ECS 160 – Discussion
Unit Tests, CI/CD

Instructor: Tapti Palit

Teaching Assistant: **Gabe Bai**

Agenda

- **Unit Tests**
- CI/CD

Review of Unit Tests

- Run snippets of code to check behavior
- Manual testing
  - System.out.println()
- Automated testing
  - assertEquals()
- Assertions should imply a logical ***invariant*** as opposed to an expected outcome
- A logical invariant is like a *contract* in your code
  - e.g. “the result of this calculation is positive”

What to Test

- Each test should verify specific behavior
  - Why?
- Each test should be independent of the others
  - Why?
- Tests shouldn't just cover expected behavior, but **unexpected behavior too**
  - Why?

A Bad Test Case

```
@Test
void testAddAndSubtract() {
    Calculator c = new Calculator();
    assertEquals(5, c.add(2, 3));
    assertEquals(1, c.subtract(3, 2));
}
```

Improving That Test Case

```
@Test void testAdd() { ... }

@Test void testSubtract() { ... }
```

Refactoring Test Cases

- Unit tests are a safety net for refactoring
  - Behavioral contract
- We want to test behavior, not implementation

```
@Test
void addShouldReturnSum() {
    Calculator c = new Calculator();
    assertEquals(5, c.add(2, 3));
}
```

Same Behavior

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

```
public int add(int a, int b) {
    return Integer.sum(a, b);
}
```

Testing Implementation

```
@Test
void addUsesPlusOperator() {
    String code = getSourceCodeOf("Calculator");
    assertTrue(code.contains("+"));
}
```

Preventing Flaky Tests

- **Flaky** tests – non-deterministic tests
  - Pass/fail unpredictably
- We want to design tests that are stable, even when automated

```
@Test
void testFetchDataFromApi() {
    ApiClient client = new ApiClient();
    String data = client.fetch("https://api.weather.com/today");
    assertTrue(data.contains("temperature"));
}
```

Testability

- Coupling and cohesion
  - Low coupling, high cohesion
- Dependency injection
  - We should be able to pass dependencies into constructors and methods, instead of just hardcoding them
- Testability ~ good design
  - If we can’t easily test your component, you probably don’t want to write your component like that

Coverage

- How much of your source code has been tested?

```
@Test void testEverything() {
    new Calculator().add(1,2);
    new Calculator().subtract(2,1);
}
```

- 70% meaningful coverage or 100% meaningless coverage?
- No simple answer about the good coverage - amount of testing necessary depends on a number of factors

What Makes a Good Unit Test?

- Focused on **one behavior**
- Quick, low latency
- Deterministic
- Independent of other tests
- Clear

EX: Code Review

```
public class BankAccount {
  private int balance;

  public BankAccount(int open) {
    // open is opening balance
    this.balance = open;
  }

  public void deposit(int amount) {
    balance += amount;
  }

  public void withdraw(int amount) {
    if (amount > balance) {
      throw new RuntimeException("oops");
    }
    balance -= amount;
  }

  public int getBalance() { return balance; }

}
```

Areas to Flag

- *open* >= 0
- *amount* in withdraw() should be positive
- withdraw() should have a descriptive exception
- No issues with getBalance()

Agenda

- Unit Tests
- **CI/CD**

CI: Continuous Integration

- Frequently merge small changes
- These changes should trigger automated tests
- Catch issues early

CD: Continuous Delivery/Deployment

- If CI passes, immediately deliver
- We can then deploy as soon as we get manual approval (if implemented)
- Deployment is a process

Pipeline Design

- Very broadly…
- Trigger (e.g. opening a pull request)
- Checkout / Build + Compile + Package
- Unit tests
- Quality checks
- Integration tests
- End-to-end tests
- Handle artifacts, move to staging (env close to prod)
- Deploy, then continuously verify

Best Practices

- “Build once”
  - … and test the same **artifact** across all of staging
- Parallelism
  - Run independent stages in parallel e.g. linters vs unit tests
- Fail fast, fail early
  - Greedily test based on ease, and halt pipeline if anything is wrong
- Security
  - Least privilege – limit who can change pipeline scripts (amongst others)
- Observability
- Scaling and maintainability

Challenges

- Pipeline as a bottleneck
  - Good pipeline design can mean the difference between a 5 hr build time and a 30 min one
- Secrets management issues
  - Keep all your secrets (API Keys for example) out of config files
  - Use a secret manager
- Flaky tests
  - Weakens confidence in CI
- Environment drift
  - Don’t want “works locally, fails in prod”