

SPW4

Test Automation & Continuous Delivery




Unit Testing

S. Wagner

Software Tests

- How can we assure that our software is correct (Verification) and valid (Validation)?
- Formal Verification
 - i.e., prove it mathematically
 - formal verification proves correctness
 - formally verifiable specifications are hard to write
 - complete verification according to a formal specification is very hard (in the best case it is "only" NP-complete)
 - (currently) formal verification is not a mainstream topic in software development
 - only applied in very critical and special cases
- Testing
 - i.e., try it empirically
 - shows that specific cases do not lead to failures
 - testing cannot guarantee that software is correct (i.e., does not contain *any* bugs)
 - effort can be scaled
 - testing has many facets (e.g., functionality, performance, usability, interaction with other systems, etc.)

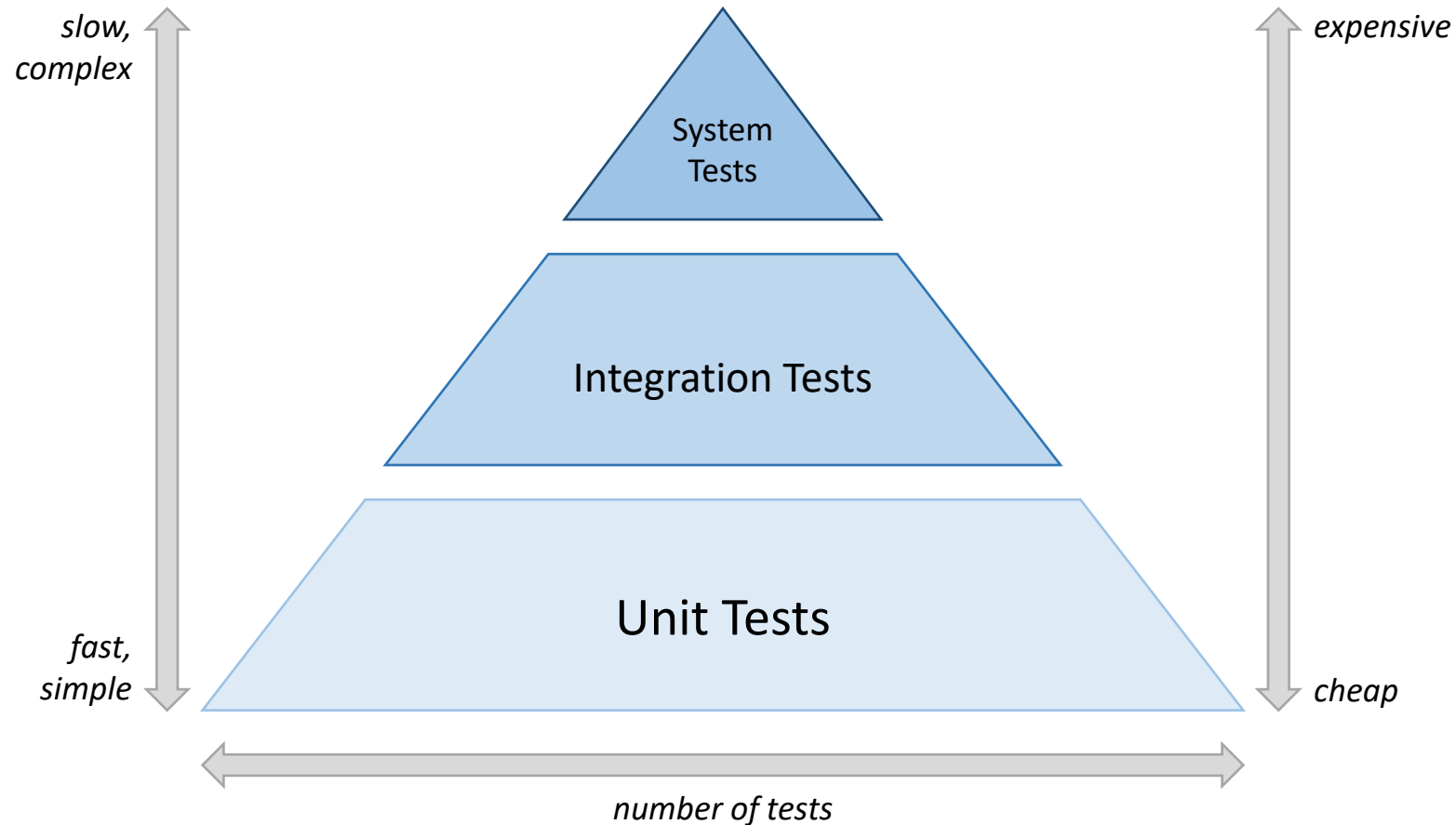
Software Defects

- A software defect (aka. bug) is a software problem in general
- Different terms describe propagation of a defect
 - **Error:** human action that produces an incorrect result
 -  **Fault:** manifestation of an error in a software system
 -  **Failure:** inability of the software system to fulfill its requirements
 -  **Incident:** symptom of a failure perceived by the user
- Static analysis tries to identify faults by analyzing code (e.g., code reviews, static code analysis, ...)
- Software testing tries to spot failures by dynamically executing code (e.g., unit tests, integration tests, ...)

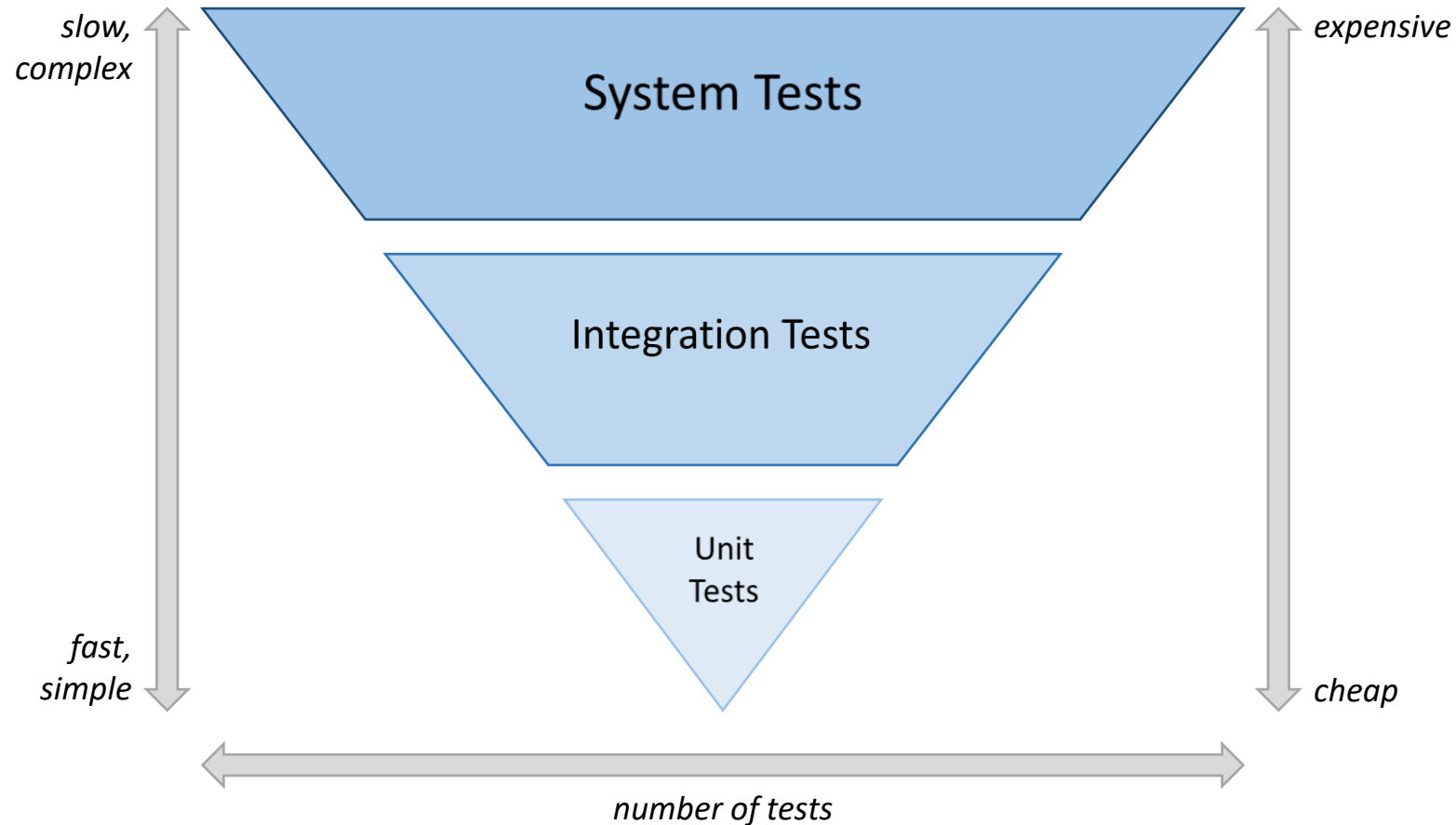
Layers of Software Tests

- Unit Tests = lowest layer
 - checks functionality of a small and clearly separated part (often a single method)
 - mostly white-box tests (tested unit and test are written by the same person)
 - tests are small, often quite simple, and can be executed very quickly
 - easy to automate
- Integration Tests
 - checks the interaction between different components (for example different classes, subsystems, layers, etc.)
 - can be hierarchically structured (bottom-up vs. top-down)
- System Tests = highest layer
 - system as a whole is tested according to the specification
 - mostly black-box tests (tester and developer are not the same person)
 - often include UI testing
 - tests are comprehensive, complex, and time-consuming
 - hard(er) to automate

Test Pyramid



Test Pyramid in Reality?



Use of Software Tests

- Tests should help to find errors as early as possible (i.e., shortly after they have been introduced)
 - context of the error is still present
 - cause of the error is easier to identify
 - fixing the error does not take long and is cheap
- Developers are responsible for product quality, and not testers
- You cannot test quality into software
- Inverted test pyramid results from manual testing
- Test automation is the key

Purpose of Unit Testing

- Unit tests ...
 - lead to a testable software design and thus to higher quality
 - give immediate feedback if something breaks
 - are a safety net when refactoring
 - enable change
 - protect from regressions
 - force us to think about "what" and not about "how"
 - are a form of compilable and executable documentation
 - can be used to explore APIs of new libraries (exploratory testing)

Testability

- Testability describes how easy or hard it is to test a software system
 - software quality criterion which belongs to maintainability (cf. ISO/IEC 25010)
 - bad testability → fewer tests → more failures → higher cost
 - bad testability inverts the test pyramid
 - testability should be kept in mind throughout the whole software lifecycle
- Test-driven development enforces testable software



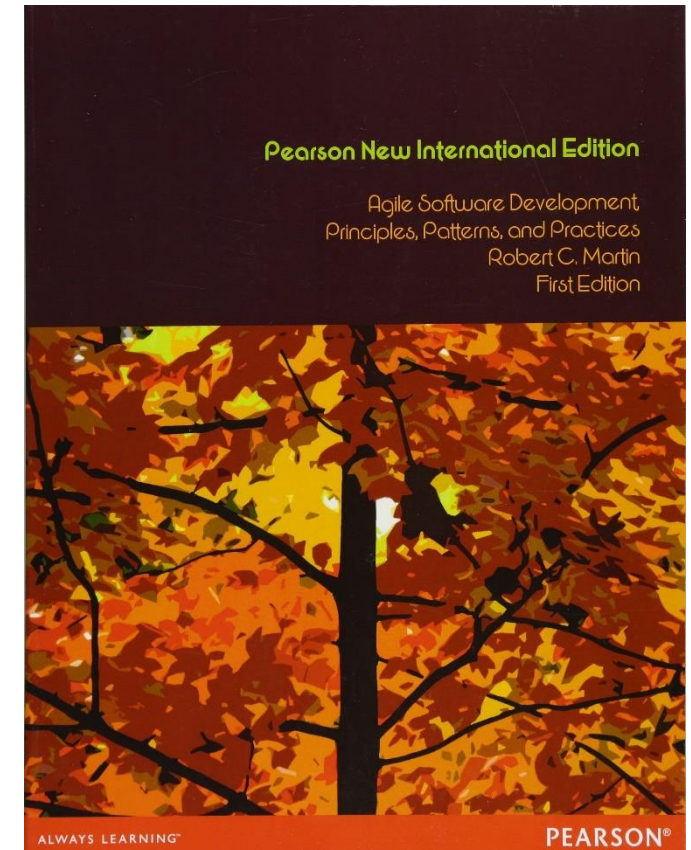
Impediments for Testability

- Testability is bad, if ...
 - software entities cannot be isolated
 - coupling between entities is high
 - initialization is inflexible
 - internal states or collaborators cannot be accessed
 - software entities cannot be controlled
 - results are hidden and cannot be observed
- Indicators for Bad Testability
 - writing tests requires much effort
 - automating tests is hard
 - many test doubles are necessary

Principles for Testable Design

... and for good software design in general

- S** • Single-Responsibility Principle (SRP)
- O** • Open-Closed Principle (OCP)
- L** • Liskov Substitution Principle (LSP)
- I** • Interface-Segregation Principle (ISP)
- D** • Dependency-Inversion Principle (DIP)



Single-Responsibility Principle

A class should have only one reason to change.

- i.e., each software entity should have a single and specific purpose
- leads to better separation of concerns and stronger cohesion
- software entities are simpler and easier to understand
- software entities are only changed on purpose
- sometimes hard to achieve (e.g., in the case of cross-cutting concerns), which motivates other programming paradigms such as aspect-oriented programming (AOP)
- software entities following the SRP are easier to test, because they have a single and clear functionality

Open-Closed Principle

Software entities should be open for extension, but closed for modification.

- introduce reasonable abstractions
- use composition and inheritance for extensions
- clearly define where extension can or cannot take place
- if extensions are necessary, the original code should not change
- abstractions make tests less brittle

Liskov Substitution Principle

Subtypes must be substitutable for their base types.

- derived classes must preserve the basic behavior of their super classes
- otherwise the is-a relationship between classes and their subclasses breaks
- objects of the subclass can replace objects of the super class at any time
- remember that a subclass has to be a specialization
- leads to good abstractions and extensibility
- well separated layers of abstraction are easier to test
- extensible classes can be used as base classes for test doubles

Interface-Segregation Principle

Clients should not be forced to depend on methods that they do not use.

- you should only depend on what you really need
- reduces unnecessary coupling
- makes code much more readable
- related to the SRP
- clear and specific interfaces are easier to test, because they represent a specific purpose

Dependency-Inversion Principle

High-level modules should not depend on low-level modules.

Both should depend on abstractions.

Abstractions should not depend on details.

Details should depend on abstractions.

