# Static and dynamic verification techniques

**Systems Engineering Laboratory 1**

A good code should

1. be syntactically correct,
2. have a good quality (readable, reusable, maintainable, ...),
3. be free of errors,
4. adhere to the specification.

(1) can be enforced simply by a compiler. (2) can be improved by keeping to certain *coding guidelines*. (3-4) can be checked by for example by manual *code review*, automated *static analysis* or *testing* (dynamic). The main difference between static and dynamic verification techniques is that dynamic techniques execute the code under analysis, while static techniques only use the code as a static input.

To have an overview of the topic of this laboratory exercise, take a look at the lecture note from the previous semester from the Systems Engineering (VIMIAC01) course.

## 1 Static techniques

The purpose of static code analysis techniques is to improve the quality of code *without executing* it.

### 1.1 Coding guidelines

Coding guidelines usually consist of a *set of rules* that give *recommendations on the style* of code (e.g., formatting, naming, structure, ...) and on certain *programming practices* (constructs, architecture, ...). Coding guidelines can be

- *industry* or *domain* specific, e.g., automotive or railway industry,
- *platform* specific, e.g., C, C++, C#, Java,
- *organization* specific, e.g., Google, CERN.

### 1.1.1 Examples

- *MISRA C* (Motor Industry Software Reliability Association) is an industry specific guideline that focuses on *safety, security, reliability and portability*. It contains 143 rules, including the following examples.
  - "RHS of `&&` and `||` operators shall not contain side effects."
  - "Test against zero should be made explicit for non-Booleans."
  - "Body of if, else, while, do, for shall always be enclosed in braces."
- *.NET Framework Design Guidelines* is a platform specific (C#) guideline that focuses on *framework and API development*. It contains rules regarding naming, type design, member design, extensibility, exceptions, usage and common design patterns. Some examples:
  - "DO NOT provide abstractions unless they are tested by developing several concrete implementations and APIs consuming the abstractions."
  - "CONSIDER making base classes abstract even if they don't contain any abstract members. This clearly communicates to the users that the class is designed solely to be inherited from."
  - "DO use the same name for constructor parameters and a property if the constructor parameters are used to simply set the property."
- *Google Java Style Guide* is an organization and platform specific guideline that focuses on *"hard-and-fast" rules* that can be checked explicitly and easily. Its main categories are source file basics, source file structure, formatting, naming, programming practices and Javadoc. Some examples:
  - "Never make your code less readable simply out of fear that some programs might not handle non-ASCII characters properly. If that should happen, those programs are broken and they must be fixed."
  - "In Google Style special prefixes or suffixes, like those seen in the examples, `mName`, `s_name` and `kName`, are not used."
  - "When a reference to a static class member must be qualified, it is qualified with that class's name, not with a reference or expression of that class's type."
  - "Local variable names are written in `lowerCamelCase`."

### 1.1.2 Using coding guidelines

Coding guidelines can be *enforced* in several ways. Many IDEs have some guidelines as their base functionality but external tools can also be used. It is important however, that they should be *integrated tightly* into the development workflow (e.g., checking the code after each commit). There are several tools (e.g., SonarQube, Coverity, ...) that can automatically check adherence to coding guidelines.

There are many coding guidelines and there is usually no absolute "best" one to pick. In many cases it is already determined by the industry, platform or organization. However, sometimes it is possible to decide on a guideline (e.g., when starting a new project). It is recommended to pick a guideline that is the most consistent with the current code base. Guidelines can also be combined, but it is not recommended to "reinvent the wheel" because it will make a hard time for new developers in the team.

## 1.2 Code review

Code review is the process of *manually reading, checking and analyzing* source code. Code review has different levels regarding its formality.

- Informal review is usually performed by other team members, controlled by the team lead.
- Walkthrough is also a more informal process, lead by the author of the code.
- Technical review is a more formal, well-prepared, documented process, which also involves experts.
- Inspection is a formal, well-prepared, documented process, which is usually performed by external experts and moderators.

A formal code review usually consists of the following steps.

- Planning: defining review criteria, allocating roles.
- Kick-off: distributing documents, explaining objectives.
- Individual preparation: reviewing artefacts, noting potential defects, questions and comments.
- Review meeting: discussing and logging results, noting defects, making decisions.
- Rework: fixing defects, recording updated status.
- Follow-up: checking fixes, checking exit criteria (e.g., whether the code quality is good enough or another review is needed).

