COMP 3550

6.4 — CROSS-CUTTING CONCERNS & ERROR-HANDLING STRATEGIES

Week 6: Alternatives to Inheritance & Dependency Injection

WHAT ARE CROSS-CUTTING CONCERNS?

Code that weaves itself into everything

WHAT ARE CROSS-CUTTING CONCERNS?

Code that weaves itself into everything

Definition:

- Logic or functionality that affects multiple parts of a system.
- Often repeated across modules if not handled centrally.

WHAT ARE CROSS-CUTTING CONCERNS?

Code that weaves itself into everything

Definition:

- Logic or functionality that affects multiple parts of a system.
- Often repeated across modules if not handled centrally.

Examples:

- Logging
- Authentication & authorization
- Input validation
- Error-handling & recovery
- Performance monitoring
- String Literals & Constants

WHAT ARE CROSS-CUTTING CONCERNS?

Code that weaves itself into everything

Definition:

- Logic or functionality that affects multiple parts of a system.
- Often repeated across modules if not handled centrally.

Examples:

- Logging
- Authentication & authorization
- Input validation
- Error-handling & recovery
- Performance monitoring
- String Literals & Constants

The Problem:

- Duplication
  - Same logic repeated in many classes.
- Tangling
  - Business logic mixed with unrelated concerns.

BETTER STRUCTURE: MIDDLE LAYERS OR INTERCEPTORS

Two Main Approaches:

WHAT HAPPENS WHERE?

UI/Presentation Layer

catch everything, show friendly message

Business/Logic Layer

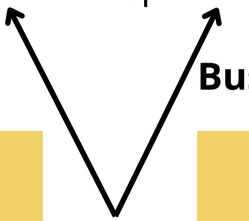
wrap/transform into domain-specific errors

Database/Persistence Layer

catch low-level issues, throw DB-specific errors

WHAT HAPPENS WHERE?

UI/P sends data/requests to
B/L and catches exceptions



UI/Presentation Layer

catch everything, show friendly message

Business/Logic Layer

wrap/transform into domain-specific errors

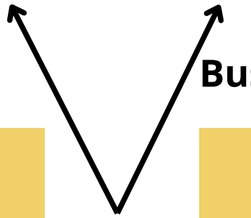
Database/Persistence Layer

catch low-level issues, throw DB-specific errors

WHAT HAPPENS WHERE?

UI/P sends data/requests to B/L and catches exceptions

B/L sends data to UI/P, throws UI/P friendly exceptions



UI/Presentation Layer

catch everything, show friendly message

Business/Logic Layer

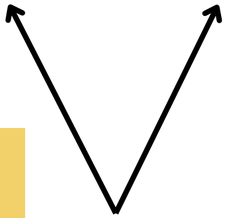
wrap/transform into domain-specific errors

Database/Persistence Layer

catch low-level issues, throw DB-specific errors

WHAT HAPPENS WHERE?

B/L requests data from D/P for UI/P use and catches D/P exceptions to be thrown as friendly UI/P exceptions



UI/Presentation Layer

catch everything, show friendly message

UI/P sends data/requests to B/L and catches exceptions

Business/Logic Layer

wrap/transform into domain-specific errors

B/L sends data to UI/P, throws UI/P friendly exceptions

Database/Persistence Layer

catch low-level issues, throw DB-specific errors

WHAT HAPPENS WHERE?

B/L requests data from D/P for UI/P use and catches D/P exceptions to be thrown as friendly UI/P exceptions

D/P connects to database to retrieve data stored and returns it to B/L or catches low-level and throws custom/specific exceptions

UI/Presentation Layer

catch everything, show friendly message

UI/P sends data/requests to B/L and catches exceptions

Business/Logic Layer

wrap/transform into domain-specific errors

B/L sends data to UI/P, throws UI/P friendly exceptions

Database/Persistence Layer

catch low-level issues, throw DB-specific errors

BETTER STRUCTURE: MIDDLE LAYERS OR INTERCEPTORS

Two Main Approaches:

