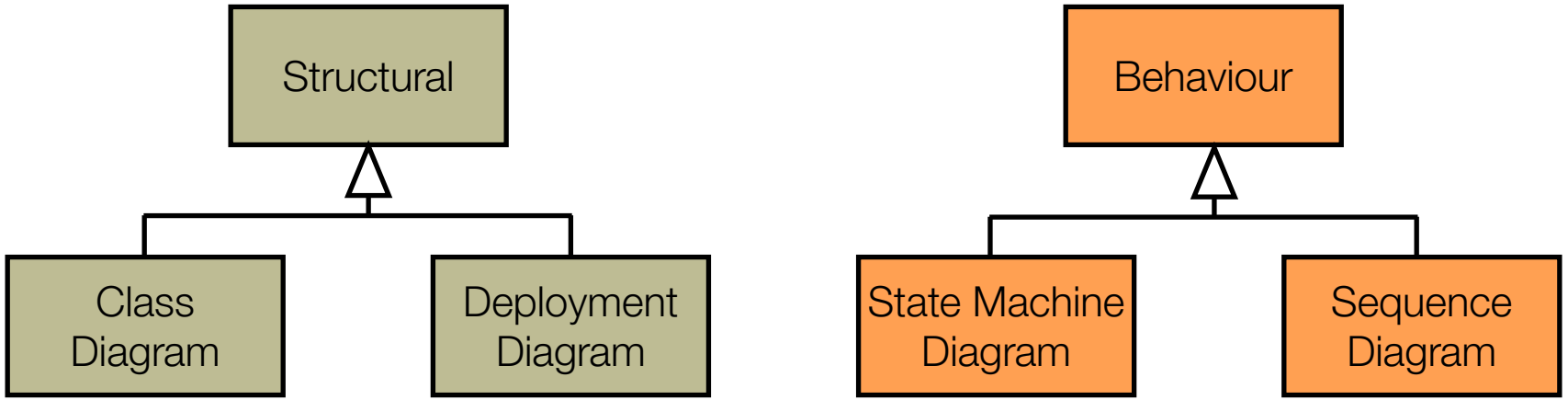


Alternate Views

- Abstractions are often presented as **‘views’** on the system.
  - Each view has a **specific goal** and focuses on subsets of elements and/or relationships.
  - Often focus on **specific concerns**, scenarios, or stakeholders within a system.
- Views **overlap**; maintaining **consistency** between views is challenging.

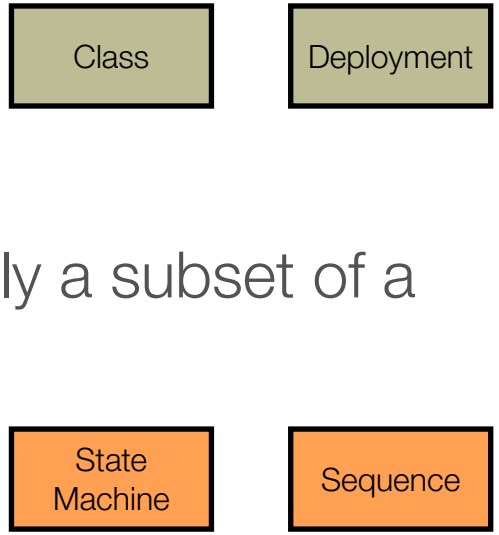
Unified Modelling Language (UML)

- Representing software systems is hard.
- UML provides a language for recording design decisions.
- Supports many alternatives views (diagrams).



Static Structure vs. Dynamic Behaviour

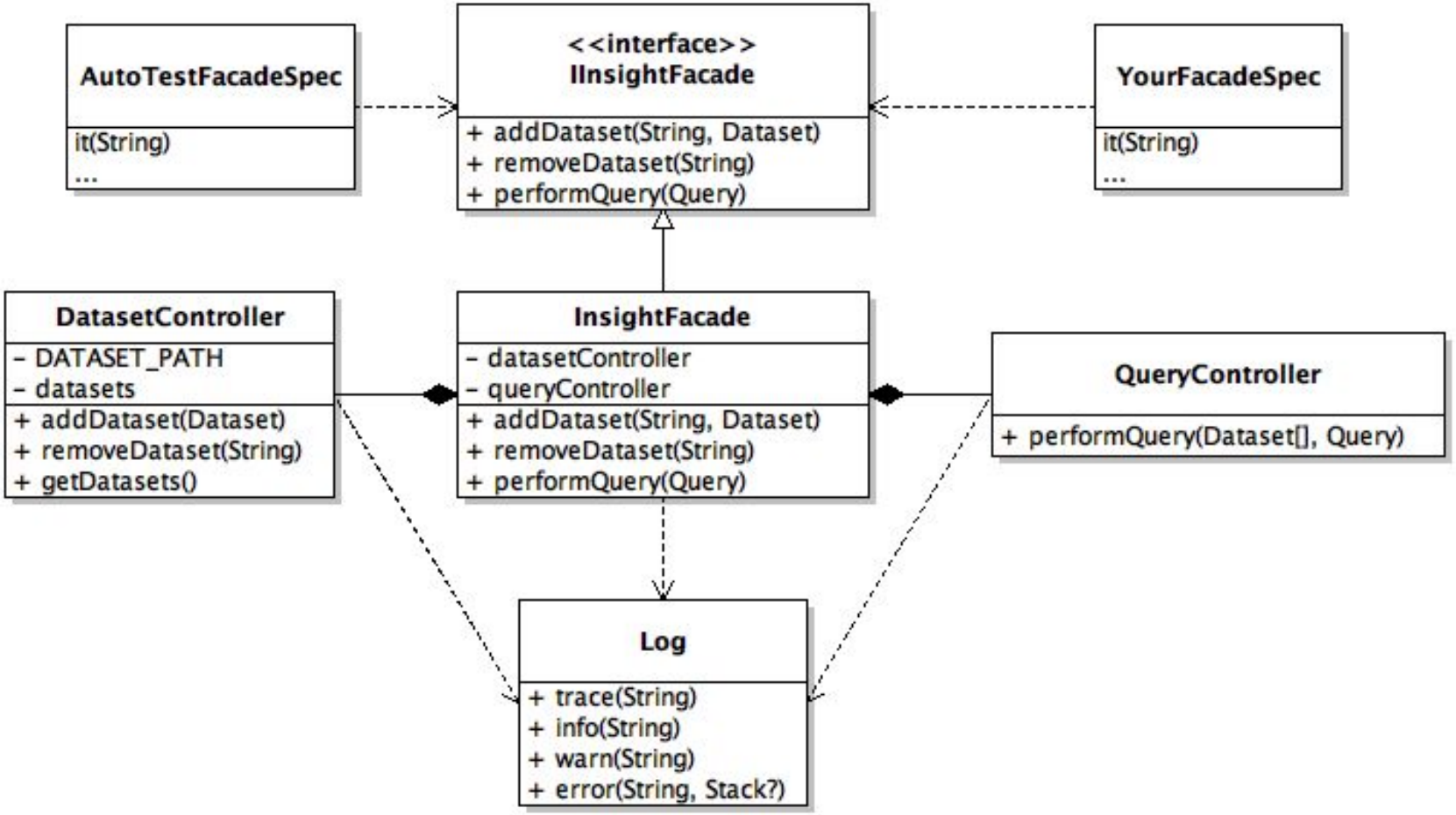
- Not necessary to compile & deploy.
  - Is it sufficient?
- Behaviour matters. Static relationships are only a subset of a complete system.
- Behaviour is inherently dynamic:
  - The code alone may not be sufficient.
  - Debugging only gives glimpses in time.
- Increased abstraction == decreased control.



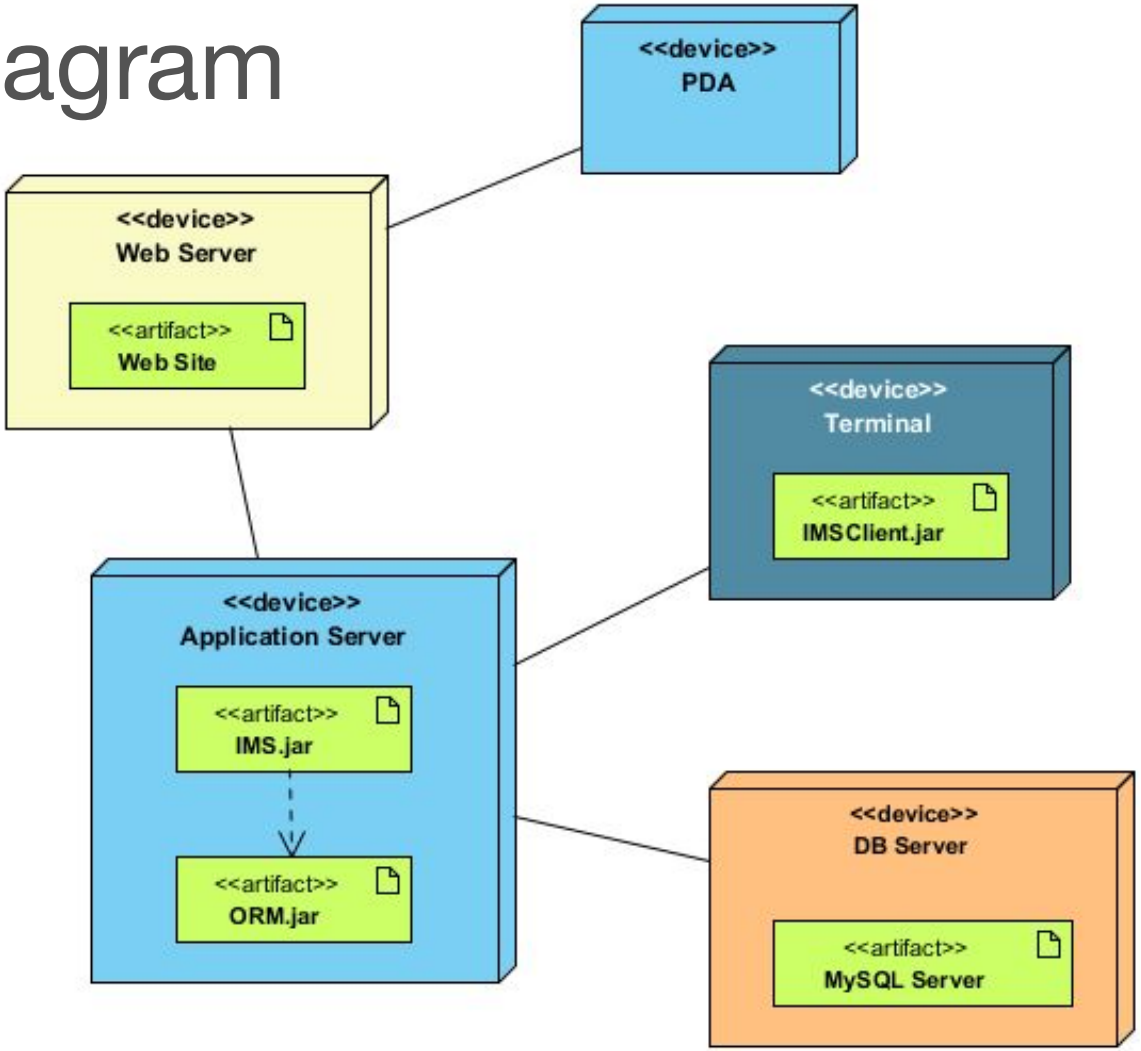
Class Diagram

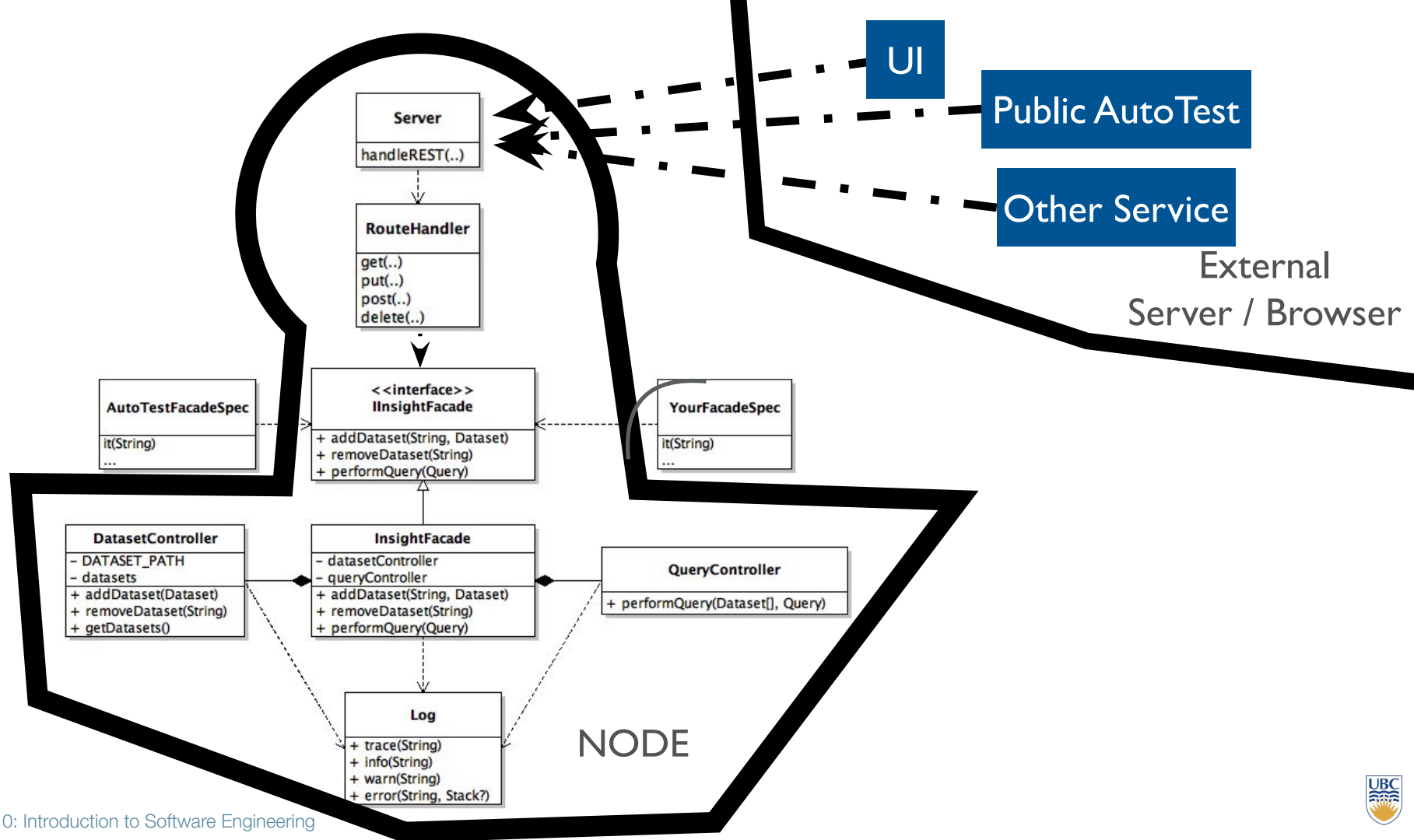
- Most common UML diagram.
- Capture the static structure of the system.
- Provide details about key methods, fields, and relationships.
- Compact representation; key development view:
  - Names
  - Types
  - Visibility
  - Inheritance
  - Ownership
  - Dependencies

InsightUBC Class Diagram



Deployment Diagram

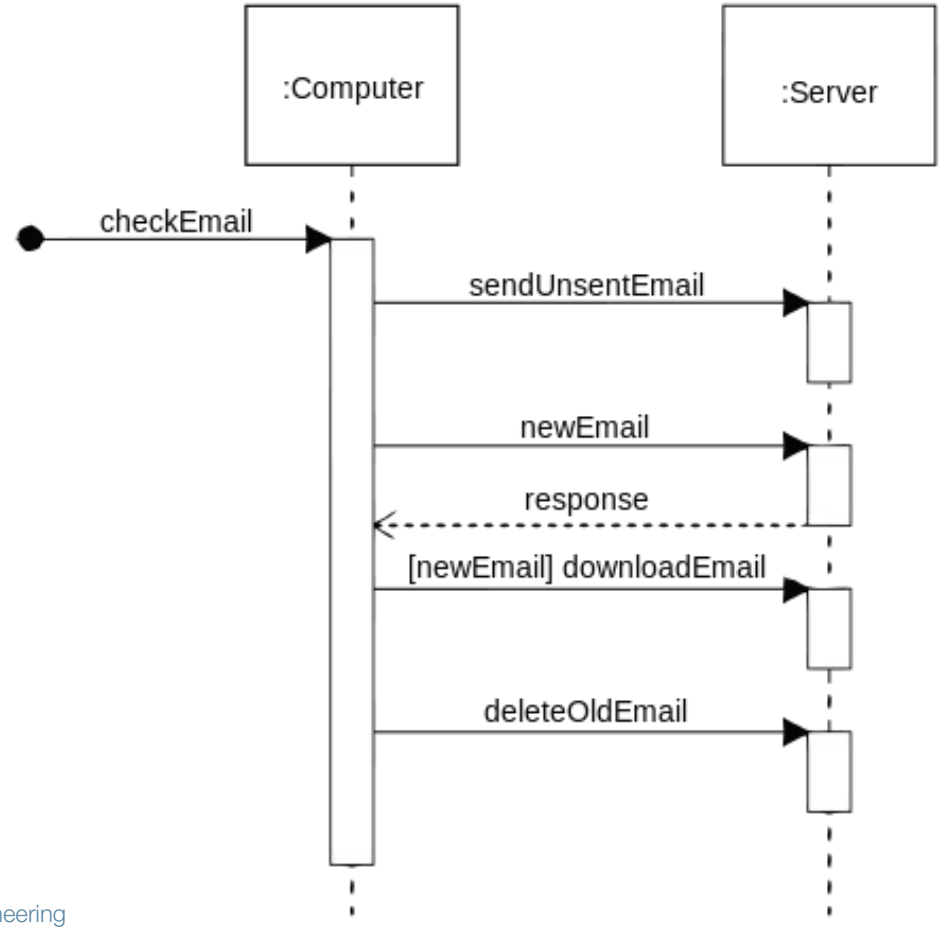



UI
Public AutoTest
Other Service
External
Server / Browser
Server
handleREST(..)
RouteHandler
get(..)
put(..)
post(..)
delete(..)
AutoTestFacadeSpec
it(String)
...
<<interface>>
IInsightFacade
+ addDataset(String, Dataset)
+ removeDataset(String)
+ performQuery(Query)
YourFacadeSpec
it(String)
...
DatasetController
- DATASET_PATH
- datasets
+ addDataset(Dataset)
+ removeDataset(String)
+ getDatasets()
InsightFacade
- datasetController
- queryController
+ addDataset(String, Dataset)
+ removeDataset(String)
+ performQuery(Query)
QueryController
+ performQuery(Dataset[], Query)
Log
+ trace(String)
+ info(String)
+ warn(String)
+ error(String, Stack?)
NODE

Sequence Diagram

- Codify how elements communicate with each other:
  - Dynamic view; how do static structures interact?
  - Good for showing protocols & interactions.
  - Enables concrete reasoning about dataflow.
- Almost always captures a specific task.
- Particularly useful for async and distributed flows.

