TVVS - Test, Verification and Validation of Software

Test management and documentation

Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

Test management

Test management

- The process of planning, scheduling, estimating, monitoring, reporting, controlling, and completing test activities.

Test management tool

- A tool that supports test management.

[ISTQB]

Test planning activities

- Determining the **scope and risks** and identifying the **objectives** of testing
- Defining the **overall approach** of testing, including the definition of the test levels and **entry and exit criteria**
- **Integrating and coordinating** the testing activities into the **software life cycle** activities
- **Making decisions** about what to test, what roles will perform the test activities, how the test activities should be done, and how the test results will be evaluated
- **Scheduling** test analysis and design activities

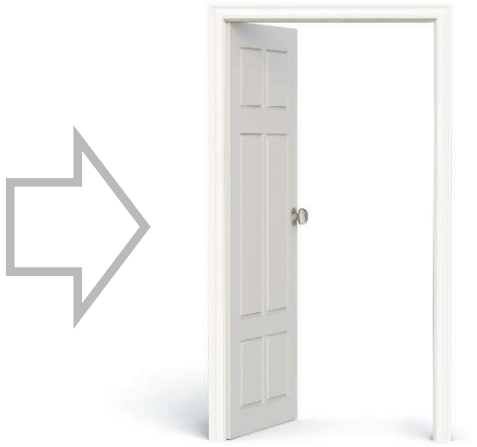
Test planning activities

- **Scheduling** test implementation, execution and evaluation
- **Assigning resources** for the different activities defined
- **Defining** the amount, level of detail, structure and templates for the **test documentation**
- **Selecting metrics** for monitoring and controlling test preparation and execution, defect resolution and risk issues
- **Selecting the level of detail** for test procedures to provide enough information to **support reproducible test preparation and execution**

Entry criteria

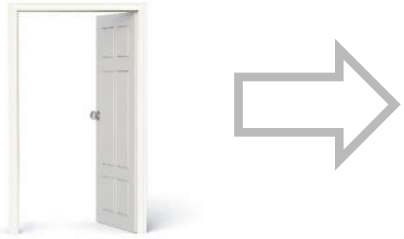
The preconditions that must be fulfilled before testing can start.

- Typically, entry criteria may cover the following:
  - Test environment availability and readiness
  - Test tool readiness in the test environment
  - Testable code availability
  - Test data availability



Exit criteria

- The conditions that must be fulfilled before testing can be concluded. Exit criteria specify when to stop testing activities and indicate whether the software product has achieved the required quality standards or goals for a particular phase of testing or the overall project.
  - Thoroughness measures, such as **coverage of code**, functionality or risk
  - **Estimates of defect density** or reliability measures
  - **Cost**
  - **Residual risks**, such as defects not fixed or lack of test coverage in areas
  - **Schedules** such as those based on time to market



Test estimation

Test estimation

- An approximation related to various aspects of testing.

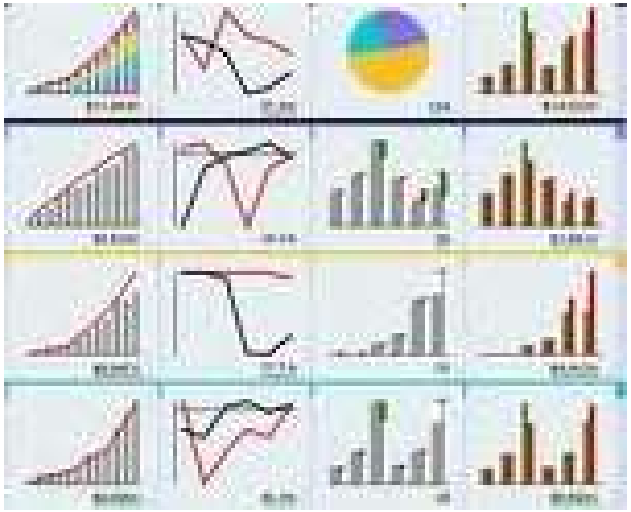
[ISTQB]

- How do you estimate the test effort

?

Test estimation

- **The metrics-based approach**: estimating the testing effort based on metrics of former or similar projects or based on typical values
  - E.g., complexity; functional point analysis, test point analysis, …
- **The expert-based approach**: estimating the tasks based on estimates made by the owner of the tasks or by experts



Test progress monitoring

- Percentage of work done in test case preparation
- Percentage of work done in test environment preparation
- Test case execution
- Defect information
- Test coverage of requirements, risks or code
- Subject confidence of testers in the product
- Dates of test milestones
- Testing costs, including the cost compared to the benefit of finding the next defect or to run the next test

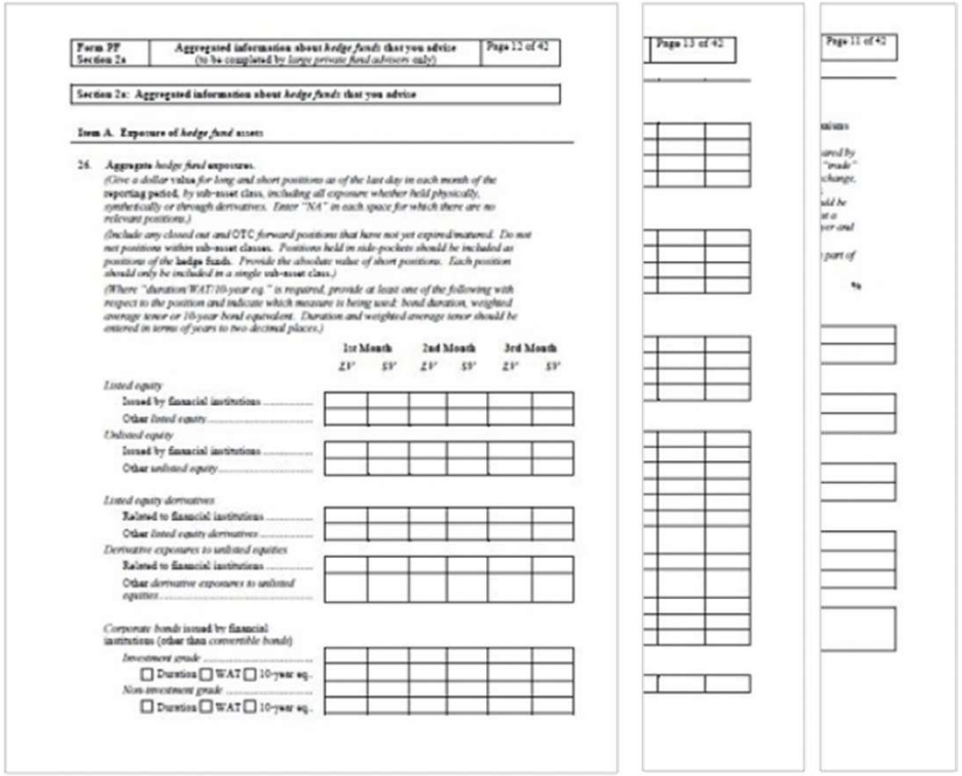
Test control

- Making decisions based on information from test monitoring
- Re-prioritizing tests when an identified risk occurs
- Changing the test schedule due to availability or unavailability of a test environment
- Setting an entry criterion requiring fixes to have been re-tested by a developed before accepting them into a build

Test reporting

- Collecting and analyzing data from testing activities and subsequently consolidating the data in a report to inform stakeholders.

