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

06b - Boundaries

CPSC310

Nick Bradley & Kyle Chin

Midterm

Details and instructions posted on Piazza!

- Oct 29th 19:00-21:00
- Location based on your last name (see list on Piazza)
- On paper, some code writing required
- Practice PL problems & practice exam

Learning goals

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

- Explain how an interface shaped like its dependency lets a storage change leak into business logic and its tests.
- Design a port in the core's terms, and an adapter that implements it.
- Separate the core (business logic) from the edge (I/O, parsing, wiring).
- Choose the test level (unit, integration or e2e) that should catch a given bug.

Real systems have two kinds of code

Business logic

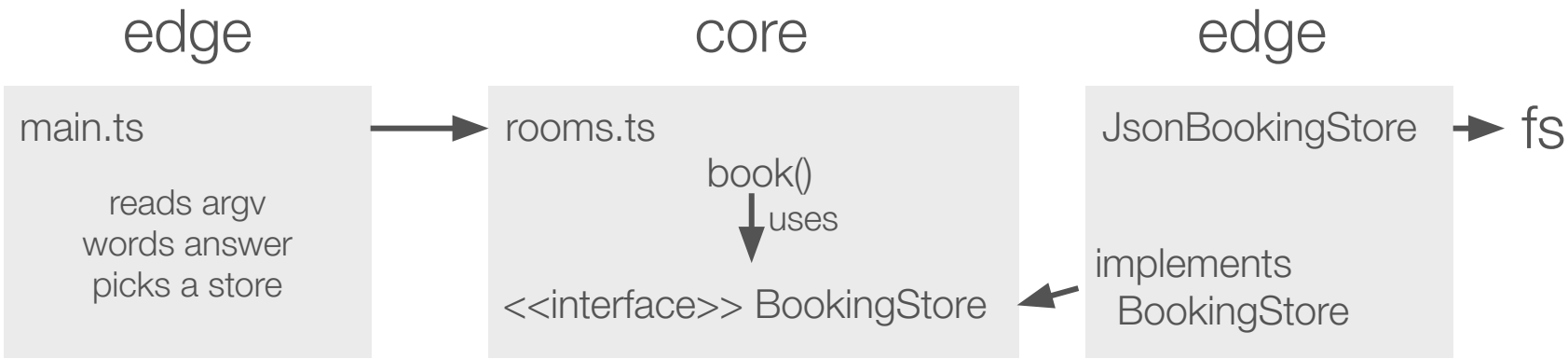
- The rules that make the system worth having. e.g., a room can't be double-booked.
- Changes when business concerns change.

Everything else

- Moves data in and out: files, databases, user interfaces, other services.
- Logging, caching, authentication.
- Changes when the outside world changes.

Goal: keep the business logic separate from everything else, so each changes for its own reasons. Builds on the mechanisms we've already seen: interfaces and test doubles (04a), Adapters (05a), and choosing the axis of change (06a).

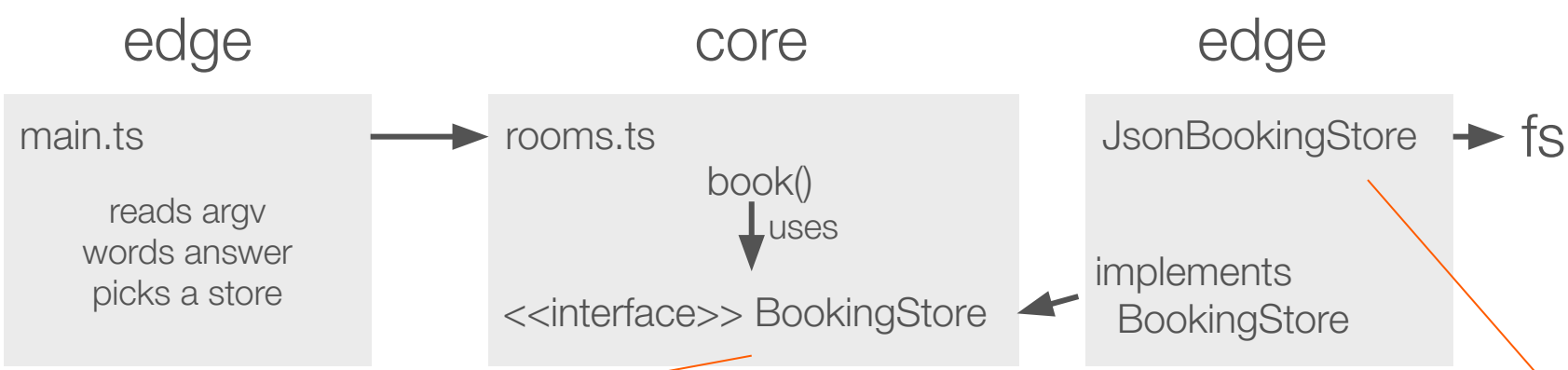
Core and Edges



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)*

Where did each responsibility go?

Responsibility	01-booking	02-booking
Read command line	main.ts	main.ts (edge)
Check the booking rules	book() in bookings.ts	refusalFor() in core
Keep a booking if valid	book() in bookings.ts	book in core
Read/write the file	book() in bookings.ts	JsonBookingStore (adapter)
Know the file's field names	book(), the rules and tests	JsonBookingStore (adapter)
Word the answer	book() in bookings.ts	main.ts (edge)
Pick where bookings are kept	bookings.ts (hard-coded path)	main.ts (edge)

Mixed Responsibilities

In 01-booking, one function had most of the jobs, so:

- **Every change risked the others.** The field rename was a storage change, and it edited the overlap rule.
- **No job could be tested alone.** Testing the overlap rule needed JSON in and a worded sentence out.
- **No job could be reused alone.** A web frontend needs the rules, but not sentences worded for the command line.

Now each change lands in one place

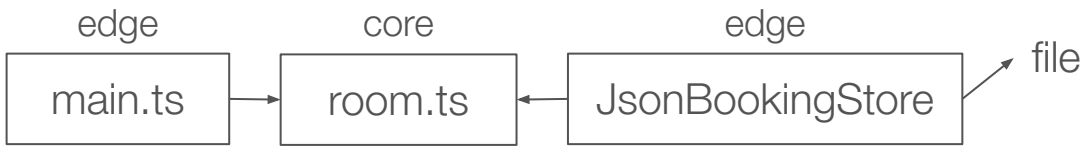
Change	01-booking	02-booking
Tool renames a field in booking.json	book() and its tests	JsonBookingStore
New rule (e.g., at most 2 bookings per day)	book(), and tests full of JSON	the core, and plain-object unit tests
New frontend: (e.g., a web page)	copy the rules out of book()	A new edge that calls the existing book()

Isolating the business logic means:

- It changes only when a rule changes.
- It's tested with plain values: no files, no wording.
- Any edge can use it: the command line, a web server, the tests.

Single responsibility principle: each part of the system has one reason to change.

How can we test the system?



		checks	how many	speed
E2E	◄——————————————————————►	wiring	1–2 tests	Slowest, most realistic
Integration	◄——————————————————►	translation	a few	fast with a temp file
Unit	◄——————► + Test Double	business logic	many	milliseconds

Summary: Boundaries

- Business logic goes in the *core*; everything else is the *edge*. The core can call the edge only through interfaces it owns, so every dependency points inward.
- A *port* is owned by the core and written in its terms. An *adapter* does the translating.
- Test each thing at the narrowest level that can see it. That keeps tests from failing for the wrong reasons.