- **Shared Utility / Service Classes**
  - Centralize cross-cutting logic (logging, validation, security)
  - Reusable across multiple modules
  - Example: LoggerService, AuthService, ValidationService

BETTER STRUCTURE: MIDDLE LAYERS OR INTERCEPTORS

Two Main Approaches:

- **Shared Utility / Service Classes**
  - Centralize cross-cutting logic (logging, validation, security)
  - Reusable across multiple modules
  - Example: LoggerService, AuthService, ValidationService
- **Interceptors / Middleware**
  - Code that runs before or after main logic automatically
  - Common in frameworks: servlet filters, Spring interceptors, Express middleware
  - Example: Every request passes through logging/auth layers before reaching controllers

BETTER STRUCTURE: MIDDLE LAYERS OR INTERCEPTORS

Two Main Approaches:

- **Shared Utility / Service Classes**
  - Centralize cross-cutting logic (logging, validation, security)
  - Reusable across multiple modules
  - Example: LoggerService, AuthService, ValidationService
- **Interceptors / Middleware**
  - Code that runs before or after main logic automatically
  - Common in frameworks: servlet filters, Spring interceptors, Express middleware
  - Example: Every request passes through logging/auth layers before reaching controllers

Strategy Pattern for swapping behaviors like logging or validation without changing core logic

- E.g., Logger interface → FileLogger, DatabaseLogger, ConsoleLogger

MINI EXAMPLE OF INTERCEPTOR

```
interface Logger {
    void log(String message);
}

class FileLogger implements Logger { /* write to file */ }
class ConsoleLogger implements Logger { /* write to console */ }

class OrderProcessor {
    private Logger logger;
    OrderProcessor(Logger logger) { this.logger = logger; }
}
```

Instead of sprinkling Logging methods and information throughout the program, we have our own classes which contain this behaviour and all code passes through them as part of processing, thus keeping our layers clean.

CENTRALIZED ERROR HANDLING

One place to catch, log, and respond.

Best Practice:

- Catch and handle errors at a top-level boundary (e.g., controller, service manager, request handler).
- Log errors consistently in one place.
- Return useful, safe responses to the caller.

CENTRALIZED ERROR HANDLING

One place to catch, log, and respond.

Best Practice:

- Catch and handle errors at a top-level boundary (e.g., controller, service manager, request handler).
- Log errors consistently in one place.
- Return useful, safe responses to the caller.

Why Not Catch Everywhere?

- Catching and ignoring hides problems.
- Scattered try-catch blocks lead to duplication and inconsistency.
- Makes debugging much harder.

CENTRALIZED ERROR HANDLING

One place to catch, log, and respond.

```
void processOrder(Order order) {
    try {
        // business logic
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Scattered logging, may be inconsistent, swallows exceptions from front end perspective

```
public void handleRequest(Request req) {
    try {
        orderProcessor.process(req.getOrder());
    } catch (Exception e) {
        logger.error("Error processing request", e);
        sendErrorResponse();
    }
}
```

All logging dependencies are contained, clean and consistent.

FAILING FAST & CLEAR MESSAGES

Fail Fast:

- Validate inputs as soon as possible, ideally in the constructor or at the controller boundary.
- Avoid letting bad data flow deeper into the system where it’s harder to trace.

Clear Messages:

- Use specific exception types to indicate what went wrong.
- Include helpful, actionable messages (what failed and why).
- Avoid generic "Something went wrong" or swallowing exceptions.

FAILING FAST & CLEAR MESSAGES

Bad

```
public void processOrder(Order order) {
    // ... lots of logic ...
    if (order.getItems().isEmpty()) {
        throw new RuntimeException("Error");
    }
}
```

FAILING FAST & CLEAR MESSAGES

Good!