- program to an interface, not to an implementation
- leads to clearer abstraction and better extensibility
- dependencies should not be hard-coded and should be set as late as possible (cf. dependency injection)
- collaborators can be replaced by test doubles easily

Principles for Good Unit Tests

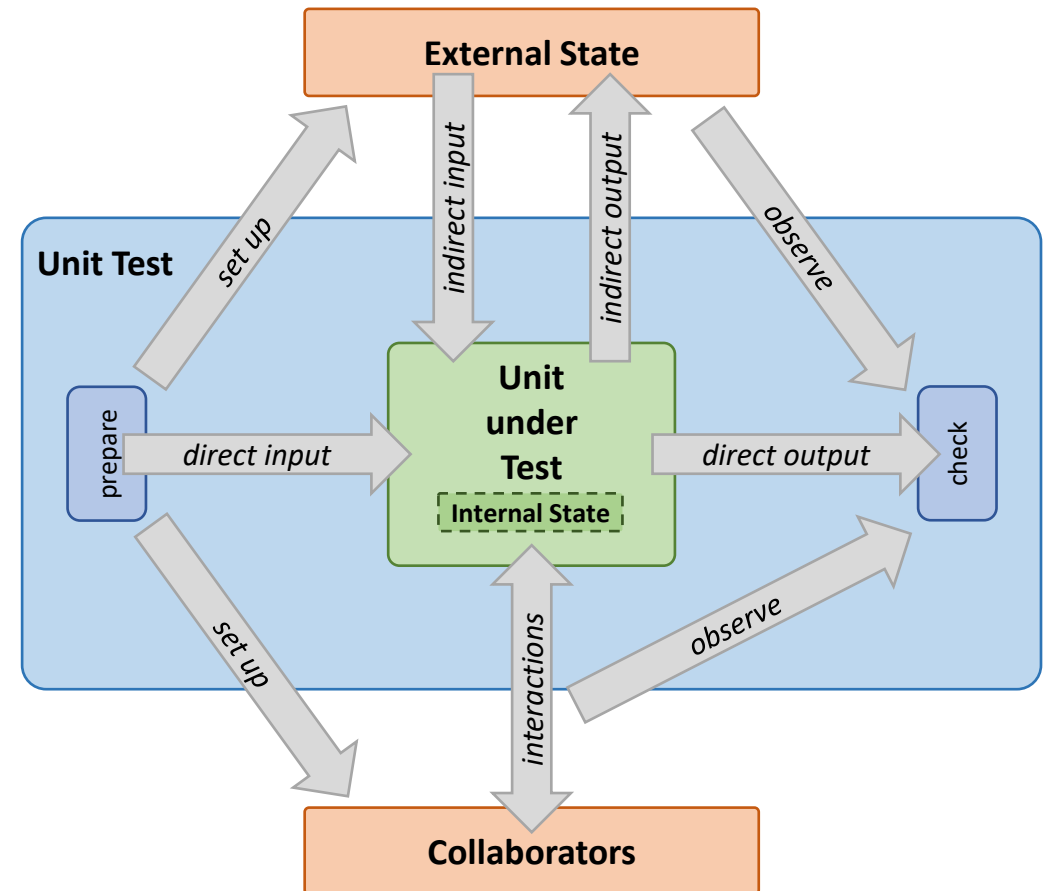
Unit tests should follow the following rules:

(cf. "Clean Code" by Robert Martin)

- F** • **fast** ... so they are often executed
- I** • **independent** ... so a failing test does not make other tests fail
- R** • **repeatable** ... so they can be run in any environment
- S** • **self-validating** ... so it is clear whether they pass or fail
- T** • **timely** ... so they will improve the design of production code

Anatomy of a Unit Test

- Direct input and direct output must be prepared and checked
- Internal state has to be initialized and checked
- External state used by the unit under test has to be set up and observed
- External collaborators called by the unit under test have to be set up
- Interactions with external collaborators have to be observed
- AAA Pattern
 - Arrange
 - Act
 - Assert



Arrange – Act – Assert (AAA) Pattern

- Each unit test should consist of three steps:
 - Arrange: prepare the unit under test
 - Act: execute the unit under test
 - Assert: verify the result
- Each unit test should test a single aspect of the system
 - avoid testing multiple test cases in a single unit test
 - if the first assertion fails, all other assertions are not checked anymore
 - each unit test should only contain a single assertion
 - if the test fails, it should be clear what is wrong without having to study the test or its output in detail

```
class CalculatorTest {  
    @Test  
    void add_withValidParams_returnsSum() {  
        // arrange  
        Calculator calc = new Calculator();  
  
        // act  
        int result = calc.add(3, 2);  
  
        // assert  
        assertEquals(5, result);  
    }  
}
```

Naming of Unit Tests

- Each unit test represents a specific test case
 - if it fails, we want to know what's wrong
 - a detailed inspection of the test should not be necessary
 - in the unit test runner, each test is listed by its name
 - the name must be so descriptive, that we know what caused the failure
- Unit tests (i.e., the test classes and methods) must be named with care
 - some unit test frameworks offer other possibilities to add longer and more descriptive names (cf. `@DisplayName` in JUnit)
 - otherwise, the method name must contain all required information
 - which unit is tested (what)
 - under which circumstances (when)
 - what is the expected result (returns, throws, ...)
 - e.g., `GetPackageLoadInfo_WithPackageWhereStatusIsNotOK_ThrowsArgumentException`

DRY vs. DAMP

- DRY = "Don't Repeat Yourself"
 - in production code, code duplication is considered evil
 - common pieces of code should be refactored into methods, classes, etc. so that they can be reused
 - DRY code is easy to maintain and to test, but harder to read
- DAMP = "Descriptive And Meaningful Phrases"
 - code can be read almost like normal text
 - understanding the code is easy and does not require special training
 - good principle for domain specific languages ... and unit tests
- Unit tests should not be too DRY and should follow the DAMP principle
 - a unit test stands by itself and is not protected by another test
 - there are no tests to test unit tests
 - simplicity and readability are of major importance for good unit tests
 - fluent APIs are great for unit tests

Unit Tests vs. Integration Tests

- Unit tests focus on a single unit
 - they do not involve and test other systems, such as filesystem, network, databases, etc.
 - if the unit under test depends on other units, these other units (collaborators) should be replaced by test doubles
- Integration tests focus on the interplay of different units
 - they test how different units work together, for example a service method and a database
- Border between unit tests and integration tests is often fuzzy

How to Define Reasonable Test Cases?

- Equivalence partitioning
 - think about in which ways the input of a unit test is processed
 - each way represents a specific equivalence class
 - each equivalence class should be tested with a single representative
- Boundary analysis
 - faults often appear at the boundary between different equivalence classes
 - in addition to the chosen representative, these boundary values should be tested aswell

JUnit

<https://junit.org>



- JUnit is an open-source unit testing framework for Java
 - development started in 1995
 - Kent Beck and Erich Gamma developed a first version of JUnit on a flight from Zurich to Washington, D.C. by porting Kent's unit testing framework from Smalltalk to Java
 - according to Martin Fowler: "Never in the field of software development was so much owed by so many to so few lines of code."
 - nowadays JUnit is the de facto standard for unit testing in Java
- JUnit 3
 - test cases are defined by inheriting from base class TestCase
 - latest version was JUnit 3.8.2 released in 2007
- JUnit 4 (<https://github.com/junit-team/junit4>)
 - test cases are defined by annotations (Java 5 or later)
 - first version was released in 2006
 - today still heavily used in Java Enterprise domain
 - limited extensibility and modularity
- JUnit 5 (<https://github.com/junit-team/junit5>)
 - new architecture, execution, and extension model
 - developed to be forward and backward compatible
 - development was seeded with a crowdfunding campaign on Indiegogo in 2015 (€ 53.937 by 474 backers)
 - first version was released in 2017



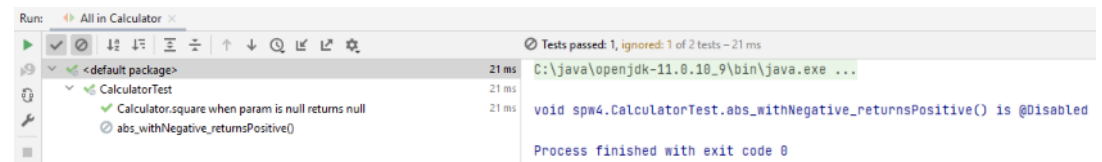
A First Unit Test

- Unit tests are regular methods and classes
- Test methods must be marked with the `@Test` annotation
- Additional annotations:
 - `@DisplayName` can be used to set a more readable name
 - `@Disabled` can be used to (temporarily ;-) deactivate test classes or test methods
 - `@Tag` can be used to tag and filter tests
- remember the AAA-Pattern

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest {
    @DisplayName("Calculator.square when param is null returns null")
    @Tag("unit")
    @Test
    void square_withNull_returnsNull() {
        Calculator calc = new Calculator(); // arrange
        var result = calc.square(0);        // act
        assertEquals(0, result);           // assert
    }

    @Disabled
    @Test
    void abs_withNegative_returnsPositive() {
        Calculator calc = new Calculator();
        var result = calc.abs(-3);
        assertEquals(3, result);
    }
}
```



Assertions

- Assertions are necessary to decide whether the test succeeded or failed
 - expected state is compared with the state of the unit under test
 - expected state comes from a so-called test oracle
 - in the simplest case the test oracle is the person who writes the unit tests and provides the expected state as a literal
 - in more complex cases creating the expected state might require additional logic
- JUnit's assertions are defined as static methods of the Assertions class
- Examples:
 - assertTrue, assertFalse, assertNotNull, assertEquals, assertSame, assertArrayEquals, assertThrows, ...
- Assertions can be grouped with assertAll
 - all assertions of a group are always executed
- Message can be provided for more descriptive errors
 - lazy evaluation of messages is also supported (Supplier<string> message)
- API reference:
<https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html>

Alternative Assertion Libraries

- Other assertion libraries can also be used
 - if more complex/specific assertions are necessary
 - in order to improve readability
- Some examples:
 - Hamcrest
 - provides different matchers
 - <http://hamcrest.org>
 - AssertJ
 - fluent API for assertions
 - <https://assertj.github.io/doc>
 - and others ...

```
assertThat(1.95, closeTo(2, 0.1));
assertThat("hello hamcrest", is(not(emptyString())));
assertThat("hello hamcrest", both(containsString("hello")).and(containsString("hamcrest")));
assertThat(intArray, arrayContainingInAnyOrder(3, 2, 1));
assertThat(persons, everyItem(hasProperty("age", is(greaterThan(40)))));
```

```
assertThat(1.95).isCloseTo(2, offset(0.1));
assertThat("hello assertj").isNotEmpty();
assertThat("hello assertj").contains("hello").contains("assertj");
assertThat(new Integer[] {1, 2, 3}).containsExactlyInAnyOrder(3, 2, 1);
assertThat(persons).allMatch(person -> person.getAge() > 40);
```

Assumptions

- Assumptions define preconditions for test cases
 - if an assumption fails, the test is skipped
 - a skipped test is not considered to be failed
- Assumptions are defined as static methods of the Assumptions class
- Examples:
 - `assumeTrue`, `assumeFalse`, `assumingThat`
- Message can be provided for more descriptive errors
 - lazy evaluation of messages is also supported (`Supplier<string> message`)
- API reference:
<https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assumptions.html>

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;
import static org.junit.jupiter.api.Assumptions.*;

class AssumptionsDemo {
    @Test
    void testOnCIServer() {
        assumeTrue("CI".equals(System.getenv("ENVIRONMENT")));
        // perform test only on CI/CD environment
        assertEquals(42, 40 + 2);
    }

    @Test
    void conditionalTest() {
        assumingThat("DEV".equals(System.getenv("ENVIRONMENT")), () -> {
            // check assertion only on local dev environment
            assertEquals(42, 40 + 2);
        });
        // always check remaining assertions
        assertEquals(42, 21 * 2);
    }
}
```

Conditional Test Execution

- Test methods can be enabled or disabled according to specific conditions
- Conditions are defined with annotations:
 - @EnabledOnOs
 - @EnabledOnJre, @EnabledForJreRange
 - @EnabledIfSystemProperty
 - @EnabledIfEnvironmentVariable
 - each annotation is also available in @Disabled... form
- Custom conditions can also be used with @EnabledIf
 - logic for a custom annotation has to be implemented in a method

```
import org.junit.jupiter.api.*;
import org.junit.jupiter.api.condition.*;
import static org.junit.jupiter.api.Assertions.*;
import static org.junit.jupiter.api.condition.JRE.*;
import static org.junit.jupiter.api.condition.OS.*;

class ConditionalTestDemo {
    @Test
    @DisabledOnOs(WINDOWS)
    void testA() { assertEquals(42, 21 * 2); }

    @Test
    @EnabledForJreRange(min = JAVA_9, max = JAVA_11)
    void testB() { assertEquals(42, 21 * 2); }

    @Test
    @EnabledIfEnvironmentVariable(named = "ENV", matches = ".*STAGING.*")
    void testC() { assertEquals(42, 21 * 2); }

    @Test
    @EnabledIf("customCondition")
    void testD() { assertEquals(42, 21 * 2); }

    boolean customCondition() {
        boolean enabled = true;
        // custom logic to decide whether
        // test should be enabled or not
        return enabled;
    }
}
```

Custom Annotations

- JUnit annotations can be used as meta-annotations
- Custom annotations can be defined to combine other annotations
- Useful to create a shortcut for
 - a specific set of tags
 - conditions
 - etc.

```
import java.lang.annotation.*;
import org.junit.jupiter.api.*;
import org.junit.jupiter.api.condition.*;
import static org.junit.jupiter.api.Assertions.*;
import static org.junit.jupiter.api.condition.OS.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Tag("unit")
@Tag("fast")
@EnabledOnOs(LINUX)
@Test
@interface UnitTestOnLinux { }

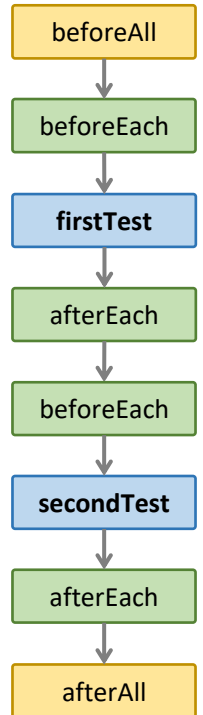
class CustomAnnotationDemo {
    @UnitTestOnLinux
    void unitTestA() {
        assertEquals(42, 15 + 27);
    }

    @UnitTestOnLinux
    void unitTestB() {
        assertEquals(42, 20 + 22);
    }
}
```

Setup and Teardown of Fixtures

- Setup and teardown of test fixtures can be done in specifically annotated methods
- Test class is instantiated for each test method
 - "per-method" test instance lifecycle
 - avoids side effects
 - instance variables of the test class can be used safely to pass data between setup, test execution and teardown
 - @BeforeAll and @AfterAll methods must be static
- Test instance lifecycle can be changed to "per-class"
 - by adding @TestInstance(Lifecycle.PER_CLASS) to the test class
 - @BeforeAll and @AfterAll then can be non-static methods
 - be aware that tests should not depend on each other and carefully avoid unwanted side effects

```
class SetupTeardownDemo {  
    @BeforeAll  
    static void beforeAll() {  
        System.out.println("beforeAll");  
    }  
  
    @BeforeEach  
    void beforeEach() {  
        System.out.println("  beforeEach");  
    }  
  
    @Test  
    void firstTest() {  
        System.out.println("    firstTest");  
        assertTrue(true);  
    }  
    @Test  
    void secondTest() {  
        System.out.println("    secondTest");  
        assertTrue(true);  
    }  
  
    @AfterEach  
    void afterEach() {  
        System.out.println("  afterEach");  
    }  
  
    @AfterAll  
    static void afterAll() {  
        System.out.println("afterAll");  
    }  
}
```



```
beforeAll  
beforeEach  
firstTest  
afterEach  
beforeEach  
secondTest  
afterEach  
afterAll
```

