Exercise 9

Dienstag, 1. Juli 2025 18:43

1)

Defects at Delivery	Impact on Product Support
Low (Well-tested software)	Fewer support tickets, less customer dissatisfaction, lower maintenance cost, better reputation.
Moderate	Noticeable increase in support queries and patch efforts, potential feature rollbacks.
High (Buggy release)	Overloaded support teams, emergency patches, hotfix cycles, negative reviews, loss of users/customers.

2)

Arguments for developers testing their own programmes:

1. **Deeper Understanding of the Code**
   Developers know the logic and architecture best, enabling precise and efficient test case creation.
2. **Faster Feedback Loop**
   Immediate testing while coding catches bugs early and reduces debugging time later.
3. **Cost Efficiency**
   Reduces the need for separate testers, especially in small teams or startups.
4. **Improved Code Quality**
   Writing tests (e.g., unit tests) encourages cleaner, modular, and more testable code.
5. **Better Ownership and Accountability**
   Developers become more responsible for the quality and reliability of their own code.

Arguments against developers testing their own programmes:

1. **Cognitive Bias / Lack of Objectivity**
   Developers may subconsciously avoid edge cases or assume their code is correct (confirmation bias).
2. **Tunnel Vision**
   Familiarity with the codebase can cause blind spots, missing errors that fresh eyes might catch.
3. **Insufficient Real-World Perspective**
   Developers might not test from the end-user's point of view, overlooking usability issues.
4. **Conflict of Interest**
   Testing your own code may reduce the incentive to rigorously test or admit flaws.
5. **Limited Time**
   Developers under deadline pressure may skip thorough testing in favor of "just making it run."

3) Equivalence classes (Folie 20)

- a concept from test theory
  - They are groups of input values that the system should treat the same
  - They help create meaningful test cases — so you don't need to test every possible input, just one **typical representative** from each group

Frequency:

Equivalence Class 1 (valid frequency):
- Input Values : 1-10
- Test Case: Store preference; user is notified accordingly

Equivalence Class 2 (invalid frequency):
- Input Values: 0, below 0, above 10
- Test Case: Throw IllegalArgumentException, preference is not stored

ChannelType:

Equivalence Class 3 (valid channel):
- Input Values: EMAIL, SMS
- Test Case: Store preference and use correct channel on update

Equivalence Class 4 (invalid channel):
- Input Values: null, unsupported value (if applicable)
- Test Case: Throw IllegalArgumentException

Website:

Equivalence Class 5 (valid website):
- Input Values: non-null object with valid URL (e.g., https://example.com)
- Test Case: User is registered or website is monitored correctly

Equivalence Class 6 (invalid website):
- Input Values: null, empty string, malformed URL
- Test Case: Throw IllegalArgumentException

Subscription ID:

Equivalence Class 7 (valid subscription id):
- Input Values: Any positive integer (e.g., 100, 1)
- Test Case: Subscription is accepted and saved

Equivalence Class 8 (invalid subscription id):
- Input Values: 0, negative integers (e.g., -5)
- Test Case: Throw IllegalArgumentException

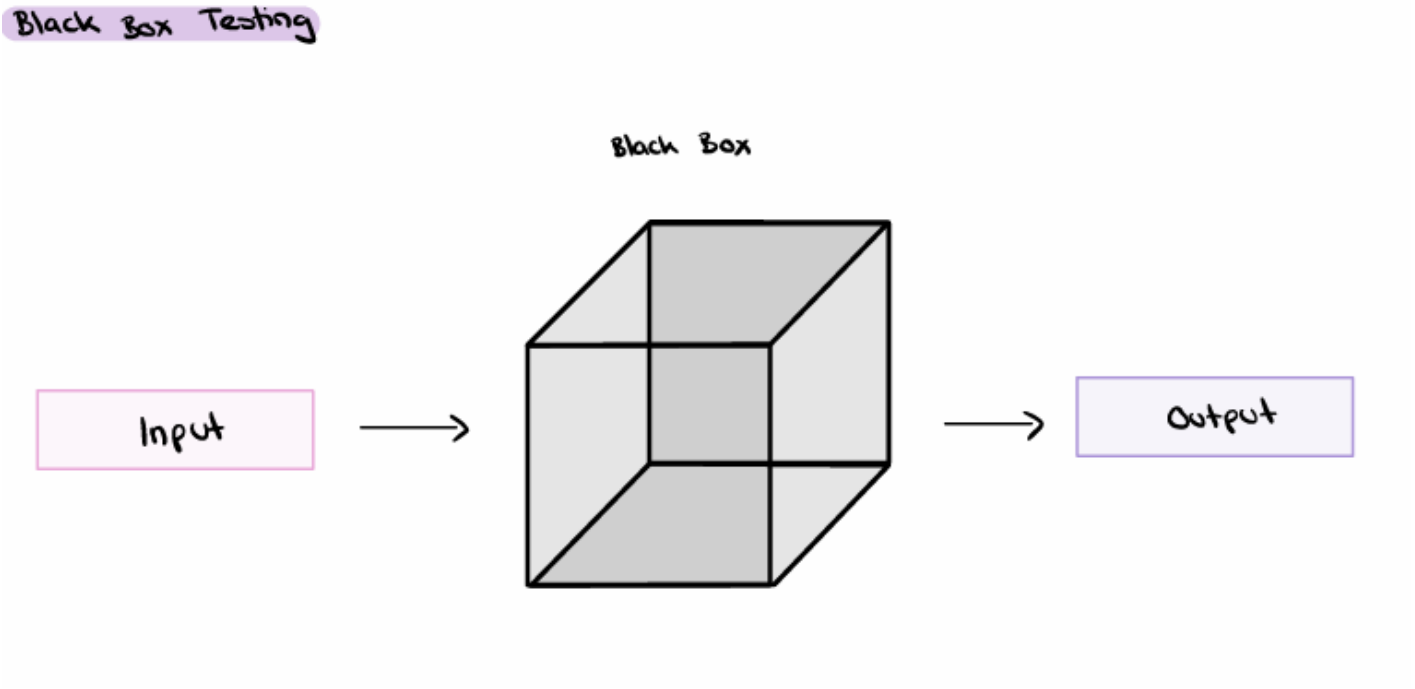
⇒ Unit tests in IntelliJ (WebsiteMonitor Project)
  - assertDoesNotThrow: ich erwarte einen gültigen Wert
  - assertThrows: ich erwarte einen Fehler