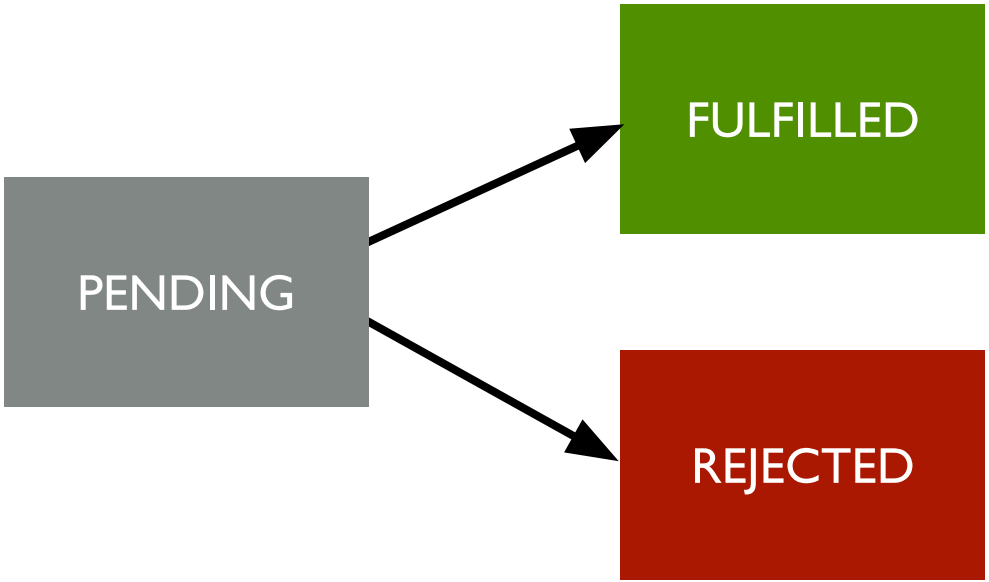
Sequence Diagram



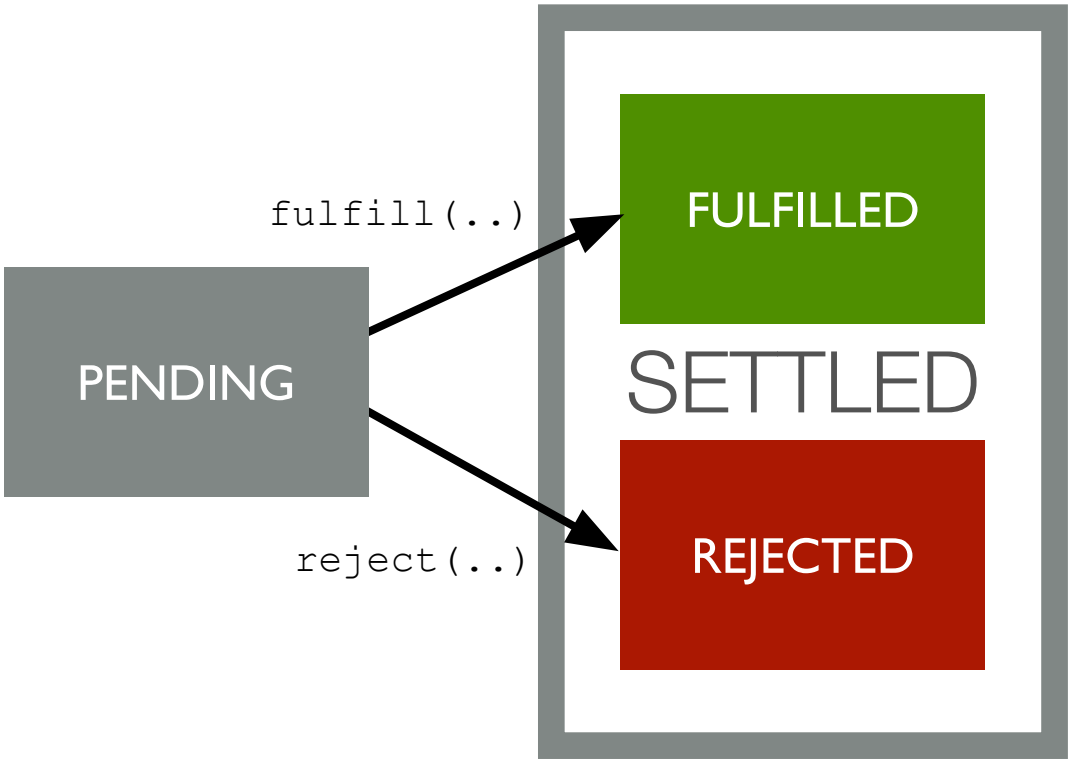
State Machine Diagram

- Used to model complex behaviours and protocols.
- Extremely useful for event-based modelling.
  - Essentially every UI.
  - From an app-perspective states can often be thought of as screens / pages. Often used for reasoning about user flows.
- Can be used to reduce the complexity of event systems.
  - e.g., reason about when events are applicable.
  - Transitions run to completion (like JavaScript methods).

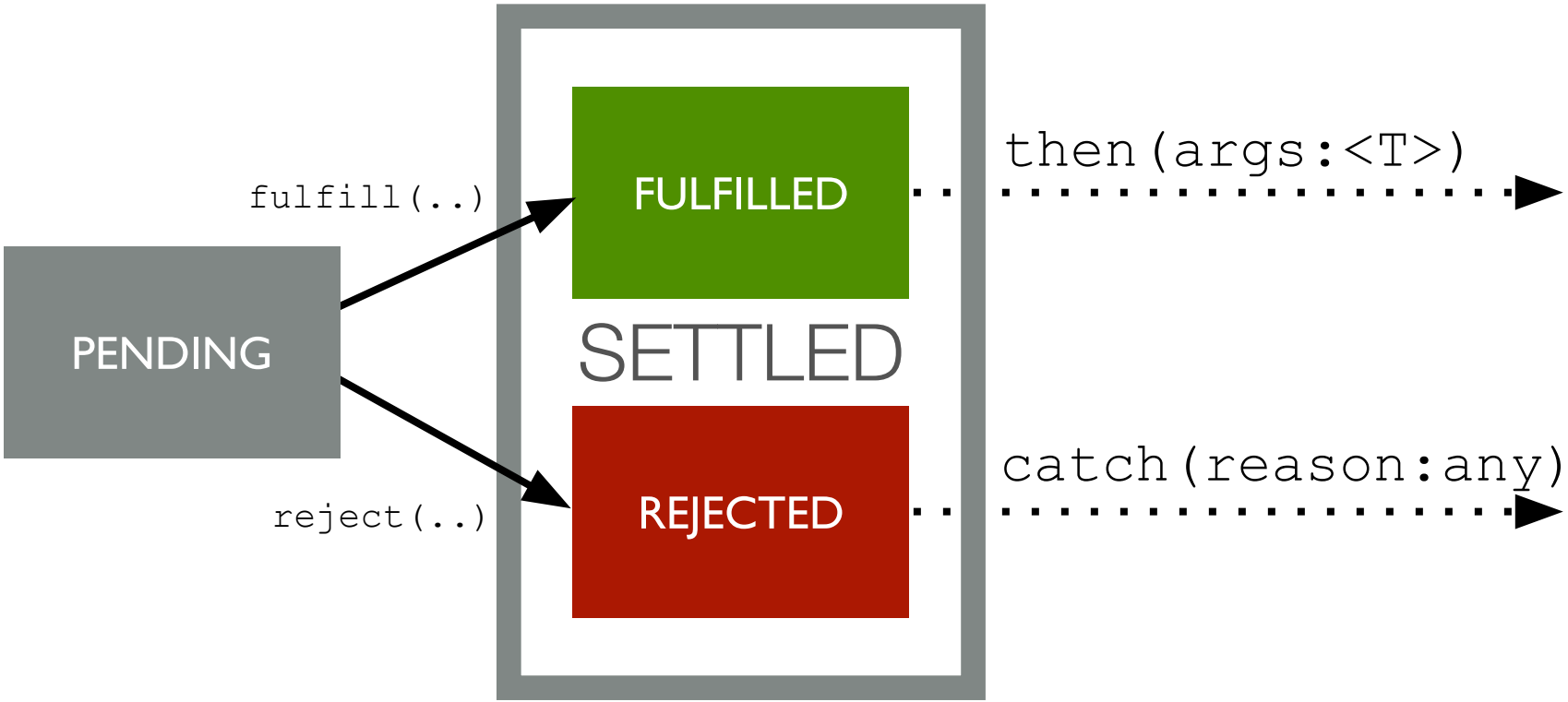
Promises



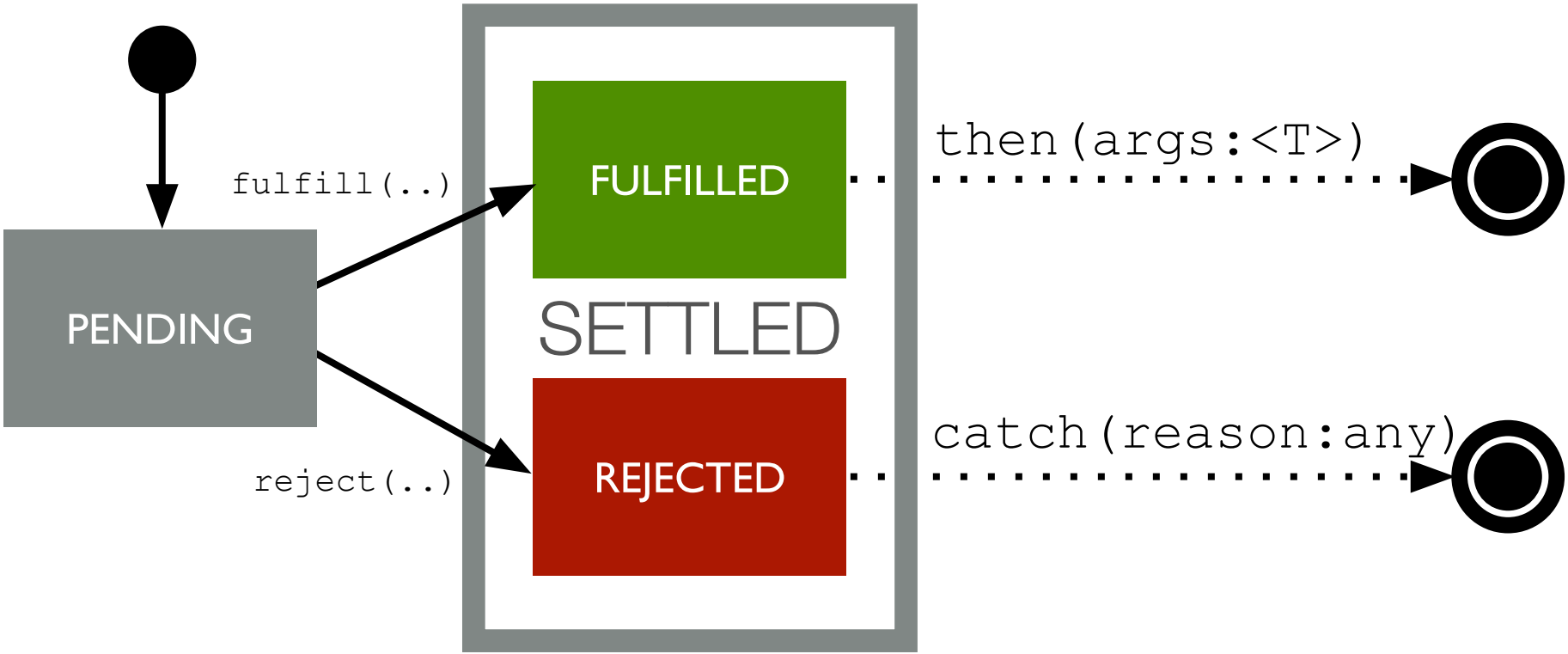
Promises



Promises


fulfill(..)
FULFILLED
then(args:<T>)
PENDING
SETTLED
reject(..)
REJECTED
catch(reason:any)

Promises

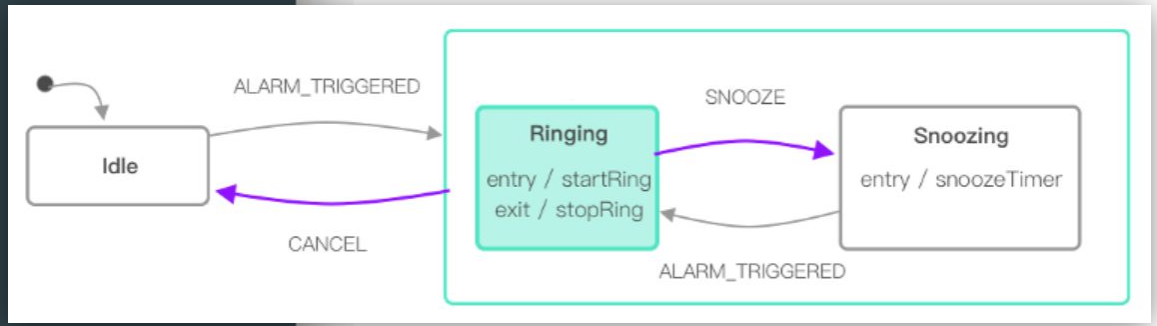

fulfill(..)
FULFILLED
then(args:<T>)
PENDING
SETTLED
reject(..)
REJECTED
catch(reason:any)

UI States

```
import { State, withStatechart } from 'react-automata'
import alarmMachine from './alarmMachine'

const AlarmClock = () => (
 <>
   <State value='Ringing'>
     <Clock ringing />
   </State>
   <State value=['Idle', 'Snoozing']>
     <Clock />
   </State>
 </>
)

export default withStatechart(alarmMachine)(AlarmClock)
```



Using Diagrams in Practice

- Different diagrams are used for different reasons during design and development. Diagrams are often short-lived and used to reason about a concrete problem.
- Choosing the right diagram for the task you are trying to solve is important. While you will probably use class diagrams the most, having the dynamic diagrams at your disposal is often extremely helpful for tricky code.
- Diagrams will vary depending on your target audience, e.g. developers, UX designers, marketing, security, administrators, and what goal you are trying to discuss.

Current Status

C2 Due Friday @ 1800.

Expecting higher AutoTest latency than C1.

J2 Will be released before C2 due date.

C3 Will be released before C2 due date.

P2 Will be released on Wednesday.

Q2 Next week (Wed-Sat).

Information Security

Topics

Examinable Skills

- Reason about how security requirements interact with software designs and implementations.
- Know how to ideate about security threat scenarios.
- Estimate the risk associated with security threats.
- Describe and apply mitigation principles (defense in depth, least privilege, etc) to abstract and concrete designs.
- Describe how the Chrome team has thought about and applied mitigation principles.
- Be able to outline how system security can be validated.

iPhone crashing 'prank' video bug and our uncertain security future

We're facing an uncertain future

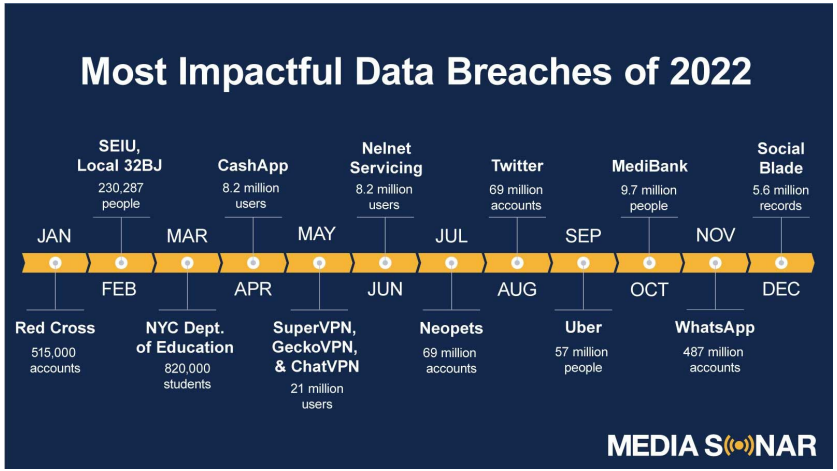
Computerworld | Nov 22, 2016 7:21 AM PT

iPhone crashing 'prank' video bug and our uncertain security future

We're facing an uncertain future

Computerworld | Nov 22, 2016 7:21 AM PT

Steps to lock up an iPhone:
1) Open a video.
2) Profit.



Average cost of a data breach in the healthcare industry (USD)

$10.1M

Increase in average breach cost in healthcare from 2020–2022

42%

IBM

Cost of a Data Breach Report 2022

Cost of a data breach 2022
https://www.ibm.com/reports/data-breach?utm_content=SRCWW

Data Breaches in October 2023

D-Link

Successful phishing attack, breaching records on a server that reached end of life in 2015, though the information itself was "of low-sensitivity and semi-public". Around 700 records breached

October 2

ROCK COUNTY WISCONSIN

Ransomware attack that encrypted files and took systems, including critical ones, offline. The attackers demanded $1.9 million (about £1.55 million), which the county refused to pay. Unknown, but no reason to assume "sensitive" personal data of employees was stolen.

October 2

23andMe

Credential stuffing attacks, resulting in initially 1 million data packs of Ashkenazi Jews leaked on a hacking forum, to which an additional 4.1 million of genetic data profiles of UK and German residents have now been added. 5,150,779 (1 million originally, plus an additional 4,150,779 from the hacker's update) records breached.

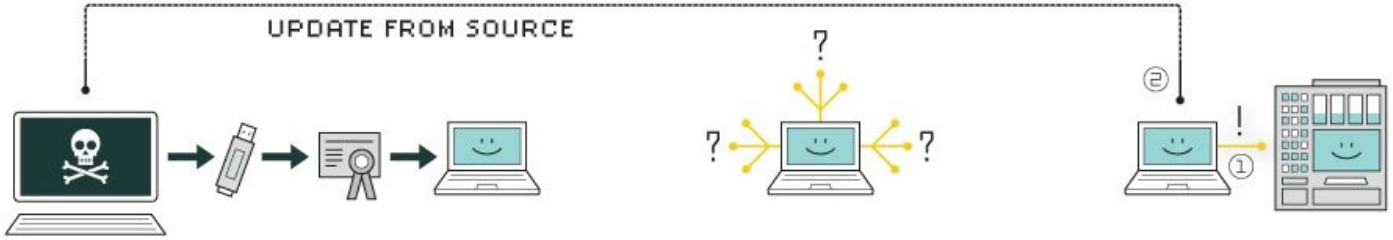
September 27

ClassPad.net

Unauthorised access to web application's server, leading to a personal data breach. The likely root cause was a misconfiguration – specifically, the disabling of

AirEuropa

The airline urged victims to cancel their credit cards following the hack. It also stated that no other personal information had been exposed in



1. infection
Stuxnet enters a system via a USB stick and proceeds to infect all machines running Microsoft Windows. By brandishing a digital certificate that seems to show that it comes from a reliable company, the worm is able to evade automated-detection systems.

2. search
Stuxnet then checks whether a given machine is part of the targeted industrial control system made by Siemens. Such systems are deployed in Iran to run high-speed centrifuges that help to enrich nuclear fuel.

3. update
If the system isn't a target, Stuxnet does nothing; if it is, the worm attempts to access the Internet and download a more recent version of itself.

4. compromise
The worm then compromises the target system's logic controllers, exploiting "zero day" vulnerabilities-software weaknesses that haven't been identified by security experts.

5. control
In the beginning, Stuxnet spies on the operations of the targeted system. Then it uses the information it has gathered to take control of the centrifuges, making them spin themselves to failure.

6. deceive and destroy
Meanwhile, it provides false feedback to outside controllers, ensuring that they won't know what's going wrong until it's too late to do anything about it.

Attacks are Increasingly:

- Frequent
- Effective
- Targeted
- Sophisticated
- Profitable
- Persistent
- Elusive

understanding
analysis
mitigation
validation

Understanding

Basic Terminology

- **Assets:** What is being secured.
- **Subject:** Who are the people using the system.
- **Policies:** The rules associated with the system.
- **Threats:** The reason we need policies.
- Dimensions:
  - **People:** Users, ops, dev, malicious actors.
  - **Process:** Architecture, design, implementation, monitoring.
  - **Technology:** How to secure the system.

Security Requirements

- Confidentiality
  - Preserving the confidentiality of information means preventing unauthorized parties from accessing the information or perhaps even being aware of the existence of the information.
- Integrity
  - Maintaining the integrity of information means that only authorized parties can manipulate the information and do so only in authorized ways.
- Availability
  - Resources are available if they are accessible by authorized parties on all appropriate occasions.
- Accountability
  - It should be possible to know how subjects interacted with sensitive systems and data.

Understanding: Security is a Huge Space

- Physical security:
  - Keyloggers
- System security:
  - Denial of service
- Network security:
  - Man-in-the-middle
- Human security:
  - Social engineering

Perfect security does not exist.

Ultimately security is all about managing risk.

What Can Be Compromised?