Test Execution Order

- Test methods are executed in a deterministic, but nonobvious order
- Unit tests should be isolated and independent of each other, therefore the order of execution does not matter
- However, order of execution might be important for integration or acceptance tests
- Order in which test methods are executed can be configured with the `@TestMethodOrder` annotation on the test class
- Built-in `MethodOrderer`:
 - `DisplayName`
 - `MethodName`
 - `OrderAnnotation`
 - `Random`

```
import org.junit.jupiter.api.*;
import org.junit.jupiter.api.MethodOrderer.*;
import static org.junit.jupiter.api.Assertions.*;

@TestMethodOrder(OrderAnnotation.class)
class ExecutionOrderDemo {
    @Test
    @Order(2)
    void testA() {
        System.out.println("testA");
        assertTrue(true);
    }

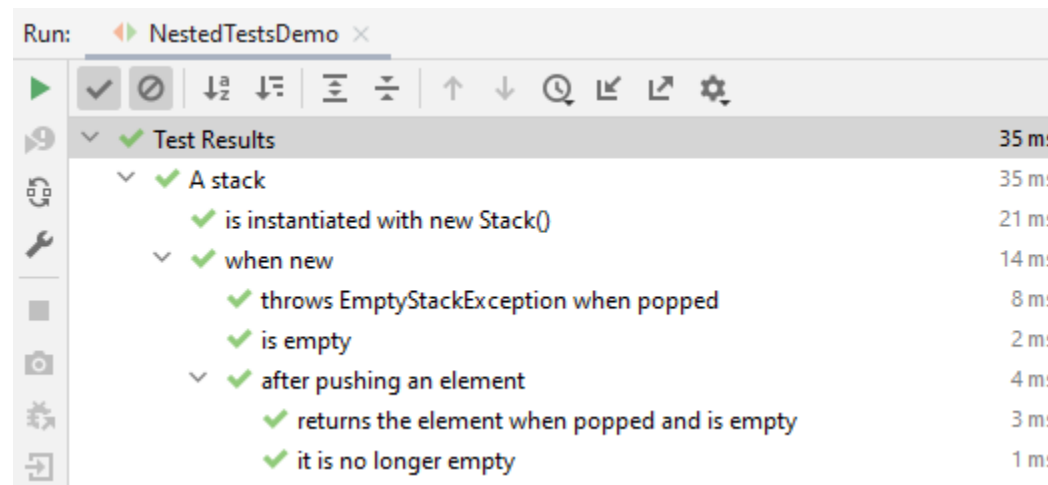
    @Test
    @Order(3)
    void testB() {
        System.out.println("testB");
        assertTrue(true);
    }

    @Test
    @Order(1)
    void testC() {
        System.out.println("testC");
        assertTrue(true);
    }
}
```

testC
testA
testB

Nested Tests

- Test classes can be nested which enables additional grouping of tests
- Nested tests are implemented as non-static nested classes and must have the `@Nested` annotation
- Setup and teardown methods of outer tests are always also executed for nested tests
- Hierarchical structure of nested tests is also reflected in the test runner



Run: NestedTestsDemo	
Test Results	35 ms
A stack	35 ms
is instantiated with new Stack()	21 ms
when new	14 ms
throws EmptyStackException when popped	8 ms
is empty	2 ms
after pushing an element	4 ms
returns the element when popped and is empty	3 ms
it is no longer empty	1 ms

```
@DisplayName("A stack")
class NestedTestDemo {
    Stack<String> stack;

    @Test
    @DisplayName("is instantiated with new Stack()")
    void isInstantiatedWithNew() {
        stack = new Stack<>();
    }

    @Nested
    @DisplayName("when new")
    class WhenNew {
        @BeforeEach
        void createNewStack() { stack = new Stack<>(); }

        @Test
        @DisplayName("is empty")
        void isEmpty() {
            assertTrue(stack.isEmpty());
        }

        @Test
        @DisplayName("throws EmptyStackException when popped")
        void throwsExceptionWhenPopped() {
            assertThrows(EmptyStackException.class, stack::pop);
        }

        @Nested
        @DisplayName("after pushing an element")
        class AfterPushing {
            String anElement = "an element";
            @BeforeEach
            void pushAnElement() { stack.push(anElement); }

            @Test
            @DisplayName("it is no longer empty")
            void isEmpty() {
                assertFalse(stack.isEmpty());
            }

            @Test
            @DisplayName("returns the element when popped and is empty")
            void returnElementWhenPopped() {
                assertEquals(anElement, stack.pop());
                assertTrue(stack.isEmpty());
            }
        }
    }
}
```

Parameterized Tests

- Parameterized tests enable to run a test multiple times with different arguments
- Arguments have to be provided by a source
 - @NullSource provides null for reference types
 - @EmptySource provides an empty string, list, set, array, etc.
 - @NullAndEmptySource is the combination of the previous two
 - @ValueSource provides single literal values
 - @EnumSource provides values of an enum type
 - @MethodSource defines a method which is called to provide the arguments
 - @CsvSource provides multiple arguments
 - @ArgumentsSource defines a class implementing ArgumentsProvider which is called
- @...Source annotations can be combined
- An individual display name for each execution can be defined
 - placeholders are available for the current arguments

```
import org.junit.jupiter.api.*;
import org.junit.jupiter.params.*;
import org.junit.jupiter.params.provider.*;
import static org.junit.jupiter.api.Assertions.*;

class ParameterizedTestDemo {
    @DisplayName("Calculator.add with two integers returns sum")
    @ParameterizedTest(name = "{0} + {1} = {2}")
    @CsvSource({"1, 2, 3", "2, 2, 4", "-1, 1, 0", "0, 0, 0",})
    void add(int x, int y, int expected) {
        assertEquals(expected, new Calculator().add(x, y));
    }

    @DisplayName("Formatter.capitalize with string returns capitalized string")
    @ParameterizedTest(name = "Input: {0}")
    @NullAndEmptySource
    @ValueSource(strings = {"abcd", "efgh"})
    void capitalize(String s) {
        if (s == null) {
            assertEquals(s, new Formatter().capitalize(s));
        } else {
            assertEquals(s.toUpperCase(), new Formatter().capitalize(s));
        }
    }
}
```

Run: ParameterizedTestDemo

Tests passed: 8 of 8 tests – 70 ms

C:\java\openjdk-11.0.10_9\bin\java.exe ...

Process finished with exit code 0

Test Results	70 ms
ParameterizedTestDemo	70 ms
Formatter.capitalize with string returns capitalized string	65 ms
Input: null	60 ms
Input:	1 ms
Input: abcd	1 ms
Input: efgh	3 ms
Calculator.add with two integers returns sum	5 ms
1 + 2 = 3	2 ms
2 + 2 = 4	1 ms
-1 + 1 = 0	1 ms
0 + 0 = 0	1 ms

Repeated Tests

- Tests can be repeated for a given number of times
 - for repeated tests the `@RepeatedTest` annotation has to be used instead of `@Test`
- Each repetition behaves exactly like a regular test execution
- An individual display name for repetitions can be defined
 - placeholders are available for current repetition, total repetitions, and original display name
- If the test method offers a parameter of type `RepetitionInfo`, the current repetition and total repetitions can also be used in the test

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

class RepeatedTestDemo {
    @DisplayName("It goes round and round ... ")
    @RepeatedTest(value = 10, name = "{currentRepetition} of {totalRepetitions}")
    void repeatedTest(RepetitionInfo repetitionInfo) {
        System.out.println("Repetition " + repetitionInfo.getCurrentRepetition());
        assertTrue(true);
    }
}
```

Run: RepeatedTestDemo

✓ Tests passed: 10 of 10 tests – 34 ms

C:\java\openjdk-11.0.10_9\bin\java.exe ...

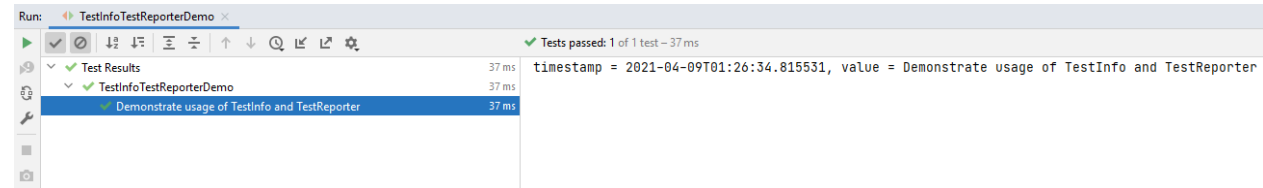
<default package>	34 ms	
Repetition 1	34 ms	
Repetition 2	34 ms	
Repetition 3	29 ms	
Repetition 4		
Repetition 5		
Repetition 6	1 ms	
Repetition 7		
Repetition 8	1 ms	
Repetition 9	1 ms	
Repetition 10	1 ms	
Process finished with exit code 0		

Metadata and Reporting

- TestInfo contains metadata about the current test, i.e.
 - display name
 - test class
 - test method
 - tags
- TestReporter enables to publish additional information about the current test
 - message and timestamp are shown in the test output
- TestInfo and TestReporter are both provided by parameter injection in constructors and methods

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

class TestInfoTestReporterDemo {
    @Test
    @DisplayName("Demonstrate usage of TestInfo and TestReporter")
    void test(TestInfo testInfo, TestReporter testReporter) {
        testReporter.publishEntry(testInfo.getDisplayName());
        assertEquals(42, 30 + 12);
    }
}
```



Interfaces and Default Methods

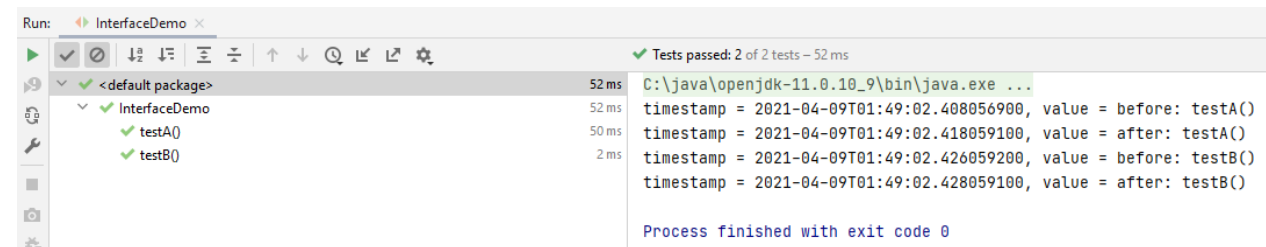
- Test annotations can also be applied on default methods in interfaces
- Common parts (e.g., setup and teardown logic) can be encapsulated
- `@BeforeAll` and `@AfterAll` can only be added to default methods, if per-class test instance lifecycle is used

```
interface TestTracer {
    @BeforeEach
    default void traceBefore(TestInfo testInfo, TestReporter testReporter) {
        testReporter.publishEntry("before: " + testInfo.getDisplayName());
    }

    @AfterEach
    default void traceAfter(TestInfo testInfo, TestReporter testReporter) {
        testReporter.publishEntry("after: " + testInfo.getDisplayName());
    }
}

class InterfaceDemo implements TestTracer {
    @Test
    void testA() {
        assertEquals(42, 15 + 27);
    }

    @Test
    void testB() {
        assertEquals(42, 20 + 22);
    }
}
```



```
Run: InterfaceDemo
52 ms C:\java\openjdk-11.0.10_9\bin\java.exe ...
52 ms timestamp = 2021-04-09T01:49:02.408056900, value = before: testA()
50 ms timestamp = 2021-04-09T01:49:02.418059100, value = after: testA()
2 ms timestamp = 2021-04-09T01:49:02.426059200, value = before: testB()
timestamp = 2021-04-09T01:49:02.428059100, value = after: testB()

Process finished with exit code 0
```