Code review is usually based on a *structured checklist*, which contains similar categories to coding guidelines (readability, maintainability, security, vulnerability, performance, common patterns, best practices, ...). There are several checklists available online, but it is recommended to *automate* the process as much as possible (e.g., formatting can be checked by an automatic tool).

There are also tools (GitHub pull request, Gerrit) that can *support* the code review process. These tools can usually attach comments and discussions to code lines or snippets and can integrate the review process as a step in the development workflow.

## 1.3 Static analysis

Static analysis is the *analysis of software without execution*. It is usually performed by *automated* tools, but code review (performed manually by humans) is sometimes also considered as static analysis.

Static analysis techniques usually belong to the following main categories.

- Pattern-based: These tools can check *basic static properties* that match certain error patterns, e.g., ignored return value, unused variable. Example tools are SonarQube, FindBugs, PMD.
- Interpretation-based: These tools can check certain *dynamic properties* as well, e.g., null pointer dereference, index out of bounds. Example tools are Coverity, Infer, PolySpace.

### 1.3.1 Examples

The following examples are potential problems that *FindBugs* can detect.

- Bad practice: random object created and used only once.
- Correctness: bitwise add of signed byte value.
- Vulnerability: expose inner static state by storing mutable object into a static field.
- Multithreading: synchronization on Boolean could lead to deadlock.
- Performance: invoke toString() on a string.
- Security: hardcoded constant database password.
- Dodgy: useless assignment in return statement.

*SonarQube* is a code quality management platform that supports more than 20 languages (including Java, C, C++, C#). It can check coding guidelines, duplicated code, test coverage, code complexity, potential bugs, vulnerabilities and can also calculate technical debt. It stores the data of its executions in a database making it possible to produce reports and evaluation graphs about the projects. It can be integrated with IDEs and CI tools as well.

*Coverity* is the static analyzer of the Synopsys suite, supporting many languages (including C, C++, C#, Java, JavaScript). It can detect dynamic properties as well such as resource leaks, null pointers, uninitialized data and concurrency issues.

### 1.3.2 Using static analysis tools efficiently

Static analysis tools should be *integrated* into the build process by for example performing the analysis before/after each commit, generating reports and sending potential problems in e-mails. These tools should be used *from the start* of a project otherwise they will find so many errors that developers will be discouraged from using them. An alternative solution is to *configure* the tools so that first they only find the most critical problems. Configuration can also be used to add new rules. The results of these tools however, should be *treated carefully* as false positives or negatives can occur. On one hand, if the tools do not find any problem, it may not mean a completely correct software (false positive). On the other hand, an error found may not cause a real failure (false negative). In the latter case the error (or the whole) rule can be suppressed, but it should always be indicated and explained in a comment.

## 1.4 Summary

The main advantage of static techniques is that they can analyze the software without execution. This is especially useful if the software is not yet in an executable state, no input is present or execution is expensive. Static techniques may find subtle errors that can be interesting even for experienced developers. Furthermore, the whole process can be in many cases fully automated and integrated into the development process.

# 2 Dynamic techniques – unit testing

As it has been introduced in other lectures of previous semesters, software testing can performed on different levels during the development process. Starting from the system test, through the integration tests, one can also perform unit tests. This laboratory focuses on unit testing among the dynamic verification techniques.

## 2.1 Introduction

The *unit* in general, is a logically well-separable part of the source code. In case of object-oriented software, this usually means a class or a small set of classes. For scripting languages, the unit is typically a module or a file. A unit usually has a well-defined interface in order to reach its features from outside. This is a crucial aspect for software testability.

The objective of *unit testing* is to detect and repair the defects found in the unit. This is the lowest level of testing (think about the V-model). Detecting bugs early in the development process may increase the quality of the implement system and might reduce the additional costs. Testing of a unit is usually performed isolated and individually. This has several benefits, including the followings.

- Complexity stays on a manageable level.
- Finding bugs is more simple.
- Increases self-confidence when performing code changes in a small unit.

A *unit test* tests a well-defined functionality or feature, and thus defines a behavioral contract for the unit under test.

## 2.2 Unit testing frameworks

Unit test frameworks are tools, which enables execution of existing unit tests in a fast, systematical way. There are many such tools (e.g., JUnit, xUnit, TestNG), however many IDEs contain a unit test framework as well (e.g., Eclipse, Visual Studio). These frameworks often have numerous features, yet the most basic and important ones are the following.