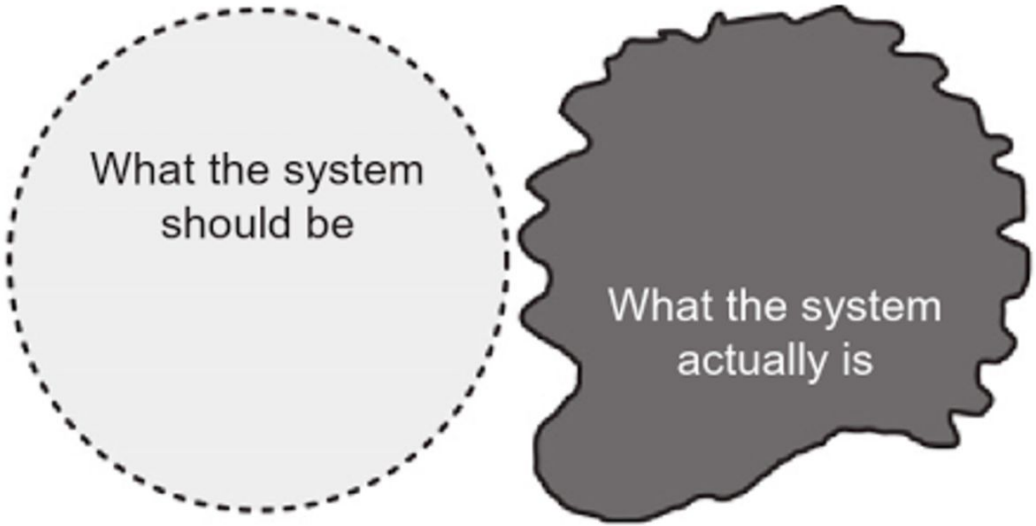
- Money:
  - Hardware costs.
  - Personnel time.
  - Customers.
  - Actual money
- Time.
- Privacy.
- Reputation.

Perfect security does not exist.

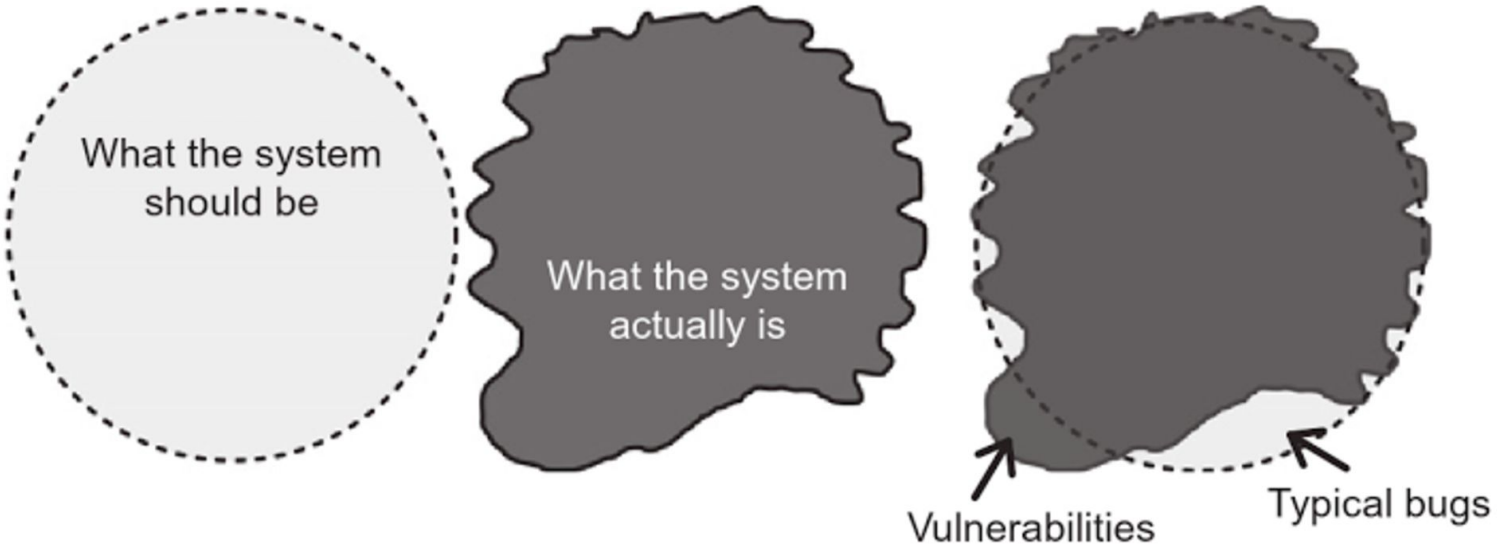
Ultimately security is all about managing risk.

Real Systems Are Really Messy

Real Systems Are Really Messy



Real Systems Are Really Messy



Analysis

Actors

- Bad actors can vary in their goals and sophistication:
  - Disgruntled employee
  - Automated malware
  - Curious attacker
  - Script kiddies
  - Motivated attacker
  - Hacktivitsts
  - Organized crime
  - Cyber terrorists
  - State actors
- It is important to realize that sometimes these threats do not arise from bad actors:
  - Unlucky users
    - Triggers race condition
  - Untrained users
    - Accident
    - Post-it note passwords

Threat Modelling

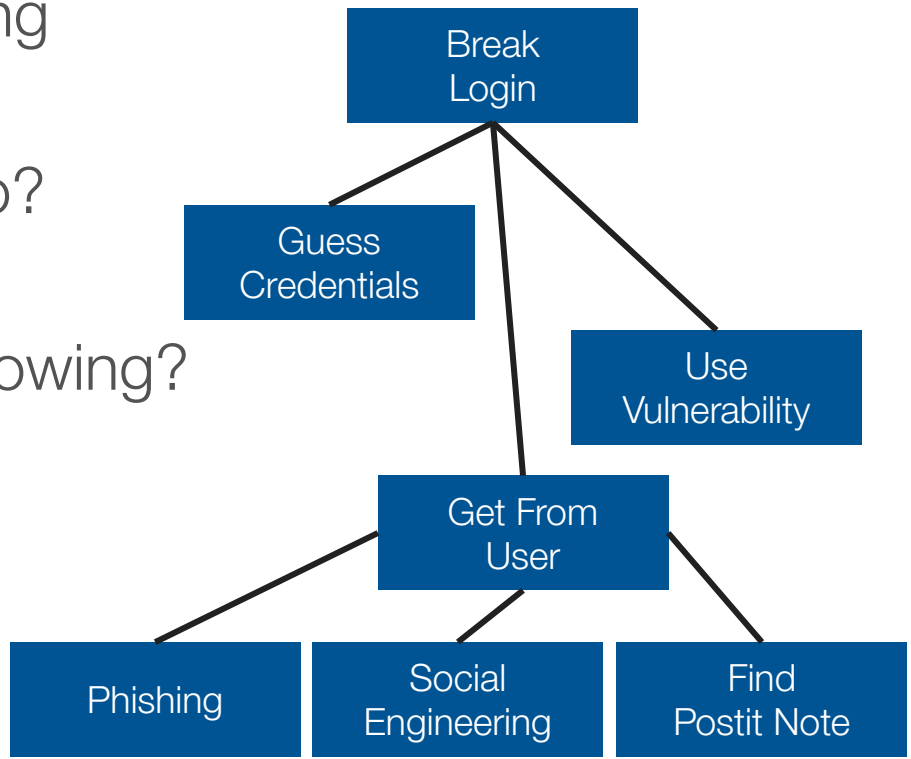
- Who are they?
- What are they trying to do?
  - Steal end-user data.
  - Direct financial gain.
  - Acquire/delete corporate data.
  - Damage reputation.
  - Steal intellectual property.
  - Decrease productivity.
  - Which vulnerabilities are they using?

Attack Tree

- A conceptual model for analysing security threats.
- Goal: What are they trying to do?
- Attacker: Who is trying to do it?
- Attack: What steps are they following?

Attack Tree

- A conceptual model for analysing security threats.
- Goal: What are they trying to do?
- Attacker: Who is trying to do it?
- Attack: What steps are they following?



risk =

threat x

exposure x

impact

DREAD

Damage: How bad would it be?

Reproducibility: How easily can it be reproduced?

Exploitability: How hard is it to do?

Affected Users: How many people impacted

Discoverability: How easy to discover

```
risk =
  damage(1..10)+
reproducibility(1..10)+
exploitability(1..10)+
   #users(1..10)+
discoverability(1..10)
```

Attack Surface

- The ‘size’ of your input/output mechanisms.
  - The fewer ways in, the easier it is to secure them.
- OWASP definition:
  - The number of i/o mechanisms.
  - The code securing those mechanisms.
  - The data being transmitted.
  - The code & mechanisms securing the data.
- Often conflicts with functional goals.

STRIDE

Coming up with meaningful scenarios for threats is hard.

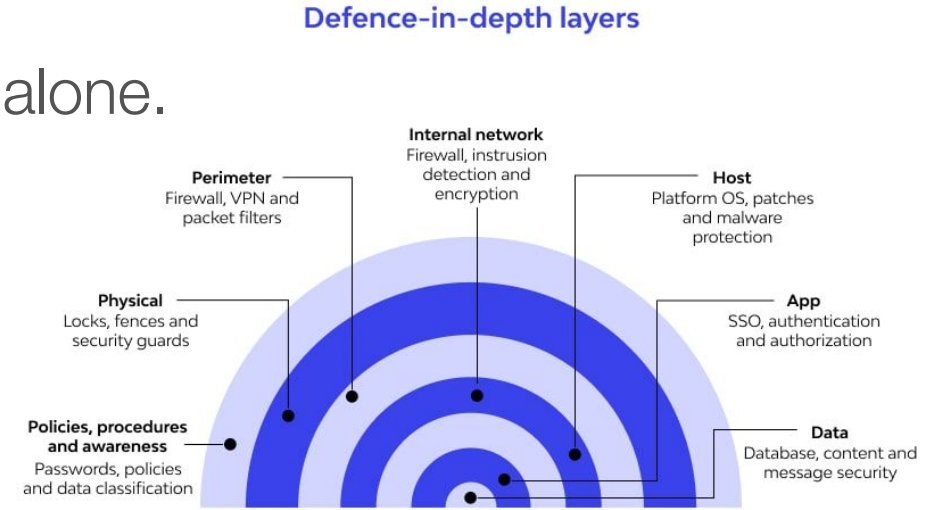
STRIDE provides a framework for ideating threat vectors:

- **S**poofing: Pretend to be someone else.
- **T**ampering: Modifying data.
- **R**epudiation: Denying performing a task.
- **I**nformation Disclosure: Access private information.
- **D**enial of Service: Halt service.
- **E**levation of Privilege: Gaining unauthorized privileges.

Mitigation

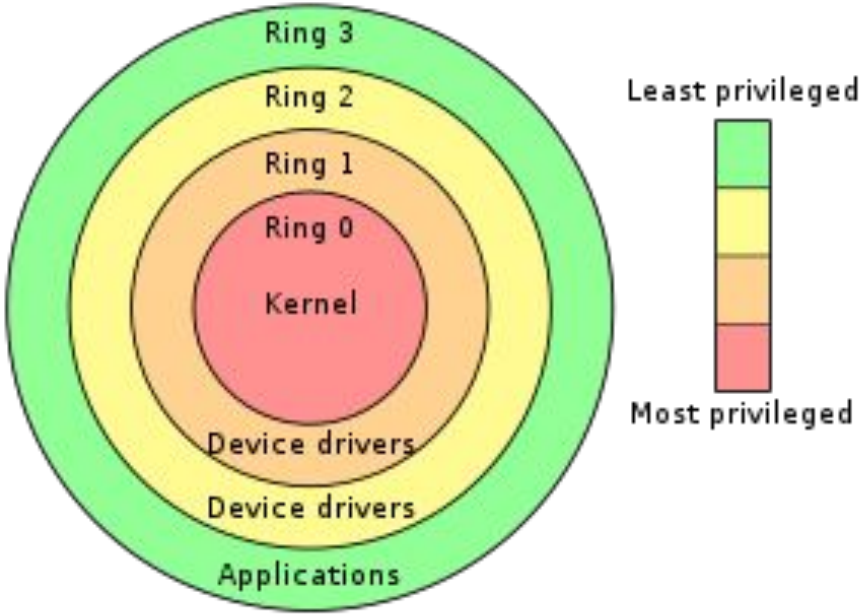
Defence in Depth

- A small breach shouldn't become a big one.
  - Need to minimize access within the system.
- Action:
  - Don't rely on border security alone.
  - Vary mechanism.
  - Avoid failure propagation.
- Downside:
  - Adds complexity, redundancy.



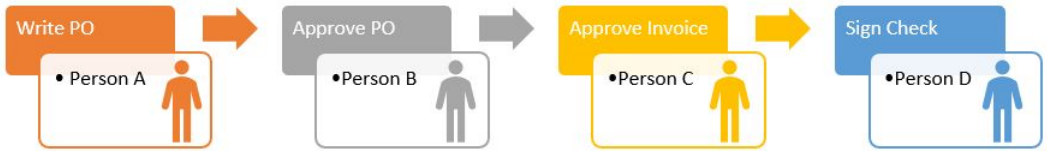
Least Privilege

- Coarse-grained privileges allow:
  - Malicious access.
  - Accidental access.
- Action:
  - Finer-grained privileges, roles, access control lists.
  - Microservices.
- Downside:
  - More complicated (in code and in usage).



Separation of Duties

- A single bad actor can cause great harm in a secure system.
  - High-impact tasks should be validated.
  - Also good for protecting against accidents.
- Action:
  - Introduce multiple steps/people for sensitive tasks.
  - Usually manual, but could also be automated.
- Downside:
  - Decreased velocity.

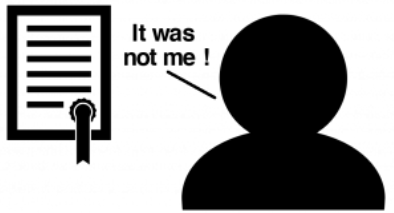


Strong Authentication

- If it is ‘easy’ for an adversary to gain access they will.
- Action:
  - Multi-factor authentication (commonly):
    - Know something.
    - Have something.
- Downside:
  - Velocity, complexity.

Non-Repudiation

- System users must be held accountable.
  - Not just for misbehaving users, also for tracking attackers.
- Action:
  - Atomic, write-only, auditing.
- Downside:
  - Audit logs are also sensitive information that could be compromised.



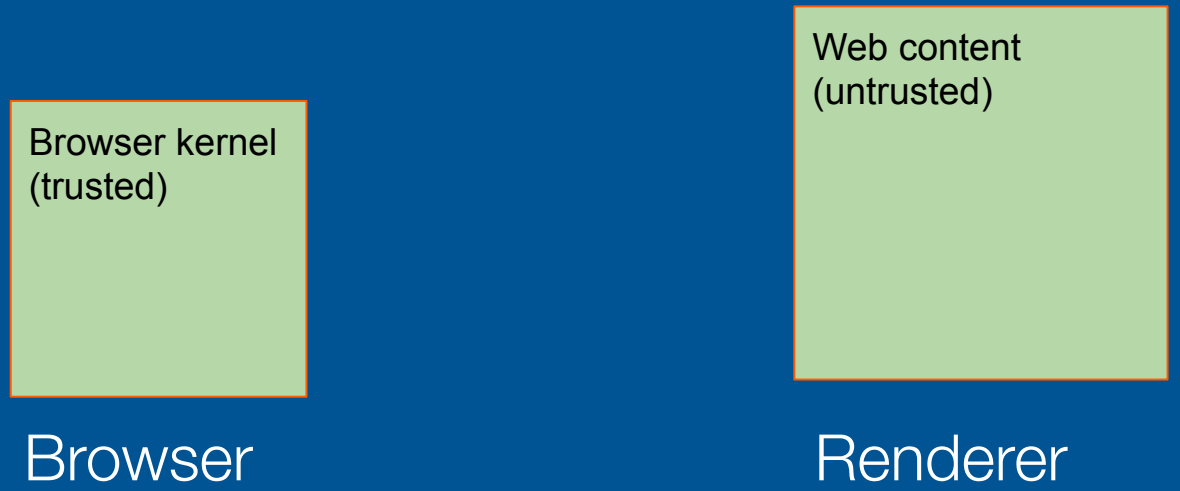
But this could never happen to

you

right?

Chrome presents many threats to core security requirements:

- **Confidentiality**: Leak user data.
- **Integrity**: Read/write arbitrary data on disk.
- **Availability**: Crash OS/host application.

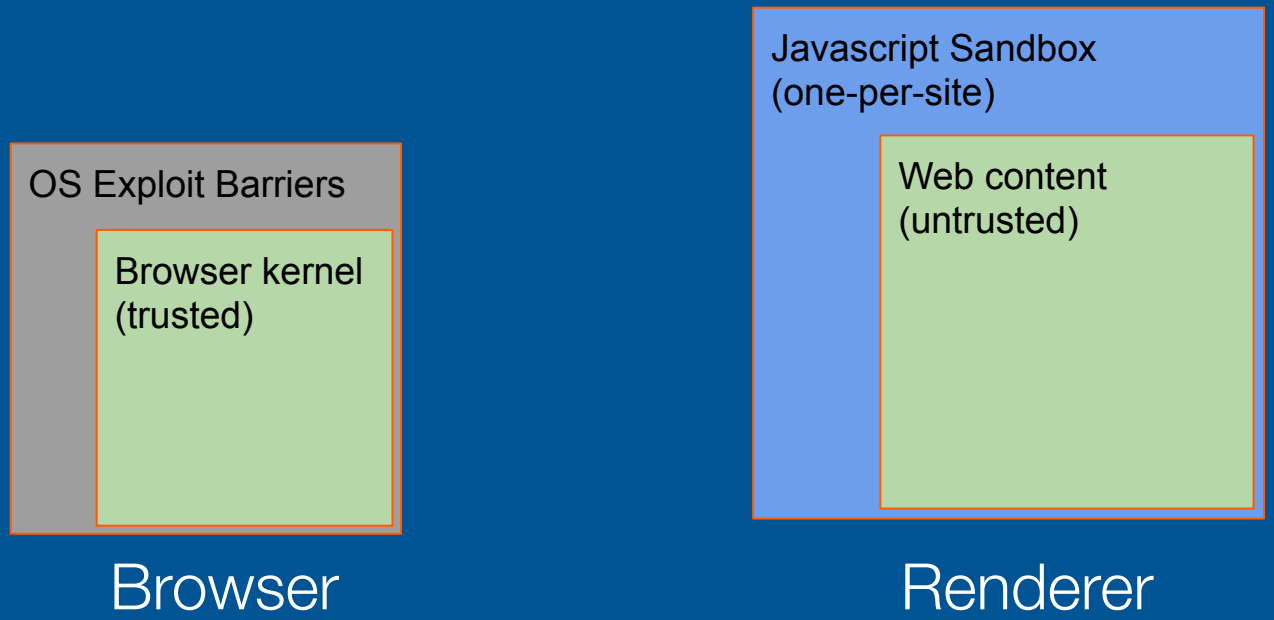


Chrome presents many threats to core security requirements:

- **Confidentiality**: Leak user data.
- **Integrity**: Read/write arbitrary data on disk.
- **Availability**: Crash OS/host application.

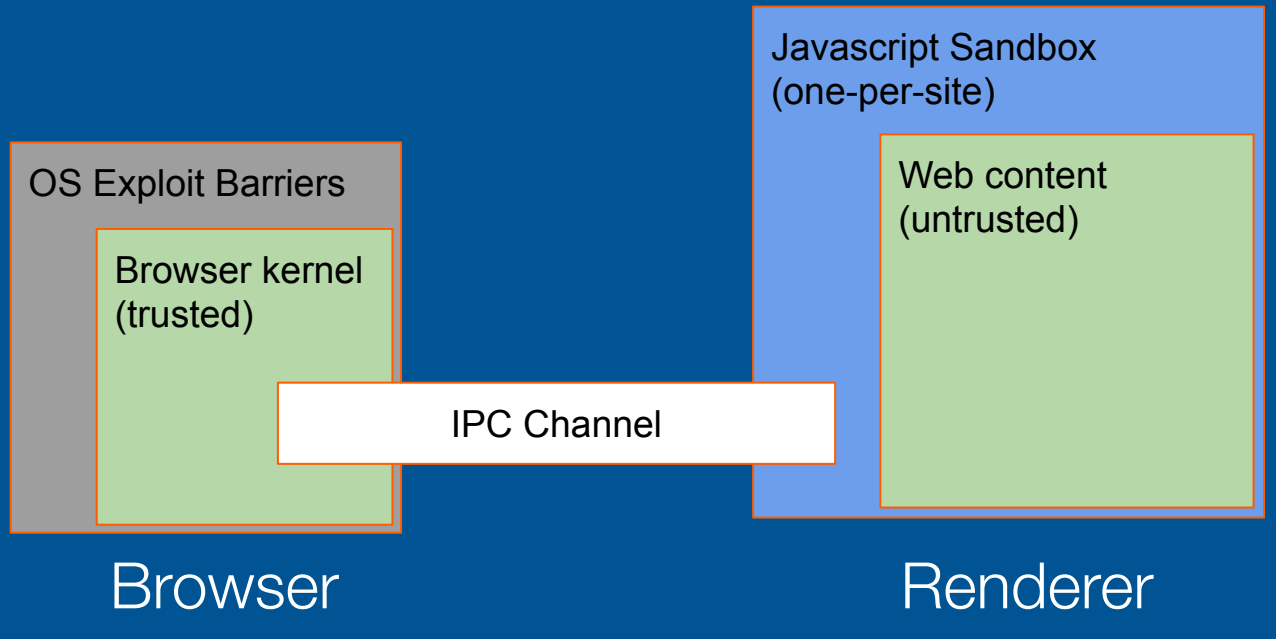
Chrome presents many threats to core security requirements:

- **Confidentiality**: Leak user data.
- **Integrity**: Read/write arbitrary data on disk.
- **Availability**: Crash OS/host application.