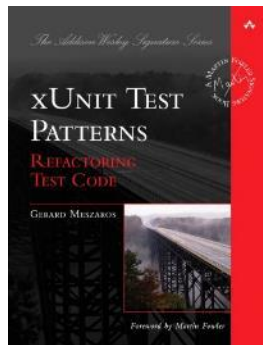
Literature



B. Garcia:
Mastering Software Testing with JUnit 5
Packt Publishing



A. Tarlinder:
Developer Testing: Building Quality into Software
Addison Wesley



G. Meszaros:
xUnit Test Patterns: Refactoring Test Code
Addison Wesley

SPW4

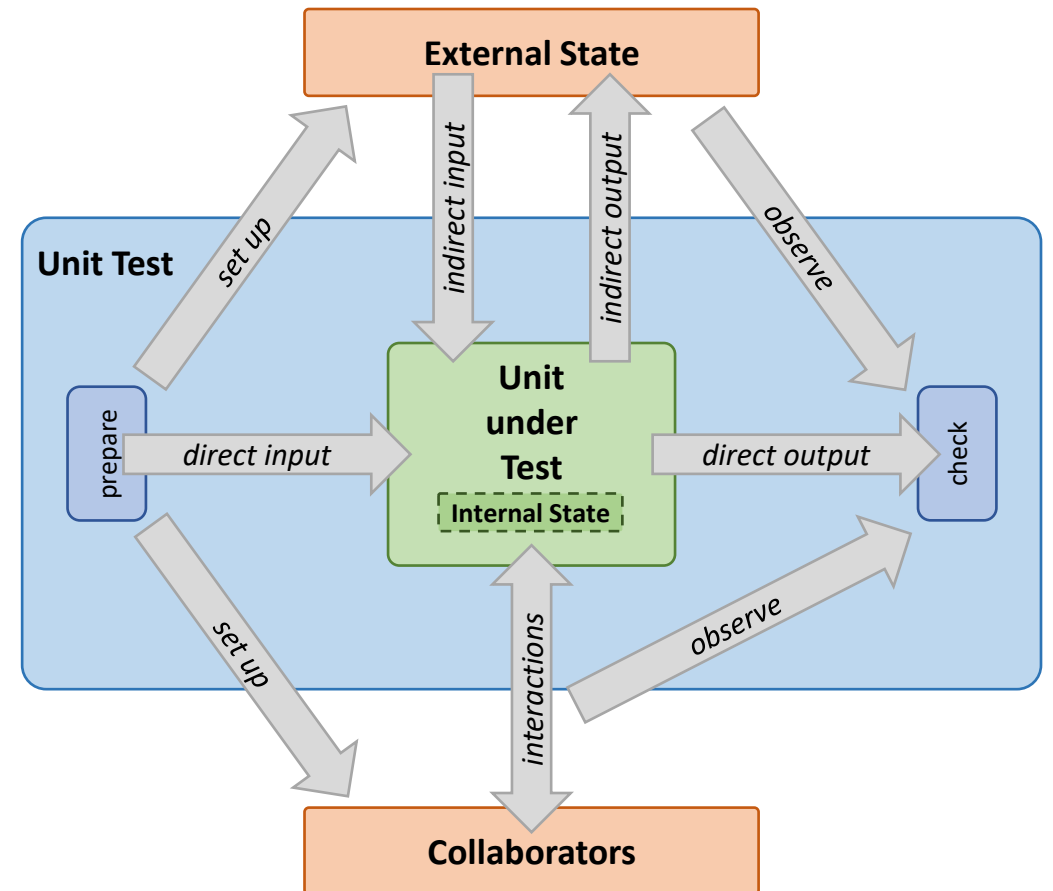
Test Automation & Continuous Delivery

Test Doubles

S. Wagner

Anatomy of a Unit Test

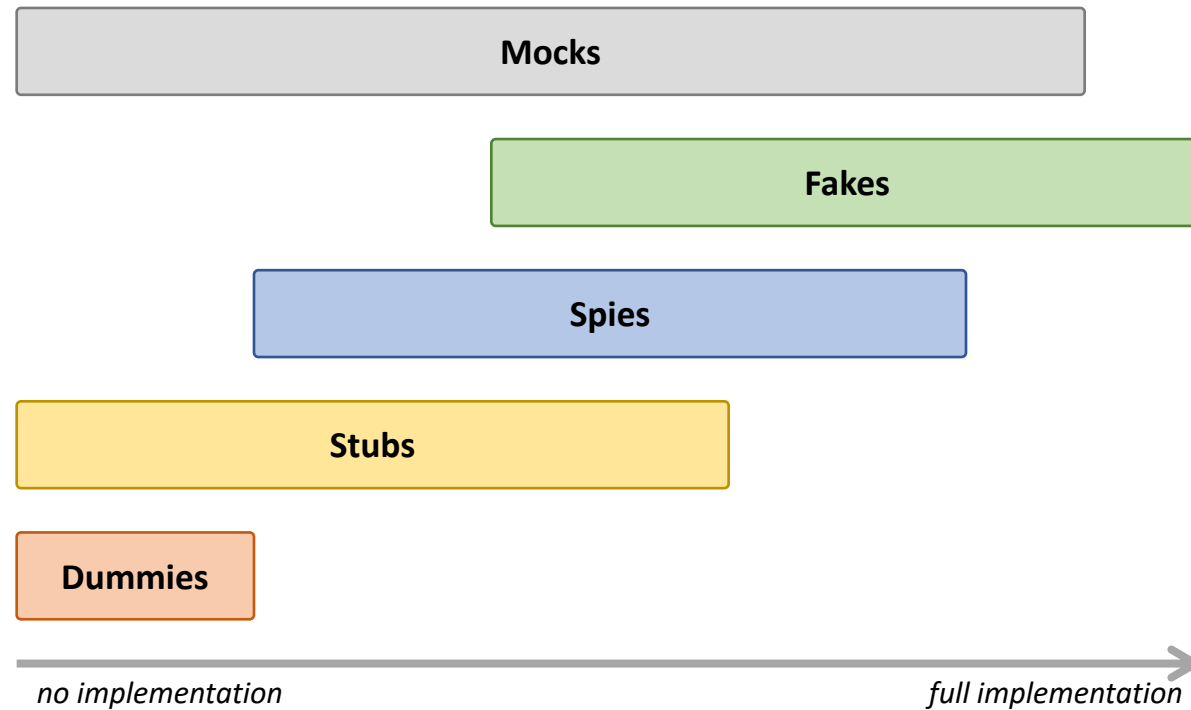
- Direct input and direct output must be prepared and checked
- Internal state must be initialized and checked
- External state used by the unit under test must be set up and observed
- External collaborators called by the unit under test must be set up
- Interactions with external collaborators must be observed
- AAA Pattern
 - Arrange
 - Act
 - Assert



Test Doubles

- Unit tests are executed in isolation
- Test doubles are needed to replace other units used by the unit under test
- Depending on their purpose, we differentiate between different kinds of test doubles:
 - **Dummy:** object which is required to fulfill an API, but which contains no implementation
 - **Stub:** object which returns hard-coded values
 - **Spy:** a stub which also tracks some interactions (partial mock)
 - **Fake:** object which replaces the original object with a simpler implementation (e.g., an in-memory database)
 - **Mock:** object which replaces the original object to track interactions
- Mocks are usually automatically generated using a mocking library
- Although you can create a mock that behaves like a dummy or a stub, be aware that mocks are intended to test behavior
- Extensive use of test doubles is an indicator for bad testability

Test Doubles



State Verification vs. Behavior Verification

- Unit test must check whether the test failed or not
- Verification can be done by checking states or by checking behavior
- State verification
 - expects a specific state after the unit under test was executed
 - uses assertions to compare current state and expected state
 - if they differ, the test fails
- Behavior verification
 - expects specific interactions between the unit under test and its collaborator
 - uses mocks or spies to track the interactions between the unit under test and its collaborator
 - if the expected interactions did not take place, the test fails
- State verification only checks the result, but not how this result has been achieved
- Behavior verification checks specific actions that lead to the desired result
- Behavior verification couples test code and production code more strongly (checks "how" and not "what")
- Behavior verification usually makes tests more brittle

Mockito

<https://site.mockito.org>



- Mockito is an open-source mocking framework for Java
 - development started in 2007 by Szczepan Faber
 - started off as a fork of EasyMock
 - first release in 2008
- Mockito does not enforce the expect-run-verify pattern
 - expectations do not have to be defined upfront
 - stubbing and verification are clearly separated
 - Mockito can verify anything after the mock object has been used

Mockito vs. EasyMock

Mockito

```
import org.junit.jupiter.api.*;
import static org.mockito.Mockito.*;

class MockitoTestDemo {
    @Test
    void testMockito() {
        List list = mock(List.class);

        when(list.get(0)).thenReturn("one");
        when(list.get(1)).thenReturn("two");

        // act ...

        verify(list).clear();
    }
}
```

EasyMock

```
import org.junit.jupiter.api.*;
import static org.easymock.classextension.EasyMock.*;

class EasyMockTestDemo {
    @Test
    void testEasyMock() {
        List list = createNiceMock(List.class);

        expect(list.get(0)).andReturn("one");
        expect(list.get(1)).andReturn("two");
        list.clear();

        replay(list);

        // act ...

        verify(list);
    }
}
```

A First Mock Object

- Working with mock objects in Mockito:
 1. create a mock object
 2. stub methods (if necessary)
 3. verify interactions (if necessary)
- remember the AAA-Pattern

```
public class PersonService {
    private PersonRepository repo;

    public PersonService(PersonRepository repo) {
        this.repo = repo;
    }

    public void register(String name, int age) {
        if (name.isBlank()) throw new Ill.Arg.Ex.("invalid name");
        if (age < 16) throw new Ill.Arg.Ex.("person too young");

        Person p = new Person(name, age);
        repo.createPerson(p);
    }

    public double getAverageAge() {
        return repo.readAllPersons()
            .collect(Collectors.averagingInt(p -> p.getAge()));
    }
}
```

```
@Test
void register_withValidNameAndAge_Succeeds() {
    PersonRepository rep = mock(PersonRepository.class);
    PersonService sut = new PersonService(rep);

    sut.register("Alice", 25);

    verify(rep).createPerson(any());
}

@Test
void register_withAgeTooYoung_throwsIllegalArgumentException() {
    PersonRepository rep = mock(PersonRepository.class);
    PersonService sut = new PersonService(rep);

    assertThrows(IllegalArgumentException.class, () -> sut.register("Bob", 12));

    verifyNoInteractions(rep);
}

@Test
void getAverageAge_whenPersonsRegistered_returnsAverageAge() {
    PersonRepository rep = mock(PersonRepository.class);
    when(rep.readAllPersons()).thenReturn(Stream.of(new Person("a", 35),
                                                    new Person("b", 28)));

    double expected = (35 + 28) / 2f;
    PersonService sut = new PersonService(rep);

    var result = sut.getAverageAge();

    assertEquals(expected, result);
}
```

Mockito JUnit 5 Extensions

- Mockito supports the JUnit 5 extension model
 - additional annotations can be used to mark mock objects
 - `@Mock` can be added to any instance variable or method parameter
- Additional dependency:

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-junit-jupiter</artifactId>
  <version>3.9.0</version>
</dependency>
```

```
@ExtendWith(MockitoExtension.class)
class MockitoJUnitExtensionDemo {

    @Mock
    PersonRepository rep;

    @Test
    void register_withValidNameAndAge_Succeeds() {
        PersonService sut = new PersonService(rep);

        sut.register("Alice", 25);

        verify(rep).createPerson(any());
    }

    @Test
    void register_withAgeTooYoung_throwsIllegalArgumentException() {
        PersonService sut = new PersonService(rep);

        assertThrows(IllegalArgumentException.class,
            () -> sut.register("Bob", 12));

        verifyNoInteractions(rep);
    }
}
```

Stubs with Fixed Parameters

- 2 Forms to stub methods:
 - when(...).thenReturn(...)
 - doReturn(...).when(...)
 - when(...) executes the stubbed method when it is defined
- Parameters can be defined explicitly
- Any method which has not been stubbed returns a default value (null, 0, empty array, etc.)

```
@Test
void stubWithFixedParameter(@Mock PersonRepository rep) {
    when(rep.readPersonById(1)).thenReturn(new Person("a"));
    doReturn(new Person("b")).when(rep).readPersonById(2);

    var personA = rep.readPersonById(1);
    var personB = rep.readPersonById(2);
    var personC = rep.readPersonById(3); // not stubbed -> returns null

    assertAll(
        () -> assertEquals("a", personA.getName()),
        () -> assertEquals("b", personB.getName()),
        () -> assertNull(personC)
    );
}
```


Stubs with any Parameters

- Argument matchers can be used for any parameters
 - anyInt(), anyDouble(), anyList(), any(MyClass.class), any(), ...
- thenReturn(...) can be called multiple times
 - at the end last value is returned every time

```
@Test
void stubWithAnyParameter(@Mock PersonRepository rep) {
    when(rep.readPersonById(anyInt())).thenReturn(new Person("a"))
    .thenReturn(new Person("b"));

    var personA = rep.readPersonById(0);
    var personB = rep.readPersonById(1);
    var personC = rep.readPersonById(2); // returns b

    assertEquals("a", personA.getName());
    assertEquals("b", personB.getName());
    assertEquals("b", personC.getName());
}
```

Multiple Matching Stubs

- If multiple stubs match, the latest stub is used
 - therefore, stubs can be overridden
 - if a default stub is defined in a setup method (@BeforeEach), it can be redefined in the test method
- Attention: strange side effects can occur, because when(...) executes the stubbed method

```
@Test
void multipleMatchingStubs(@Mock PersonRepository rep) {
    when(rep.readPersonById(anyInt())).thenReturn(new Person("x"));
    when(rep.readPersonById(anyInt())).thenReturn(new Person("a"))
        .thenReturn(new Person("b"));
    when(rep.readPersonById(3)).thenReturn(new Person("c"));

    var personA = rep.readPersonById(1); // returns b
    var personB = rep.readPersonById(2);
    var personC = rep.readPersonById(3);

    assertAll(
        () -> assertEquals("b", personA.getName()),
        () -> assertEquals("b", personB.getName()),
        () -> assertEquals("c", personC.getName())
    );
}
```

```
@Test
void multipleMatchingStubs(@Mock PersonRepository rep) {
    when(rep.readPersonById(anyInt())).thenReturn(new Person("x"));
    when(rep.readPersonById(anyInt())).thenReturn(new Person("a"))
        .thenReturn(new Person("b"));
    doReturn(new Person("c")).when(rep).readPersonById(3);

    var personA = rep.readPersonById(1); // returns a
    var personB = rep.readPersonById(2);
    var personC = rep.readPersonById(3);

    assertAll(
        () -> assertEquals("a", personA.getName()),
        () -> assertEquals("b", personB.getName()),
        () -> assertEquals("c", personC.getName())
    );
}
```

Stubs Throwing Exceptions

- Stubbed methods can also throw exceptions
 - if a void-method is stubbed, doThrow(...) has to be used

```
@Test
void stubThrowsException(@Mock PersonRepository rep) {
    when(rep.readPersonById(-1)).thenReturn(IllegalArgumentException.class);

    assertThrows(IllegalArgumentException.class,
        () -> rep.readPersonById(-1));
}
```

```
@Test
void voidStubThrowsException(@Mock PersonRepository rep) {
    doThrow(IllegalArgumentException.class).when(rep).createPerson(any());

    assertThrows(IllegalArgumentException.class,
        () -> rep.createPerson(new Person("a")));
}
```

Unnecessary Stubs

- Unnecessary stubs lead to exceptions
 - clean code = no unnecessary code
 - if a stub method is not used (i.e. called), an exception is thrown automatically
 - unnecessary stubs should be avoided
- Lenient stubs can be used, if these exceptions are not desired
 - be sure what you do, before you allow unnecessary stubs

```
@Test
void unnecessaryStubs(@Mock PersonRepository rep) {
    when(rep.readPersonById(0)).thenReturn(new Person("a"));
    when(rep.readPersonById(1)).thenReturn(new Person("b"));

    var personA = rep.readPersonById(0);

    assertEquals("a", personA.getName());
}
```

Tests failed: 1 of 1 test – 536 ms

C:\java\openjdk-11.0.10_9\bin\java.exe ...

org.mockito.exceptions.misusing.UnnecessaryStubbingException:
Unnecessary stubbings detected.
Clean & maintainable test code requires zero unnecessary code.
Following stubbings are unnecessary (click to navigate to relevant line of code):
1. -> at spw4.exploration.MockitoStubMethodsDemo.unnecessaryStubs([MockitoStubMethodsDemo.java:96](#))
Please remove unnecessary stubbings or use 'lenient' strictness. More info: javadoc for UnnecessaryStubbingException class.

```
@Test
void unnecessaryStubs(@Mock PersonRepository rep) {
    when(rep.readPersonById(0)).thenReturn(new Person("a"));
    lenient().when(rep.readPersonById(1)).thenReturn(new Person("b"));

    var personA = rep.readPersonById(0);

    assertEquals("a", personA.getName());
}
```

Stubs with Callbacks and without Verification

- Stubs can also call custom code
 - InvocationOnMock offers access to arguments, mock object, method, etc.
 - be careful that your stubs does not become too complicated
- Each stub can also be verified
 - if a real stub without verification is sufficient, stubOnly() can be defined in the MockSettings
 - other settings are defaultAnswer, extraInterfaces, etc.

```
@Test
void stubWithCallback(@Mock PersonRepository rep) {
    String[] names = { "a", "b", "c" };
    when(rep.readAllPersons()).then(new Answer() {
        public Object answer(InvocationOnMock invocation) {
            return Stream.of(new Person(names[0]),
                            new Person(names[1]),
                            new Person(names[2]));
        }
    });

    var result = rep.readAllPersons().map(p -> p.getName()).toArray();

    assertEquals(names, result);
}
```

```
@Test
void stubWithoutVerification() {
    PersonRepository rep = mock(PersonRepository.class,
                                withSettings().stubOnly());
    when(rep.readPersonById(anyInt())).thenReturn(new Person("a"));

    var personA = rep.readPersonById(0);

    assertEquals("a", personA.getName());
    // verify(rep).readPersonById(0); -> fails as mock is stubOnly()
}
```

Mocks vs. Spies

- Mocks

- new objects which do not contain any implementation
- stub methods can be added
- can also be created for interfaces

```
@Test
void mockDemo() {
    List<Integer> list = mock(List.class);
    list.add(1);
    var result = list.get(0);
    assertEquals(null, result);
    verify(list).get(0);
}
```

- Spies

- objects of existing classes which are extended with behavior tracking
- objects contain default implementation
- methods can be replaced by stubs
- can only be created for instantiable types

```
@Test
void spyDemo() {
    List<Integer> list = spy(new ArrayList<>());
    list.add(1);
    var result = list.get(0);
    assertEquals(1, result);
    verify(list).get(0);
}
```

Verify for Fixed or any Parameters

- Fixed parameter values or argument matchers can also be used for verification

```
@Test
void verifyForFixedParameter(@Mock PersonRepository rep) {
    when(rep.readPersonById(anyInt())).thenReturn(new Person("a"));

    var personA = rep.readPersonById(0);

    verify(rep).readPersonById(0);
}
```

```
@Test
void verifyForAnyParameter(@Mock PersonRepository rep) {
    when(rep.readPersonById(anyInt())).thenReturn(new Person("a"));

    var personA = rep.readPersonById(0);

    verify(rep).readPersonById(anyInt());
}
```

Verify by Number of Calls

- Verify how often an interaction has occurred
- `verifyNoMoreInteractions` checks if any additional interactions took place
- `verifyNoInteractions` assures that no interactions took place at all
- Be careful, do not oververify your tests!

