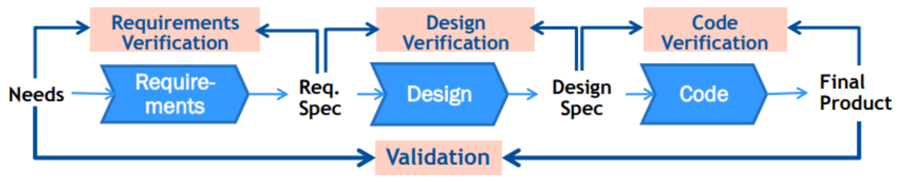
1 Part I – Software Reviews & Inspections

1.1 Verification versus Validation

- **Verification – are we building the product right**
  - Ensure (mainly through reviews) that intermediate work products and the final product are "well built", i.e., conform to their specifications.
- **Validation – are we building the right product?**
  - Ensure (manly through tests) that the final product will fulfill its intended use in its intended environment.
  - Can also be applied to intermediate work products, as predictors of how well the final product will satisfy user needs.

Verification shows conformance with specification. Validation shows that the program meets the customer's needs.



1.2 Técnicas V&V Estáticas e Dinâmicas

- **Static Techniques** — involve analyzing the static system representations to find problems and evaluate quality.
  - Reviews and inspections.
  - Automated static analysis (e.g., with lint).
  - Formal verification (e.g., with Dafny)
- **Dynamic Techniques** – involve executing the system and observing its behavior.
  - Software testing.
  - Simulation.

Static verification techniques involve examination and analysis of the program for error detection. Dynamic techniques involve executing the program for error detection.

They are complementary and not opposing techniques. Both should be used during the V&V process.

1.3 Software reviews and inspections

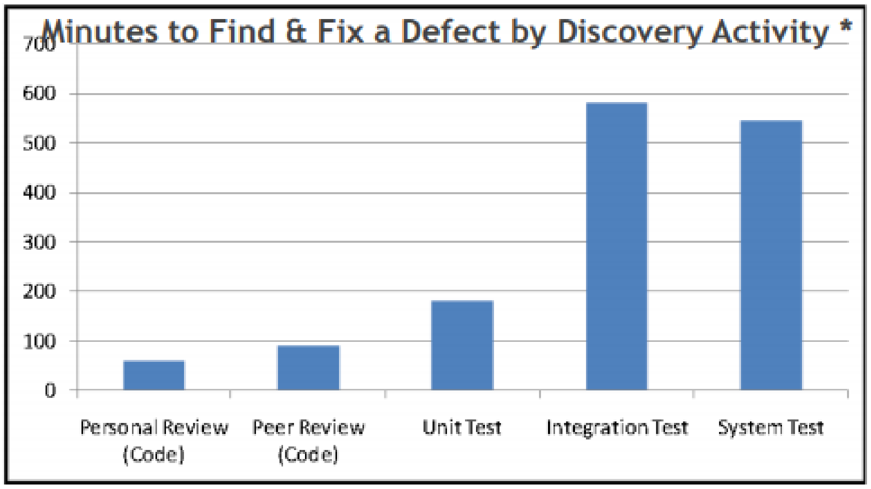
Analysis of static system representations to find problems.

- Manual analysis of requirements specs, design specs, code, etc.
- May be supplemented by tool-based static analysis.

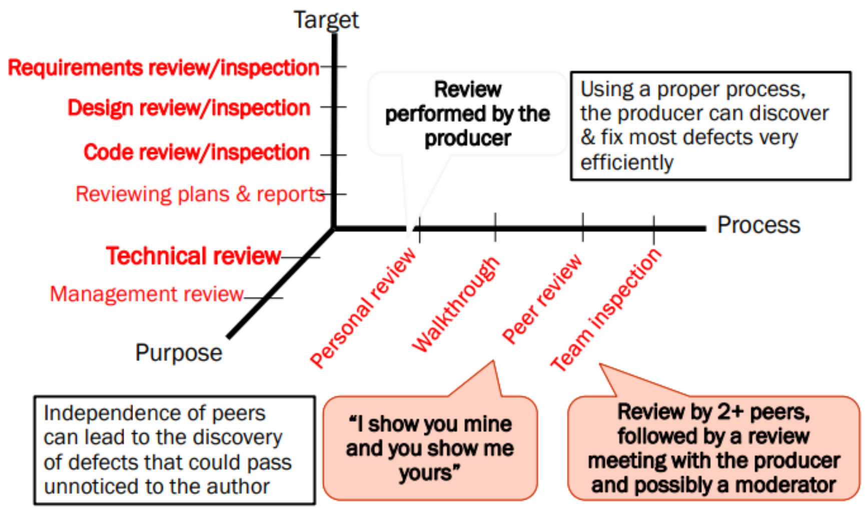
Advantages (as compared to testing):