[ISTQB]



Form PF — Aggregated information about *hedge funds* that you advice — Page 12 of 42
Section 2c — (to be completed by *large private fund advisers* only)

Section 2c: Aggregated information about *hedge funds* that you advice

Item A. Exposure of *hedge fund* assets

26. Aggregate *hedge fund* exposures.

(Give a dollar value for long and short positions as of the last day in each month of the reporting period, by sub-asset class, including all exposure whether held physically, synthetically or through derivatives. Enter "NA" in each space for which there are no relevant positions.)

(Include any closed out and OTC forward positions that have not yet expired/matured. Do not net positions within sub-asset classes. Positions held in side-pockets should be included as positions of the hedge funds. Provide the absolute value of short positions. Each position should only be included in a single sub-asset class.)

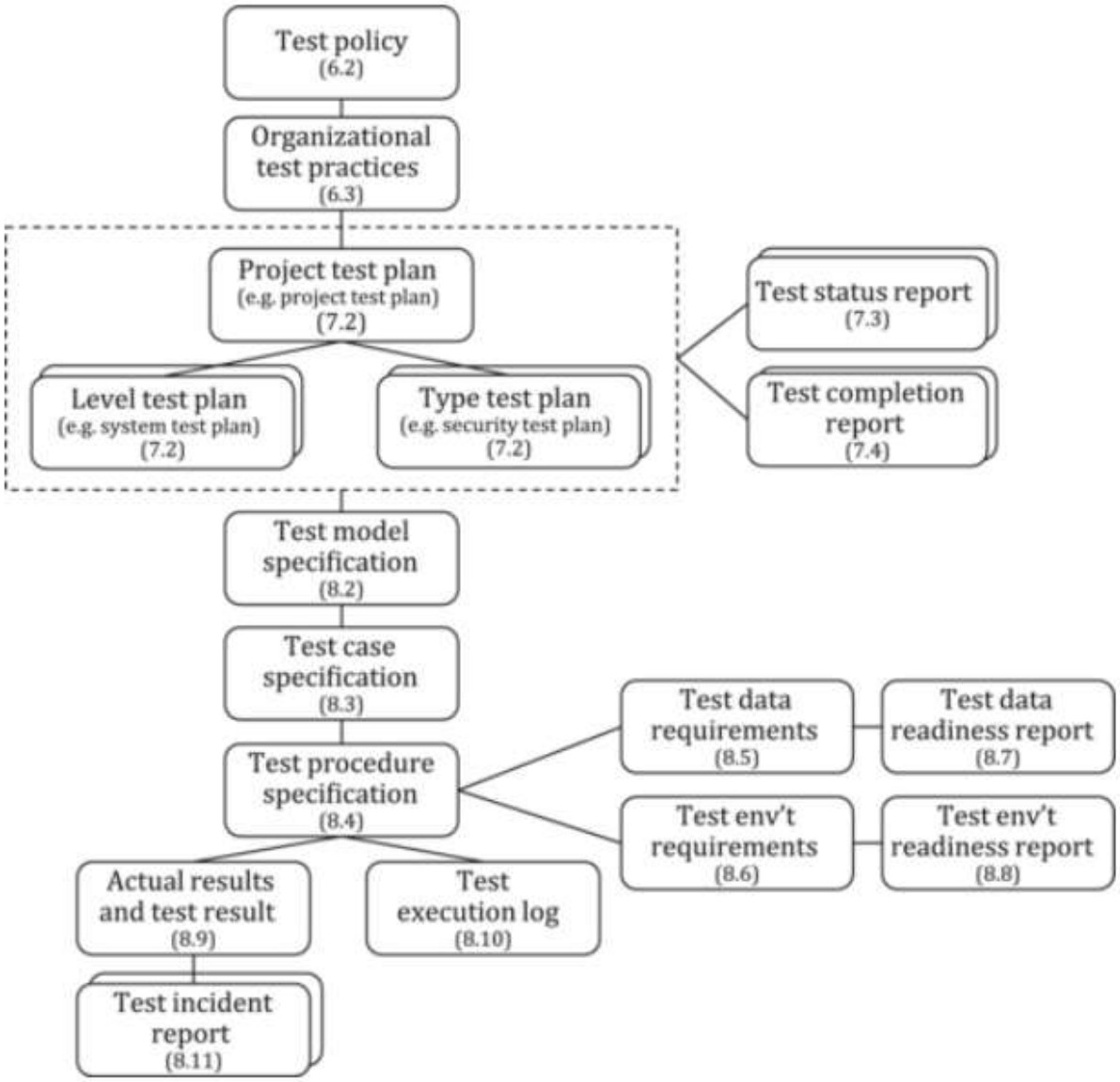
(Where "duration/WAT/10-year eq." is required, provide at least one of the following with respect to the position and indicate which measure is being used: bond duration, weighted average tenor or 10-year bond equivalent. Duration and weighted average tenor should be entered in terms of years to two decimal places.)

	1st Month		2nd Month		3rd Month	
	LV	SV	LV	SV	LV	SV
Listed equity						
Issued by financial institutions						
Other *listed equity*						
Unlisted equity						
Issued by financial institutions						
Other *unlisted equity*						
Listed equity derivatives						
Related to financial institutions						
Other *listed equity derivatives*						
Derivative exposures to unlisted equities						
Related to financial institutions						
Other *derivative exposures to unlisted equities*						
Corporate bonds issued by financial institutions (other than *convertible bonds*)						
Investment grade						
☐ Duration ☐ WAT ☐ 10-year eq.						
Non-investment grade						
☐ Duration ☐ WAT ☐ 10-year eq.						

