For personal use only,
please do not
distribute on
non-UBC domains.

06b - API Design & Evolution

CPSC310

Nick Bradley & Kyle Chin

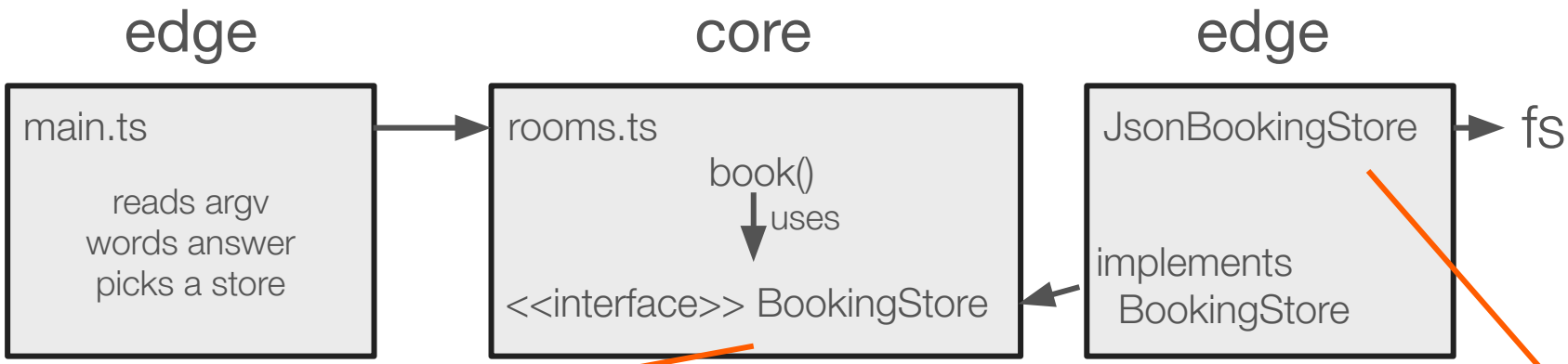
Announcements

- Congratulations on finishing D2!
- D3: posted, due Nov 6th
  - Complete the planning task BEFORE Lab 07
- Midterm next week: Thursday, Oct 29 @ 19:00-21:00
  - Details & Practice Exam posted on Piazza

Core and Edges

Business logic
Imports nothing from outside & does no I/O of its own

External concerns
I/O, Parsing, Validation, Config (file paths), Services (auth, email, APIs)



Port: *an interface owned by core, defined in its own domain*

Adapter: *a class that implements a port using real resources (fs, network)*

Testing at the Boundary

Boundaries provide a natural way to structure a test suite:

- Test across boundaries
  - does the entire application work as expected (typically from the user's perspective)?
- Test within a boundary
  - does this unit of code behave as expected?
- Test between boundaries
  - do the adapters properly translate between the domain and the system?

E2E: Testing *Across* Boundaries

Natural to write tests the way the system is used (e.g., the tests in the project). These are called *end-to-end (E2E)* tests.

E2E tests can give assurance code behaves as expected, but come with several challenges:

- Poor fault localization
- Multiple reasons for failure
- Cumbersome to write
- Expensive to run

Unit: Testing *Within* Boundaries

Test small pieces of code (usually functions/methods). They are typically:

- Easy to write (and later to read as documentation)
- Very fast to run
- The most direct way to localize faults
  - test doubles in place of dependencies
  - natural place to set breakpoints

But:

- They check behaviour in isolation!

Integration: Testing *Between* Boundaries

Bridge the gap between unit tests and E2E tests.

Checks that adapters between the core and the system translate correctly.