```
@Test
void verifyByNumberOfCalls(@Mock PersonRepository rep) {
    when(rep.readPersonById(anyInt())).thenReturn(new Person("a"));

    var personA = rep.readPersonById(0);
    var personB = rep.readPersonById(1);

    verify(rep, times(2)).readPersonById(anyInt());
    verify(rep, atLeastOnce()).readPersonById(anyInt());
    verify(rep, atLeast(2)).readPersonById(anyInt());
    verify(rep, atMost(2)).readPersonById(anyInt());
    verify(rep, never()).readPersonById(2);
}
```

```
@Test
void verifyNoMoreInteractionsOccurred(@Mock PersonRepository rep) {
    when(rep.readPersonById(anyInt())).thenReturn(new Person("a"));

    var personA = rep.readPersonById(0);

    verify(rep).readPersonById(0);
    verifyNoMoreInteractions(rep);
}
```

```
@Test
void verifyNoInteractionsOccurred(@Mock PersonRepository rep) {
    verifyNoInteractions(rep);
}
```


Verify Order of Interactions

- Verify that interactions occur in a specific sequence
 - inOrder-Object has to be the same for each verify(...)

```
@Test
void verifyOrderOfInteractions(@Mock PersonRepository rep) {
    when(rep.readPersonById(anyInt())).thenReturn(new Person("a"));

    var personA = rep.readPersonById(3);
    var personB = rep.readPersonById(2);
    var personC = rep.readPersonById(1);

    InOrder inOrder = inOrder(rep);
    inOrder.verify(rep).readPersonById(3);
    inOrder.verify(rep).readPersonById(2);
    inOrder.verify(rep).readPersonById(1);
}
```

Verify with Argument Matchers or Captors

- Argument matchers can be used to process given parameter values
 - can also be combined with Hamcrest matchers
 - it is also possible to write custom matchers
- Argument captors can be used to retrieve parameter values
 - can then be checked by assertions

```
@Test
void verifyWithArgumentMatchers(@Mock List<String> list) {
    list.add("abcde");

    verify(list).add(or(contains("c"), endsWith("de")));
}
```

```
@Test
void verifyCapturedParameters(@Mock PersonRepository rep) {
    Person personA = new Person("a");

    rep.createPerson(personA);

    ArgumentCaptor<Person> captor = ArgumentCaptor.forClass(Person.class);
    verify(rep).createPerson(captor.capture());
    assertEquals(personA, captor.getValue());
}
```

Dangers and Pitfalls

- Oververification
 - each verification of a mock object couples the test to the internal implementation of the SUT
 - tests become brittle and sensitive to change
 - refactoring might break tests which should still be green
 - avoid too many and too precise verifications
- Mocks everywhere
 - creating mocks is easy, maybe too easy
 - if a real class exists that is sufficient for the current test, use it
 - do not mock types that you do not own
- Transitive mocking
 - be alarmed if a mock returns a mock
 - transitive verification of interactions usually is not reasonable

Test Driven Development

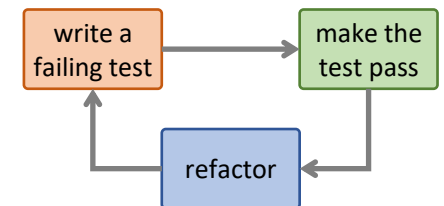
- 3 laws of TDD

You may not write production code until you have written a failing unit test.

You may not write more of a unit test than is sufficient to fail, and not compiling is failing.

You may not write more production code than is sufficient to pass the currently failing test.

- Tests and production code are developed together in a very tight loop
- Thinking about and writing tests first has some benefits
 - you must express first, which functionality and API you want to offer
 - your production code will be testable and therefore better designed
 - your production code will be fully covered by tests
- However, pure TDD is hard and a bit unrealistic, but the principle is valuable



Better Bad Tests than No Tests?

- Test suite has to evolve together with the system
- If tests are bad ...
 - bad tests make it harder to keep production code and test code in sync
 - tests consume more and more time
 - keeping up the test suite becomes unaffordable
 - test are silenced
 - finally test suite is abandoned
 - without tests, more defects arise
 - more defects lead to a fear of change
 - without change, code starts to rot
- Bad tests can be worse than no tests at all
- Test code deserves (and requires) the same level of care as production code
- Test code is not second-class code

Test Smells

Code Smells

- **Obscure tests**
 - test cannot be understood at a glance
- **Conditional test logic**
 - tests contain conditional code that may or may not be executed
- **Hard-to-Test Code**
 - testability of the unit under test is bad
- **Test Code Duplication**
 - same test code is repeated multiple times
- **Test Logic in Production**
 - production code contains routines which are only for testing

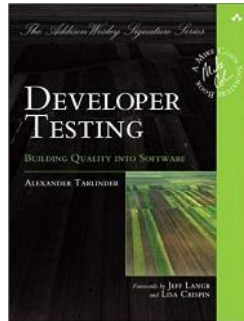
Behavior Smells

- **Assertion roulette**
 - it is hard to tell which assertion caused a test to fail
- **Erratic tests**
 - tests seem to pass/fail non-deterministically
- **Fragile tests**
 - tests fail although they should not
- **Frequent Debugging**
 - manual debugging is required to find out why a test failed
- **Manual intervention**
 - tests require manual action when they are run
- **Slow tests**
 - tests take too long to run

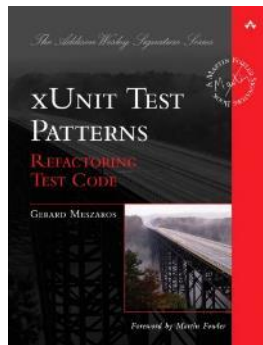
Literature



B. Garcia:
Mastering Software Testing with JUnit 5
Packt Publishing



A. Tarlinder:
Developer Testing: Building Quality into Software
Addison Wesley



G. Meszaros:
xUnit Test Patterns: Refactoring Test Code
Addison Wesley

SPW4

Test Automation & Continuous Delivery

Continuous Integration/Delivery/Deployment

S. Wagner & J. Karder

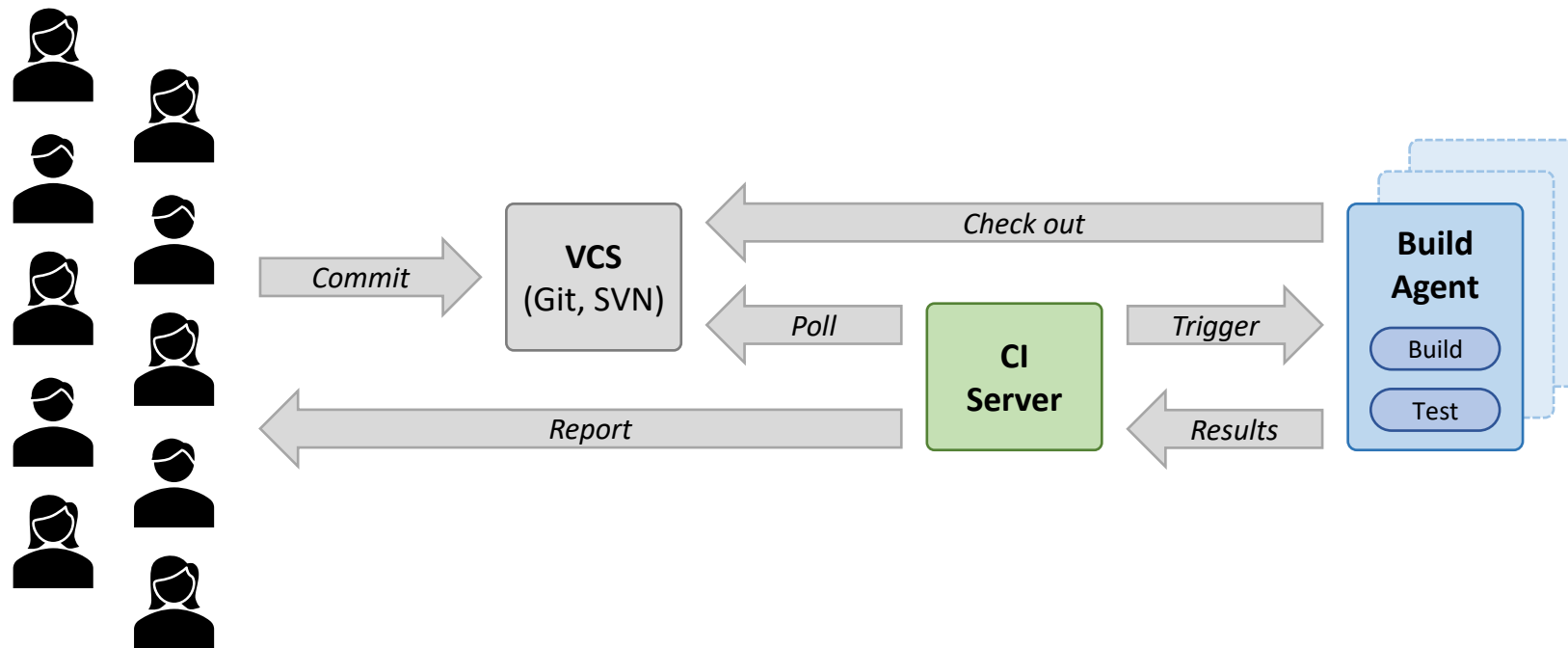
Why Test Automation is so Important

- Benefits of executing tests automatically:
 - you can be sure that the code in the repository does not break anything
 - if something breaks, you get immediate feedback
 - you do not have to execute tests manually and therefore you save time
 - you must keep your test suite up to date and preserve it from rotting
 - you even might get additional feedback (e.g., code coverage, code complexity, adherence to coding standards, performance)
 - you ensure that everything works in a well-defined environment
 - you (eventually) get rid of "It works on my machine."

Continuous Integration (CI)

- Continuous integration automates compilation and executes unit and maybe also integration tests
 - helps to identify problems early (e.g., build breaks, failing tests)
 - CI without automated tests is like a car without wheels
- Prerequisites of CI:
 - version control system (either centralized or decentralized; mostly Git in these days)
 - CI server (aka. build server)
 - optionally some client machines to compile and execute tests on them
 - executable description of the build and test process
 - channel to provide feedback (e.g., e-mail)
 - frequent integration of changes (i.e., commits/pushes to the VCS)

Continuous Integration (CI)



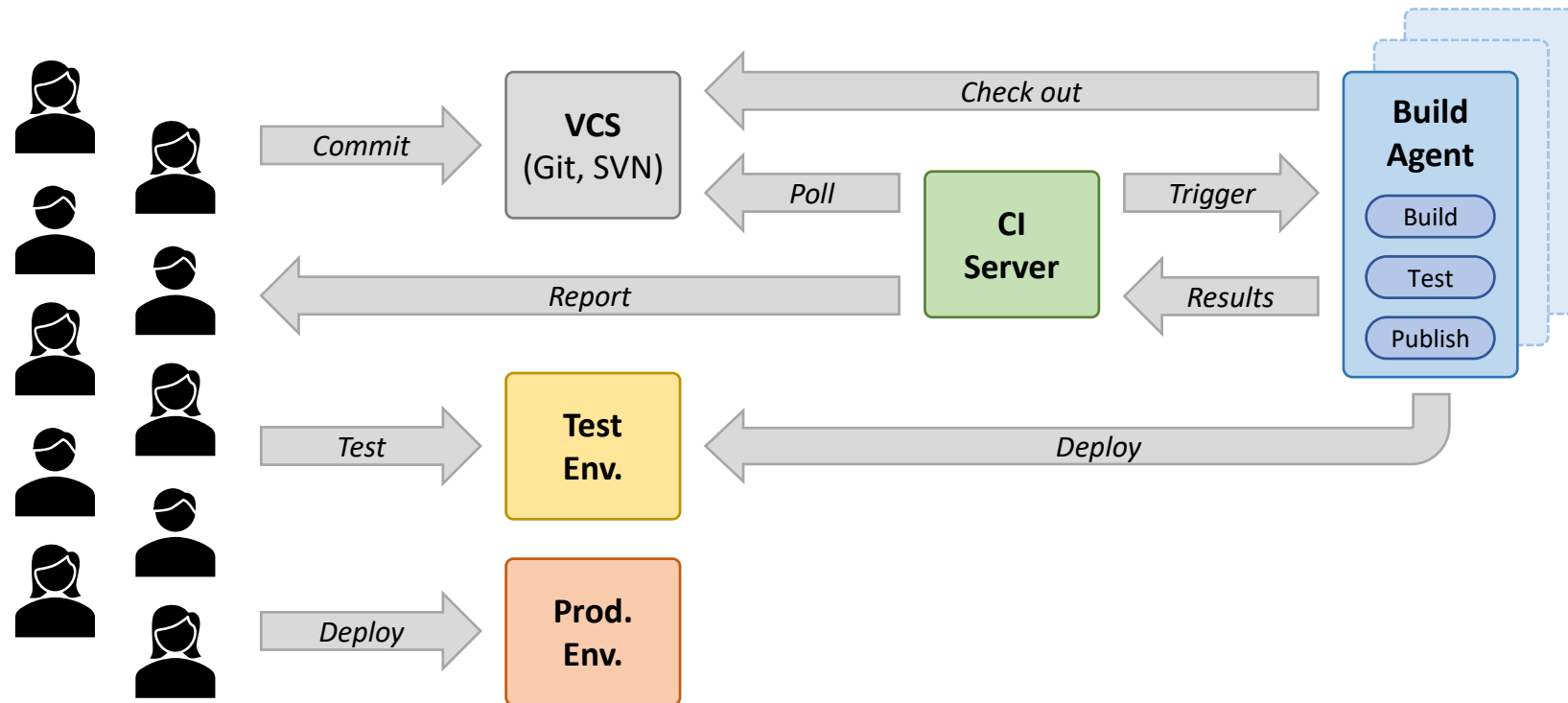
CI Pipelines

- Definition of a CI pipeline
 - consists of all steps which should be executed to create a build and execute tests
 - are executed regularly after each commit/push
 - structured in stages, at least *build* and *test*
 - CI pipelines are often extended by additional steps to check code quality (e.g., static code analyzers)
 - often build agents (i.e., dedicated machines) compile software and execute tests in order to spare resources on the CI server
- When to trigger your CI pipeline?
 - on demand? nightly? on each change?
 - as often as you can afford it – the more often the better
 - if executing tests takes too long, make your tests faster
 - or use different test groups (e.g., unit tests on each change, integration/performance tests nightly)
- Golden rule: "If your CI pipeline fails, it has to be fixed immediately."

Continuous Delivery (CD)

- Compiling the code and executing the tests is not sufficient
- Software should be tested in an environment which is technically very similar to the production environment
 - often there is a significant difference between production environments (multiple machines, security, load balancing, etc.) and development environments (single machine of a developer)
- Deployed versions can be used by testers, product owners, customers, etc. (cf. principles of agile software: deliver working software)
 - system tests can be done manually
- Continuous delivery extends the CI pipeline by this step
- Deployment to the production environment is also often automated, but still manually triggered

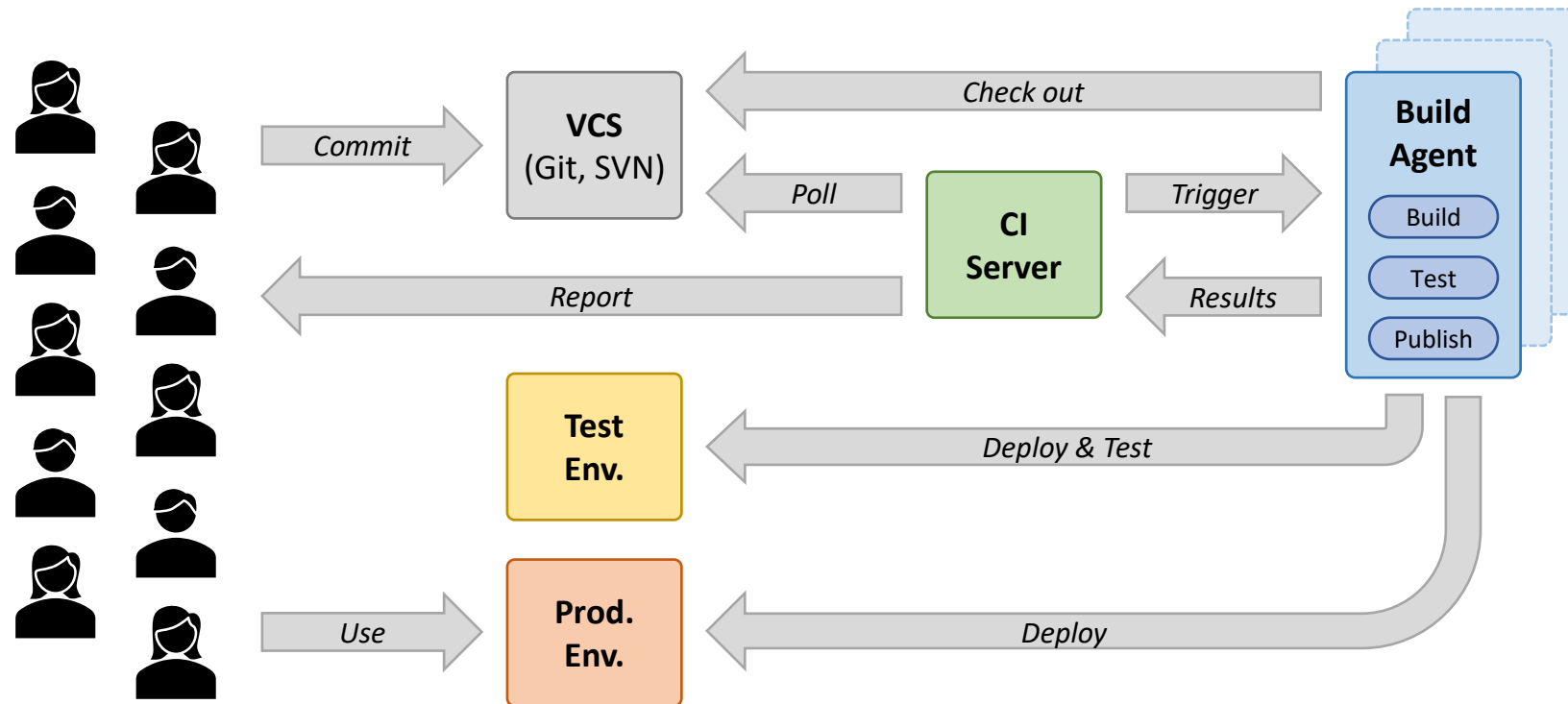
Continuous Delivery (CD)



Continuous Deployment

- System tests in a test environment are also fully automated
- Software is automatically deployed to the production environment, if no errors occur

Continuous Deployment



CI/CD Systems

- Many different tools and vendors
 - open-source, commercial, on premises, SaaS, ...
 - some examples: Jenkins, TeamCity, GitLab, GitHub Actions, Azure Pipelines, CircleCI, CloudBees CI, ...
- Core functionality is always the same
 - monitor VCS for changes
 - retrieve most recent version
 - prepare environment
 - execute pipeline (compile, test, inspect, deploy)
 - store build artifacts
 - report results

GitLab CI/CD

<https://gitlab.com> / <https://gitlab.fh-ooe.at>



- GitLab supports CI/CD
 - definition of CI/CD pipeline is stored in the repository (`.gitlab-ci.yml`)
 - each pipeline is executed on a runner
 - runners support different executors (e.g., docker, shell)
- Components of a GitLab CI/CD pipeline:
 - pipelines consist of stages and each stage contains one or more jobs
 - a stage defines when to run jobs, a job defines what to do
 - jobs are isolated from each other
 - all jobs in a stage can run in parallel, if there are enough runners
 - jobs can be restricted to specific runners
 - jobs can be triggered by or excluded from specific branches
 - jobs can also be triggered manually
 - if a single job fails, (usually) the whole pipeline fails
- Online documentation of GitLab's CI/CD features:
<https://docs.gitlab.com/ee/ci/README.html>

GitLab CI/CD

<https://gitlab.com> / <https://gitlab.fh-ooe.at>



- Types of CI/CD pipelines in GitLab:

- basic pipelines
 - execute one stage after another and run all jobs in each stage in parallel
- directed acyclic graph (DAG) pipelines
 - relationships between jobs can be defined as DAG
- multi-project pipelines
 - a pipeline in one project can trigger pipelines of other projects
- parent-child pipelines
 - pipelines can be hierarchically structured
- pipelines for merge requests
 - pipelines can be restricted to commits which belong to a merge request
- pipelines for merged results
 - pipelines can be executed on the result of a merge request before the merge request is accepted
- merge trains
 - pipelines for merged results can be queued one after the other

GitLab CI/CD

<https://gitlab.com> / <https://gitlab.fh-ooe.at>



- Simple example of a `.gitlab-ci.yml` file:

```
stages:                                # List of all stages
  - build
  - test

image: maven:3.8.1-jdk-11              # docker image, if docker executor is used
                                       # (can also be defined individually in each job)

build_project:                         # id of job
  stage: build
  tags:                                # List of tags, which a runner must have to execute this job
    - java
  script:                              # List of commands
    - mvn compile

test_project:
  stage: test
  tags:
    - java
  script:
    - mvn test
```

GitLab CI/CD – Stages



<https://gitlab.com> / <https://gitlab.fh-ooe.at>

stages

- defines the sequence of stages each pipeline is going through
- if missing, defaults to build, test, deploy
- two implicit stages `.pre` and `.post`
 - do not have to be defined
 - can be used to execute jobs before or after all other user-defined stages
 - order of `.pre` and `.post` cannot be changed

tags

- runners can be tagged with arbitrary labels
- job can be restricted to runners which have all specified tags
- per default, runners do not execute jobs without any tags (can be changed in runner settings)
- tags can be used to mark attributes of runners (e.g, OS, installed frameworks/services, ...)

GitLab CI/CD – Variables



<https://gitlab.com> / <https://gitlab.fh-ooe.at>

variables

- can be defined in the pipeline configuration
- can also be defined in the web frontend (Settings → CI/CD → Variables)
 - variables can be protected and/or masked
 - useful for secrets
- values can be used everywhere
- variables are exposed as environment variables
- can be set globally or per job

```
variables:  
  maven_packages_cache: ".m2/repository"  
  MAVEN_OPTS: "-Dmaven.repo.local=./$maven_packages_cache"
```

- GitLab also exposes many predefined variables:
 - e.g., \$CI_JOB_NAME, \$CI_COMMIT_REF_SLUG, \$CI_COMMIT_SHORT_SHA, ...
 - cf. https://docs.gitlab.com/ee/ci/variables/predefined_variables.html

Variables

Variables store information, like passwords and secret keys, that you can use in job scripts. [Learn more.](#)

Variables can be:

- **Protected:** Only exposed to protected branches or tags.
- **Masked:** Hidden in job logs. Must match masking requirements. [Learn more.](#)

Type	↑ Key	Value	Protected	Masked	Environments	
Variable	DB_CONNECTION_STRING	*****	✓	✗	test	
Variable	DB_CONNECTION_STRING	*****	✓	✗	production	
Variable	DB_PASSWORD	*****	✓	✓	test	
Variable	DB_PASSWORD	*****	✓	✓	production	

Collapse

GitLab CI/CD – Triggers



<https://gitlab.com> / <https://gitlab.fh-ooe.at>

only / except

- a pipeline is usually started after each push
- jobs can be restricted to or excluded from specific triggers
- only or except can be used to restrict to or excluded from:
 - only:refs or except:refs → branches/tags
 - only:variables or except:variables → values of variables
 - only:changes or except:changes → specific changes
- execution of pipelines can also be scheduled at specific times

```
analyze_coverage:
  stage: test
  only:
    refs:
      - main           # branch name
      - tags            # all tags
      - merge_requests # all MR
    changes:
      - **/*.java
  except:
    variables:
      - $CI_COMMIT_MESSAGE =~ /skip-coverage/
```

```
job1:
  stage: build
  only:
    refs:
      - main

job2:
  stage: build
  only:
    - main           # defaults to refs
                    # -> job1 and job2 are equivalent
```

GitLab CI/CD – Artifacts



<https://gitlab.com> / <https://gitlab.fh-ooe.at>

artifacts

- artifacts are results of a job which should be stored on the GitLab server (e.g., binaries, reports, ...)
- as jobs are isolated and independent of each other, artifacts can be used to pass results of one job to the next one (typical example build → test to reuse compiled binaries in test stage)
- by default, each jobs receives all artifacts from all previous jobs
- jobs can define dependencies to explicitly set which artifacts are required

```
build:
  stage: build
  artifacts:
    name: $CI_COMMIT_SHORT_SHA # filename of generated artifact
    expire_in: 1 week          # duration to store artifact on GitLab server
    paths:                     # list of paths to include in artifact
      - "target/*"
  script:
    - mvn compile

test:
  stage: test
  dependencies:                # list of artifacts to include from previous jobs
    - build
  script:
    - mvn test
```


GitLab CI/CD – Caches



<https://gitlab.com> / <https://gitlab.fh-ooe.at>

cache

- if the docker executor is used, each job is executed in a new container
- required dependencies are downloaded each time
- jobs can define paths which should be cached
- cached files are stored locally in the runner (default) or in a cloud-based cache (e.g., S3 buckets)
- `cache:key` can be used to create independent caches (e.g., **key: "\$CI_COMMIT_REF_SLUG"** → cache per branch/tag)
- `cache:policy` defaults to `pull-push` and can be set to `pull`, if cache is not changed

```
image: maven:3.8.1-jdk-11

variables:
  maven_packages_cache: ".m2/repository"
  MAVEN_OPTS: "-Dmaven.repo.local=.$maven_packages_cache"

build:
  stage: build
  cache:
    paths:                # list of paths to include in cache
      - $maven_packages_cache
  script:
    - mvn compile
```

GitLab CI/CD – Environments



<https://gitlab.com> / <https://gitlab.fh-ooe.at>

environment

- jobs can create environments to test, review, or operate an application
- environments can be accessed in the web frontend (Operations → Environments)
- specific jobs can be defined to stop a deployed environment again
- environments can be created manually or automatically

```
deploy_test_environment:
  stage: deploy
  dependencies:
    - build
  environment:
    name: test
    action: start
    url: https://my.test.environment.com # URL of the deployed environment
    on_stop: "stop_test_environment"
  when: manual # deployment is triggered manually
  script:
    - ... # commands to deploy test environment
```

```
stop_test_environment:
  stage: deploy
  environment:
    name: test
    action: stop
  script:
    - ... # commands to stop test environment
```

Available 2		Stopped 0				Enable review app		New environment		
Environment	Deployment	Job	Commit	Updated	Upcoming	Auto stop in				
test	#116 by	start test env #55730	dev ↪ 685f90c5 Code Cleanup	37 minutes ago						
production	#107 by	start production env #5...	master ↪ b0d02e96 Merge branch 'dev' into...	2 weeks ago						

GitHub Actions



<https://github.com/features/actions>

- GitHub offers a VM-based CI/CD platform
 - definition of CI/CD pipeline is stored in the repository (`.github/workflows/my_workflow.yml`)
 - available operating systems of runners are Windows Server, Ubuntu Linux, macOS
 - each pipeline is executed on a fresh Azure VM (Dv2- and DSv2-series)
 - 2 cores, 7GB memory, 14GB SSD disk space
 - macOS machines are hosted directly by GitHub
 - 3 cores, 14GB memory, 14GB SSD disk space
 - maximum runtime of a job on a GitHub hosted runner is 6 hours
 - runners can also be self-hosted
- Actions are building blocks of pipelines
 - many actions available which are developed and maintained by the community
 - execution of shell commands is also supported
- Online documentation of GitHub Actions:
<https://docs.github.com/en/actions>

GitHub Actions

<https://github.com/features/actions>



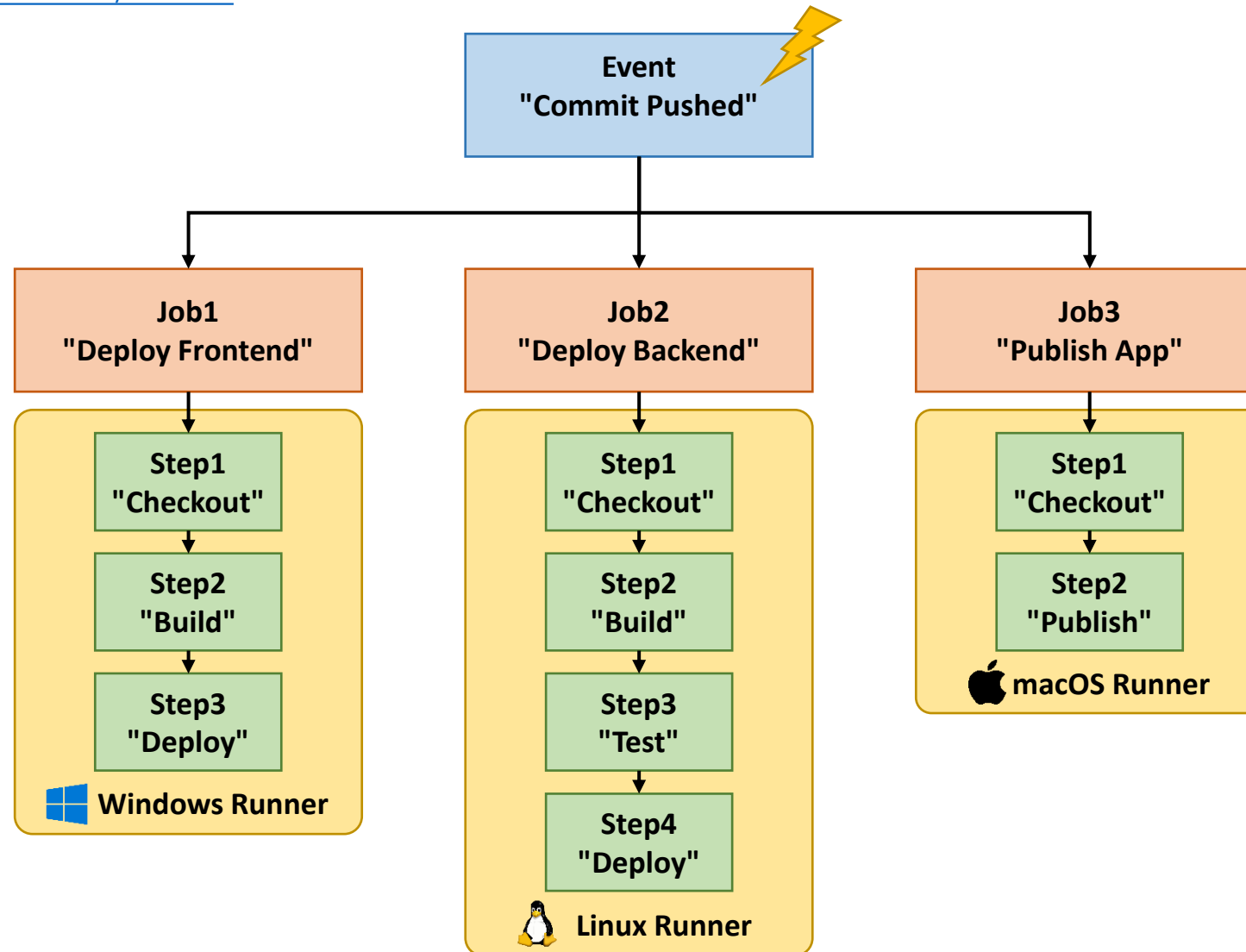
- Event
 - activity that triggers a workflow
 - e.g., push to repository, create a PR, ...
- Workflow
 - a number of jobs
 - scheduled or triggered by an event
 - jobs executed in parallel per default
- Job
 - a series of steps
 - executed by a runner
- Step
 - action or shell command
 - all steps within a job executed by the same runner
- Action
 - standalone commands
 - lots of community actions
 - e.g., checkout repository, coverage tests, installation of software, ...
- Runner
 - client that executes a job
 - self-hosted or hosted by GitHub
 - Windows, Linux, macOS images

GitHub Actions

<https://github.com/features/actions>



GitHub Actions



GitHub Actions



GitHub Actions

<https://github.com/features/actions>

- Configuration is stored in the repository
 - actions are defined in a YAML file (*.yaml)
 - must be committed to .github/workflows
- Simple example: learn-github-actions.yml

