Summary of Software Testing - Teaching Notes

1. Introduction to Software Testing

Objectives of Software Testing

Defect Detection

- Identify and fix bugs before release.
- Example: Function intended to add numbers but uses the wrong operator.

```
def add(a, b):
    return a - b  # Incorrect operator
```

Quality Assurance

- Ensures software meets **functional and non-functional** requirements.
- Example: Checking if a login system works correctly.

```
@Test
public void testLogin() {
    User user = new User("alice","password123");
    assertTrue(user.login());
}
```

Validation vs. Verification

Validation: "Are we building the right product?" (*user needs*).
Verification: "Are we building the product right?"(*code correctness*).

2. Key Principles of Testing

Early Testing (Test-Driven Development - TDD)

- Write tests before implementing the actual code.

```
def test_multiply():
    assert multiply(3, 4) == 12  # Fails initially

# Implement function
def multiply(a, b):
    return a * b
```

Defect Clustering

- Most defects are in a small number of modules.
- Example: Payment processing modules often have critical bugs.

Pesticide Paradox

- Repeating the same tests becomes ineffective.
- Solution: Update test cases to **include edge cases**.

```
def test_divide_by_zero():
    with pytest.raises(ZeroDivisionError):
        divide(10, 0)
```

Testing Shows Presence of Defects, Not Absence

- Just because tests pass doesn't mean software is bug-free.

Exhaustive Testing is Impossible

- A function that takes two 32-bit integers has $\mathbf{2^{64}}$ possible input combinations → **Not feasible to test all.**

Testing is Context-Dependent

- Different industries require different testing strategies:
    - **Banking apps**: Security-critical.
    - **Gaming apps**: Focus on performance.

Absence of Errors Fallacy

- Even if software has no bugs, it may **fail to solve the intended problem**.

3. Types of Software Testing

Functional Testing (Does the software do what it's supposed to?)

✅ **Manual Testing** – Human-driven testing.
✅ **Automated Testing** – Uses tools like **Selenium, PyTest, JUnit**.

```
from selenium import webdriver

def test_google_search():
    driver = webdriver.Chrome()
    driver.get("https://www.google.com")
    search_box = driver.find_element("name", "q")
    search_box.send_keys("software testing")
    search_box.submit()
    assert "software testing" in driver.title
    driver.quit()
```

✅ **API Testing** – Validate API responses.

```
import requests
def test_api_response():
    response =
requests.get("https://api.example.com/users/1")
    assert response.status_code == 200
    assert response.json()["id"] == 1
```

Non-Functional Testing (Performance, Security, Usability)

Performance Testing – Load and stress testing using **JMeter**.
Security Testing – Identify vulnerabilities (e.g., SQL injection attacks).
Usability Testing – Check user experience.

Static vs. Dynamic Testing

✅ **Static Testing** – Code review, linting, without execution.
✅ **Dynamic Testing** – Running the program to detect runtime defects.

4. Software Testing Levels

Unit Testing

📌 Definition: Tests individual software components (functions/methods) independently.
✅ When to Use:

- Checking specific logic, loops, and input validation.
- Testing error handling and edge cases.

⚙️ Techniques:

- White-box Testing (check internal code logic: statements, branches, paths).
- Mocking/Stubbing (isolate external dependencies like databases, APIs).

👍 Pros:

- Quick, targeted tests.
- Early detection of bugs.
- Easy debugging.
- Easily automated.

👎 Cons:

- Doesn't catch integration/system-wide problems.
- Mocking can add complexity.
- Risk of false confidence (passing tests ≠ fully working system).

🛠️ Common Tools:

- JUnit, pytest, Mockito (for mocking)

📌 Best Practices:

- Write clear, simple, and targeted tests.
- Mock external systems.
- Aim for high code coverage (e.g., 80%).

```
def calculate_discount(price, discount_percent):
    if discount_percent < 0 or discount_percent > 100:
        raise ValueError("Invalid discount")
    return price * (1 - discount_percent / 100)

# Unit test
def test_calculate_discount():
    assert calculate_discount(100, 20) == 80.0
    assert calculate_discount(100, 0) == 100.0
    with pytest.raises(ValueError):
        calculate_discount(100, 110)
```