Page 13 of 42

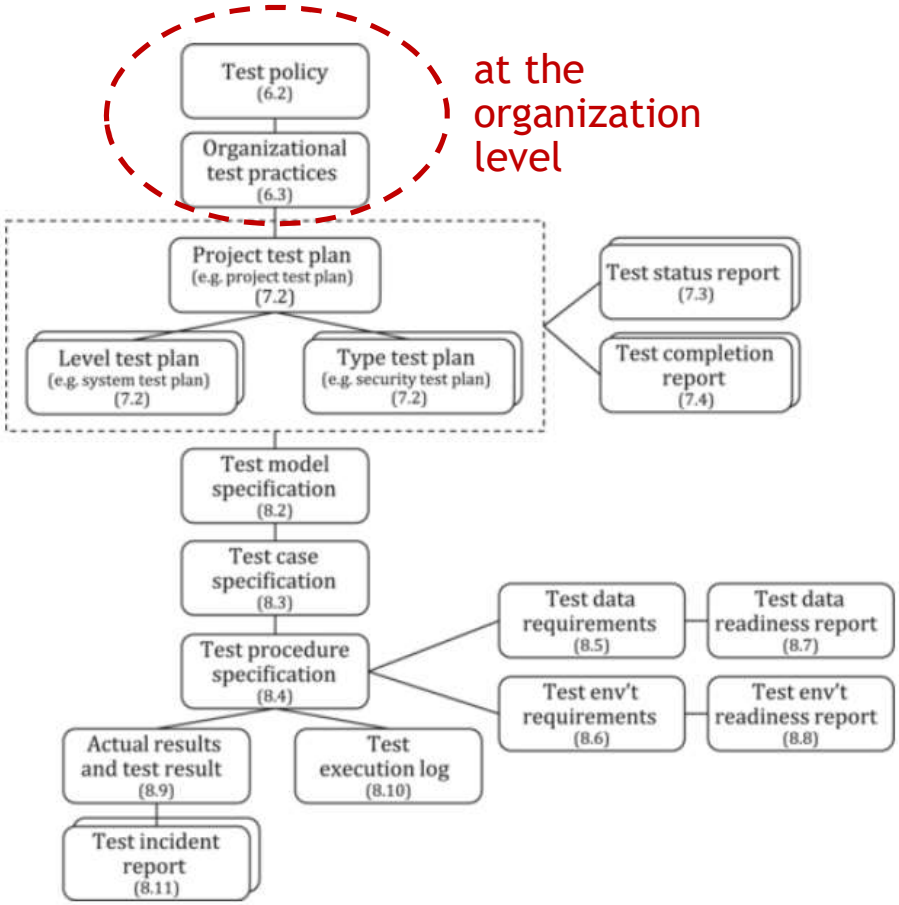
Page 11 of 42

Test Documentation
ISO/IEC/IEEE 29119-3:2021



Test Documentation
ISO/IEC/IEEE 29119-3:2021

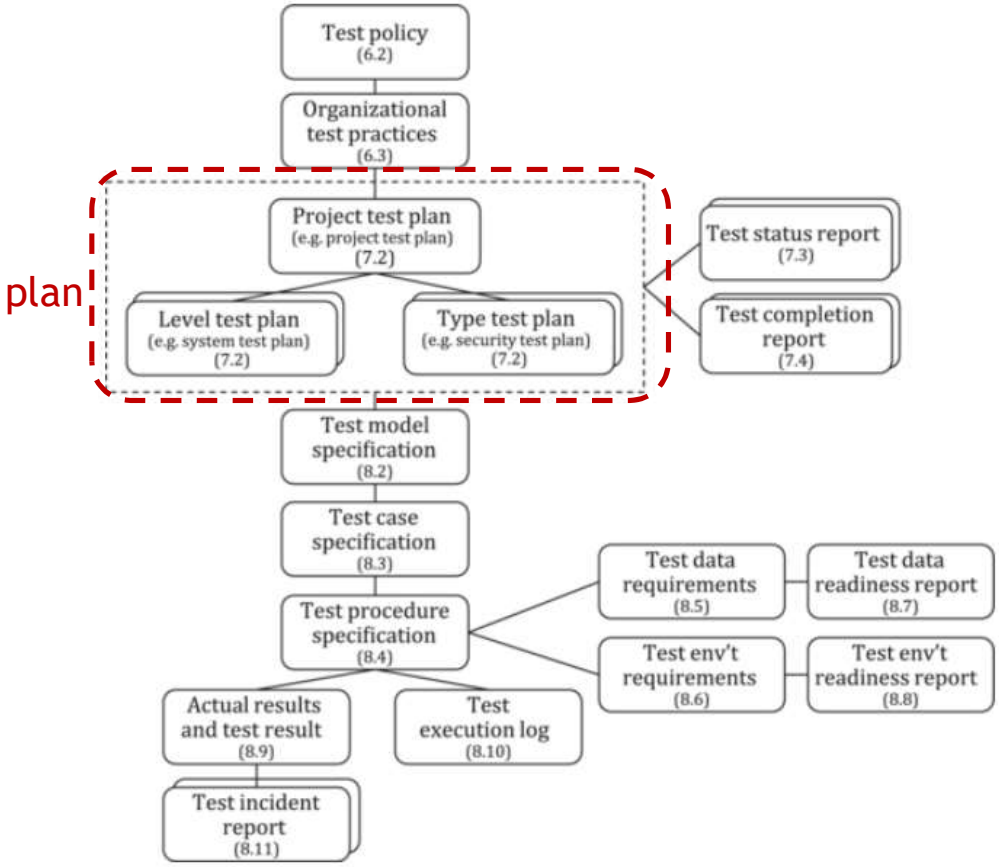
- test Policy
  - A high-level document describing the principles, approach and major objectives of the organization regarding testing
- organizational test practices
  - documentation that expresses the recommended approaches or methods for the testing to be performed within an organization, providing detail on how the testing is to be performed



Test Documentation
ISO/IEC/IEEE 29119-3:2021

- test plan
  - detailed description of test objectives to be achieved and the means and schedule for achieving them, organized to coordinate testing activities for some test item or set of test items

    [Note: A project can have more than one test plan for example, there can be a project test plan (also known as a master test plan) that encompasses all testing activities on the project; further detail of particular test activities can be defined in one or more **test level / test type plans** (e.g., a system test plan or a performance test plan).],



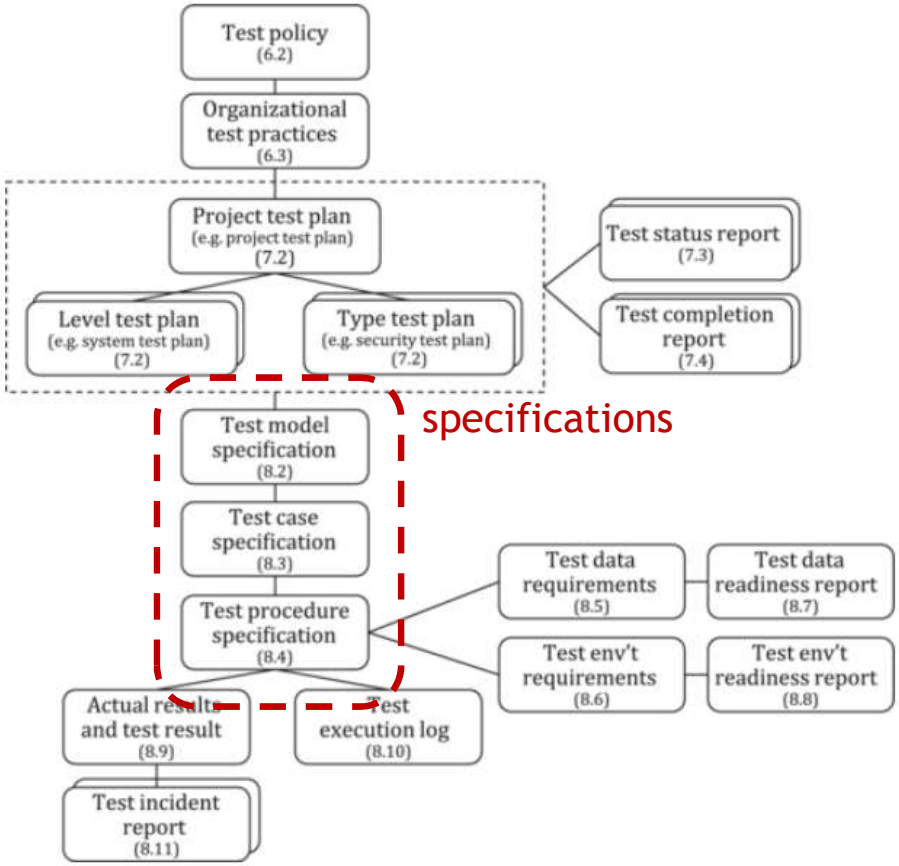
Test Documentation
ISO/IEC/IEEE 29119-3:2021

- test model specification
  - documentation specifying the test model

    [
    test model - representation of the test item, which allows the testing to be focused on particular characteristics or qualities.