Approach:

- Test the core code with the real adapter instead of test double.
- The adapter uses an actual system resource (e.g., file on disk)

Test Suite Composition

End-to-End Tests

1 **A few end-to-end tests ensure the whole application works.**

Integration Tests

2 Several tests at each boundary



Unit Tests

3 **The majority of tests should be unit tests.**

Learning goals

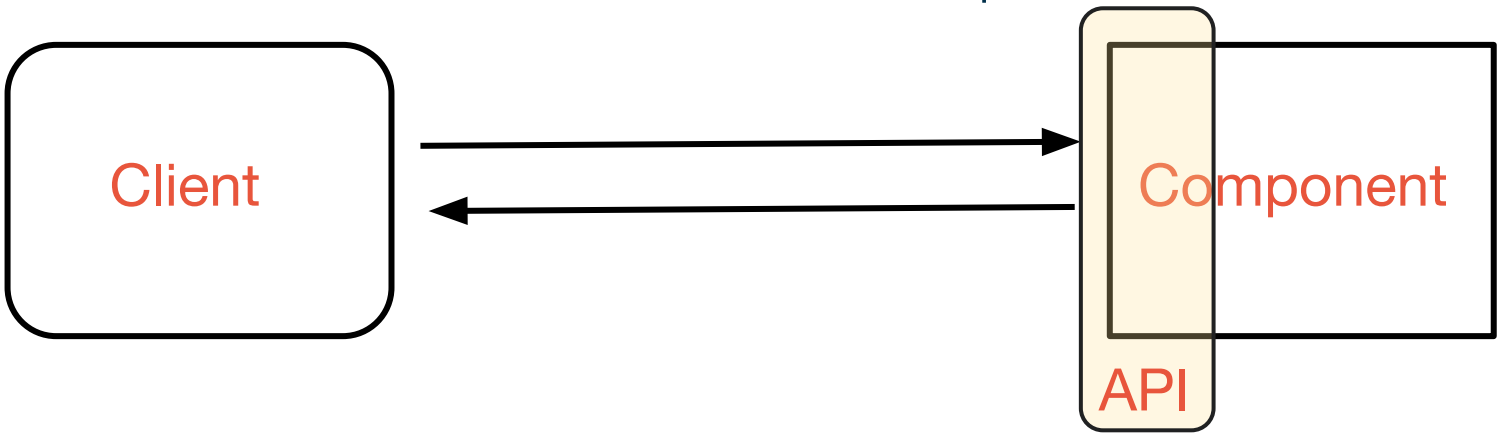
By the end of the lecture you should be able to:

- Analyze an API using API design criteria: information hiding, obliviousness, usability, and completeness
- Describe the impact API changes have on clients in terms of severity: Major, Minor, Patch
- Identify alternative API designs and changes to mitigate and address these issues

What is an API?

Application **P**rogramming **I**nterface

"*a way* for two or more computer programs to ***communicate*** with each other"- Wikipedia



Examples of APIs



Chai Asser

- Introduction
- Expect / Should
- Assert
- Plugin Utilities
- Online Test Suite
- language chains
- not
- deep
- nested
- own

Node.js v25.7.0 | ▸ Table of contents | ▸ Index | ▸ Other versions | ▸ Options

Stability: 2 - Stable

fs.readFile(path[, options], callback)

▸ History

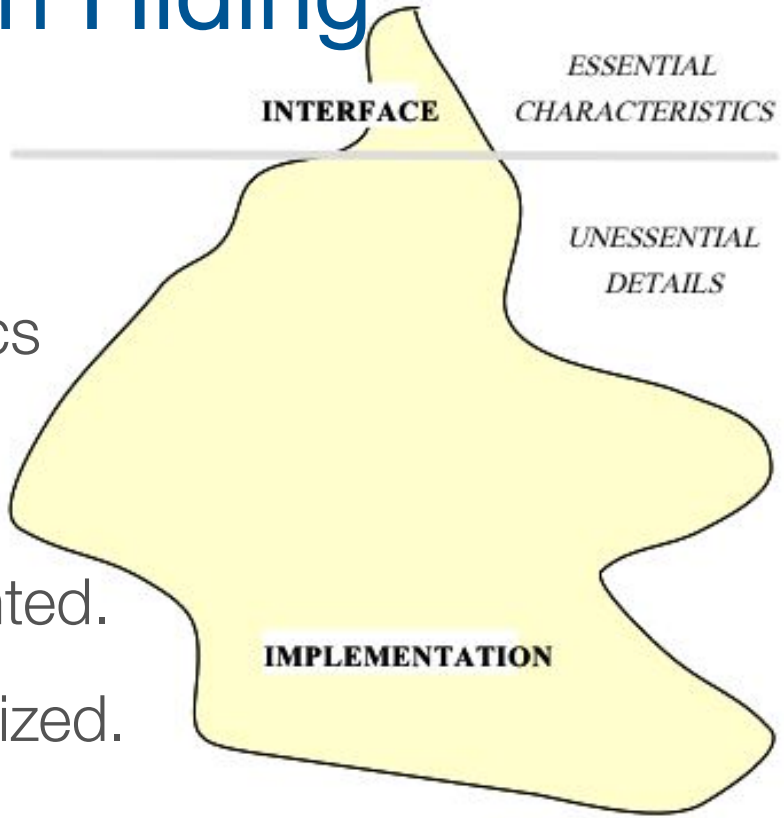
- `path` <string> | <Buffer> | <URL> | <integer> filename or file descriptor
- `options` <Object> | <string>
  - `encoding` <string> | <null> **Default:** `null`
  - `flag` <string> See support of file system `flags`. **Default:** `'r'`.
  - `signal` <AbortSignal> allows aborting an in-progress readFile
- `callback` <Function>
  - `err` <Error> | <AggregateError>
  - `data` <string> | <Buffer>