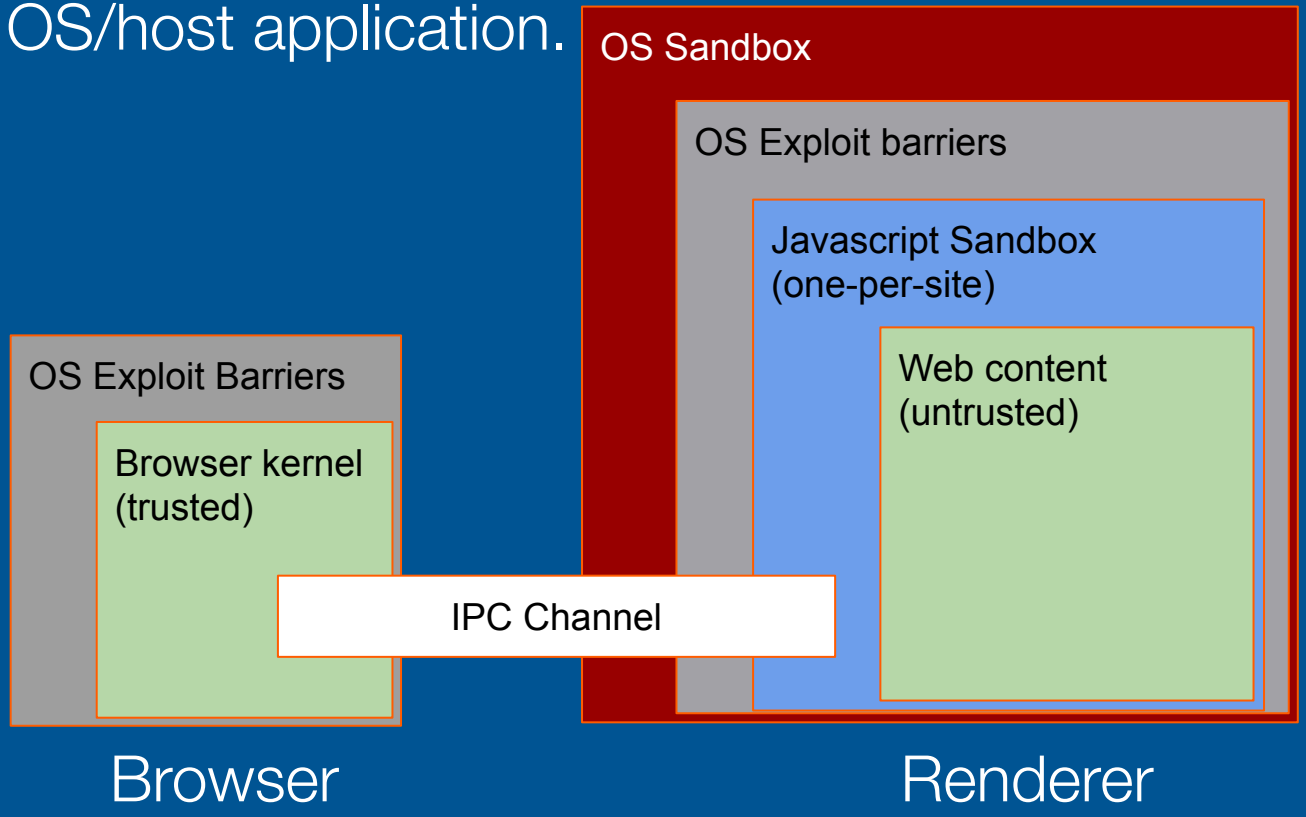
Chrome presents many threats to core security requirements:

- **Confidentiality**: Leak user data.
- **Integrity**: Read/write arbitrary data on disk.
- **Availability**: Crash OS/host application.



Chrome presents many threats to core security requirements:

- **Confidentiality**: Leak user data.
- **Integrity**: Read/write arbitrary data on disk.
- **Availability**: Crash OS/host application.



Security Infrastructure @ Google

- High-level goals:
  - Secure deployment of services.
  - Secure storage of data with end user privacy safeguards.
  - Secure communications between services.
  - Secure and private communication with customers.
  - Safe operation by administrators.
- All Google services are built on this infrastructure.

https://cloud.google.com/security/security-design/

Security Layers @ Google

- Operational security
- Internet communication
- Storage services
- User identity
- Service deployment
- Hardware infrastructure

Google Infrastructure Security Layers

Operational Security

- Intrusion Detection
- Reducing Insider Risk
- Safe Employee Devices & Credentials
- Safe Software Development

Internet Communication

- Google Front End
- DoS Protection

Storage Services

- Encryption at rest
- Deletion of Data

User Identity

- Authentication
- Login Abuse Protection

Service Deployment

- Access Management of End User Data
- Encryption of Inter-Service Communication
- Inter-Service Access Management
- Service Identity, Integrity, Isolation

Hardware Infrastructure

- Secure Boot Stack and Machine Identity
- Hardware Design and Provenance
- Security of Physical Premises

Validation

Validation

There are many approaches to help validate the security properties of your system:

Code review: Checking your code as as it is written / updated to ensure it conforms to security best practice is one of the first lines of defense.

Static analysis: Programs can analyze your source code to check for specific vulnerabilities and patterns that may disclose sensitive information.

Dynamic analysis: Executing programs against the running system is called dynamic analysis. These can include automated bots, fuzzers, or other malware-based tools.

Penetration testing: A specialized form of dynamic analysis whereby users explicitly try to break the security policies of a system using their own creative means. As with faults, this can show the presence of security weaknesses but not their absence. It is good practice to use pen testing, but bad practice to rely on it exclusively. Often uses *red teams* internally, or external actors.

SOLID

Topics

Modular Design Principles & Heuristics

- **SOLID** (one nice set of principles)
  - S **S**ingle Responsibility Principle
  - O **O**pen/Closed Principle
  - L **L**iskov Substitution Principle
  - I **I**nterface Segregation Principle
  - D **D**ependency Inversion Principle

Single Responsibility Principle

S O L I D

A class should have only a single responsibility. Specifically, only one potential change in the system's specification should be able to affect the implementation of the class.

Divergent changes often signal SRP violations because they suggest that one class is doing the work of two (or more).

SRP: Learning Outcomes

Be able to:

- Evaluate whether/how SRP principle is being adhered to.
- Spot glaring SRP violations.
- Understand different kinds of coupling and cohesion to reason about how they impede or enable evolution.

- **Divergent changes:** One class doing work of two.
- **Feature envy:** Envious code relies too heavily on the other class's internals.
- Link code smells to refactorings that lead to and improve SRP.
  - **Extract class** (use delegation).
  - **Move method** (reduce feature envy by re-homing methods).
  - Identify and explain how principles are connected (the violation of one may lead to the violation of another).
- Be able to repair principle violations through refactoring.

Open/Closed Principle

SOLID

A class must be **closed** for ***internal change***.
But must be **open** for ***extension.***

When designing classes, *do not plan* for brand new functionality to be added *by modifying the core* of the class.

Instead, design your class *so that extensions can be made* in a modular way, to provide new functionality by leveraging the power of the inheritance facilities of the language, or through pre-accommodated addition of methods.

OCP

What if you only wanted to specialise this bit?

We would have to override the entire method.

```
public init(): void {
    this.bp = new BeeperPanel();
    this.validate();
    try {
        const sampleRate = this.getParameter("sampleRate");
        if (sampleRate !== null) {
            const sR = Number.parseInt(sampleRate);
            this.bp.setSampleRate(sR);
        }
    } catch(err) { }
    try {
        const framesPerWave = this.getParameter("fpw");
        if (framesPerWave !== null) {
            const fpw = Number.parseInt(framesPerWave);
            const slider = this.bp.getFPWSlider();
            slider.setValue(fpw);
        }
    } catch(err) { }
    try {
        const harmonic = (this.getParameter("harmonic") !== null)
        this.bp.setupSound();
    } catch(err) { }
    try {
        if (this.getParameter("autoloop") !== null) {
            const loopCount = this.getParameter("autoloop");
            if (loopCount !== null) {
                const lc = Number.parseInt(loopCount);
                this.bp.loop(lc);
            }
        }
    } catch(err) { }
}
```

OCP Example

SOLID

Extract method refactoring

```
public init(): void {
  this.setupPanel();
  this.getSampleRate();
  this.getFramesPerWave();
  this.setupHarmonicsAndSound();
  this.setLoopCount();
}
```

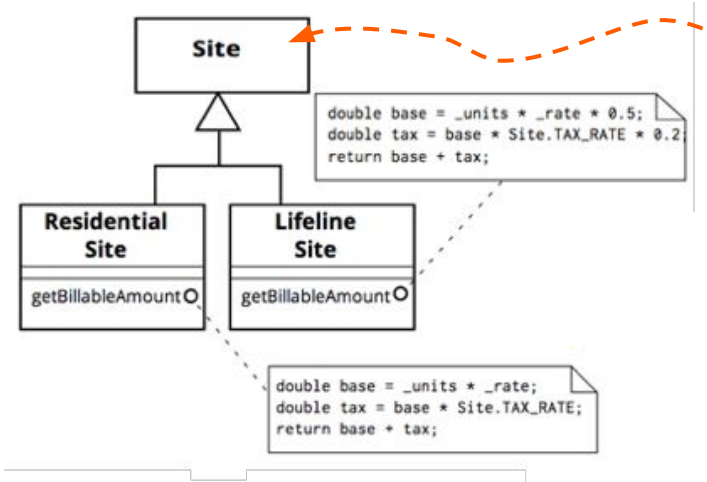
public enough

Why is this refactoring better?

If these methods are protected not private, we can *override* individual portions of the behaviour.

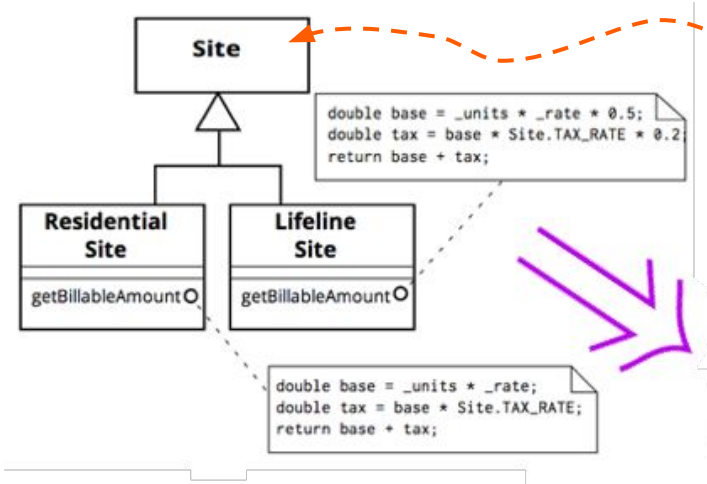
This gives us another motivation for *decomposing* long methods: to *facilitate* the Open/Closed principle!

OCP-Guided Refactoring

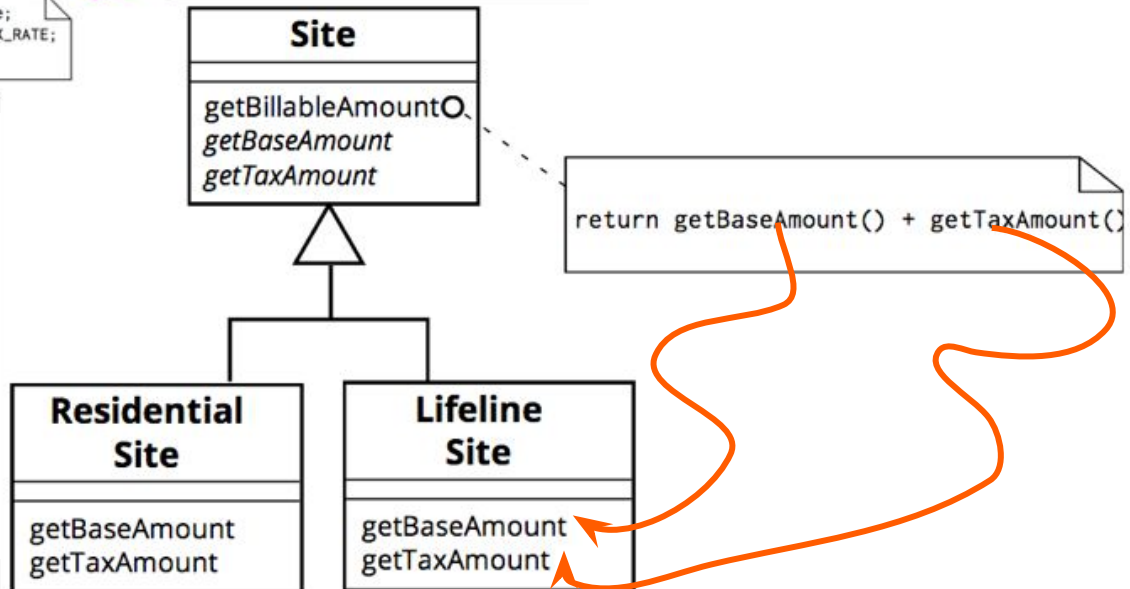


Violation: Open for modification :(. Subtypes must override/re-implement method entirely to specialise.

OCP-Guided Refactoring



Violation: Open for modification :(. Subtypes must override/re-implement method entirely to specialise.



This arrangement achieves the OCP. Site is *closed for modification* (no need to change its code), and *open for extension* (subclasses can specialise straightforwardly).

OCP Red Flag

SOLID

- Whenever your code is making behavioural changes based on internal **flags** or **instanceof** you are likely violating Open/Close. E.g.,

```
class BillingService {
 public chargeOrder(order: PizzaOrder, cc: CreditCard): Receipt {
   if (cc instanceof VisaCard) {
        …
   } else if (cc instanceof MasterCard) {
        …
   }
}
```


UBC

OCP: Learning Outcomes

S O L I D

Be able to:

- Evaluate whether/how OCP principle is being adhered to.
- Spot glaring OCP violations.
- Describe how smells can arise in designs that do not follow OCP:
  - E.g., Potentially: Shotgun surgery BETWEEN supertype and subtype, or across several subtypes all overriding the same method.
- Link code smells to refactorings that lead to principles:
  - E.g., refactor to template; extract method.
  - E.g., You are faced with a class that has a method that is very long, with multiple stages the order of which matters, and your subtype needs to change one part. What principle violation am I?
- Perform fixes of principle violations through refactorings.

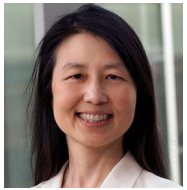
Liskov Substitution Principle

SOLID

Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

[Barbara Liskov and Jeanette Wing, A Behavioral Notion of Subtyping, ACM TOPLAS, 1811-1841, 1994.]

Barbara Liskov



Jeanette Wing

[http://en.wikipedia.org/wiki/Liskov_substitution_principle]

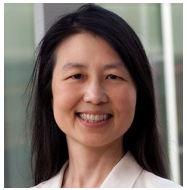
Liskov Substitution Principle

SOLID

Subtype Requirement: **Let ϕ(x) be a property provable about objects x of type T**. Then ϕ(y) should be true for objects y of type S where S is a subtype of T.

For a given property, ϕ(x), of a Supertype, T, ...

Barbara Liskov



Jeanette Wing

[http://en.wikipedia.org/wiki/Liskov_substitution_principle]

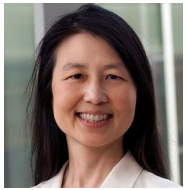
Liskov Substitution Principle

S O L I D

Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T. **Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.**

For a given property, $\phi(x)$, of a Supertype, T, **that property must hold for all Subtypes of T.**

Barbara Liskov



Jeanette Wing

[http://en.wikipedia.org/wiki/Liskov_substitution_principle]

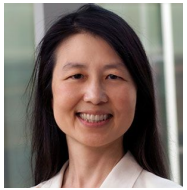
Liskov Substitution Principle

SOLID

Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

[Barbara Liskov and Jeanette Wing, A Behavioral Notion of Subtyping, ACM TOPLAS, 1811-1841, 1994.]

Barbara Liskov



Jeanette Wing

```
c: Cookie = new OatmealCookie();
c.eat()
```

If S is a subtype of T, then objects of type T (supertype) in a program may be replaced with objects of type S (subtype) *without altering* any of the desirable properties of that program.

[http://en.wikipedia.org/wiki/Liskov_substitution_principle]

What is LSP?

S O L I D

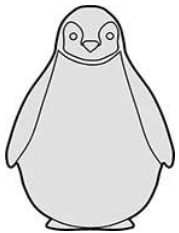
Substitutability:
A subclass should not break the expectations set by its superclass!

```
class GoodPerson {
    public freeBird(b: Bird) {
      b.flyFromHeight(100);
      // ... watch bird fly!
    }
}
```

```
class Bird {
    // all heights work!
    public flyFromHeight(height: number) {
      // ... some implementation
    }
}
```

```
class Seagull extends Bird {
    // all heights work!
    public flyFromHeight(height: number) {
      // ... some implementation
    }
}
```

```
class Penguin extends Bird {
    // heights under 50cm only!
    public flyFromHeight(input: number) {
      // ... another implementation
    }
}
```



Facets of LSP

SOLID

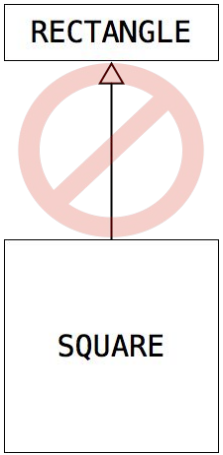
Substitutability
A subclass should not break the expectations set by its superclass!

Precondition Rule
Preconditions should not be strengthened (for the same inputs)
Ok to widen!

Postcondition Rule
Postconditions should not be weakened (for the same inputs)
Ok to narrow!

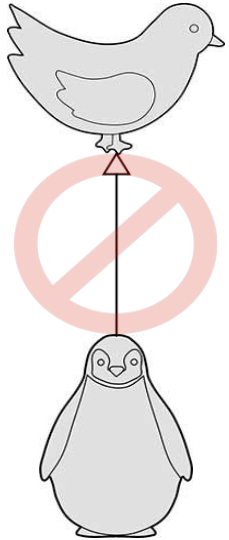
The Methods Rule

Resize only lets you specify one dimension.



Changing width will also change height.
Bad: widened postcondition!