Integration Testing

Purpose: Verifies interactions between different components of a software system.
Use When: Ensuring modules integrate correctly.
Approaches:
Big Bang: Test all modules together (risky for large systems
Incremental:
Top-down: Test higher-level modules first, using stubs for lower modules.
Bottom-up: Test lower-level modules first, using drivers for higher modules.
Sandwich: Combines top-down and bottom-up methods.
✅ Pros:
Catches integration issues early.
Ensures components integrate correctly.
Supports system stability.
❌ Cons:
Slower execution due to setup (databases, networks).
External dependencies (APIs, third-party services) can cause instability or flaky tests.
Complex debugging since failures involve multiple components.
🛠️ Tools:
API Testing: Postman, RestAssured
Messaging/API Integration: Kafka, RabbitMQ
REST APIs: Postman, RestAssured
📌 Key Takeaway: Integration testing validates correct interactions between modules, despite potential slowness, complexity, and external instability.

```
import requests
def test_payment_integration():
    cart_response =
requests.post("https://api.example.com/cart", json={"item":
"book", "price": 50})
    cart_id = cart_response.json()["cart_id"]
    payment_response =
requests.post("https://api.example.com/pay", json={"cart_id":
cart_id, "card": "4111-1111-1111-1111"})
    assert payment_response.status_code == 200
```

System Testing

✅ **When to Use:**

- Validating complete user workflows (e.g., login → checkout).
- Checking performance, security, and usability.

🎯 **Types:**

- **Functional:** User scenarios (end-to-end).
- **Performance:** Load/stress testing.
- **Security:** Penetration testing, vulnerability scanning.
- **Compatibility:** Cross-browser and device tests.

👍 **Pros:**

- Comprehensive: Tests entire system realistically.
- Identifies security and performance issues.

⚠️ **Cons:**

- Slow and resource-intensive.
- Complex setup.

🛠️ **Common Tools:**

- Selenium, Cypress (functional)
- JMeter, Gatling (performance)
- OWASP Zap (security)

📌 **Best Practices:**

- Simulate real-world scenarios accurately.
- Include security and load testing regularly.

```
// Java + Selenium example
@Test public void testLogin() {

WebDriver driver = new ChromeDriver();

driver.get("https://app.example.com/login");
driver.findElement(By.id("username")).sendKeys("user@example.com");
driver.findElement(By.id("password")).sendKeys("securepassword");
driver.findElement(By.id("submit")).click();

WebElement welcomeMessage = driver.findElement(By.id("welcome"));
assertTrue(welcomeMessage.getText().contains("Welcome, User"));
driver.quit();
}
```

Acceptance Testing

What it is:

- Validates software from the user's perspective, ensuring it meets business and user requirements.

Types:

- **User Acceptance Testing (UAT)**
  End-users validate system in production-like environment.

- **Alpha/Beta Testing**
  Early version tested by internal (Alpha) or external (Beta) users to collect feedback.

- **Contract Acceptance
  Checks if the system meets specific legal or regulatory standards (e.g., GDPR, HIPAA).

Tools:

- Behavior-Driven Development (BDD): Cucumber, SpecFlow, Behave
  Bridges business requirements and testing code clearly.

```
Feature: Password Reset

Scenario: Successful password reset
  Given a user has forgotten their password
```

```
  When they request a password reset with a valid email
  Then they receive a reset link via email
```

```
from behave import *

@given('a user has forgotten their password')
def step_user_forgot_password(context):
    context.user_email = 'user@example.com'

@when('they request a password reset with a valid email')
def step_request_reset(context):
    context.response =
request_password_reset(context.user_email)

@then('they receive a reset link via email')
def step_check_email(context):
    assert context.response.status_code == 200
    assert email_received(context.user_email, 'Password
Reset')
```

Pros:

- Ensures product meets user expectations.
- Validates regulatory compliance.
- Enhances user and stakeholder satisfaction.

Cons:

- Criteria can be subjective.
- Can lead to additional testing complexity.
- Setup requires realistic user environments.

✅ **Best Practices:**

- Engage users and stakeholders early.
- Clearly define acceptance criteria.
- Utilize realistic scenarios and data for thorough validation.