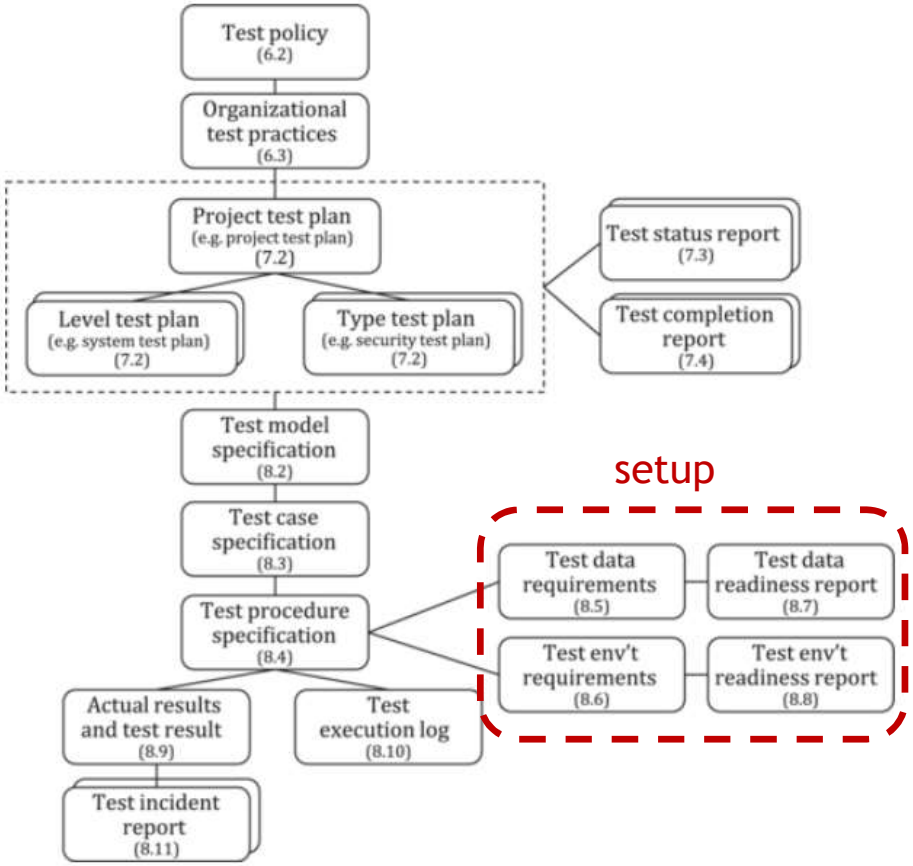
    EXAMPLE Requirements statements, equivalence partitions, state transition diagram, use case description, decision table, input syntax, source code, control flow graph, parameters and values, classification tree, natural language.

    ]
- test case specification
  - documentation of a set of one or more test cases
- test procedure specification or test script
  - documentation specifying one or more test procedures



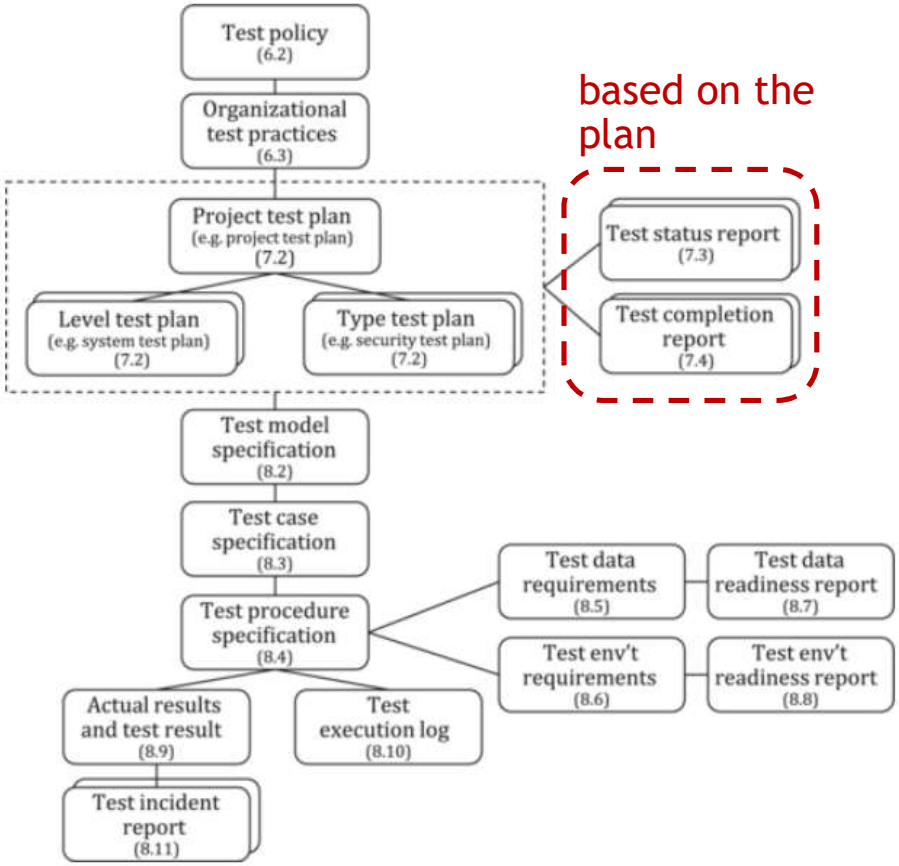
Test Documentation
ISO/IEC/IEEE 29119-3:2021

- test data requirements
  - specifications and criteria necessary for the selection, creation, and preparation of test data used in software testing processes
- test environment requirements
  - documentation of the necessary properties of the test environment
- test data readiness report
  - documentation describing the status of each test data requirement
- test environment readiness report
  - documentation that describes the status of each test environment requirement



