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

08a - Midterm Review

CPSC310

Nick Bradley & Kyle Chin

Announcements

REST is not on the midterm (will be covered next week)

Midterm review materials:

- Practice Midterm
- Cheatsheet
- PL labs
- Activities (in the lecture_code repo)

What is a Design Pattern?

A tried and true solution to a commonly encountered problem.

A design pattern is not:

- A class diagram you copy
- A framework or library
- A rule you must always follow

A design pattern is:

- A *reusable way of thinking* about a recurring design problem
- What emerges when you keep a likely change local
- A vocabulary for discussing trade-offs

Why / Why Not Design Patterns?

Easy to blindly ‘force’ a pattern into a context where it does not make sense.

✅ Patterns help when:

- Change is likely
- Variability already exists
- Coupling is becoming painful

❌ Patterns hurt when:

- Requirements are stable
- There’s only one concrete case
- The abstraction cost outweighs future benefit

Designing for Change

Every system you build will be changed:

- New requirements: "Can we add a QA role to the org chart?"
- New dependencies: "We're switching payment providers."
- New scale: more users, more developers, more code

We can't predict what the change will be, but we can often predict where it will land.

When the change arrives, how many places do you have to edit, and how much unrelated code do you have to touch?

How to understand a design pattern

1) **Change**: which decision is likely to vary?
2) **Pain**: where does that decision scatter (a coupling problem) or tangle (a cohesion problem) in the current code?
3) **Mechanism**: how does the pattern give that decision one home?
4) **Cost**: what indirection did we add, and was it worth it?

Finding the Mechanism

1. **Abstraction**: what new interface or class gives the decision a single home?
2. **Polymorphism/Obliviousness**: what methods does that abstraction declare, and who calls them without knowing the concrete type? Can someone add a new implementation and override them safely?
3. **Trigger**: when are those methods called, and by whom?

Summary: Design Patterns

Each pattern puts an interface between the client and the thing that varies. A new variant is a new class; the client doesn't change.

Pattern	Interface	New Variant	Unchanged
Adapter	MusicSource	YouTube Music	Reports
Composite	Employee	DataScientist	Manager, org totals
Decorator	Drink / AddOn	Mocha add-on	Menu drinks, cart
State	PhoneState	Lockout state	Phone

OCP: Choosing the Axis

Every extension point costs indirection: an interface, a dispatch, more classes. Spend it only where variation will actually arrive.

Before adding an extension point, ask:

- What will vary? (new services, roles, add-ons, states…)
- What must stay stable for the code that depends on it?
- Is that variation *really* likely?

Every Pattern has a Closed Axis

Open to new implementations, closed to new operations.

Pattern	Cheap	Expensive (touches every class)
Adapter	new music service	new query on MusicSource
Composite	new role	new operation (e.g., avg salary)
Decorator	new add-on	new property (e.g., calories)
State	new state	new button

Open/Closed Principle (OCP)

A unit of code (function, class, module) should be:

- Open for extension: new behaviour can be added
- Closed for modification: adding the behaviour doesn't change code that already works.

How: depend on an abstraction, then extend by writing a new implementation of it. The four patterns are four ways of putting the abstraction in place.

Recall: Two Axes of Testability

Controllability

The degree to which a test can determine the inputs and state that the code under test depends on.

Observability

The degree to which a test can inspect the results and effects that the code under test produces.

What makes this code hard to test?

```
function printStatus(offeringId: string): void {
    const offering = findOffering(offeringId);
    if (offering === undefined) {
        console.log(`no offering '${offeringId}'`);
        return;
    }
     const offeringStart = Date.parse(SEASON[offering.season])
    const daysLeft = offeringStart - Date.now()
```

What makes this code hard to test?

```
function printStatus(offeringId: string): void {
    const offering = findOffering(offeringId);
    if (offering === undefined) {
        console.log(`no offering '${offeringId}'`);
        return;
    }
     const offeringStart = Date.parse(SEASON[offering.season])
    const daysLeft = offeringStart - Date.now()
```

Observability: No way to assert on log output

What makes this code hard to test?

```
function printStatus(offeringId: string): void {
    const offering = findOffering(offeringId);
    if (offering === undefined) {
        console.log(`no offering '${offeringId}'`);
        return;
    }
     const offeringStart = Date.parse(SEASON[offering.season])
    const daysLeft = offeringStart - Date.now()
```

Observability: No way to assert on log output

Controllability: No way to set the date

Test Doubles

- A test double stands in for a dependency that the code under test uses.
- A *stub* controls what the dependency gives back, like our approving and declining readers.
- A *spy* records what the code under test sent to it, like our recording reader.