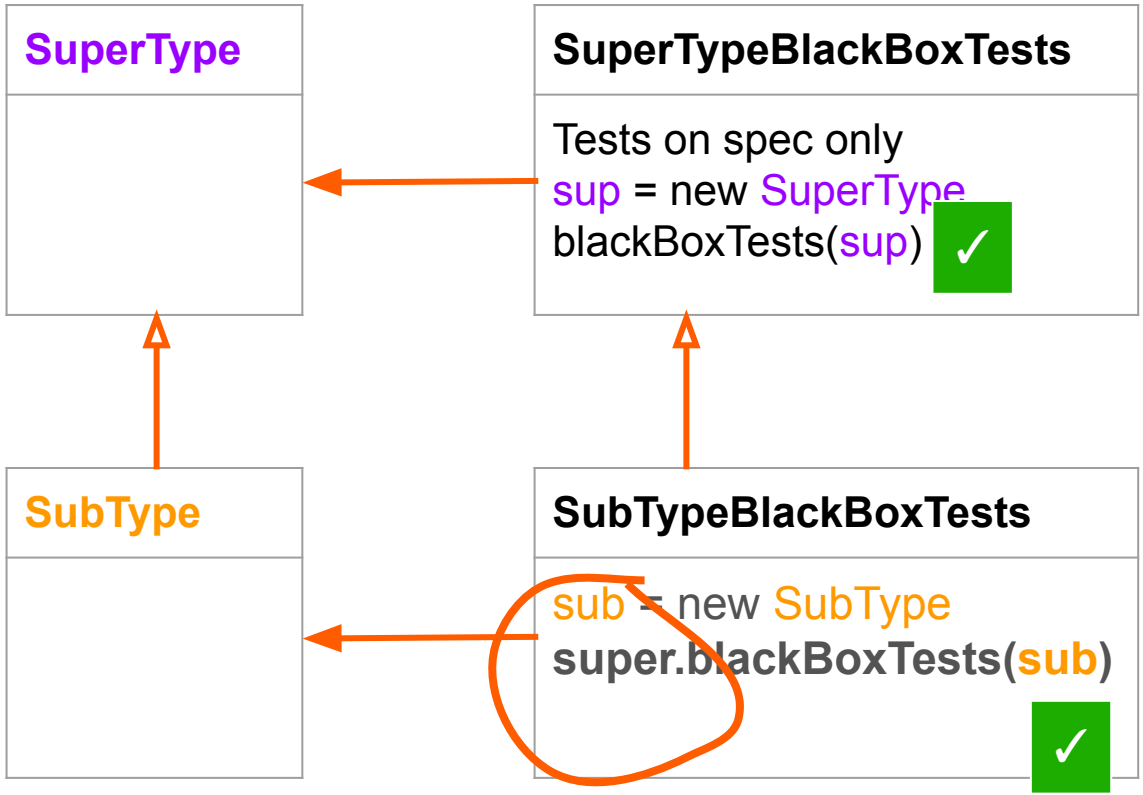
Can fly.



Additional effect: Penguin perishes
Bad: Widened postcondition!

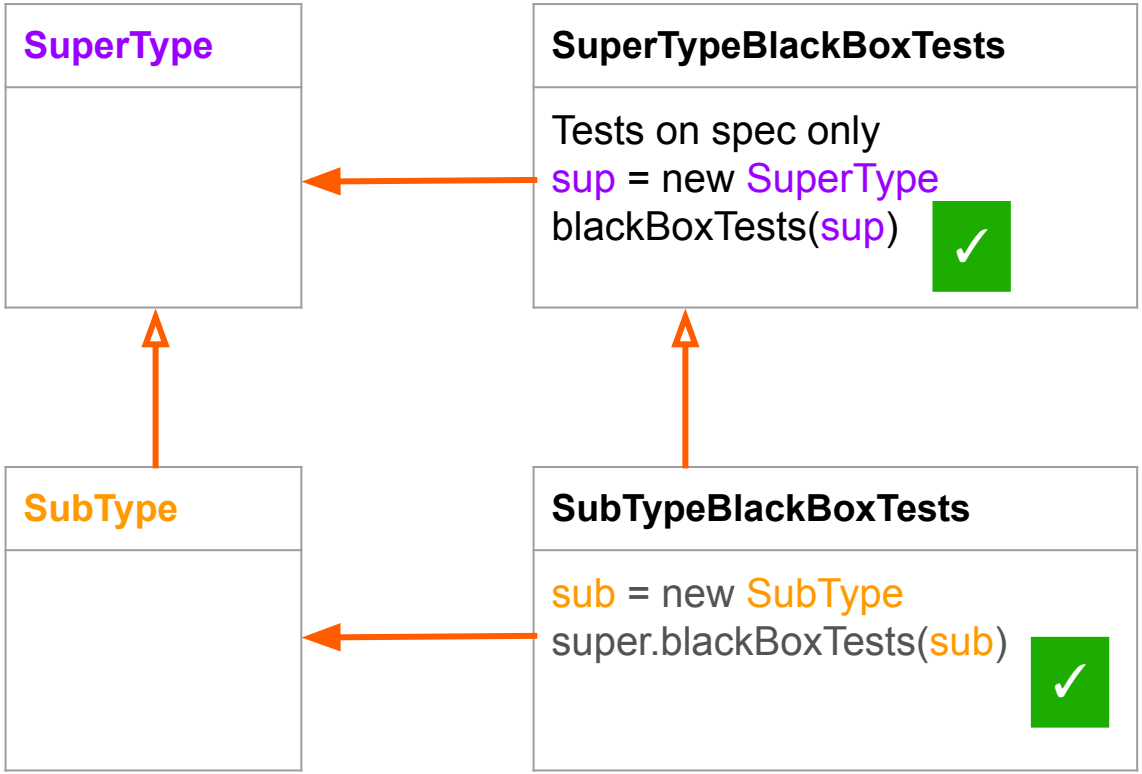
LSP & Testing

Tests *within the specification* of the superclass *should still pass* if the subclass is substituted.



LSP & Testing

Tests *within the specification* of the superclass *should still pass* if the subclass is substituted.



For a class to be substitutable for its supertype, All the **supertype black box tests must pass**, when run on an instance of the subtype.

Interface Segregation Principle

S O L I D

"Many client-specific interfaces are better than one **general-purpose** interface."

[http://en.wikipedia.org/wiki/Interface_segregation_principle]

Interface Segregation Principle

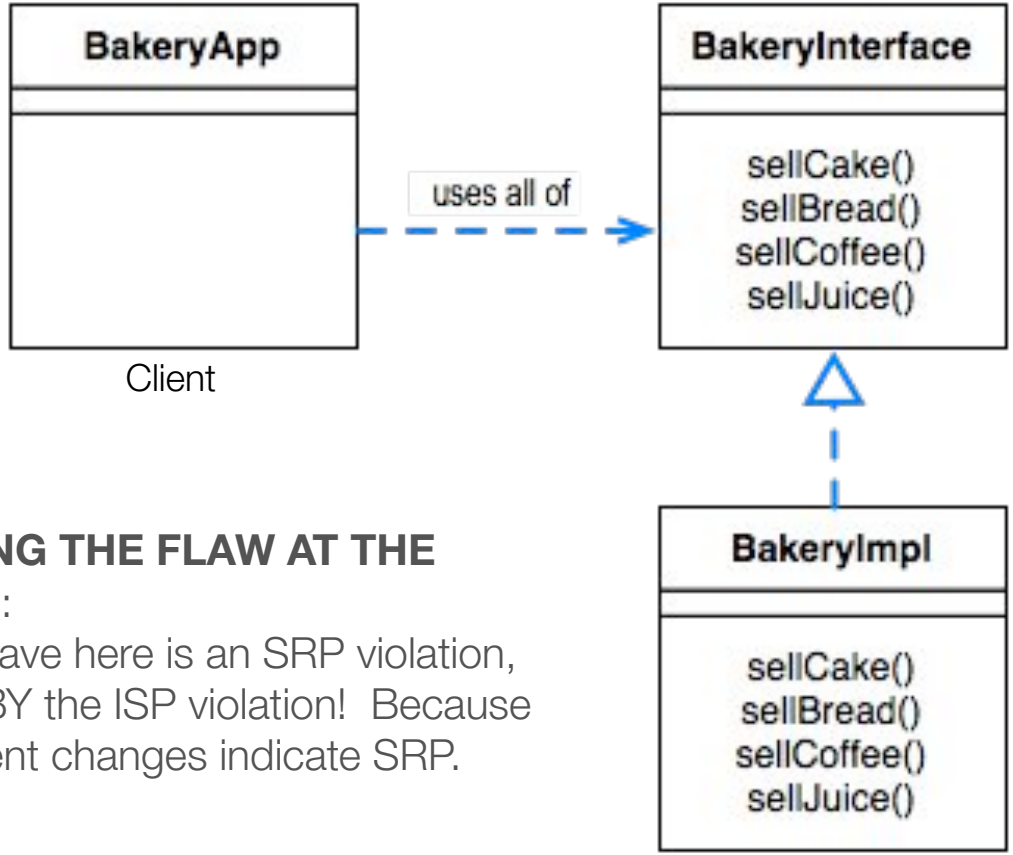
SOLID

- Clients should not be required to depend on methods they do not need.
- No implementation should be forced to provide methods that do not fit into their abstraction!
  - This is a classic pathway to SRP and LSP violations!

- **Solution**: Move towards role-based interfaces.
  - Clients need only know about the methods that are of interest to them.
  - This relates strongly to the concept of high cohesion.

Starting a Bakery

S O L I D



DETECTING THE FLAW AT THE SUBTYPE:
What we have here is an SRP violation, CAUSED BY the ISP violation! Because the divergent changes indicate SRP.

We can immediately see that merge conflicts will become a problem in a multi-person team.
Food versus drinks → Divergent Changes.

const b: BakeryInterface = new BakeryImp();

UBC

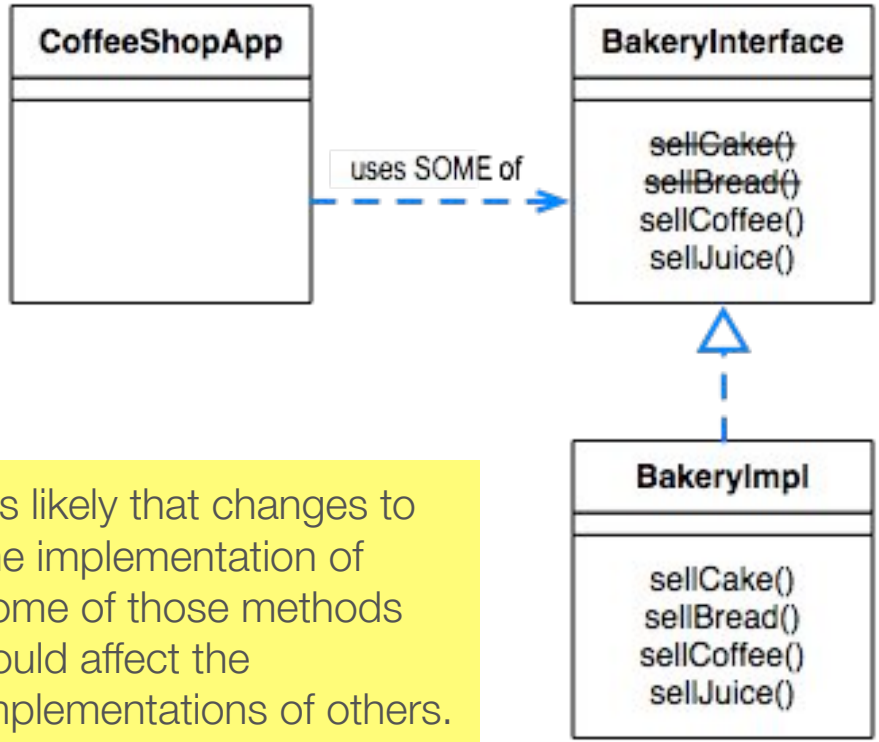
Business is Good. Coffee Cart?

SOLID

CoffeeShopApp forced to rely on methods it doesn't need because it only has BakeryInterface to work with.

This is not as good!

Now the CoffeeShopApp client is only using part of the BakeryInterface, but it *has to know* about all o it.



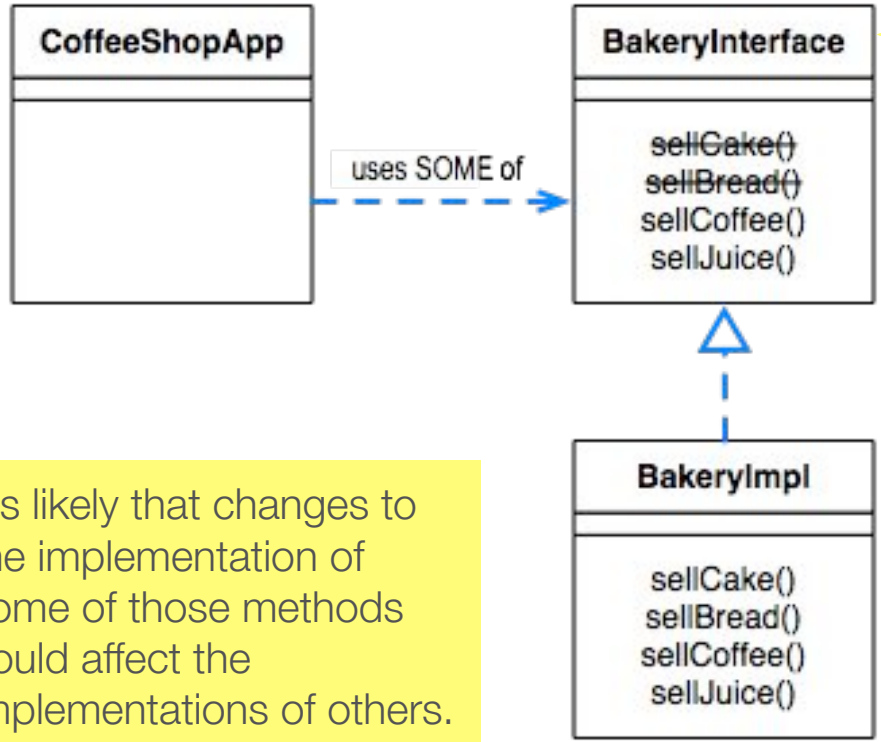
It's likely that changes to the implementation of some of those methods could affect the implementations of others.

Business is Good. Coffee Cart?

SOLID

This is not as good!

Now the CoffeeShopApp client is only using part of the BakeryInterface, but it *has to know* about all o it.



It's likely that changes to the implementation of some of those methods could affect the implementations of others.

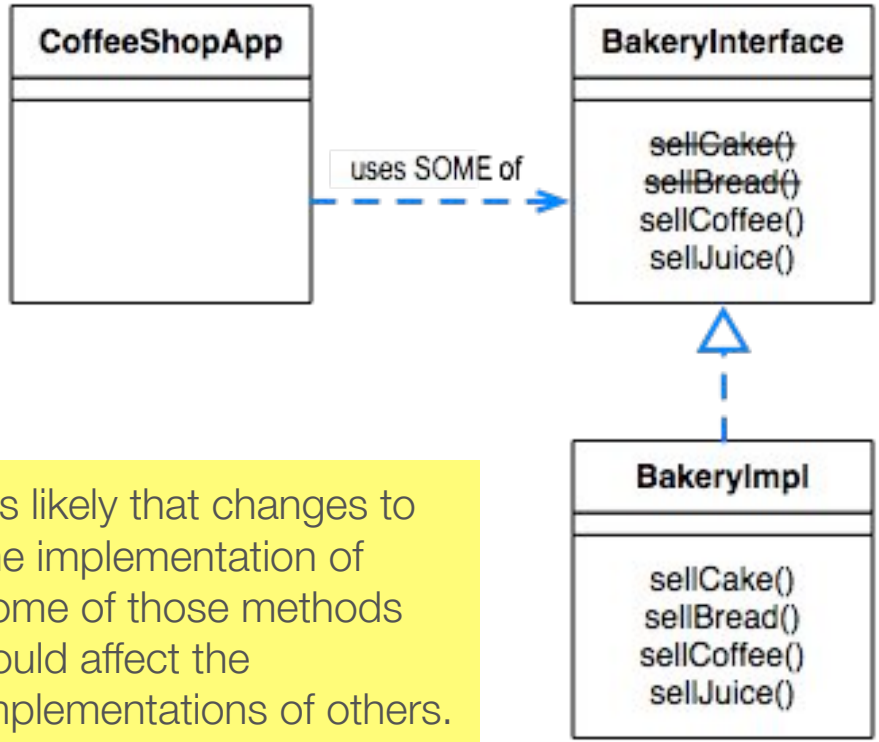
Food versus drinks → Divergent Changes.

Business is Good. Coffee Cart?

SOLID

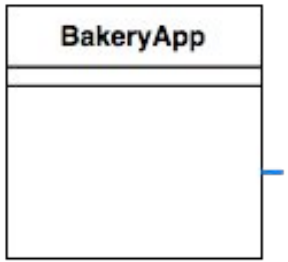
This is not as good!

Now the CoffeeShopApp client is only using part of the BakeryInterface, but it *has to know* about all o it.

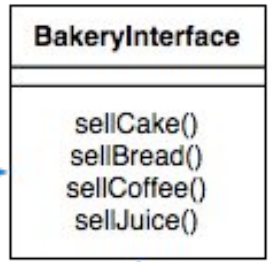


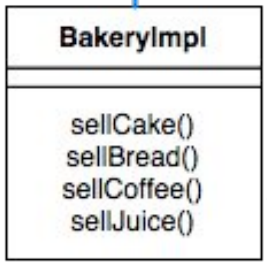
It's likely that changes to the implementation of some of those methods could affect the implementations of others.

Ultimately BakeryInterface might be a good design for a bakery but is bad at being a Coffee Shop.



uses all of





ISP forces responsibilities on others (the client, the implementers).

SRP describes *implementation* level collisions.

ISP Violation

SRP Violation

Divergent changes

S O L I D

BakeryApp

....
...

BakeryInt
---
sellCake()
sellBread()

CoffeeShpInt
---
sellCoffee()
sellJuice()

BakeryImpl
---
sellCake()
sellBread()

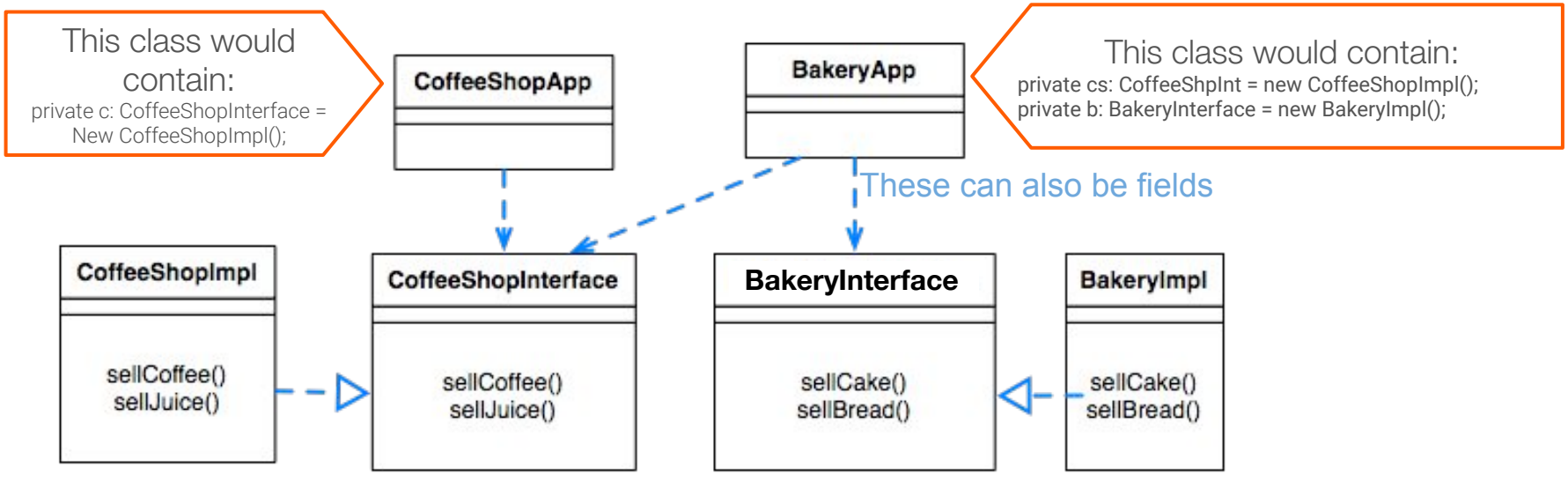
Same thing

BakeryApp
---
const cs: CoffeeShpInt = new CoffeeShopImpl();
const b: BakeryInterface = new BakeryImpl();

CoffeeShpImpl
---
sellCoffee()
sellJuice()

Extract Interface Refactoring.
Extract **sellCoffee sellJuice → CoffeeShopInterface**

An Improved Design



No LSP violation!