- Can be applied to any artefact, and not only code
- Can be applied earlier (thus reducing impact and cost of errors)
- Fault localization (debugging) is immediate
- Allows evaluating internal quality attributes (e.g., maintainability)
- Usually more efficient and effective than testing in finding security vulnerabilities and checking exception handling
- Very effective in finding multiple defects
- Peer reviews promote knowledge sharing

1.4 Efficiency of defect removal methods

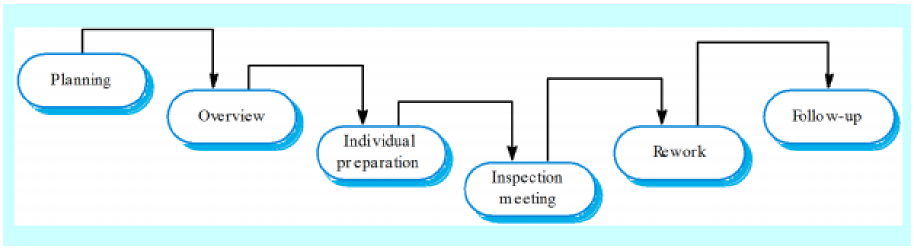


1.5 Types of Reviews


Target
Requirements review/inspection
Design review/inspection
Code review/inspection
Reviewing plans & reports
Technical review
Management review
Purpose
Process
Personal review
Walkthrough
Peer review
Team inspection
Review performed by the producer
Using a proper process, the producer can discover & fix most defects very efficiently
Independence of peers can lead to the discovery of defects that could pass unnoticed to the author
"I show you mine and you show me yours"
Review by 2+ peers, followed by a review meeting with the producer and possibly a moderator

1.6 Review Best Practices

- **Use a checklist derived from historical defect data:**
  - Makes the review more effective and efficient, by focusing the attention on the most frequent and important problems.
  - CPersonal checklists make sense, because each person tends to repeat his/her own mistakes.
- **Take enough review time :** 200 LOC/hour is a recommended review rate by some authors(LOC-Lines of Code).
- **Take a break between developing and reviewing (in personal reviews).**
- **Combine personal reviews with peer reviews or team inspections:** Team inspections comprise individual reviews performed by 2+ peers (Individual preparation), followed by a meeting (Inspection meeting) with the producer and possibly a moderator.