When to use a test double

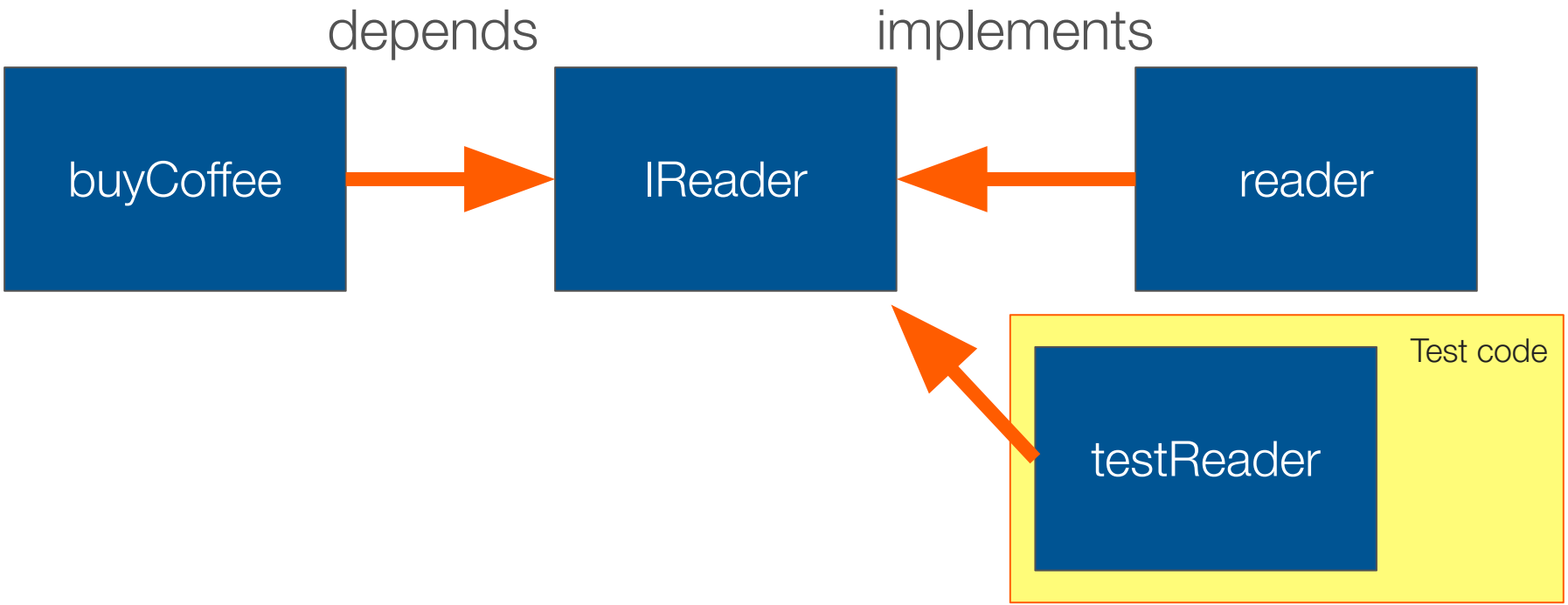
Use one when the real dependency is:

- uncontrollable: a bank, a clock
- unobservable: you can't see what was sent
- unsafe or slow: real money, the network

Otherwise, use the real thing. If it's fast, deterministic and has no side effects, don't double it.

DIP Enables Test Doubles

testReader is called a *Test Double*



Valid Test Doubles

Any object with a charge method that returns a boolean compiles as a IReader.

```
class FakeWorkingReader implements IReader {
    charge = (n: number) => {
        if (n < 0) throw new Error("Invalid charge!");
        return true;
    };
}
```

Is compiling good enough?

- What does IReader guarantee buyCoffee? Where is that written down?
- Does this double keep that guarantee? The declined test fails: who's wrong, buyCoffee or the double?

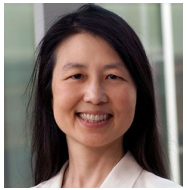
Liskov Substitution Principle

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]

Recap: Testability with test doubles

- **To test code that calls something**, depend on an interface and pass the dependency in (DIP).
- **Test doubles stand in for it.** Stubs control what comes back, and spies record what goes out.
- **A double has to keep the interface's contract**, not just its shape (LSP).
- **The cost:** another layer; doubles can drift from the real thing & need to be updated when the interface changes. The real implementation still needs tests of its own.