Test Documentation
ISO/IEC/IEEE 29119-3:2021

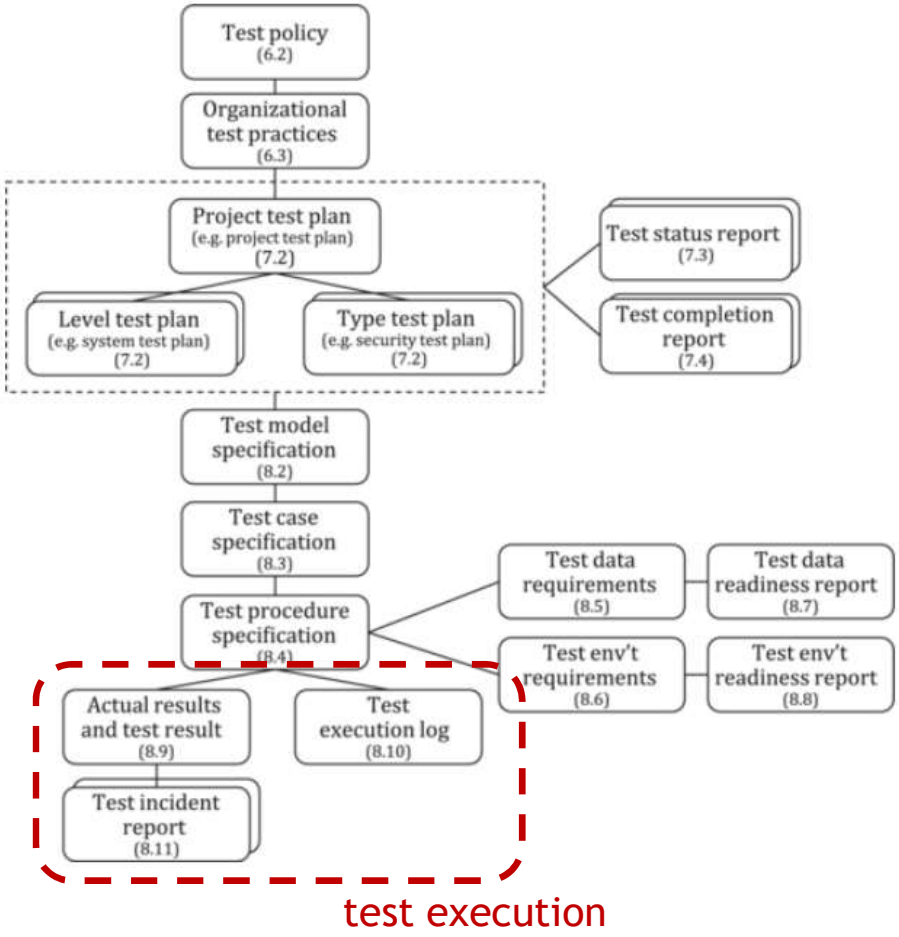
- test status report
  - report that provides information about the status of the testing that is being performed in a specified reporting period
- test completion report or test summary report
  - report that provides a summary of the testing that was performed



Test Documentation
ISO/IEC/IEEE 29119-3:2021

- test execution log
  - record of the execution of one or more test procedures
- actual results & test result
  - actual result: set of behaviours or conditions of a test item, or set of conditions of associated data or the test environment, observed as a result of test execution
  - test result: indication of whether or not a specific test case has passed or failed, i.e. if the actual results corresponds to the expected results or if deviations were observed
- incident report
  - documentation of the occurrence, nature, and status of an incident

    [test incident - event occurring during the execution of a test that requires investigation]

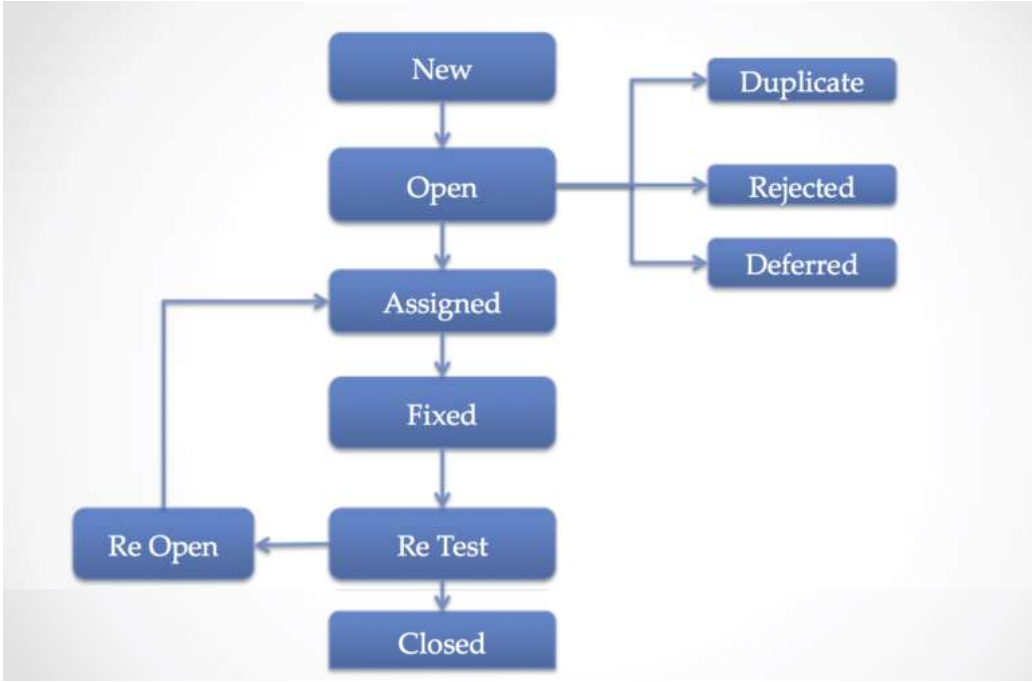


Incident

- anomalous or unexpected event, set of events, condition, or situation at any time during the life cycle of a project, product, service, or system