4) Regression testing (Folie 16)
   - Re-run existing test suites to ensure new changes don't break existing features
   - Goals:
     - Confirm that new features haven't affected existing features.
     - Detect unexpected side effects from code changes.
     - Maintain system stability over time.
   - When is it performed:
     - After bug fixes
     - After new features are added
     - After code refactoring or performance improvements
     - Before a release cycle

- How it's done:
  - Manual Testing: Running key test scenarios again (not efficient for large systems).
  - Automated Testing: Using tools (e.g., JUnit, Selenium, TestNG) to re-run test suites quickly and reliably.

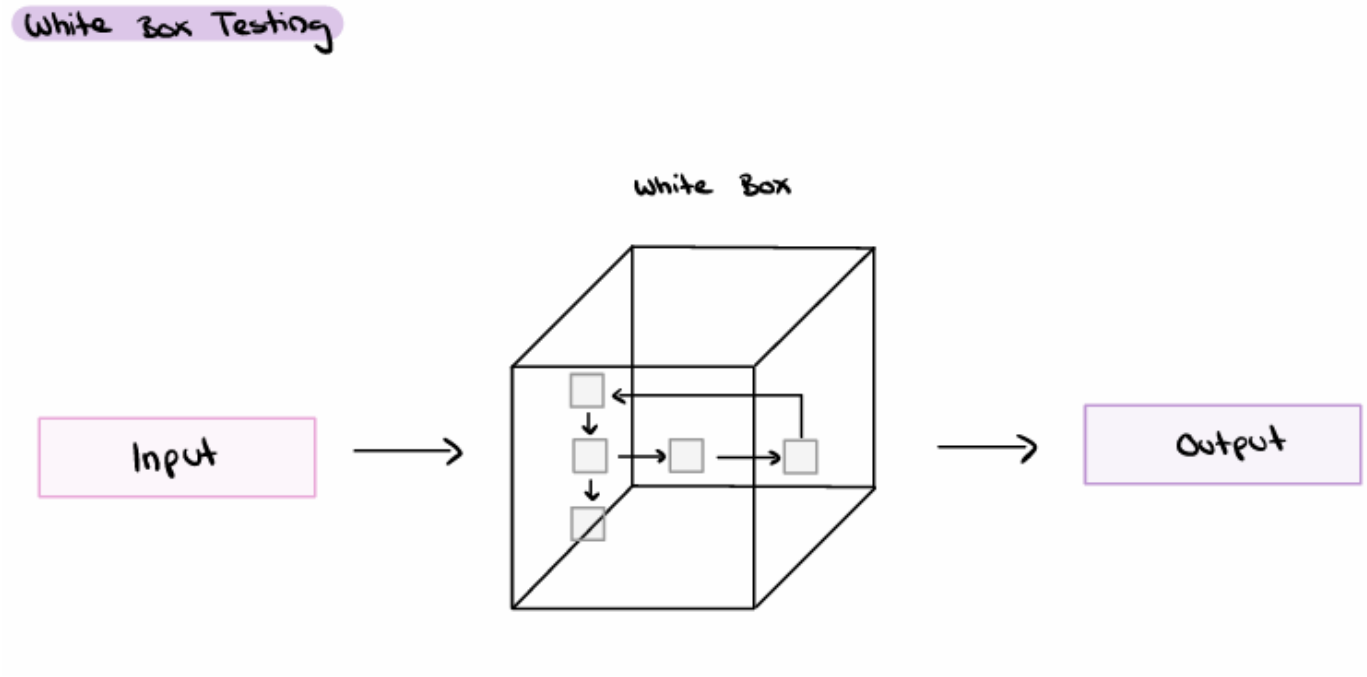
5) Black Box vs. White Box Testing:

Black Box Testing:



- Implementation knowledge is not required
- Also known as closed box testing, data driven testing and functional testing
- Performed by end users and also by software testers
- Programming knowledge is not required
- Types: functional testing, non-functional testing, regression testing

White Box Testing:



- Implementation knowledge is required
- Also known as clear box testing, structural testing or code based testing
- Normally done by software developers
- Programming knowledge is required
- Types: path testing, loop testing, condition testing

⇒ Black Box Testing focuses on what the software does, without looking at the code. White Box Testing looks inside the code to check how it works