---

5. Testing Techniques

✅ **White-Box Testing** (tests internal logic)
Objective: Validate internal code structures, logic, and paths
○ Focus: Code coverage, error handling, and algorithmic correctness.
○ Pros:
■ Uncovers hidden code flaws (e.g., dead code, boundary errors).
■ High precision in debugging.
○ Cons:
■ Time-consuming for complex systems.
■ Requires code access and technical expertise

- **Statement Coverage**: Ensure every line runs at least once.

```
def calculate_grade(score):
if score >= 90:
return "A"
elif score >= 80:
return "B"
else:
return "C"
# Test case for statement coverage
def test_calculate_grade():
assert calculate_grade(95) == "A" # Covers 'if'
assert calculate_grade(85) == "B" # Covers 'elif'
assert calculate_grade(70) == "C" # Covers 'else
```

- **Branch Coverage**: Test all conditional branches.

```
def is_valid(x, y):
return (x > 0) and (y < 10)
# Test all conditions:
assert is_valid(5, 5) == True # (True and True)
assert is_valid(-1, 5) == False # (False and True)
assert is_valid(5, 15) == False # (True and False)
```

- **Path Coverage**: Test all possible execution paths.

```
def process_orders(orders):
total = 0
for order in orders:
if order["status"] == "valid":
total += order["amount"]
return total
# Test multiple paths
orders = [
{"status": "valid", "amount": 100},
{"status": "invalid", "amount": 50}
]
assert process_orders(orders) == 100 # Covers loop and
condition
```

- More on Coverage

```
def check_access(is_admin, is_verified, balance):
    # Rule 1: Admins always have access
    if is_admin:
        return "Admin Access"
    # Rule 2: Non-admins need verification and sufficient balance
    elif is_verified and balance >= 100:
        return "User Access"
    # Rule 3: Deny access otherwise
    else:
        return "Access Denied"
```

- Statement Coverage
  - All lines are executed (100% statement coverage).
  - Not all branches/paths are tested (e.g. is_verified=True and balance=50 is untested)

Test Case	is_admin	is_verified	balance	Lines Executed
1	True	False	0	if is_admin, return "Admin Access"
2	False	True	200	elif, return "User Access"
3	False	False	0	else, return "Access Denied"

- Branch Coverage
  - All branches (decision outcomes) are tested.

- Not all paths are tested (e.g. is_verified=True and balance=50 is covered, but combinations like is_verified=False and balance=200 are missing).

Test Case	is_admin	is_verified	balance	Branches Covered
1	True	Any	Any	if is_admin → True
2	False	True	200	elif → True
3	False	False	Any	elif → False, else → True
4	False	True	50	elif → False, else → True

- Path Coverage
  - All paths are tested.
  - Path coverage sometimes can be redundant with branch coverage in simple logic.

Test Case	`is_admin`	`is_verified`	`balance`
1	`True`	Any	Any
2	`False`	`True`	`200`
3	`False`	`False`	Any
4	`False`	`True`	`50`

- Mutation Testing
  - Inject artificial defects ("mutants") to evaluate test effectiveness.

```python
# Original code
def add(a, b):
    return a + b

# Mutant (a - b instead of a + b)
def mutant_add(a, b):
    return a - b

# Test should fail for the mutant
def test_add():
    assert add(2, 3) == 5  # Passes for add, fails for mutant_add
```

- Loop Testing
  - Validate loops (e.g., for, while) under different conditions:
    - Zero iterations (loop never executes).
    - One iteration (minimum boundary).
    - Two iterations (check loop variables).
    - N iterations (typical case).
    - N+1 iterations (exit condition check).

```python
def sum_squares(n):
    total = 0
    for i in range(n):
        total += i ** 2
    return total

# Loop Test Cases
assert sum_squares(0) == 0   # Zero iterations
assert sum_squares(1) == 0   # i=0 → 0² = 0
assert sum_squares(3) == 5   # 0² + 1² + 2² = 0 + 1 + 4 = 5
```

✅ **Black-Box Testing** (tests input/output without looking at code)

Test functionality without internal code knowledge.

On larger test you may need to mock the behaviour

○ Focus: Input/output validation against requirements.

○ Pros:

No coding expertise required.

Mimics real user behavior.

○ Cons:

Misses internal logic errors.

Combinatorial explosion in complex systems.

Equivalence Partitioning: Group similar inputs to test fewer cases.

- Equivalence Partitioning
  - Group inputs into classes with similar behavior.
  - Example: Testing a login form's password field:
    - Valid partition: 8-12 characters.
    - Invalid partitions: <8, >12, special characters (if disallowed).

```python
def validate_password(password):
    return 8 <= len(password) <= 12

# Test cases
assert validate_password("Pass1234") == True    # Valid
assert validate_password("Short") == False      # Invalid (too short)
assert validate_password("VeryLongPassword") == False  # Invalid (too long)
```

Boundary Value Analysis (BVA): Test at the **edges** of valid ranges.