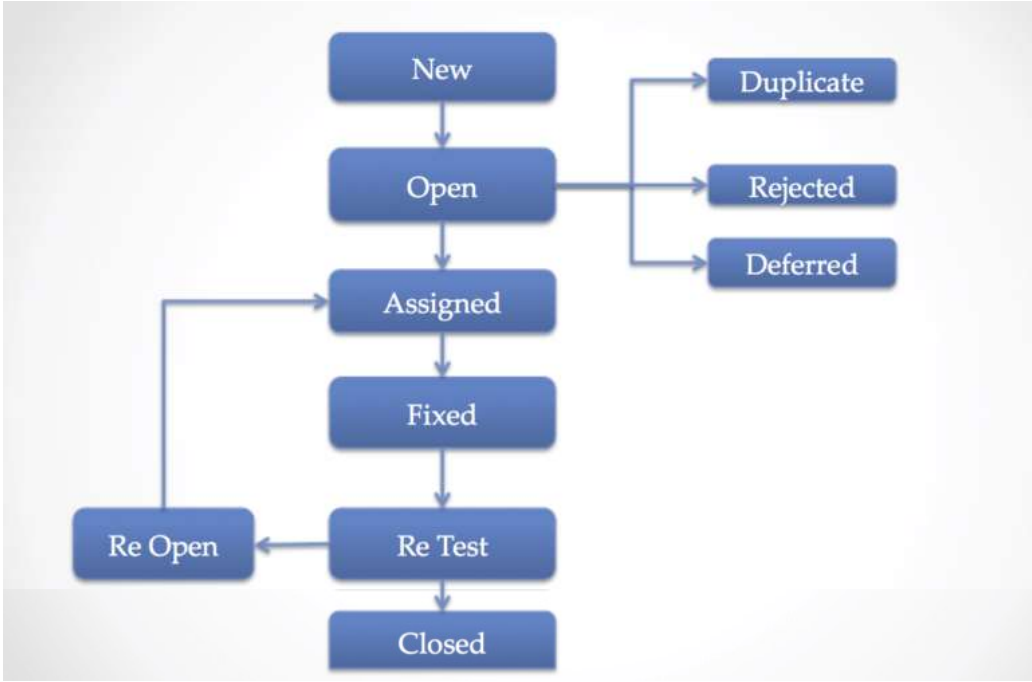
Incident lifecycle

- The common states of an incident in the defect lifecycle include:
  - **New/Open**: When first discovered and reported, an incident enters this state. At this stage, it has not been analyzed or assigned for further action.
  - **Assigned**: Once the incident is reviewed, analyzed, and categorized, it is assigned to an appropriate individual or team for further investigation or resolution.
  - **Fixed**: When the incident's root cause has been identified and corrected, the incident is marked as "Fixed." This means that the issue has been resolved by implementing necessary changes in the sw.
  - **Retest**: After the incident has been fixed, it needs to be retested to ensure that the resolution has successfully addressed the problem. The incident is marked for retesting to verify if the fix is effective.



Incident lifecycle

- The common states of an incident in the defect lifecycle include:
  - **Reopen**: If the issue reoccurs or if the reported problem persists after being marked as fixed, it is reopened for further investigation.
  - **Closed**: When the reported incident has been successfully resolved, retested, and verified, it is closed. No further action is required.
  - **Deferred**: the reported incident or defect is acknowledged and valid, but the decision is made to postpone its resolution to a future release or iteration
  - **Duplicate**: the reported incident is found to be a duplicate of another already reported and existing issue in the defect tracking system.
  - **Rejected:** the reported defect or issue is reviewed and deemed invalid or not actionable for various reasons. When an issue is marked as "Rejected," it does not meet further investigation or resolution criteria.



Some test related roles

Test manager

- The person responsible for project management of testing activities, resources, and evaluation of a test object.

Test leader

- On large projects, the person who reports to the test manager and is responsible for project management of a particular test level or a particular set of testing activities.

Test director

- A senior manager who manages test managers.

Tester

- A person who performs testing.

[ISTQB]

Tasks of the Test Leader

- Coordinate the test strategy and plan with project managers and others
- Write or review a test strategy for the project, and test policy for the organization
- Contribute the testing perspective to other project activities, such as integration planning
- Plan the tests – considering the context and understanding the test objectives and risks
- Initiate the specification, preparation, implementation and execution of tests, monitor the test results and check the exist criteria
- Adapt planning based on test results and progress and take any action necessary to compensate for problems

Tasks of the Test Leader

- Set up adequate configuration management of testware for traceability
- Introduce suitable metrics for measuring test progress and evaluating the quality of the testing and the product
- Decide what should be automated, to what degree, and how
- Select tools to support testing and organize any training in tool use for testers
- Decide about the implementation of the test environment
- Write test summary reports based in the information gathered during testing

Tasks of the Tester

- Review and contribute to test plans
- Analyse, review and assess user requirements, specifications and models for testability
- Create test specification
- Set up the test environment
- Prepare and acquire test data

Tasks of the Tester

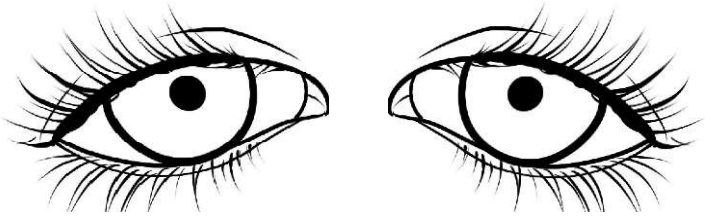
- Implement tests on all test levels, execute and log the tests, evaluate the results and document the deviations from expected results
- Use test administration or management tools and test monitoring tools as required
- Automate tests
- Measure performance of components and systems
- Review tests developed by others

Test independence

- No independent testers – developers test their own code
- Independent testers within the development teams
- Independent test team or group within the organization, reporting to project management or executive management
- Independent testers from business organization or user community
- Independent test specialists for specific test types such as usability
- Independent testers outsourced or external to the organization

Benefits of independence

- Independent testers see other and different defects and are unbiased
- Independent testers can verify assumptions people made during specification and implementation of the system



Drawbacks of test independence

- Isolation from the development team (if treated as totally independent)
- Developers may lose a sense of responsibility for quality
- Independent testers may see as a bottleneck or blamed for delays in release

Test smells

- Test smells are defined as bad programming practices in unit test code (such as how test cases are organized, implemented and interact with each other) that indicate potential design problems in the test source code.
- [https://testsmells.org/]

Assertion Roulette

```
@MediumTest
public void testCloneNonBareRepoFromLocalTestServer() throws Exception {
    Clone cloneOp = new Clone(false, integrationGitServerURIFor("small-repo.early.git"), helper().newFolder());

    Repository repo = executeAndWaitFor(cloneOp);

    assertThat(repo, hasGitObject("ba1f63e4430bff267d112b1e8afc1d6294db0ccc"));

    File readmeFile = new File(repo.getWorkTree(), "README");
    assertThat(readmeFile, exists());
    assertThat(readmeFile, ofLength(12));
}
```

Rationale: The assertThat() method is called 3 times within the test method. Each assert statement checks for a different condition, but the developer does not provide a explanation message for each assert statement. Hence, if one of the assert statements were to fail, identifying the cause of the failure is not straightforward.

Conditional Test Logic

```
@Test
public void testSpinner() {
    for (Map.Entry entry : sourcesMap.entrySet()) {

        String id = entry.getKey();
        Object resultObject = resultsMap.get(id);
        if (resultObject instanceof EventsModel) {
            EventsModel result = (EventsModel) resultObject;
            if (result.testSpinner.runTest) {
                System.out.println("Testing " + id + " (testSpinner)");
                //System.out.println(result);
                AnswerObject answer = new AnswerObject(entry.getValue(), "", new CookieManager(), "");
                EventsScraper scraper = new EventsScraper(RuntimeEnvironment.application, answer);
                SpinnerAdapter spinnerAdapter = scraper.spinnerAdapter();
                assertEquals(spinnerAdapter.getCount(), result.testSpinner.data.size());
                for (int i = 0; i < spinnerAdapter.getCount(); i++) {
                    assertEquals(spinnerAdapter.getItem(i), result.testSpinner.data.get(i));
                }
            }
        }
    }
}
```

Rationale: The test method, testSpinner(), contains multiple control statements (i.e. control flow statements). The success or failure of the test is based on the result of the assertion method which is within the control flow blocks and hence not predictable. This also increases the complexity of the test method and hence has a negative impact on maintenance of the test.