- Definition of unit test cases and sequences.
- Execution of unit tests.
- Display of test execution results.

### 2.2.1 Example JUnit test case

```
public class ListTest {
    List list; // Reference to the unit

    // Initialization before all test cases
    @Before public void setUp(){
```

```
        list = new List(); // Instantiation of the unit
    }

    @Test public void add_EmptyList_Success(){
        list.Add(1); // Invocation of the functionality under test
        assertEquals(1, list.getSize()); // Checking the results
    }
}
```

### 2.2.2 JUnit annotations

The list below is only an excerpt, you can find many extended lists on the web, such as this.

- `@Test`: Definition of a test method
- `@Before`: Invoked before every test cases.
- `@After`: Invoked after every test case.
- `@BeforeClass`: Invoked before all test cases found in a test class (thus, test cases may affect each other).
- `@AfterClass`: Invoked after all test cases found in a test class (e.g., deleting a global state).
- `@Ignore`: Omitting the given test case from execution.
- `@Test(expected=IllegalArgumentException.class)`: A test case with this annotation only passes, when `IllegalArgumentException` is thrown.
- `@Test(timeout=100)`: A test case with this annotation will fail when the execution requires more than 100 ms to finish.

## 2.3 What is a good unit test?

The followings are guidelines worth considering.

- *Simplicity, reliability*: Does not contain complex logic (e.g., loops, try-catches, etc.).
- *One test case, one functionality*: Every test case focuses on a well-defined functionality. However, this does not imply that only one assertion could be used.
- *Code quality matters*: No duplications, good readability, and easy understanding.
- *Independence*: Test cases shall not have effect on each other, since it can lead to unexpected behaviors and unreliable tests.
- *Checks*: The assertions in one test case shall not be over-specified (too many assertions may result in this), and shall verify a well-defined functionality (see one test case, one functionality).

### 2.3.1 Typical conventions

- Test class name: [Unit_name]Test
- Test method name:
  - [Method]_[StateUnderTest]_[ExpectedBehavior]

  - [Method]_[WhatItDoes]_[IfTheseConditionsApply]
  - [TheFunctionalityUnderTest]
- Structure
  - Arrange-Act-Assert
  - Given-When-Then

## 2.4 Isolation

Unit testing usually comes along with a problem, namely the *dependencies* of the unit under test (e.g., other modules, file system, calls to network, etc.). These dependencies may affect the outcome of unit the tests, which must be avoided. In order to overcome this, two approaches are employed in general: 1) design for testability and 2) usage of test doubles. Test double is a common name for replacement object that can be used in the unit tests instead of the original ones. The most frequently used test doubles are the followings.

- *Stub*: Stubs have no verification logic inside, they usually only return a simple value or an object based on predefined rules. The tests verify the *state* of the unit under test (e.g., by checking the return value of the tested method or querying getters).
- *Mock*: Can return values by predefined rules, but also stores the calls issued to itself. Therefore unit tests can verify the *interactions* of the unit under test by using checks on the mocks. For example, using mocks it can be verified how many times the unit under test has called a method with various arguments.
- *Dummy*: A test double, which is only used to have an compilable and executable code, however it is not being used during the execution.
- *Fake*: A test double, which has a minimal functionality, yet must stay rather simple.

### 2.4.1 Isolation frameworks

Such tools are used to isolate the dependencies of the unit under test using special APIs. These APIs usually return a test double from an interface or a class description. Mockito, Rhino Mocks and TypeMock are three of the most frequently used isolation frameworks.

### 2.4.2 Example for using Mockito

```
public class PriceServiceTest {

    DataAccess mockDA;

    @Before public void init() {
        // creating a mock for the DataAccess class
        mockDA = mock(DataAccess.class);
        // injecting the mock into the unit under test
        ps = new PriceService(mockDA);
    }
```

```
    @Test public void SuccessfulPriceQuery() {
        // Arrange
        // If getProdPrice is called with "A100", then it returns 50.
        when(mockDA.getProdPrice("A100")).thenReturn(50);

        // Act
        int p = ps.getPrice("A100");

        // Assert
        // Verifying that the mocked call has been invoked only once with argument "A100".
        verify(mockDA, times(1)).getProdPrice("A100");
    }
    ...
}
```