Asynchronously reads the entire contents of a file.

```
import { readFile } from 'node:fs';
```

APIs Provide 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/]

APIs Are Forever

APIs accumulate clients and become long-lived contracts

- Once published, an API becomes a dependency for many teams, services, and applications.
- Changes to a widely-used API can break thousands of clients.

A good API amplifies its value

- A well-designed API makes developers more productive and reduces errors.
- A stable, predictable API encourages reuse and long-term adoption.

But APIs must evolve

- Requirements change, systems grow, and new capabilities are needed.
- Internals should change freely — the API surface should change rarely.
- When an API need to change, it must do so *without breaking clients*.

API Design: Start with the Task

What is the goal of this API?

- What task should a developer be able to accomplish easily?
- What sequence of actions should feel natural and predictable?
- What mistakes should the API help the developer avoid?

→ The API should present an abstraction that matches how developers think about the task.

UserStore Usability Flaws

As a client of this API, we had to…

- Look in multiple places to get the result of one task
- Write extra code to do a simple task
- Touch important internal variables
- Infer ambiguous behaviour

API Design Principles

Principle	Test
Information hiding	Is internal state or structure exposed that callers shouldn't see or touch?
Obliviousness	Does the caller need to know something about the implementation to use the API correctly?
Easy to use/Hard to misuse	Can the type system prevent incorrect usage? (e.g. wrong call order, incorrect parameters)
Contract completeness	Do the signatures and JSDoc fully capture what can go wrong and what callers can rely on?

December 18, 2025 • malcolm-matalka

github api tooling automation terrateam

GitHub changed their API without warning and broke our production system

> In 2022, I **wrote about GitHub Actions API limitations**, specifically that workflow_dispatch didn't return a run ID. GitHub finally fixed it this week. Then broke Terrateam and likely other systems that haven't noticed yet.

There is a special class of failure that only shows up once people actually depend on you.

Not a bug. Not an outage. Not a missing feature.

A contract violation.

On this page

- What changed
- Why this broke things
- The real problem
- What should have happened
- What we are doing about it
- APIs are promises

Change Severity

- An API is an agreement with a client.
- Changing an API changes the agreement with the client
- **Goal**: understand how a change to code will affect the client

WHEN
will the client experience the change?

WHAT
will the client need to do to adopt the change?

Change Severity Levels

- MAJOR when making *incompatible* API changes
  - Changes that require clients to update their code
- MINOR when adding *backwards-compatible functionality*
  - Structural changes that do not require clients to update their code (e.g., adding new methods, optional parameters, etc.)
- PATCH when making backwards-compatible improvements
  - Behavioural changes that are not essential for clients to adopt and do not require them to update their code to adopt (e.g., minor bug fixes, performance improvements, etc.)
- NONE when making changes that do not affect clients
  - e.g. Refactoring

API Design Summary

- **APIs define stable boundaries.** They act as contracts that many clients depend on.
- **Good API design minimizes change.** Clear tasks, small surfaces, and hiding internals keep APIs stable.
- **Internals should change freely; APIs should not.** A well-designed API protects clients from internal evolution.
- **But APIs must evolve sometimes.** Important to consider whether a change will break clients.