Constructor initialization

```
public class TagEncodingTest extends BrambleTestCase {
        private final CryptoComponent crypto;
        private final SecretKey tagKey;
        private final long streamNumber = 1234567890;

        public TagEncodingTest() {
                crypto = new CryptoComponentImpl(new TestSecureRandomProvider());
                tagKey = TestUtils.getSecretKey();
        }

        @Test
        public void testKeyAffectsTag() throws Exception {
                Set set = new HashSet<>();
                for (int i = 0; i < 100; i++) {
                        byte[] tag = new byte[TAG_LENGTH];
                        SecretKey tagKey = TestUtils.getSecretKey();
                        crypto.encodeTag(tag, tagKey, PROTOCOL_VERSION, streamNumber);
                        assertTrue(set.add(new Bytes(tag)));
                }
        }
 ...
}
```

Rationale: The test class utilizes a constructor instead of a setUp method to initialize fields.

Default test

```
public class ExampleUnitTest {
    @Test
        public void addition_isCorrect() throws Exception {
                assertEquals(4, 2 + 2);
        }

    @Test
    public void shareProblem() throws InterruptedException {
        .....
                Observable.just(200)
                .subscribeOn(Schedulers.newThread())
                .subscribe(begin.asAction());
                begin.set(200);
                Thread.sleep(1000);
                assertEquals(beginTime.get(), "200");
        .....
        }
        .....
}
```

Rationale: The default test class provided by Android Studio is utilized to hold actual test methods. The developer should rename this test class (and remove the example test method).

Duplicate assert

```java
@Test
public void testXmlSanitizer() {
    boolean valid = XmlSanitizer.isValid("Fritzbox");
    assertEquals("Fritzbox is valid", true, valid);
    System.out.println("Pure ASCII test - passed");

    valid = XmlSanitizer.isValid("Fritz Box");
    assertEquals("Spaces are valid", true, valid);
    System.out.println("Spaces test - passed");

    valid = XmlSanitizer.isValid("Frützbüx");
    assertEquals("Frützbüx is invalid", false, valid);
    System.out.println("No ASCII test - passed");

    valid = XmlSanitizer.isValid("Fritz!box");
    assertEquals("Exclamation mark is valid", true, valid);
    System.out.println("Exclamation mark test - passed");

    valid = XmlSanitizer.isValid("Fritz.box");
    assertEquals("Exclamation mark is valid", true, valid);
    System.out.println("Dot test - passed");

    valid = XmlSanitizer.isValid("Fritz-box");
    assertEquals("Minus is valid", true, valid);
    System.out.println("Minus test - passed");

    valid = XmlSanitizer.isValid("Fritz-box");
    assertEquals("Minus is valid", true, valid);
    System.out.println("Minus test - passed");
}
```

Rationale: In this test method, testXmlSanitizer(), the developer tests 'Exclamation mark is valid', 'Frützbüx is invalid' and 'Minus is valid' multiple times in the same test method.

Eager test

```
@Test
public void NmeaSentence_GPGSA_ReadValidValues(){
    NmeaSentence nmeaSentence = new NmeaSentence("$GPGSA,A,3,04,05,,09,12,,,24,,,,,2.5,1.3,2.1*39");
    assertThat("GPGSA - read PDOP", nmeaSentence.getLatestPdop(), is("2.5"));
    assertThat("GPGSA - read HDOP", nmeaSentence.getLatestHdop(), is("1.3"));
    assertThat("GPGSA - read VDOP", nmeaSentence.getLatestVdop(), is("2.1"));
}
```

Rationale: In this test method, NmeaSentence_GPGSA_ReadValidValues(), the developer calls multiple methods of the production class. Testing multiple methods of the production class in a single test method causes confusion as to what exactly the test method is testing.

Empty test

```
public void testCredGetFullSampleV1() throws Throwable{
//        ScrapedCredentials credentials =  innerCredTest(FULL_SAMPLE_v1);
//        assertEquals("p4ssw0rd", credentials.pass);
//        assertEquals("user@example.com",credentials.user);
}
```

Rationale: The test method, testCredGetFullSampleV1(), contains only comments (i.e. no executable statements). A test method without executable statements will be marked as passing when executed.

Exception Handling

```
@Test
public void realCase() {
    Point p34 = new Point("34", 556506.667, 172513.91, 620.34, true);
    Point p45 = new Point("45", 556495.16, 172493.912, 623.37, true);
    Point p47 = new Point("47", 556612.21, 172489.274, 0.0, true);
    Abriss a = new Abriss(p34, false);
    a.removeDAO(CalculationsDataSource.getInstance());
    a.getMeasures().add(new Measure(p45, 0.0, 91.6892, 23.277, 1.63));
    a.getMeasures().add(new Measure(p47, 281.3521, 100.0471, 108.384, 1.63));

    try {
        a.compute();
    } catch (CalculationException e) {
        Assert.fail(e.getMessage());
    }

    // test intermediate values with point 45
    Assert.assertEquals("233.2405",
        this.df4.format(a.getResults().get(0).getUnknownOrientation()));
    Assert.assertEquals("233.2435",
        this.df4.format(a.getResults().get(0).getOrientedDirection()));
    Assert.assertEquals("-0.1", this.df1.format(
        a.getResults().get(0).getErrTrans()));

    // test intermediate values with point 47
    Assert.assertEquals("233.2466",
        this.df4.format(a.getResults().get(1).getUnknownOrientation()));
    Assert.assertEquals("114.5956",
        this.df4.format(a.getResults().get(1).getOrientedDirection()));
    Assert.assertEquals("0.5", this.df1.format(
        a.getResults().get(1).getErrTrans()));

    // test final results
    Assert.assertEquals("233.2435", this.df4.format(a.getMean()));
    Assert.assertEquals("43", this.df0.format(a.getMSE()));
    Assert.assertEquals("30", this.df0.format(a.getMeanErrComp()));
}
```

Rationale: In this example, the developer fails the test when a specific exception occurs. Ideally, the developer should split this test method into multiple test methods that (1) knowingly generate the exception and (2) do not generate the exception. The developer should utilize the @Test expected attribute in JUnit 4 to fail the test when the exception occurs instead of explicitly catching or throwing the exception.

General Fixture

```
protected void setUp() throws Exception {
    assetManager = getInstrumentation().getContext().getAssets();
    certificateFactory = CertificateFactory.getInstance("X.509");

    infoDebianTestCA = loadCertificateInfo("DebianTestCA.pem");
    infoDebianTestNoCA = loadCertificateInfo("DebianTestNoCA.pem");
    infoGTECyberTrust = loadCertificateInfo("GTECyberTrustGlobalRoot.pem");

    // user-submitted test cases
        infoMehlMX = loadCertificateInfo("mehl.mx.pem");
}

public void testIsCA() {
    assertTrue(infoDebianTestCA.isCA());
    assertFalse(infoDebianTestNoCA.isCA());
    assertNull(infoGTECyberTrust.isCA());

    assertFalse(infoMehlMX.isCA());
}
```

Rationale: The setup/fixture method initializes a total of 6 fields (variables). However, the test method, testIsCA(), only utilizes 4 fields.