Bad cohesion → Divergent changes → Extract Class → Extract Interface → Interface Seg P.

Dependency Inversion

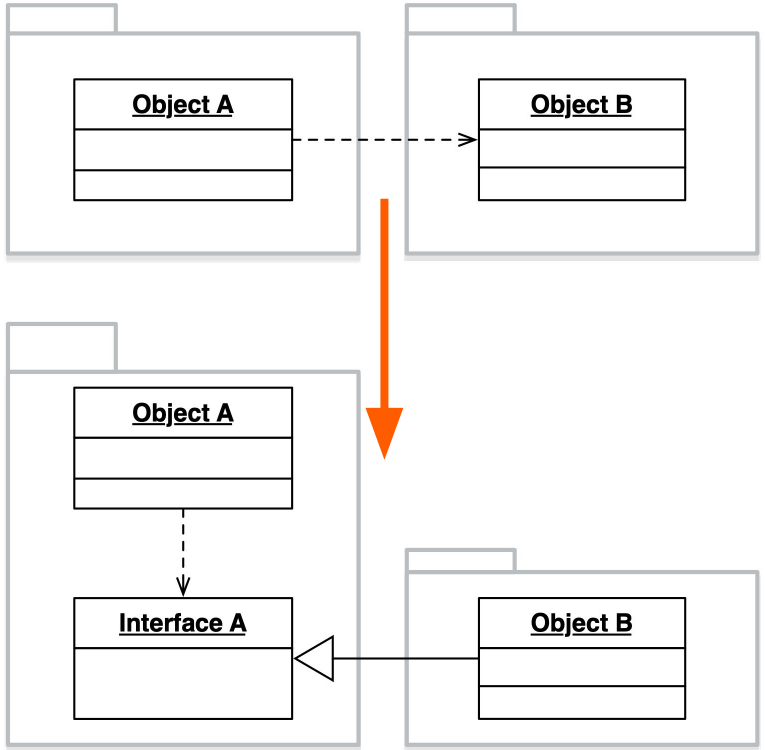
SOLID

Depend upon means **DECLARING** an entity of that type.

> "Depend upon *abstractions*, do not depend upon *implementations*."

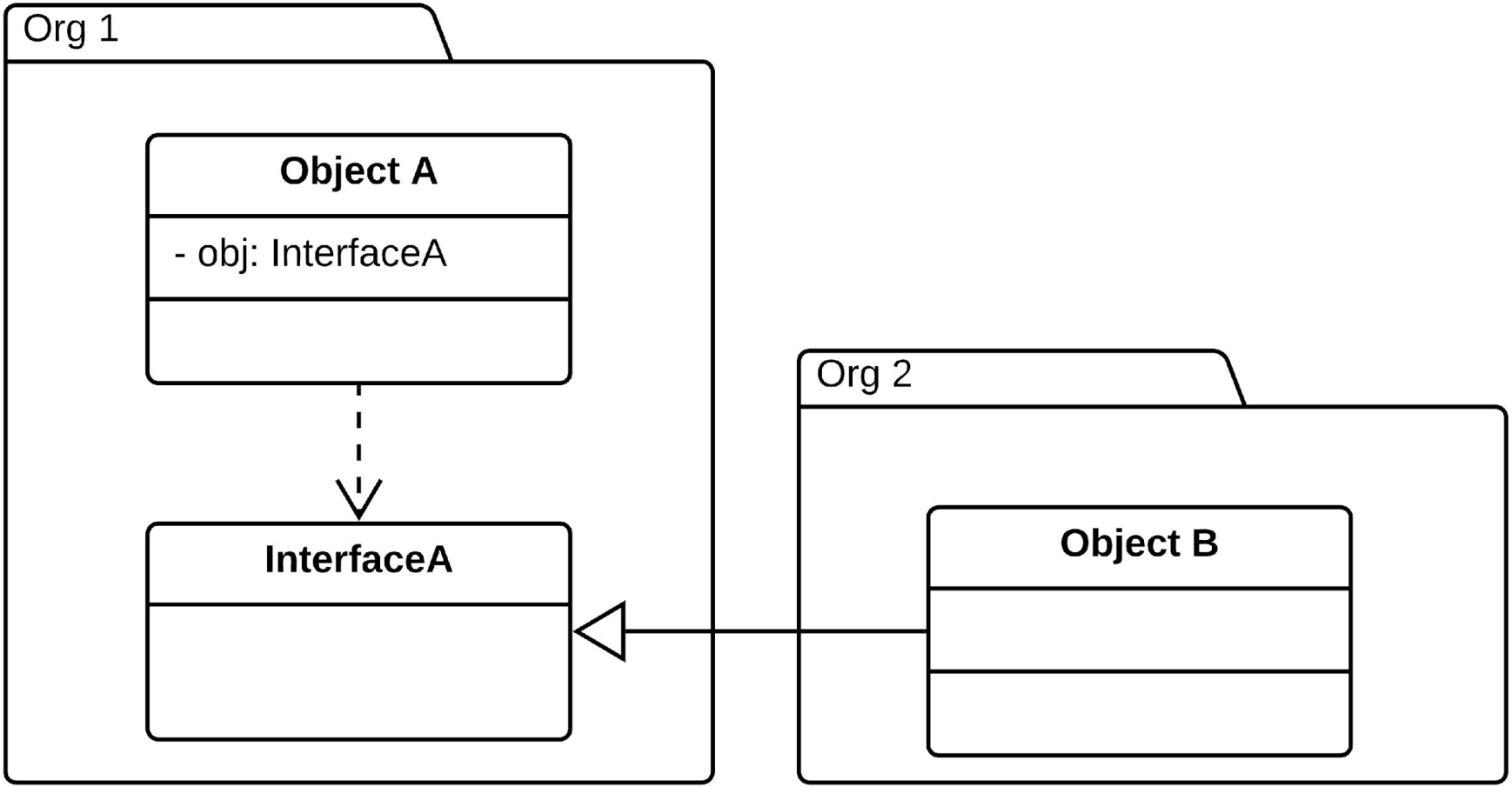
High-level modules should *not* depend on low-level modules. Both *should* depend on abstractions.

Restated: Abstractions should not depend on *details*. Details should depend on *abstractions*.



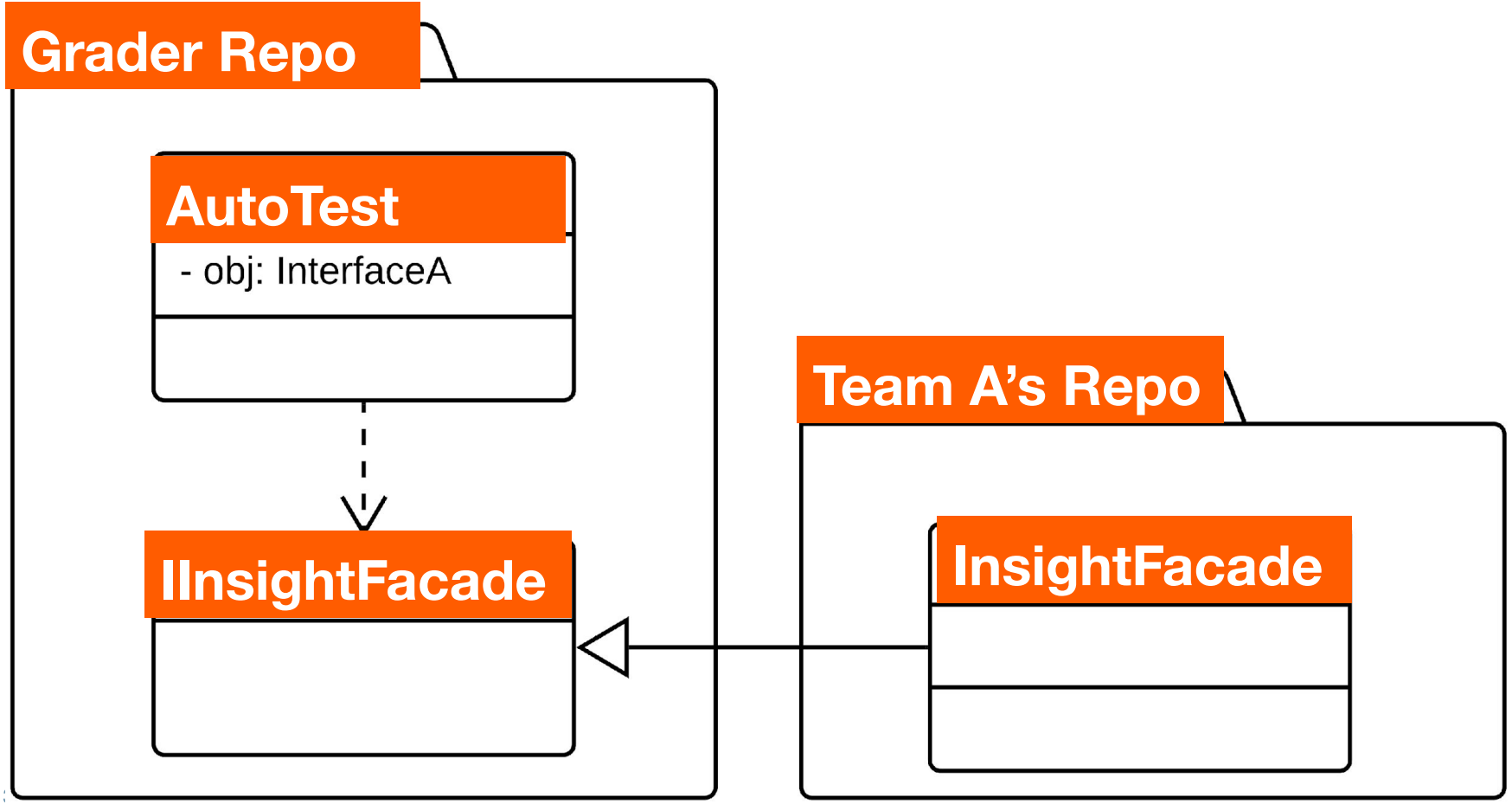
[http://en.wikipedia.org/wiki/Dependency_inversion_principle]

Dependency Inversion



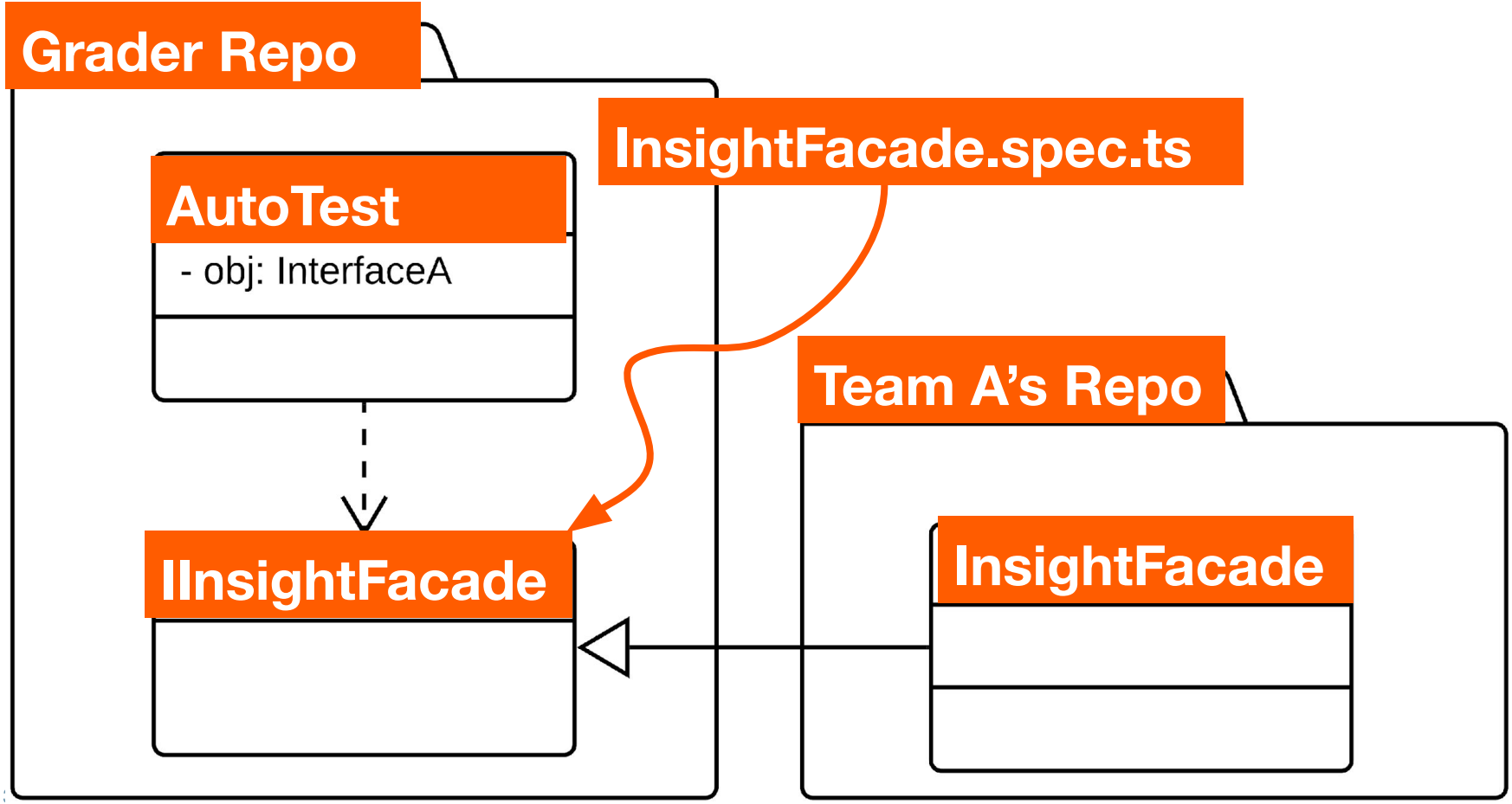
Dependency Inversion

SOLID



Dependency Inversion

SOLID



Concrete DI Examples

S O L I D

OLD: const data: ArrayList = new ArrayList();
NEW: const data: Collection = new ArrayList( );
→ What class of future changes does this enable?

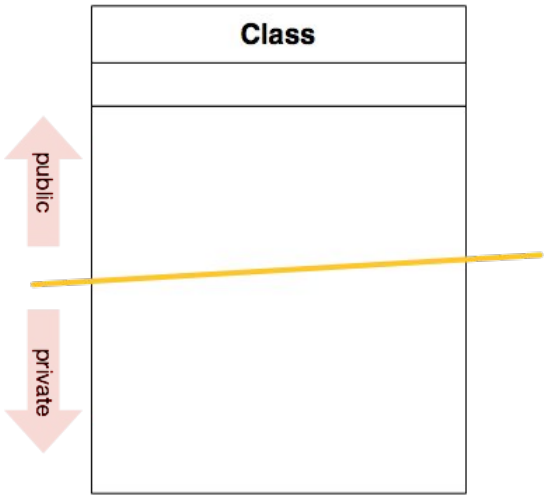
OLD: const s: Server = new Server( );
NEW: const s: ServerInterface = new ServerImplementation( );
→ What class of future changes does this enable?

data.add('cs310'); // only knows data as a Collection interface, not its concrete type.
s.start(); // only knows about s as the interface type.

UBC

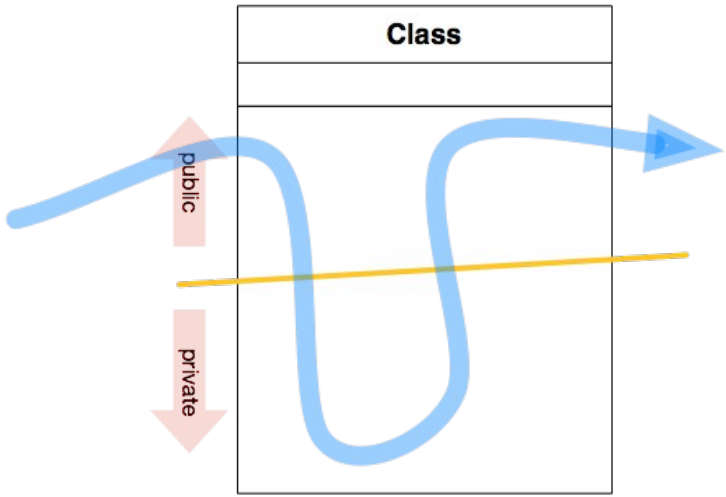
Typical Class Design

SOLID



UBC

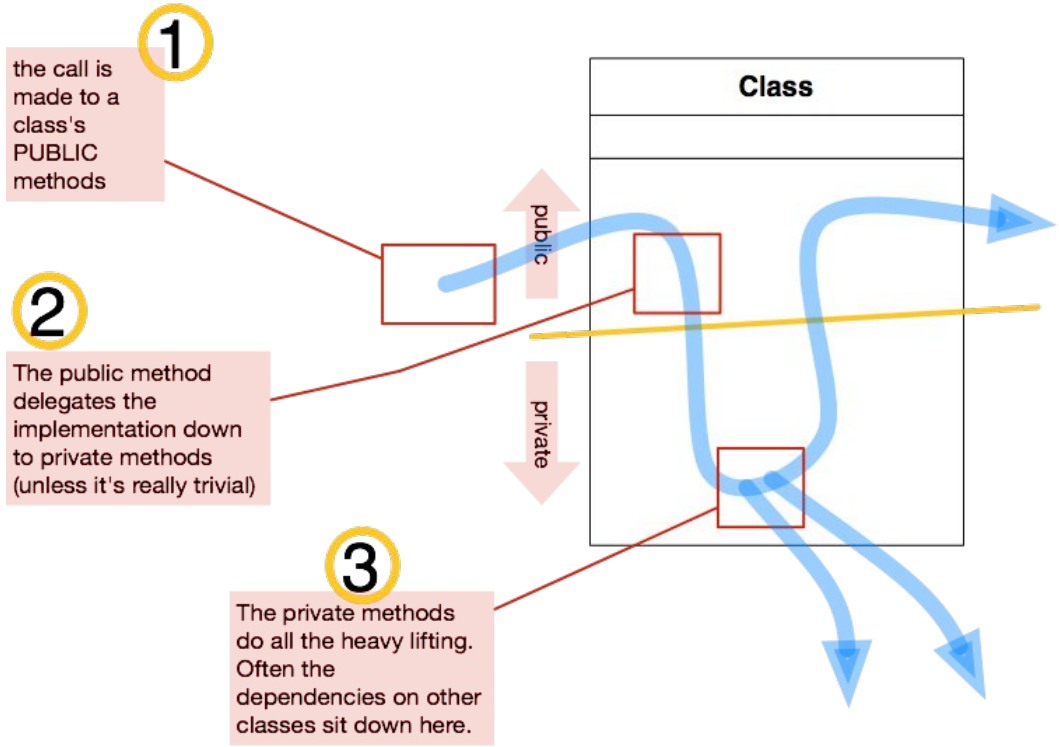
Public Methods are the ‘Interface’



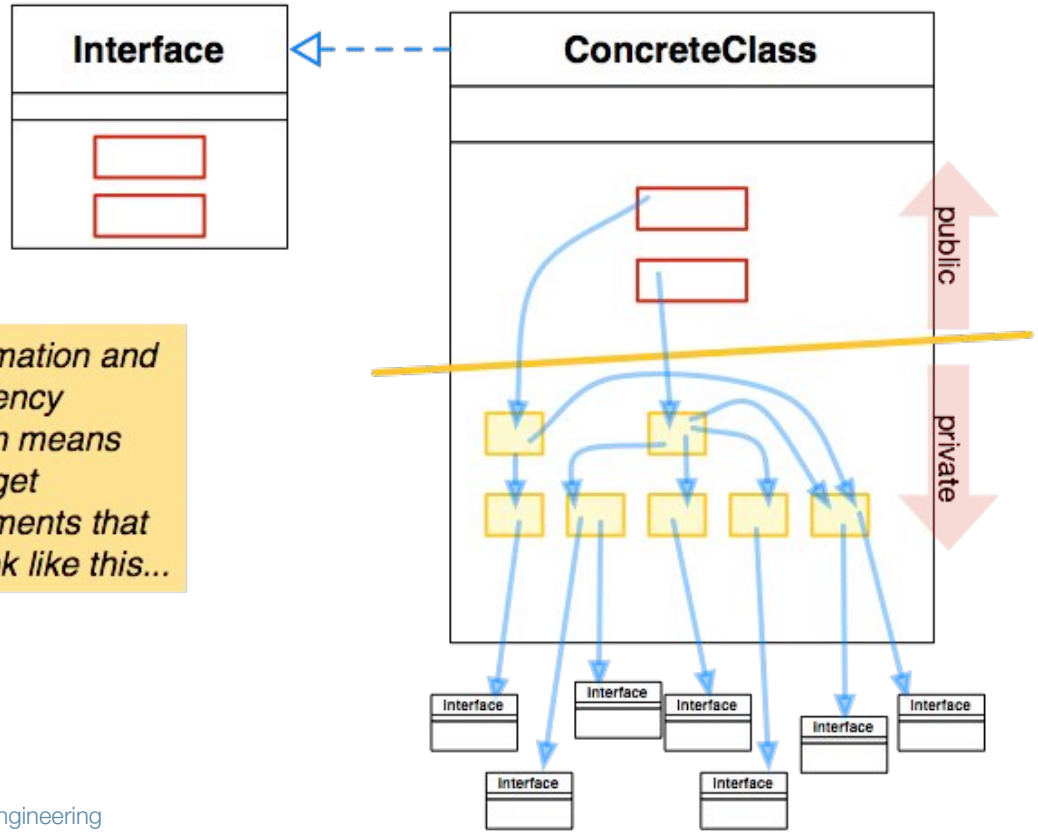
Nice clean calls down from within public methods.