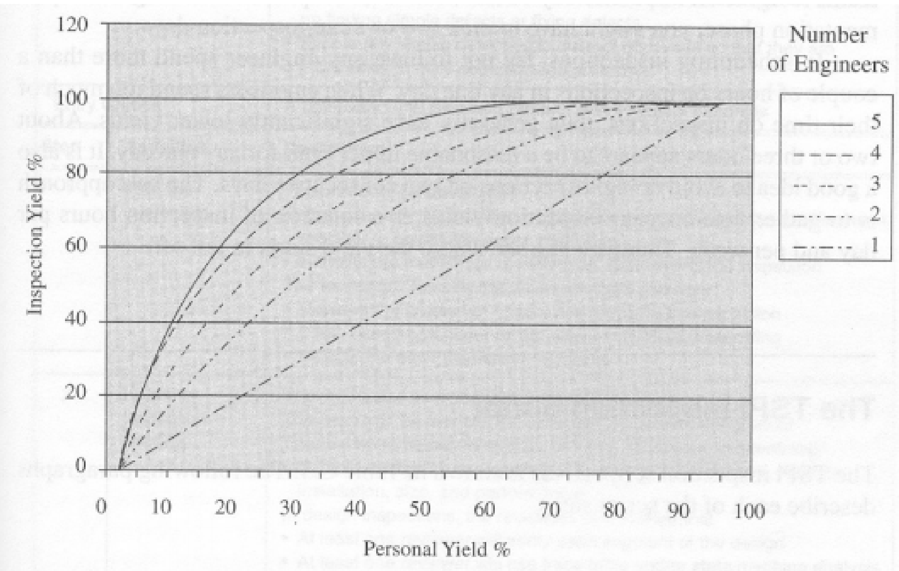
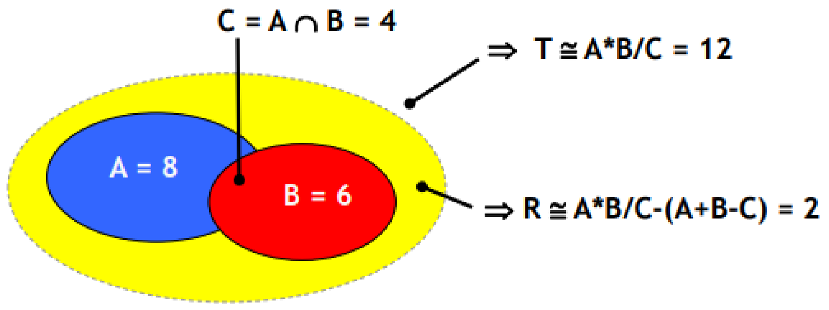


FIGURE C.1 INSPECTION YIELD VERSUS PERSONAL YIELDS FOR VARIOUS NUMBERS OF ENGINEERS

- **Measure the review process & use data to improve:** size, time spent, defects found, defects escaped (found later).

1.7 Estimate Missed defects

The capture-recapture method is used to estimate the total defects (T) and number of defects remaining (R) based on the degree of overlapping between defects detected by different inspectors (A, B).



In case of more than 2 inspectors, **A** refers to the inspector that found more unique defects, and **B** refers to the union of all other inspectors

2 Part II – Software Testing

2.1 Test Concepts

2.1.1 Testing goals and limitations

Goals:

- exercise the software with defined test cases and observe its behaviour to discover defects.
- increase the confidence on the software correctness and to evaluate product quality.

Limitations:

- Testing can show the presence of bugs, not their absence

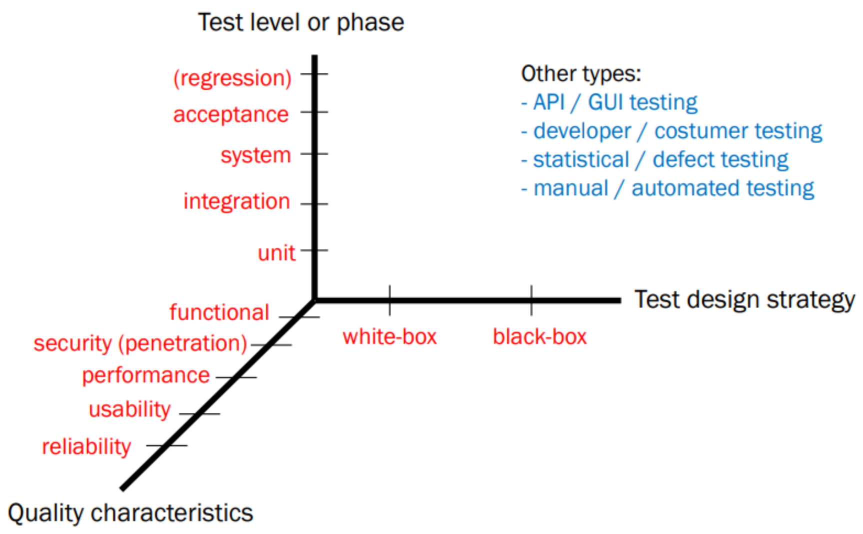
2.1.2 Test Cases

- **Test Case:** A set of test inputs, execution conditions, and expected results developed to exercise a particular program path.
- **Test Script:** concrete definition of test steps / procedure(can be parameterized for reuse with multiple test data).