Ignored Test

```
@Ignore("disabled for now as this test is too flaky")
public void peerPriority() throws Exception {
        final List addresses = Lists.newArrayList(
        new InetSocketAddress("localhost", 2000),
        new InetSocketAddress("localhost", 2001),
        new InetSocketAddress("localhost", 2002)
        );
    peerGroup.addConnectedEventListener(connectedListener);
    .....
}
```

Rationale: This test will not be executed due to the @Ignore annotation.

Lazy Test

```
@Test
public void testDecrypt() throws Exception {
    FileInputStream file = new FileInputStream(ENCRYPTED_DATA_FILE_4_14);
    byte[] enfileData = new byte[file.available()];
    FileInputStream input = new FileInputStream(DECRYPTED_DATA_FILE_4_14);
    byte[] fileData = new byte[input.available()];
    input.read(fileData);
    input.close();
    file.read(enfileData);
    file.close();
    String expectedResult = new String(fileData, "UTF-8");
    assertEquals("Testing simple decrypt",expectedResult, Cryptographer.decrypt(enfileData, "test"));
}

@Test
public void testEncrypt() throws Exception {
    String xml = readFileAsString(DECRYPTED_DATA_FILE_4_14);
    byte[] encrypted = Cryptographer.encrypt(xml, "test");
    String decrypt = Cryptographer.decrypt(encrypted, "test");
    assertEquals(xml, decrypt);
}
```

Rationale: Both test methods, testDecrypt() and testEncrypt(), call the same SUT method, Cryptographer.decrypt().

Magic Number Test

```
@Test
public void testGetLocalTimeAsCalendar() {
    Calendar localTime = calc.getLocalTimeAsCalendar(BigDecimal.valueOf(15.5D), Calendar.getInstance());
    assertEquals(15, localTime.get(Calendar.HOUR_OF_DAY));
    assertEquals(30, localTime.get(Calendar.MINUTE));
}
```

Rationale: In this test method, it is not known the significance of the two numeric literals that are passed as parameters in the assertion method.

Mystery Guest

```
public void testPersistence() throws Exception {
    File tempFile = File.createTempFile("systemstate-", ".txt");
    try {
        SystemState a = new SystemState(then, 27, false, bootTimestamp);
        a.addInstalledApp("a.b.c", "ABC", "1.2.3");

        a.writeToFile(tempFile);
        SystemState b = SystemState.readFromFile(tempFile);

        assertEquals(a, b);
    } finally {
        //noinspection ConstantConditions
        if (tempFile != null) {
            //noinspection ResultOfMethodCallIgnored
            tempFile.delete();
        }
    }
}
```

Rationale: As part of the test, the test method, testPersistence(), creates a File (tempFile) in a specific directory and then utilizes this file in the test process.

Redundant Print

```
@Test
public void testTransform10mNEUAndBack() {
    Leg northEastAndUp10M = new Leg(10, 45, 45);
    Coord3D result = transformer.transform(Coord3D.ORIGIN, northEastAndUp10M);
    System.out.println("result = " + result);
    Leg reverse = new Leg(10, 225, -45);
    result = transformer.transform(result, reverse);
    assertEquals(Coord3D.ORIGIN, result);
}
```

Rationale: The test method, testTransform10mNEUAndBack(), contains a statement that prints the value of a variable to the console. This is a redundant statement that might have been added by a developer, for debugging purposes, at the time of writing the test method.

Redundant Assertion

```
@Test
public void testTrue() {
    assertEquals(true, true);
}
```

Rationale: The test method, testTrue(), will always pass as since the assert statement compares a Boolean value of true against another Boolean value of true.

Resource Optimism

```
@Test
public void saveImage_noImageFile_ko() throws IOException {
File outputFile = File.createTempFile("prefix", "png", new File("/tmp"));
ProductImage image = new ProductImage("01010101010101", ProductImageField.FRONT, outputFile);
Response response = serviceWrite.saveImage(image.getCode(), image.getField(), image.getImguploadFront(), image.getImguploadIngredients(), image.getImguploadNutrition()).execute();
assertTrue(response.isSuccess());
assertThatJson(response.body())
    .node("status")
        .isEqualTo("status not ok");
}
```

Rationale: The test method accesses a file without verifying if the file exists before using it in the test operations.

Sensitive Equality

```
@Test
public void test1() throws UnknownHostException {

    String peersPacket = "F8 4E 11 F8 4B C5 36 81 " +
        "CC 0A 29 82 76 5F B8 40 D8 D6 0C 25 80 FA 79 5C " +
        "FC 03 13 EF DE BA 86 9D 21 94 E7 9E 7C B2 B5 22 " +
        "F7 82 FF A0 39 2C BB AB 8D 1B AC 30 12 08 B1 37 " +
        "E0 DE 49 98 33 4F 3B CF 73 FA 11 7E F2 13 F8 74 " +
        "17 08 9F EA F8 4C 21 B0";

    byte[] payload = Hex.decode(peersPacket);

    byte[] ip = decodeIP4Bytes(payload, 5);

    assertEquals(InetAddress.getByAddress(ip).toString(), ("/54.204.10.41"));
}
```

Rationale: Use of the default value returned by an objects toString() method, to perform string comparisons, runs the risk of failure in the future due to changes in the objects implementation of the toString() method.

Sleepy Test

```
public void testEdictExternSearch() throws Exception {
    final Intent i = new Intent(getInstrumentation().getContext(), ResultActivity.class);
    i.setAction(ResultActivity.EDICT_ACTION_INTERCEPT);
    i.putExtra(ResultActivity.EDICT_INTENTKEY_KANJIS, "空白");
        tester.startActivity(i);
        assertTrue(tester.getText(R.id.textSelectedDictionary).contains("Default"));
        final ListView lv = getActivity().getListView();
        assertEquals(1, lv.getCount());
        DictEntry entry = (DictEntry) lv.getItemAtPosition(0);
        assertEquals("Searching", entry.english);
        Thread.sleep(500);
        final Intent i2 = getStartedActivityIntent();
        final List result = (List) i2.getSerializableExtra(ResultActivity.INTENTKEY_RESULT_LIST);
        entry = result.get(0);
        assertEquals("(adj-na,n,adj-no) blank space/vacuum/space/null (NUL)/(P)", entry.english);
        assertEquals("空白", entry.getJapanese());
        assertEquals("くうはく", entry.reading);
        assertEquals(1, result.size());
}
```

Rationale: The developer causes an artificial delay in test execution using Thread.sleep(). Without comments, it is assumed that the developer performs the delay to stimulate an actual activity (i.e., searching).

Unknown Test

```
@Test
public void hitGetPOICategoriesApi() throws Exception {
    POICategories poiCategories = apiClient.getPOICategories(16);
    for (POICategory category : poiCategories) {
      System.out.println(category.name() + ": " + category);
    }
}
```

Rationale: Due to a missing assertion method, it is not possible to know what this method is testing.