- Boundary Value Analysis (BVA)
  - Test values at the edges of partitions.
    - Example: For a password length of 8-12:
      - Test 7, 8, 9, 11, 12, 13.

```python
assert validate_password("A" * 7) == False
assert validate_password("A" * 8) == True
assert validate_password("A" * 12) == True
assert validate_password("A" * 13) == False
```

- Decision Table Testing
  - Map inputs to outputs using combinatorial logic.
  - Example: A discount rule:

Membership	Order Total	Discount
Yes	≥ $100	20%
Yes	< $100	10%
No	Any	0%

```python
def apply_discount(is_member, total):
    if not is_member:
        return 0
    return 20 if total >= 100 else 10

# Parameterized tests (using pytest)
@pytest.mark.parametrize("is_member, total, expected", [
    (True, 150, 20),
    (True, 50, 10),
    (False, 200, 0)
])

def test_discount(is_member, total, expected):
    assert apply_discount(is_member, total) == expected
```

- State Transition Testing
  - Test transitions between system states.

```python
# Assume you dont know the code below but know the state transition.
class LoginSystem:
    def __init__(self):
        self.attempts = 0
    def login(self, password):
        if password == "secret":
            self.attempts = 0
            return "Success"
        else:
            self.attempts += 1
            if self.attempts >= 3:
                return "Account Locked"
            return "Fail"

# Test state transitions
def test_login_states():
    system = LoginSystem()
    assert system.login("wrong") == "Fail"           # State: 1 attempt
    assert system.login("wrong") == "Fail"           # State: 2 attempts
    assert system.login("wrong") == "Account Locked" # State: Locked
```

- Grey-Box Testing
  - Blend white-box and black-box techniques with partial code knowledge.
  - Pros:
    - Balances depth and efficiency.
    - Useful for integration and security testing.
  - Cons:
    - Requires partial code/architecture knowledge.
    - May miss edge cases in hidden logic.
  - Example
    - Validate CRUD operations with SQL query insights.:

```python
def test_user_creation():
    # Black-box: API call to create user
    response = api.post("/users", {"name": "Alice"})
    assert response.status_code == 201

    # White-box: Direct database check
    user = db.query("SELECT * FROM users WHERE name = 'Alice'")
    assert user is not None
```

- Regression Testing (Idk where to put, but it is a common technique)
    - New code changes (e.g. bug fixes, feature updates) do not break existing functionality.
    - Re-running previously executed test cases to verify that the system still behaves as expected
    - It can be either blackbox or whitebox, say for a bank app
        - Black-box regression tests:
            - Verify users can still log in, view balances, and transfer funds via the UI.
        - White-box regression tests:
            - Ensure the updated transfer_funds() method handles currency conversions correctly.

---

6. Automated Testing & CI/CD

✅ **Jenkins for CI/CD**

- Automates testing during development.
- **CI**: Automatically integrates code changes.
- **CD**: Deploys them to production.

✅ **Example Jenkins Pipeline**

```
pipeline {
    agent any
    stages {
        stage('Checkout') {
            steps { git 'https://github.com/example-repo.git'
}
        }
        stage('Build') {
            steps { sh 'make build' }
        }
        stage('Test') {
            steps { sh 'pytest tests/' }
        }
    }
    post {
        always { junit 'test-results.xml' }
    }
}
```

---

7. Test Documentation & Management

✅ **Test Plan** – Defines strategy, scope, environment, and criteria.
✅ **Defect Management** – Track bugs using **Jira, Bugzilla**.
✅ **Metrics & Reporting** – Measure coverage, defect density, execution progress.

Example Test Plan

1. **Scope:** Test login, checkout, and payment flows.
2. **Exit Criteria: 95% test pass rate, 0 critical defects.**
3. **Tools Used:** Selenium, PyTest, Postman.

✅ **Test Traceability Matrix**

Requirement ID	Test Case	Status
REQ-001	Validate login functionality	✅ Passed

✅ **Defect Lifecycle**

Defect ID	Description	Priority	Status
DEF-001	Login fails in Safari	High	In Progress

✅ **Test Closure Report**

- **Summary:** 300 test cases executed, 98% pass rate.
- **Lessons Learned:** Automate smoke tests **earlier**.

Final Summary

1️⃣ Testing detects defects but cannot guarantee bug-free software.
2️⃣ TDD encourages early testing and helps avoid defects before implementation.
3️⃣ Functional testing ensures correct behavior; non-functional tests evaluate performance/security.
4️⃣ Different levels (Unit, Integration, System, Acceptance) ensure reliability.
5️⃣ Automated testing with CI/CD (Jenkins) improves efficiency.
6️⃣ Proper documentation (test plans, defect tracking) ensures accountability.