2.1.3 Test Activities

- **Test Planning:** define the objectives of testing and the approach for meeting test objectives within constraints imposed by the context.
- **Test monitoring and control:** compare actual progress against the plan, and take actions necessary to meet the objectives of the test plan.
- **Test analysis:** identify testable features and test conditions.
- **Test design:** derive test cases.
- **Test implementation:** create automated scripts.
- **Test execution:** run test suites.
- **Test completion:** collect data from completed test activities.

2.1.4 Test Types



2.2 Test Levels

2.2.1 Unit Testing/Component Testing/Module Testing

- Testing of individual hardware or software units or groups of related units.
- Detect functional (e.g., wrong calculations) and non-functional (e.g., memory leaks) defects in the unit.

- Usually API testing.
- Responsibility of the developer.
- Usually based on experience, specs and code.

2.2.2 Integration Testing

- Software and/or hardware components are combined and tested to evaluate the interaction between them.
- Two levels of integration testing:
  - **Component integration testing:** interactions between components.
  - **System integration testing:** interactions between systems.
- Responsibility of an independent test team.
- Usually based on a system spec (technical/design spec).
- Detect defects that occur on the units' interfaces.
- For easier fault localisation, integrate incrementally/continuously.

2.2.3 System Testing

- Conducted on a complete, integrated system to evaluate the system's compliance with specified requirements.
- Both functional behavior and quality requirements (performance, usability, reliability, security, etc.) are evaluated.
- Usually GUI testing.
- Responsibility of an independent test team.
- Usually based on requirements document.

2.2.4 Acceptance Testing

- Determine whether or not a system atisfies its acceptance criteria.
- Enable a customer,user, or other authorized entity to determine whether or not to accept the system.
- Usually the responsibility of the customer.
- Based on a requirements document or contract.
- Check if customer requirements and expectations are met.

2.2.5 Regression Testing

- Tests to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.
- Changes to software, to enhance it or fix bugs, are a very common source of defects.
- Not really a new test level, but just the repetition of testing at any level.

2.3 Test case design techniques

2.3.1 Design goals:

- Create a set of test cases (test suite) that are effective in validation and defect testing.
- A good test suite should have a small/manageable size and have a high probability of finding most of the defects.

2.3.2 Design Strategies:

Black-Box Testing: Derivation of test cases based on some external specification.

- **Equivalence class partitioning:** partition the input domain into classes of equivalent behavior, separating classes of valid and invalid inputs, and select at least one test case from each class.
- **Boundary value analysis:** select test values at the boundaries of each partition (e.g., immediately below and above), besides typical values.
- **Decision table testing:** test all possible combinations of a set of conditions and actions (each combination corresponding to a business rule).
- **State transition testing:** derive test cases from a state-machine model of the system.
- **Use case testing:** derive test cases from a use case model of the system (with use cases possibly detailed with scenarios, pre/post-conditions, etc).

White-box Testing: Derivation of test cases according to program structure.

- **Using coverage analysis tools:** (e.g., Eclemma) to analyse code coverage achieved by black-box tests and design additional tests as needed.
- **Testing statement coverage:** Assure that all statements are exercised.
- **Decision/Branch coverage:** Assure that all decisions (if, while, for, etc.) take both values true and false.

2.4 Test automation tools

- **Unit testing frameworks:** JUnit, NUnit.
- **Mock object frameworks:**
  - Facilitate simulating external components in unit testing.
  - EasyMock, jMock.
- **Test coverage analysis tools:**
  - Measure degree of code coverage achieved by the execution of a test suite.
  - Useful for white-box testing.
  - Eclemma, Clover.
- **Mutation testing tools:**
  - Evaluate the quality of a test suite by determining its ability to 'kill' mutants (with common fault types) of the program under test.
  - pitest, muJava.
- **Acceptance testing frameworks:**
  - Allows creating test cases by people without technical knowledge.
  - Cucumber, JBehave, Fitnesse.
- **Capture/replay tools (aka functional testing tools):**
  - Capture user interactions in scripts that can be edited and replayed.
  - Useful for GUI testing, particularly regression testing.
  - Selenium, IBM Rational Functional Tester.
- **Performance/load testing tools:**
  - Execute test suites simulating many users and measure system performance.
  - IBM Rational Performance Tester, Compuware QA Load.
- **Penetration testing tools:** Metasploit, ZAP.
- **Test case generation tools:**
  - Automatically generate test cases from models or code.
  - IParTeG (UML), EvoSuite (Java), Spec Explorer (Spec#), Conformiq (UML).