All the guts down here.

Public Calls ‘Down’ to Impl

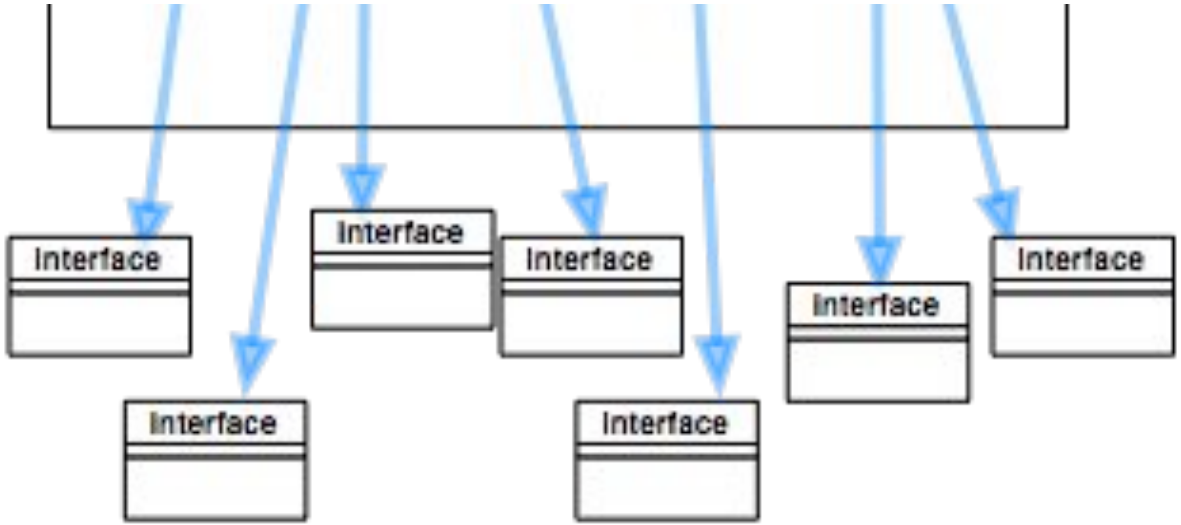


Combining These…



So information and Dependency Inversion means that we get arrangements that often look like this...

Private Behaviours Depend on Abstractions!



Design Principles Recap

Pragmatic Programmer:
Eliminate effects between unrelated things
by designing components that are:
self-contained, independent,
and have a single, well-defined purpose.

S O L I D

Single Responsibility

S O L I D

A class should have only a single responsibility

A class should have only one reason to change.

Open/Closed

SOLID

A class must be **closed** to internal changes

but still be **open** for extensions

When designing classes, do not plan for brand new functionality to be added by modifying the core of the class. Instead, design your class so that extensions can be made in a modular way, to provide new functionality by leveraging the power of the inheritance facilities of the language, or through pre-accommodated addition of methods.

Liskov Substitution

S O L I D

Objects of a **supertype**
in a program may be replaced with
objects of a **subtype**
without altering any desirable
properties of the program

If S is a subtype of T, then objects of type T (supertype) in a program may be replaced with objects of type S (subtype) without altering any of the desirable properties of that program.

Interface Segregation

S O L I D

No client methods should be forced to depend on it does not use

A move towards role-based interfaces: No implementation class should be forced to provide methods that do not fit into its abstraction! (SRP and LSP violations!) Clients need only know about the methods that are of interest to them.

Dependency **Inversion**

S O L I D

Depend on
abstractions not
implementations

High-level modules should not depend on low-level modules; instead, they should depend on abstractions.

UBC

Microservices and API Design

Topics

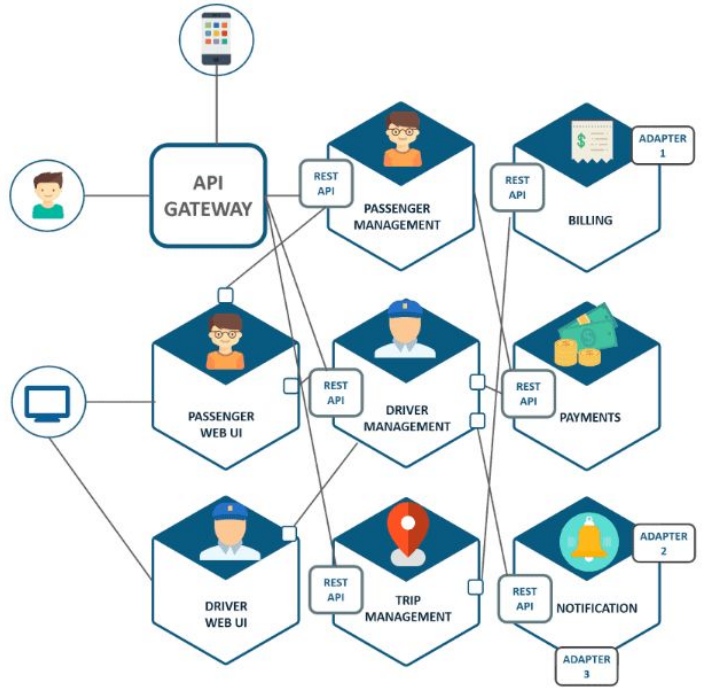
Examinable Skills

- Know what an API is and why they are used.
- Describe considerations for designing high-quality APIs.
- Design and justify an API for a given set of requirements.
- Understand the role APIs play in microservice-based architectures.
- Describe the strengths and weaknesses of versioning APIs in microservice and non-microservice API environments.

Microservice Architecture at Medium

[https://medium.engineering/microservice-architecture-at-medium-9c33805eb74f]

Monolith to Microservice



What is Microservice Architecture?

First of all, let's take a moment to think about what microservice architecture is and is not. "Microservice" is one of those overloaded and confusing software engineering trends. This is what we at Medium think what it is:

> *In microservice architecture, multiple loosely coupled services work together. Each service focuses on a single purpose and has a high cohesion of related behaviors and data.*

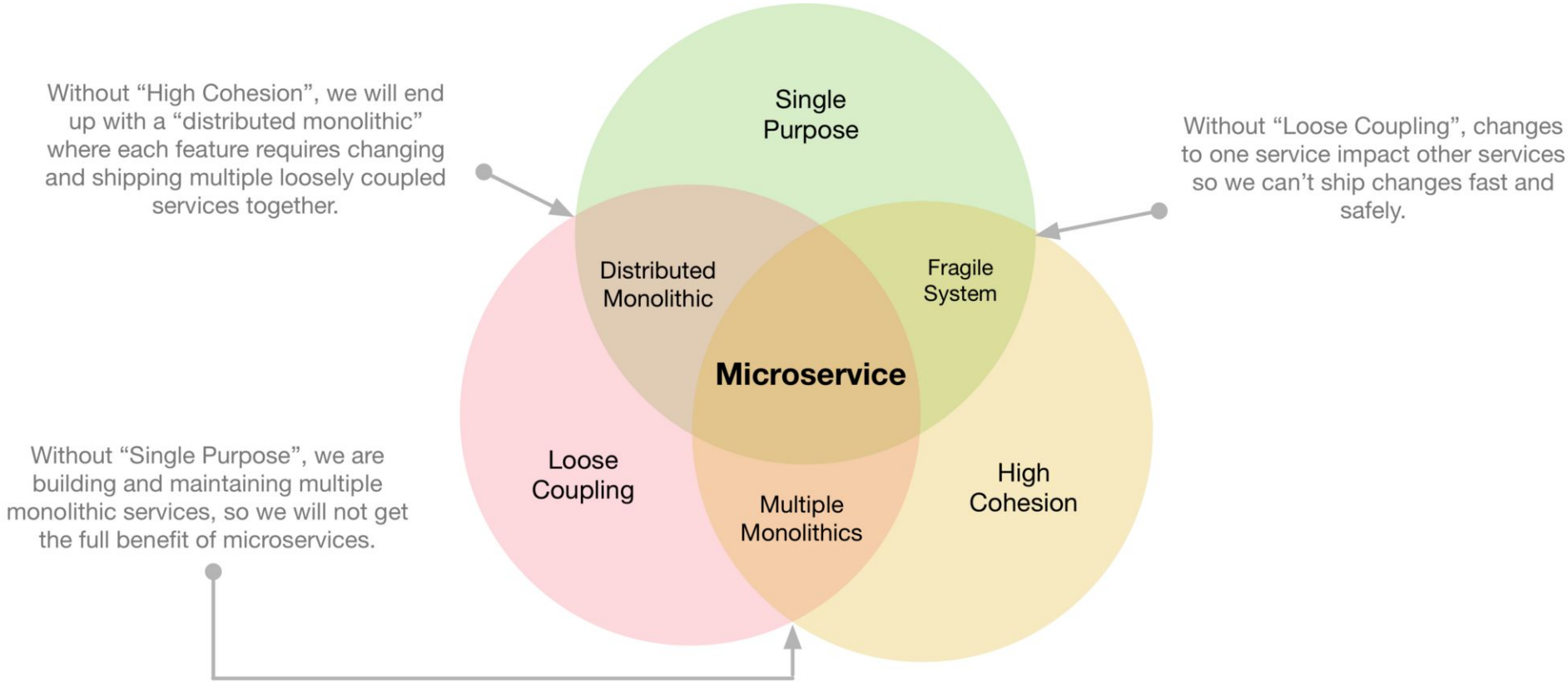
What is Microservice Architecture?

First of all, let's take a moment to think about what microservice architecture is and is not. "Microservice" is one of those overloaded and confusing software engineering trends. This is what we at Medium think what it is:

> *In microservice architecture, multiple loosely coupled services work together. Each service focuses on a single purpose and has a high cohesion of related behaviors and data.*

- *Single purpose* — each service should focus on one single purpose and do it well.
- *Loose coupling* — services know little about each other. A change to one service should not require changing the others. Communication between services should happen only through public service interfaces.
- *High cohesion* — each service encapsulates all related behaviors *and data* together. If we need to build a new feature, all the changes should be localized to just one single service.

Microservice Properties



Design Discipline

When we model microservices, we should be disciplined across all three design principles. It is the only way to achieve the full potential of the microservice architecture. Missing any one of them would become an anti-pattern.

Microservice “Syndromes”

- **Poorly modelled** microservices cause more harm than good, especially when you have **more than a couple** of them.
- Lack of **observability**, which makes it difficult to **triage** performance issues or failures.
- When facing a problem, teams tend to create a new service instead of **fixing the existing one** even though the latter may be a better option.
- Even though the services are **loosely coupled**, lack of a **holistic picture** of the whole system could be problematic.

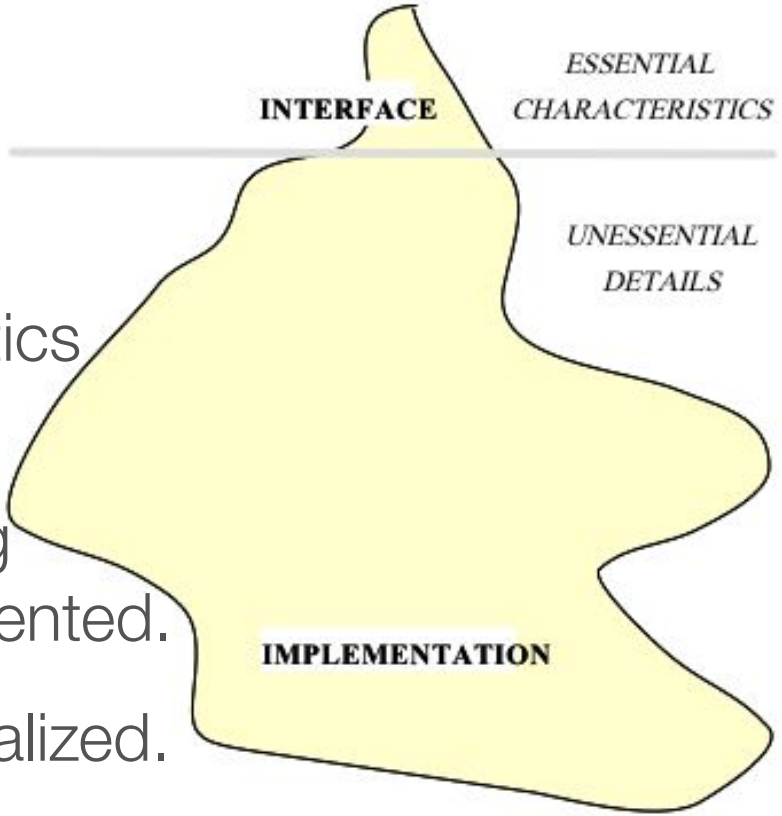
High-level

APIs

Low-level

Information Hiding

- Only expose necessary functions.
- Abstraction hides complexity by emphasizing essential characteristics and suppressing detail.
- Caller should not assume anything about how an interface is implemented.
- Effects of internal changes are localized.



[http://www.fatagnus.com/program-to-an-interface-not-an-implementation/]

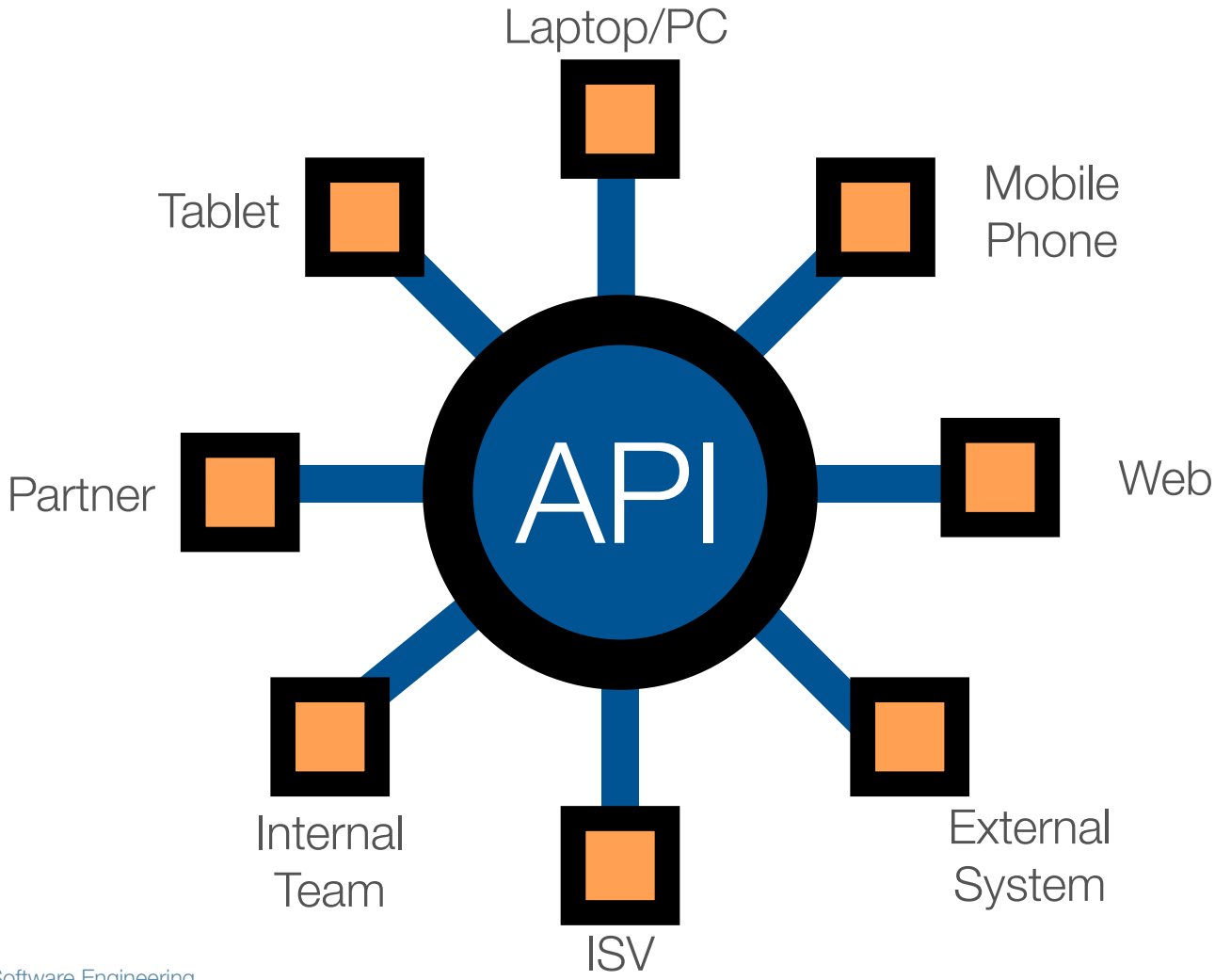
Information Hiding Example

```
public getElevationInMeters(pressureInMB: number): number;
```

```
class ElevationController implements BarometricElevation {
    private correction: number; // geoid correction, set elsewhere

    // for a pressure in millibars, return elevation in meters.
    // assumes the correction field has been set for the geoid, or is 0
    public getElevation(pressure: number): number {
        let altpress = (1 - Math.pow((pressure / 1013), 0.19)) * 145366;
        altpress = (0.305 * altpress)  + this.correction ;
    }
}
```


API


Laptop/PC
Tablet
Mobile Phone
API
Partner
Web
Internal Team
ISV
External System

API Design Principles

1. Do one thing and do it well.
2. APIs should be as small as possible but no smaller.
   - When in doubt, leave it out. APIs are forever.
3. Implementation should not impact API.
   - Leaking API details is a fundamental mistake.
4. Minimize access. Increases independence.
5. Names matter.
6. Documentation matters.

API Design

- Two main questions will drive many design decisions:
  - What is the goal of this API?
  - Who will be using this API?
- And technical considerations:
  - Data formats, protocols.
  - Authentication, security.
  - Versioning, licensing.

Easy to use, hard to misuse

How To Write an API

- Start with one-page spec to create use cases.
- Talk to as many stakeholders as possible.
- Code tests against API to simulate client code.
- Write *multiple* clients before release.
  - If you only write one, you'll only ever have one.
  - If you write two, you can support more with difficulty.
  - If you write three, it will work fine.
- Most APIs have fundamental limitations that are non-optimal.

Designing API Methods

- Don’t make a client do anything that could be internal.
  - Look for clones in your sample client systems.
- Users should not be surprised by API behaviour.
- Fail fast: report errors as soon as possible.
  - Compiler better than runtime.
  - First error rather than last at runtime.
- Provide basic type access (e.g., convert data to strings).
- Use consistent params and types (e.g., time always number milliseconds).
- Avoid returns that demand exceptional handling (e.g., checking error codes).