6. Automated Test Case Generation (More on Formal Verification)

- Program Paths
    - a sequence of statements executed from program entry to exit.

```
void func(int x) {
    if (x > 0) {
        printf("Positive");
    } else {
        printf("Non-positive");
    }
}

# Paths: 2 (one for x > 0, one for x <= 0).
```

- Loop Paths
    - Loops create multiple paths based on iterations.

```
int sum(int n) {
    int total = 0;
    for (int i = 0; i < n; i++) {
        total += i;
    }
    return total;
}

# Loop Paths:
n <= 0: Loop not entered.
n = 1: 1 iteration.
n > 1: Multiple iterations.
```

- Infeasible Paths
    - Paths that cannot be executed due to contradictory conditions.

```
void check(int x) {
    if (x > 10) {
        if (x < 5) {  // Infeasible (x cannot be >10 and <5)
            printf("Impossible");
        }
    }
}
```

- Loop Invariants
    - A property that holds before and after each loop iteration.

```
int sum(int arr[], int size) {
    int total = 0;
    int i = 0;
    while (i < size) {
        total += arr[i];  // Invariant: total = sum(arr[0..i-1])
```

```
        i++;
    }
    return total;
}
```

- Loop Unrolling
    - Replacing loops with repeated code for a fixed number of iterations.

```
# Original
for (int i = 0; i < 3; i++) {
    a += i;
}

# Unrolled:
a += 0;
a += 1;
a += 2;
```

- Symbolic Execution for Generate Test Case

```
#int complexFunction(int x) {
    int y = 0;
    for (int i = 0; i < x; i++) {
        if (i % 2 == 0) {
            y += i * 2;  // Even iteration
        } else {
            y -= i;      // Odd iteration
        }
    }
    return y;
}
```

- Step 1: Convert Code to Single Static Assignment (SSA) Form
    - SSA assigns each variable a unique version after each assignment, simplifying constraint tracking.

```
int complexFunction(int x) {
    y_1 = 0;
    i_1 = 0;
    while (i_1 < x) {
        if (i_1 % 2 == 0) {
            y_2 = y_1 + i_1 * 2;
        } else {
            y_2 = y_1 - i_1;
        }
        i_2 = i_1 + 1;
     }
    return y_1;
}
```

- Step 2: Loop Unrolling
    - Unroll them for a fixed number of iterations (e.g., 2 times).
    - Bounded loop notion

```
// Unrolled Code (for 2 iterations):
int complexFunction(int x) {
    y_1 = 0;
    i_1 = 0;

    // Iteration 1
    if (i_1 < x) {
        if (i_1 % 2 == 0) {
            y_2 = y_1 + i_1 * 2;
        } else {
            y_2 = y_1 - i_1;
        }
        i_2 = i_1 + 1;
        // Iteration 2
        if (i_2 < x) {
            if (i_2 % 2 == 0) {
                y_3 = y_2 + i_2 * 2;
            } else {
                y_3 = y_2 - i_2;
            }
            i_3 = i_2 + 1;
        }
    }
    return y_3;
}
```

- Step 3: Collect Path Constraints
    - For each unrolled iteration, track conditions and assignments as symbolic equations.
    - Then you solve equations and pick inputs from feasible solution regions for each program path.

```
Path 1: Both iterations take the even branch.
Constraints:
Iteration 1: i_1 < x and i_1 % 2 == 0.
Iteration 2: i_2 = i_1 + 1 < x and i_2 % 2 == 0.
Symbolic Variables:
i_1 = 0, i_2 = 1, x > 1.
Equations:
0 % 2 == 0 → true.
1 % 2 == 0 → false → Infeasible path (contradiction).

Path 2: Iteration 1 even, Iteration 2 odd.
Constraints:
Iteration 1: i_1 < x and i_1 % 2 == 0.
Iteration 2: i_2 = i_1 + 1 < x and i_2 % 2 ≠ 0.
Symbolic Variables:
i_1 = 0, i_2 = 1, x > 1.
Equations:
0 % 2 == 0 → true.
```

```
1 % 2 ≠ 0 → true.
Solution: x ≥ 2 (e.g., x=2, x=3).

Path 3: Loop runs once (early exit).
Constraints:
Iteration 1: i_1 < x and i_1 % 2 == 0.
Iteration 2: i_2 = i_1 + 1 ≥ x.
Symbolic Variables:
i_1 = 0, x = 1.
Solution: x=1.
```