2.5 Test management

Tools:

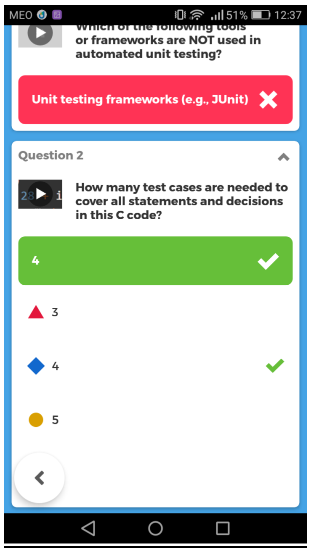
- Manage test information and status.
- Integrate with other management tools: requirements management, project management, bug tracking, configuration management.
- Integrate with test automation tools.
- TestLink.

Test Management charts are used to track progress of testing and bug fixing activities.

2.6 Testing best practices

- **Test as early as possible** The cost of finding and fixing bugs increases exponentially with time.
- **Automate the tests**.
- **Test first (write tests before the code):** Helps clarifying requirements and specifications since test cases are partial specifications of system behavior.
- **Black-box first:** Start by designing test cases based on the specification and then add any tests needed to ensure code coverage.

2.7 Useful Kahoots



Which of the following tools or frameworks are NOT used in automated unit testing?

Unit testing frameworks (e.g., JUnit) ✗

Question 2

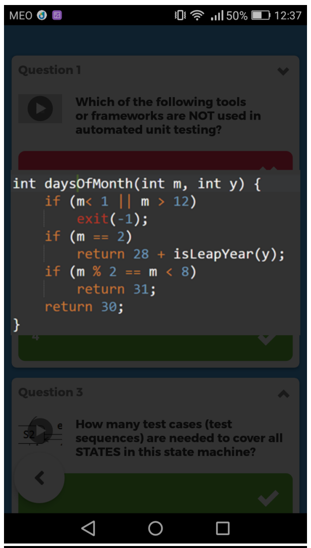
How many test cases are needed to cover all statements and decisions in this C code?

4 ✓

3

4 ✓

5



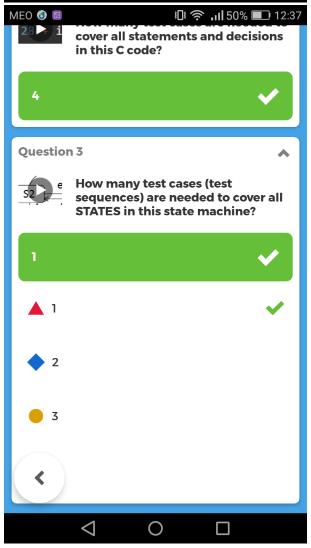
Question 1

Which of the following tools or frameworks are NOT used in automated unit testing?

```
int daysOfMonth(int m, int y) {
    if (m< 1 || m > 12)
        exit(-1);
    if (m == 2)
        return 28 + isLeapYear(y);
    if (m % 2 == m < 8)
        return 31;
    return 30;
}
```

Question 3

How many test cases (test sequences) are needed to cover all STATES in this state machine?



cover all statements and decisions in this C code?

4 ✓

Question 3

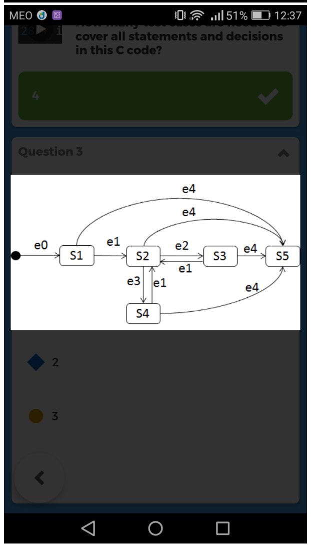
How many test cases (test sequences) are needed to cover all STATES in this state machine?

1 ✓

1 ✓

2

3



cover all statements and decisions in this C code?

4

Question 3

2

3