Designing for Constant Change

- As Jeff Dean WSDM Keynote: the abstractions you choose will change over time as new parameters are learned.
- Thinking concretely about what parts of the system are likely to change in the short and medium terms are more likely to lead to useful and valuable abstraction layers, rather than taking an 'anything can change' view to design.
- While APIs are forever, their internals will change more than you can usually imagine in advance!
- The right design for one system will probably be different at 10X load or 100X load (premature optimization).

API Design Advice (Josh Bloch @ GOOG)

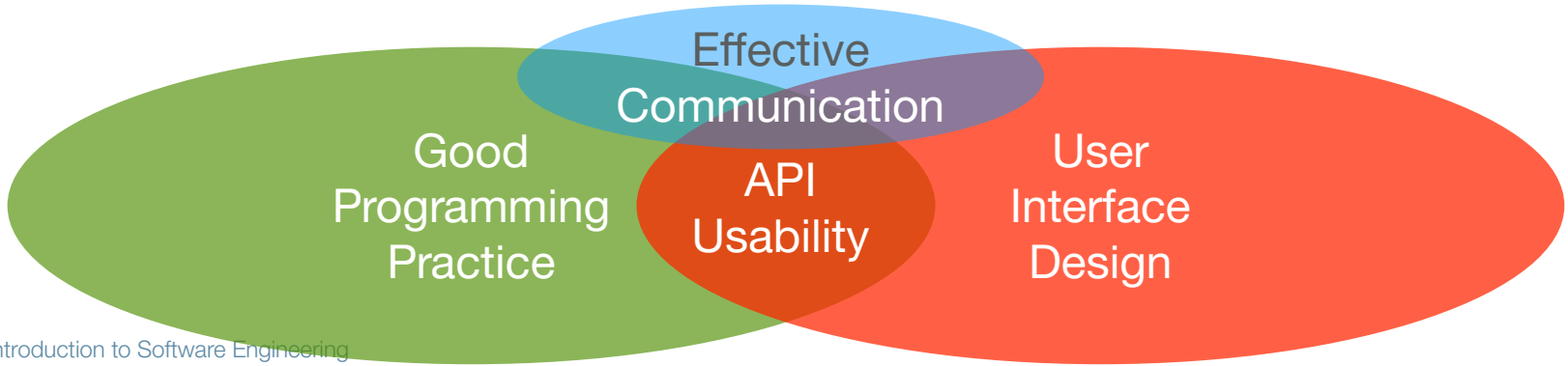
- APIs should **do one thing**, and do it well; easy to name.
  - `getPaidUsersAndSortByName() -> getUsers(paid: boolean, sortIndex: number)`
- APIs should **never** expose internal implementation details.
  - These make it difficult for the consumer to use and for the maintainer when they want to change in internal implementation detail.
  - `getUsers(paidOnly: boolean, sortKey: Users.Keys)` (index exposes imp'l)
- APIs should be kept as **small** and simple as possible.
  - 'When in doubt, leave it out'; hard to remove API.
  - Best: `getPaidUsers()` ← sortKey not needed, client sorts.
- The developer '**usability**' of a system dictated by API quality.

Programmatic Usability

- What is usability to a developer?

Programmatic Usability

- What is usability to a developer?
- APIs are affordances used by developers for understanding
- API design forces you to think from a client's point of view.
- Good APIs encourage reusability.
- Broad valuable APIs can be widely used.



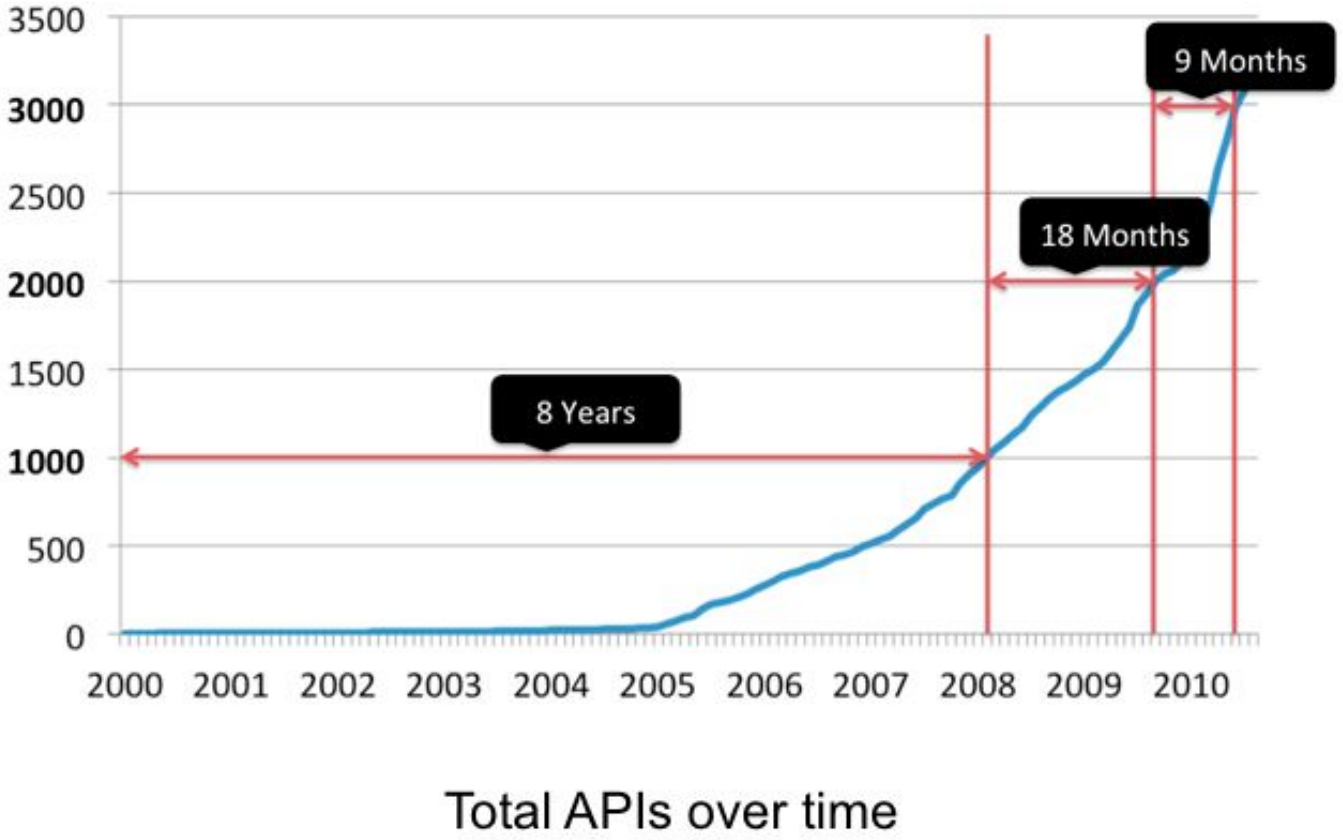
Principles of Programmatic Usability

- **Good visibility**: Possible actions and states must be exposed.
  - anti-example: `items.sort('clearance')`
- **Good model**: Helpful, consistent, and complete abstractions clarify the correct model of the system.
  - anti-example: `store.get(id: string)`
- **Good mapping**: Natural mapping between actions & results.
  - anti-example: `store.getProduct(pid: string): any`
- **Good feedback**: Continuous feedback about results of actions.
  - anti-example: `error: OperationFailure`

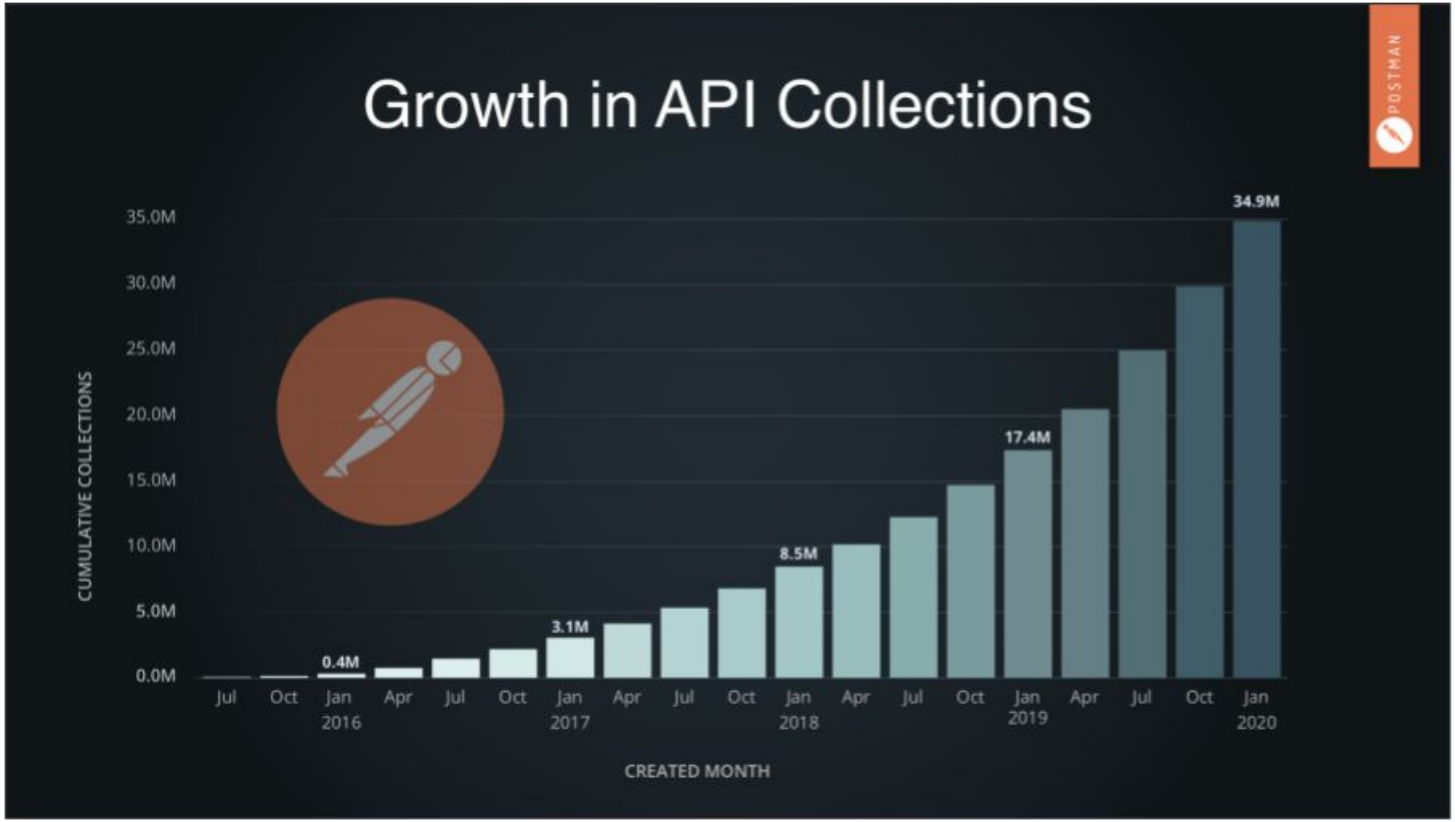
DEVELOPER EXPERIENCE

USER EXPERIENCE APPLIED TO DEVELOPERS, BECAUSE DEVELOPERS ARE PEOPLE TOO.

API Growth

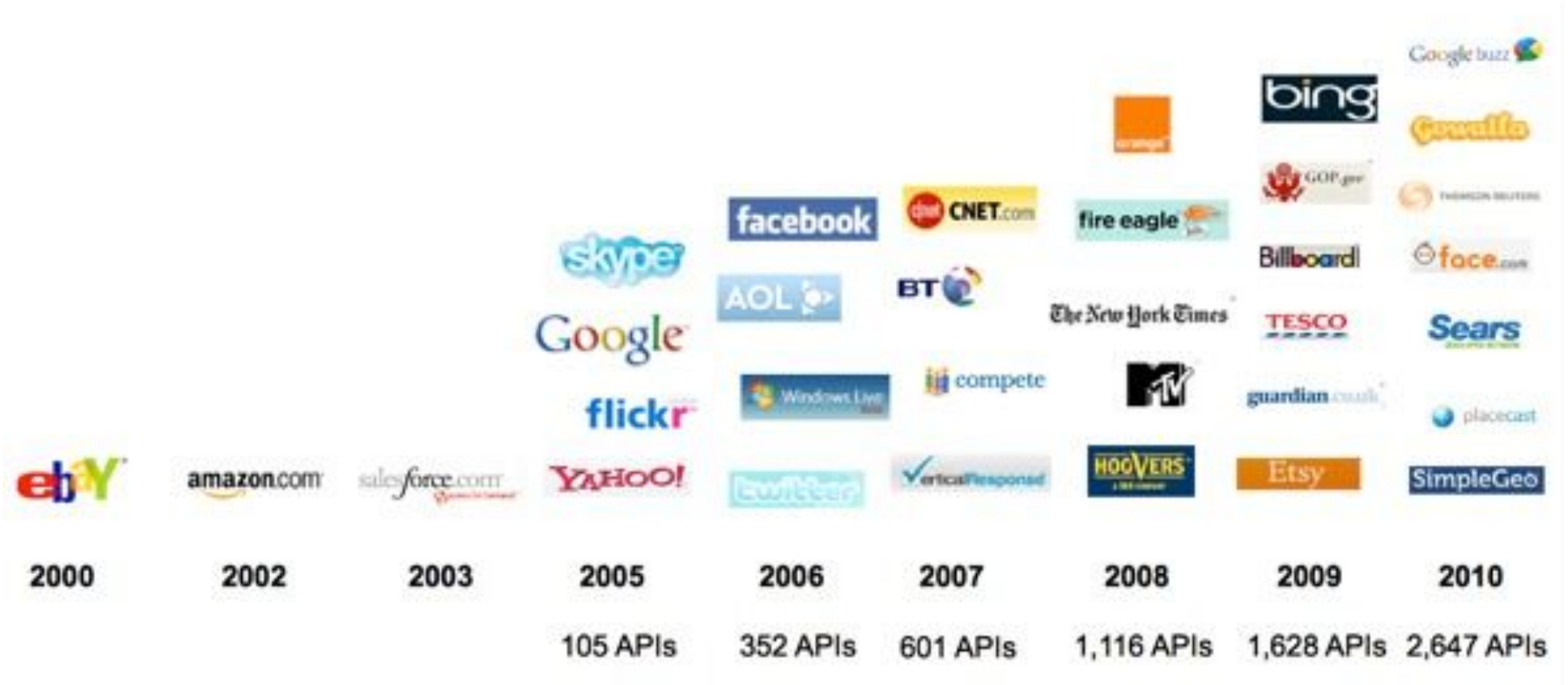

3500
3000
2500
2000
1500
1000
500
0
2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010
8 Years
18 Months
9 Months
Total APIs over time

API Growth



https://blog.postman.com/api-growth-rate/

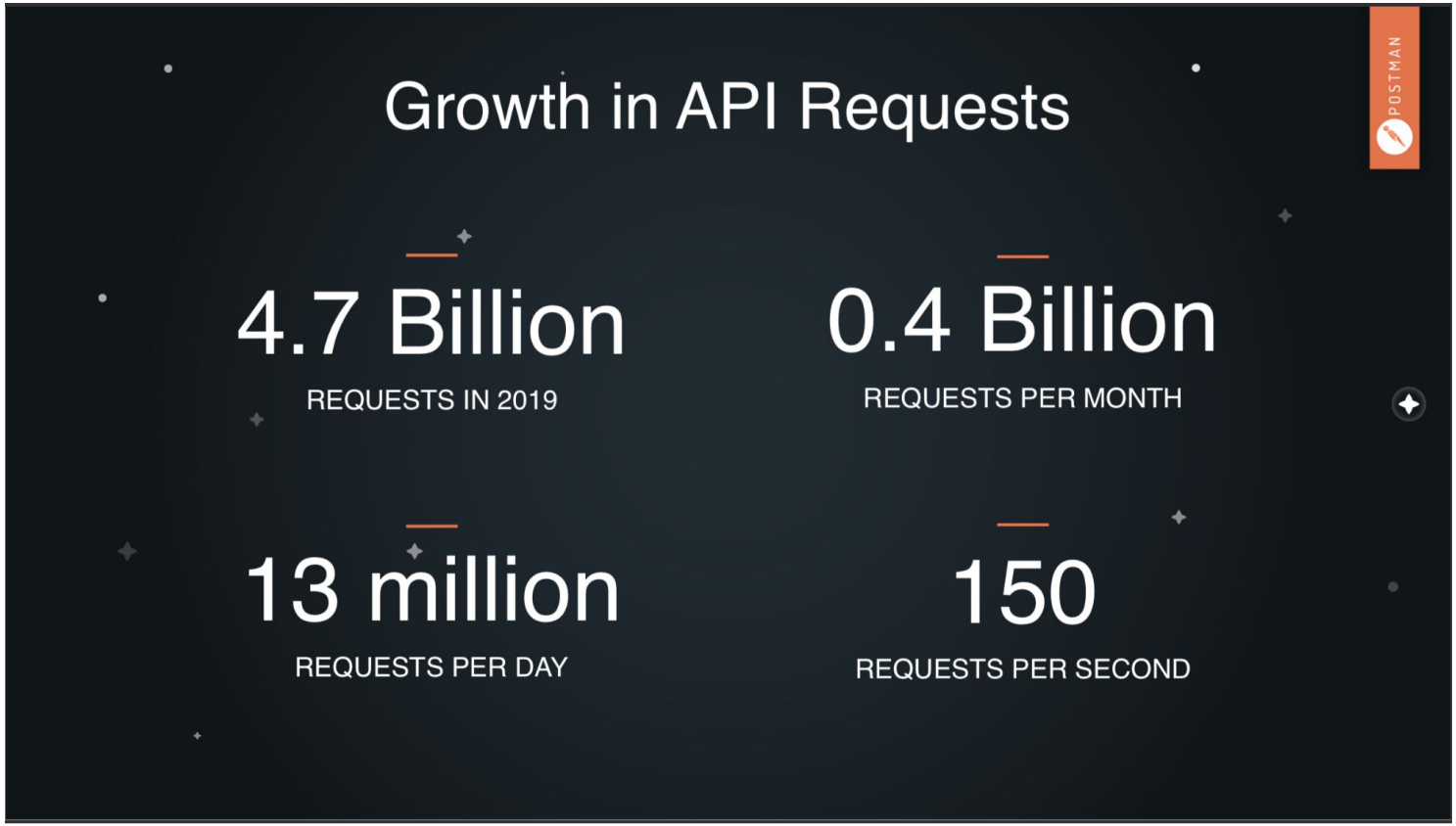
API Platforms



API Platforms

- Rather than building all possible products, provide a platform.
  - Others integrate and extend platform (often for free).
  - But users pay for basic platform access.
  - Platform consumers do not need to be ‘the public’.
  - Platforms are also heavily used within organizations.
    - e.g., CWL at UBC.

API Requests



https://blog.postman.com/api-growth-rate/

High-Level Design Summary

Abstraction (high level concept at the root of all that follows)

- Manage complexity by focusing on key aspects for given task and stakeholder.
- Both too much abstraction or too little abstraction inhibit understanding.
- Useful for discussing viewpoints that makes sense for individual stakeholders.

Decomposition (mechanism for ideating about abstractions)

- Mechanism for breaking down complex description into more manageable pieces.
- Goal is to make common tasks simple while not prohibiting exceptional tasks.
- Can be done both top down or bottom up (or both).

Information hiding (how programs leverage abstraction)

- Hides implementation details from high level interfaces.
- Separate that which varies from that which stays the same.
- Separate names from implementations (aka API from impl).

Encapsulation (language approach for implementing information hiding)

- Mechanism for implementing abstractions in a program.
- Usually through the use of language interfaces.
- Captures data and behaviour and separate these from their implementation.

Design Symptoms and Principles

- The unconstrained nature of software makes it easy to make decisions that will end up being ‘bad’ in the long run.
- Symptoms provide a set of attributes you can use to evaluate your design (both for refactoring and during initial design).
- Design principles provide a set of techniques that can help you to find ways to mitigate symptoms of poor design.
- Neither symptoms or principles we have discussed here are complete; they will vary by organization, domain, and team.

▼ coupling ▲ cohesion

SOLID

Encapsulate what varies
Design to interfaces
Composition over inheritance