```
public OrderProcessor(Order order) {
    if (order.getItems().isEmpty()) {
        throw new InvalidOrderException("Order must contain at least one item");
    }
}
```

RESILIENCE THROUGH ERROR BOUNDARIES

Key Idea:

- Isolate risky code so failures don’t propagate unchecked.
- One module’s failure shouldn’t bring down the whole system.
- Wrap unstable calls with protective boundaries.

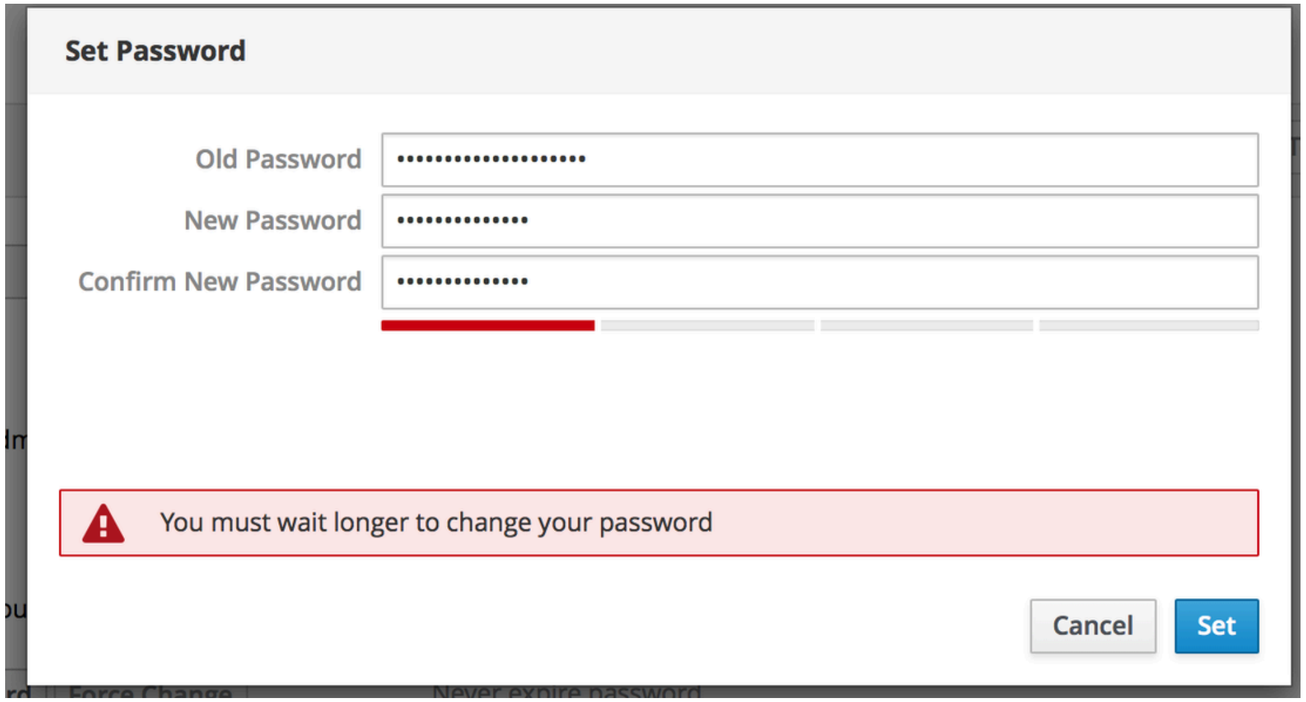
Example Scenario:

- You call a 3rd-party API for shipping rates.
  - API fails or returns bad data.
- Without boundaries:
  - raw exceptions crash your order processing.
- With boundaries:
  - API wrapper catches the raw error and throws a safe, internal exception that your system can handle.

RESILIENCE THROUGH ERROR BOUNDARIES

```
class ShippingApiWrapper {
    ShippingRate getRate(Address address) {
        try {
            return api.getRate(address);
        } catch (ApiException e) {
            throw new ShippingServiceUnavailableException(
                "Could not retrieve shipping rate", e
            );
        }
    }
}
```

USER FRIENDLY?



How long must I wait? What makes it a strong vs weak password? WHAT DO I DO?!

USER FRIENDLY?

User messaging should be clear and actionable. I should know what I, as a user, need to do investigate, or restart, or who to contact based on the error message.

PROJECT PAUSE & REFLECT

Add a validation or error layer to one part of your codebase this week.

Be sure you store messaging in a central location to make it easier to reuse. Take some time to plan out what this should look like.