```
name: learn-github-actions      # name of the workflow, optional
on: [push]                     # event that triggers the workflow
jobs:
  check-bats-version:          # id of the first job
    runs-on: ubuntu-latest     # specifies the runner to use
                                # > provisions fresh container hosted by github
    steps:
      - uses: actions/checkout@v2 # step 1: executes community action "checkout@v2"
                                    # > checks out repository
      - uses: actions/setup-node@v1 # step 2: executes community action "setup-node@v1"
                                    # > sets up node environment, provides npm command
      - run: npm install -g bats   # step 3: shell command
                                    # > installs node.js package "bats"
      - run: bats -v              # step 4: shell command
                                    # > executes "bats -v"
```

GitHub Actions – Triggers



GitHub Actions

<https://github.com/features/actions>

on

- single GitHub event or list of events that trigger the workflow

```
on: [push, pull_request] # events that trigger the workflow
```

- further trigger behavior specification possible

```
on:
  # triggers workflow when pushing to branch 'main' or tags 'v1.0' .. 'v1.9'
  # triggers workflow when a PR for branch 'main' is created
  push:
    branches:
      - main
    tags:
      - v1.* # wildcard support
  pull_request:
    branches:
      - main
```

GitHub Actions – Triggers



GitHub Actions

<https://github.com/features/actions>

- workflows can be scheduled using the schedule event

```
on:
  schedule:
    # POSIX cron syntax
    # when using *, string needs to be quoted
    - cron: '30 5,17 * * *' # triggers at 05:30 and 17:30
```

- workflows can also be triggered manually (on GitHub actions page)

```
on: workflow_dispatch
```


GitHub Actions – Jobs



<https://github.com/features/actions>

jobs

- lists the jobs that define the workflow

```
jobs:
  my_first_job:      # id of first job
    name: My first job
  my_second_job:     # id of second job
    name: My second job
```

- all jobs within a workflow are executed in parallel per default
- a job is always executed by a specific runner

GitHub Actions – Jobs



<https://github.com/features/actions>

jobs.<job_id>.runs-on

- specifies the runner that should execute the job

```
runs-on: ubuntu-latest
```

- GitHub provides various runners to choose from
 - windows-latest → Windows Server 2019
 - ubuntu-latest → Ubuntu 20.04
 - macOS-latest → macOS Catalina 10.15
 - ...
- self-hosted runners can also be used
 - runners can be added in the repository settings

```
runs-on: [self-hosted, linux]
```

GitHub Actions – Jobs



<https://github.com/features/actions>

`jobs.<job_id>.needs`

- allows to specify which jobs need to complete first

```
jobs:
  job1:
  job2: # will be executed when 'job1' succeeds
    needs: job1
  job3: # will be executed when 'job1' and 'job2' succeed
    needs: [job1, job2]
```

- conditional expressions can be used to determine job execution

```
jobs:
  job1:
  job2:
    needs: job1
  job3:
    # will always be executed after 'job1' and 'job2',
    # regardless of whether or not they succeed
    if: always()
    needs: [job1, job2]
```

GitHub Actions – Steps

<https://github.com/features/actions>



GitHub Actions

jobs.<job_id>.steps

- specifies the steps of a job

```
jobs:  
  job:  
    steps:  
      - name: Step 1  
      - name: Step 2
```

- used to execute shell commands or run actions

GitHub Actions – Steps

<https://github.com/features/actions>



GitHub Actions

`jobs.<job_id>.steps[*].run`

- executes a shell command

```
jobs:
  job:
    steps:
      - name: Build Project
        run: mvn compile
```

```
jobs:
  job:
    steps:
      - name: Step 1
        run: | # multi-line command
              echo "GitHub"
              echo "Actions"
```

- working directory can also be set

```
jobs:
  job:
    steps:
      - name: Build Project
        run: mvn compile
        working-directory: ./src
```

GitHub Actions – Actions



GitHub Actions

<https://github.com/features/actions>

`jobs.<job_id>.steps[*].uses`

- executes a GitHub action (i.e., a reusable unit of code)

```
steps:  
  - uses: actions/checkout@v2
```

- many community actions available
- specific version to be used can be defined via
 - tag: actions/foo-bar@v1.0.0
 - branch name: actions/foo-bar@main
 - commit SHA: actions/foo-bar@06911ba...

GitHub Actions – Actions

<https://github.com/features/actions>



GitHub Actions

`jobs.<job_id>.steps[*].with`

- specifies input parameters defined by an action

```
jobs:
  job:
    steps:
      - name: Setup Java
        uses: actions/setup-java@v2
        with:
          distribution: 'adopt'
          java-version: 11
```

GitHub Actions – Strategies



GitHub Actions

<https://github.com/features/actions>

`jobs.<job_id>.strategy`

- used to create different job variations using a build matrix

`jobs.<job_id>.strategy.matrix`

- defines a matrix for various job configurations
- a maximum of $2^8 = 256$ variations can be created per workflow run

GitHub Actions – Strategies



GitHub Actions

<https://github.com/features/actions>

- this example will create 6 jobs which might all run in parallel

```
jobs:
  build:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-18.04, ubuntu-20.04]
        java: [8, 11, 15]
    name: build on ${{ matrix.os }} using java ${{ matrix.java }}
    steps:
      - uses: actions/checkout@v2
      - name: Setup Java
        uses: actions/setup-java@v2
        with:
          java-version: ${{ matrix.java }}
      - run: mvn compile
```

jobs.<job_id>.strategy.max-parallel

- limits parallel execution

```
strategy:
  max-parallel: 2
```

GitHub Actions – Env. Vars



<https://github.com/features/actions>

env

- a map of environmental variables available to all jobs within the workflow

```
env:  
  VARIABLE1: value1  
  VARIABLE2: value2
```

- can also be specified only for
 - jobs: `jobs.<job_id>.env`
 - steps: `jobs.<job_id>.steps[*].env`

```
jobs:  
  job:  
    env:  
      FOO: bar  
    steps:  
      - name: Step 1  
        env:  
          TEST: "GitHub Actions"  
        run: |  
          echo "FOO = $FOO"  
          echo "TEST = $TEST"
```

GitHub Actions – Env. Vars



GitHub Actions

<https://github.com/features/actions>

- do **not** store secrets in workflow configuration files!
- use GitHub secrets to set values of environmental variables

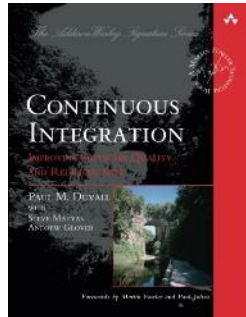
```
jobs:  
  job1:  
    env:  
      SUPER_SECRET: ${ secrets.SUPER_SECRET }
```

- secrets can be created in the repository settings

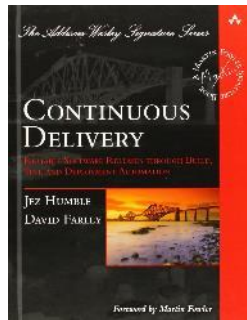
Automatically Provide IT Infrastructure

- Continuous delivery/deployment requires additional tools to provide test or even production environments automatically
- Configuration management (CM) tools manage and automate the configuration of IT infrastructure (e.g., virtual machines, databases, containers)
 - examples for CM tools: Puppet, Chef, Ansible
- Infrastructure as Code (IaC)
 - required infrastructure is defined in configuration files which are managed like source code (e.g., version controlled)
 - foundation for automating infrastructure provisioning

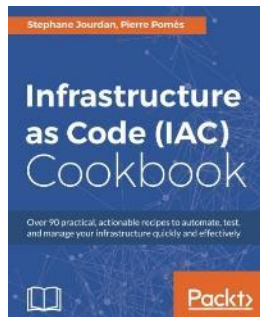
Literature



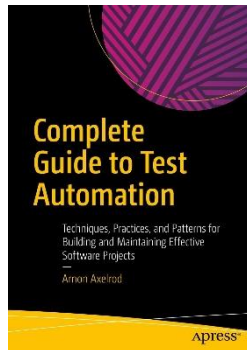
P. Duvall:
Continuous Integration: Improving Software Quality and Reducing Risk
Addison Wesley



J. Humble, D. Farley:
Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation
Addison Wesley



S. Jourdan, P. Pomes:
Infrastructure as Code (IAC) Cookbook
Packt Publishing



A. Axelrod:
Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects
Apress

SPW4

Test Automation & Continuous Delivery

Static Code Analysis

S. Wagner

Checking Code Quality by Inspection

- Code Reviews
 - fundamental principle in agile development
 - very effective for improving code quality
 - improve communication and skills in a team
 - suffer from "human factors" (subjectivity, emotions, etc.)
 - bind many resources and are very expensive
- Static Code Analysis
 - automatically and repeatedly check code by a set of rules (continuous inspection)
 - benefit from findings and best practices of many others
 - rules can be customized to specific needs and processes
 - results are objective and do not have any "human factors"
 - can be integrated into the CI/CD pipeline
 - does not require human resources and is very cheap

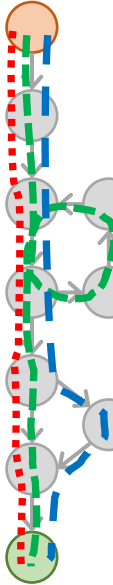
Testing vs. Inspection

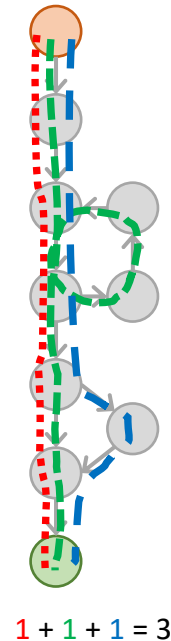
- Testing
 - software is executed to check its functionality
 - used to find failures, i.e., inability of the software to fulfill its requirements
 - tests require more effort, as they have to be written/defined
- Inspection
 - code is analyzed by a set of rules
 - software metrics are calculated (e.g., complexity metrics)
 - helps to identify problems before they turn into failures
 - predefined rules can be used right away
 - rules can cover many different aspects of code quality (e.g., formatting, anti-patterns, naming, complexity, duplication)

Software Metrics

- Software metrics describe measurable properties of software
 - extend measurement of software quality
 - indicate units which might be problematic
 - visualize quality changes over time
- Metrics can be categorized into several topics:
 - size metrics (SLOC, number of classes, number of methods, etc.)
 - complexity metrics (cyclomatic complexity, cognitive complexity, etc.)
 - test metrics (number of tests, test coverage, etc.)
 - duplications (duplicated lines of code, etc.)

Code Complexity

- Correlation of code complexity and defects
 - long methods (i.e., many lines of code and many different paths) are much more likely to contain defects
 - code complexity and defect rate are directly proportional
 - Cyclomatic Complexity
 - number of distinct paths through a method
 - cyclomatic complexity > 10 is considered harmful
 - cyclomatic complexity should correlate with the number of test cases
 - How to deal with high cyclomatic complexity?
 - write tests
 - refactor by extracting methods
 - but before you start refactoring, write tests
- 



Code Coupling

- Objects with a lot of dependencies are brittle
 - defects can arise in the object, if one of its dependencies change
 - a change in the object can lead to defects in its dependencies (side effects)
- Afferent Coupling (Fan In)
 - number of relationships to the object
 - high afferent coupling means that many other objects depend on that object
 - objects with high afferent coupling should have many unit tests
- Efferent Coupling (Fan Out)
 - number of relationships from the object
 - high efferent coupling means that this objects uses many other objects
- Instability
 - $\text{Instability} = \text{Efferent Coupling} / (\text{Efferent Coupling} + \text{Afferent Coupling})$
 - indicates an object's resilience to change (0 = stable, 1 = instable)

		Efferent Coupling (Fan Out)							
		10	20	30	40	50	60	70	80
Afferent Coupling (Fan In)	10	0,50	0,67	0,75	0,80	0,83	0,86	0,88	0,89
	20	0,33	0,50	0,60	0,67	0,71	0,75	0,78	0,80
	30	0,25	0,40	0,50	0,57	0,63	0,67	0,70	0,73
	40	0,20	0,33	0,43	0,50	0,56	0,60	0,64	0,67
	50	0,17	0,29	0,38	0,44	0,50	0,55	0,58	0,62
	60	0,14	0,25	0,33	0,40	0,45	0,50	0,54	0,57
	70	0,13	0,22	0,30	0,36	0,42	0,46	0,50	0,53
	80	0,11	0,20	0,27	0,33	0,38	0,43	0,47	0,50

Code Duplication

- Copy & Paste Programming
 - code blocks are simply copied without thinking about better ways to reuse or to generalize
 - is often not even noticed
- Problems of duplicated code
 - higher maintenance costs
 - increased testing effort
- Static code analysis can easily identify duplicated code

Code Coverage

- Code coverage indicates how much code is executed by automated tests
 - different levels of granularity
 - class coverage, method coverage, line (or statement) coverage, branch (or path) coverage
- Coverage data is retrieved by sampling or instrumenting code while tests are executed
 - can have a significant impact on the performance of test execution
 - coverage reports are usually not needed as immediate feedback, as they show trends in the long run
 - analyzing coverage might be postponed to nightly builds, if performance is an issue

SonarQube

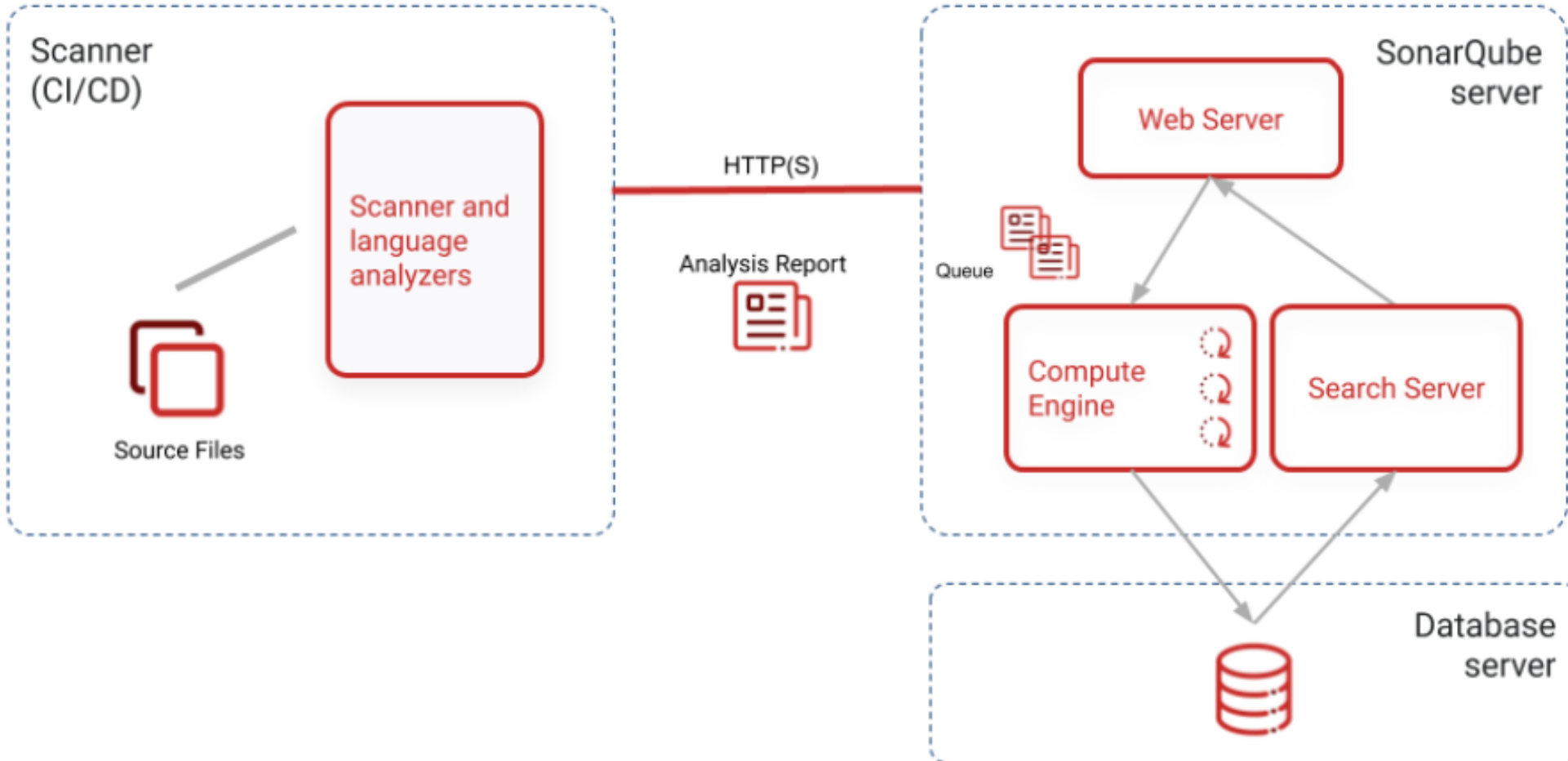
<https://sonarqube.org>



- SonarQube is a web-based open-source tool for continuously inspecting code quality
 - supports 25+ programming languages (e.g., Java, C#, C/C++, JavaScript, TypeScript, Python, Go, PHP, etc.)
 - detects bugs, code smells, security vulnerabilities
 - provides reports on test coverage and code duplication
- SonarQube is maintained by SonarSource
 - development started in 2006
 - company SonarSource was founded in 2008
 - located in Switzerland
 - received \$45 million USD venture capital in 2016

SonarQube – Architecture

<https://sonarqube.org>



<https://docs.sonarqube.org/latest/setup/install-server>

SonarQube – Test Coverage



<https://sonarqube.org>

- SonarQube does not execute tests itself
- Separate coverage tool is required to provide coverage data
 - e.g., Java Code Coverage Library (JaCoCo), <https://www.jacoco.org/jacoco>
 - JaCoCo provides a Maven plugin to integrate coverage analysis into the build process
 - coverage data is written to target/site/jacoco in XML format (required for SonarQube) and in human-readable HTML format

TSPSolver > spw4.tsp

spw4.tsp

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
TSP	<div><div></div></div>	0 %	<div><div></div></div>	0 %	10	10	28	28	6	6	1	1
TSPSolver	<div><div></div></div>	0 %	n/a	n/a	3	3	20	20	3	3	1	1
TSPLibParser	<div><div></div></div>	81 %	<div><div></div></div>	82 %	12	54	23	138	1	14	0	1
GA	<div><div></div></div>	100 %	<div><div></div></div>	100 %	0	25	0	56	0	11	0	1
Permutation	<div><div></div></div>	100 %	<div><div></div></div>	100 %	0	16	0	26	0	6	0	1
OrderCrossover	<div><div></div></div>	100 %	<div><div></div></div>	100 %	0	8	0	21	0	2	0	1
CyclicCrossover	<div><div></div></div>	100 %	<div><div></div></div>	100 %	0	8	0	20	0	2	0	1
TournamentSelector	<div><div></div></div>	100 %	<div><div></div></div>	100 %	0	7	0	11	0	2	0	1
Solution	<div><div></div></div>	100 %	<div><div></div></div>	100 %	0	9	0	14	0	5	0	1
InversionMutator	<div><div></div></div>	100 %	<div><div></div></div>	100 %	0	4	0	9	0	2	0	1
RandomSelector	<div><div></div></div>	100 %	<div><div></div></div>	100 %	0	6	0	7	0	2	0	1
TSPLibFormatException	<div><div></div></div>	100 %	n/a	n/a	0	1	0	2	0	1	0	1
Total	368 of 1 620	77 %	21 of 185	88 %	25	151	71	352	10	56	2	12

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.7</version>
  <executions>
    <execution>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>report</id>
      <phase>test</phase>
      <goals>
        <goal>report</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```


SonarQube – Pipeline Integration



<https://sonarqube.org>

- Integration of SonarQube into the build process is very easy
 - program SonarScanner is executed during the build, which reports analysis results to the SonarQube server
 - several build automation tools are directly supported (e.g., Maven, Ant, Gradle)
- SonarQube provides a Maven goal:

```
mvn sonar:sonar -Dsonar.host.url=$sonar_url  
                -Dsonar.projectKey=$sonar_project  
                -Dsonar.login=$sonar_login  
                -Dsonar.qualitygate.wait=true
```

- `sonar.host.url` = URL of the SonarQube server
- `sonar.projectKey` = project ID
- `sonar.login` = access token generated for the project
- `sonar.qualitygate.wait` = true, if the build should fail when a quality gate is not passed

SonarQube – Rules & Quality Profiles



<https://sonarqube.org>

- Rules
 - static code analysis of SonarQube is based on rules
 - e.g., 639 rules for Java in SonarQube 8.9.1
 - 4 categories: bugs, vulnerabilities, code smells, security hotspots
 - rules can be extended by plugins from the Sonar Marketplace
 - rules from other analyzers can be integrated (e.g., Checkstyle, Findbugs)
 - custom rules can also be integrated as plugins
- Quality Profiles
 - rules can be activated or deactivated as needed
 - quality profile represents the set of active rules and is defined per language
 - quality profiles can be set individually for each project
 - arbitrary many custom quality profiles can be defined

SonarQube – Quality Gates



<https://sonarqube.org>

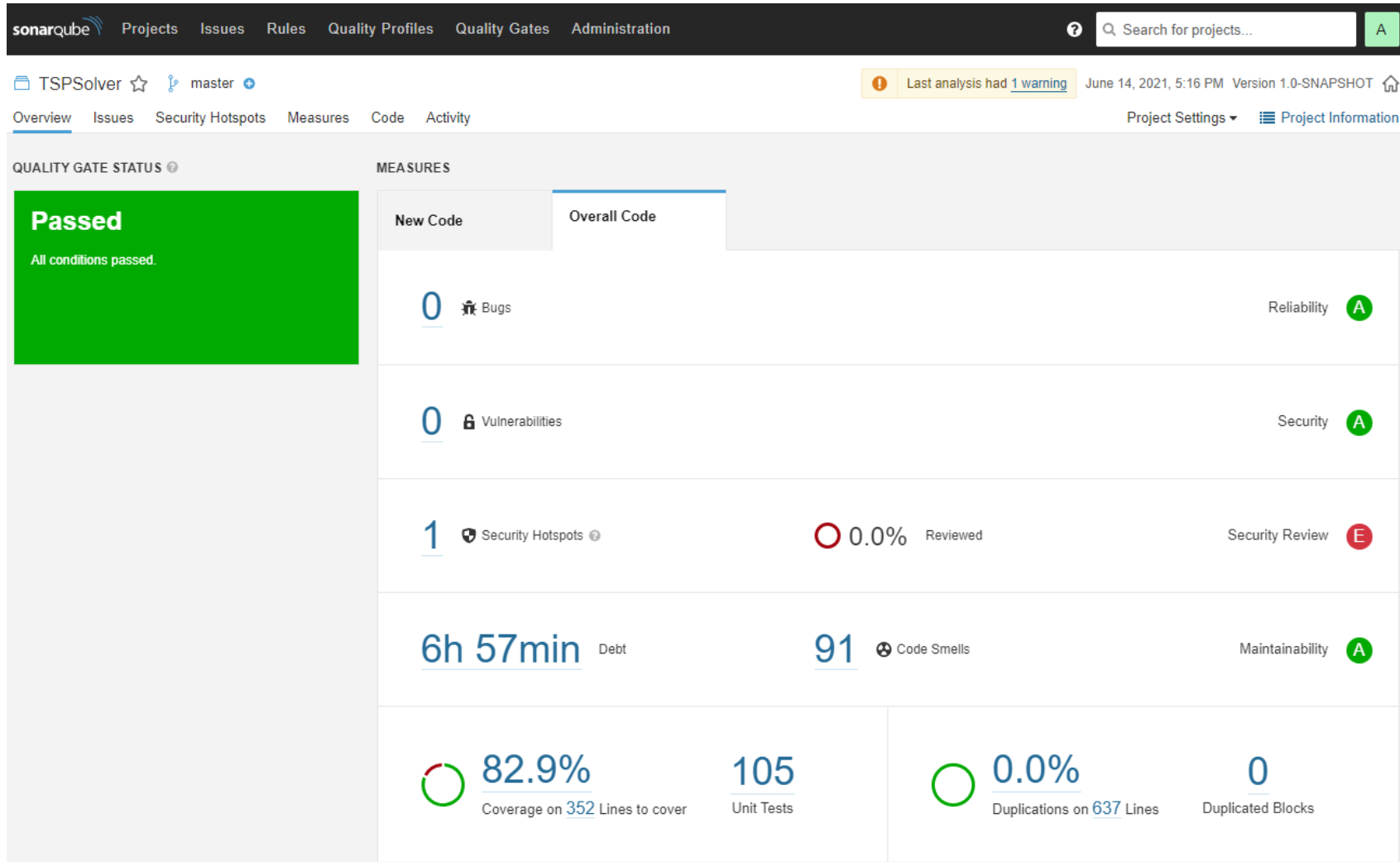
- Quality gates represent conditions which have to be fulfilled by a project to be acceptable
 - ensure minimum standards for specific measures
 - conditions can be defined only for new code or for all code
- Arbitrary many custom quality gates can be defined
- Each project is associated with one quality gate
- Default quality gate in SonarQube 8.9.1:

Conditions on New Code

Metric	Operator	Value
Coverage	is less than	80.0%
Duplicated Lines (%)	is greater than	3.0%
Maintainability Rating	is worse than	A
Reliability Rating	is worse than	A
Security Hotspots Reviewed	is less than	100%
Security Rating	is worse than	A

SonarQube – Screenshot

<https://sonarqube.org>



SPW4

Test Automation & Continuous Delivery

Deployment & Monitoring

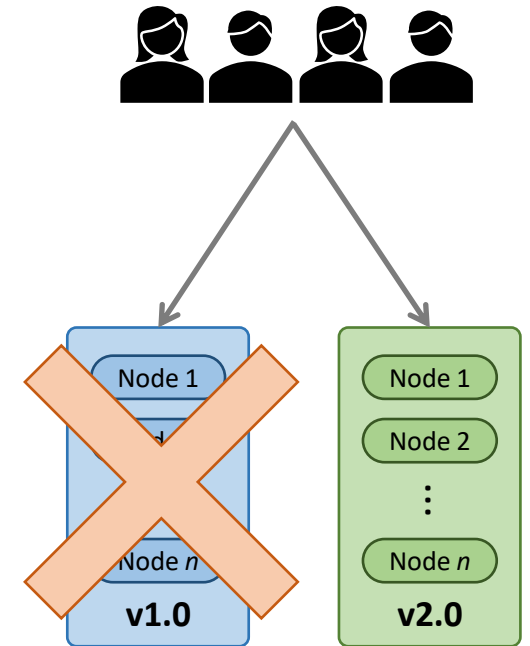
S. Wagner

Deployment

- Goals
 - provide software to users (cf. agile manifesto)
 - do not negatively impact user experience of current users
 - if problems occur, roll back to a good version
- Manual vs. Automated Deployment
 - manual deployment is time consuming and error prone
 - stable and safe deployment process has to be automated
 - deployment should be integrated into CI/CD pipeline
- Deployment Strategies
 - define the general process for deploying a new version

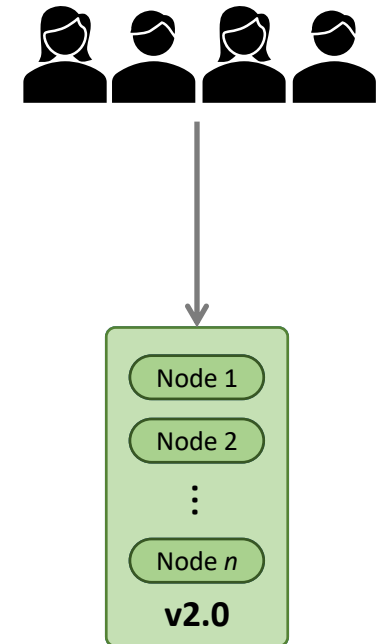
Reckless Deployment

- aka Recreate, Basic, Simple
 - very simple and naïve strategy
 - negative effects on users are not considered
 - leads to an effective downtime
 - very easy to implement
- Steps:
 - decommission old version
 - deploy new version



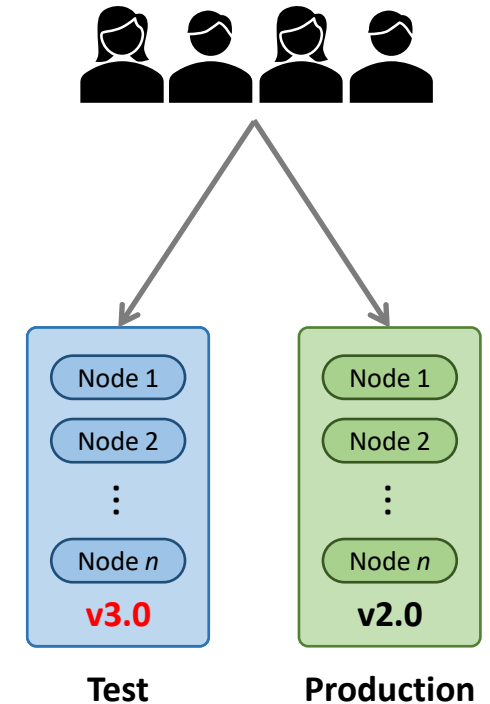
Rolling Deployment

- aka Gradual, Ramped, Incremental
 - system is gradually transitioned into new version
 - system is heterogeneous during the transition phase
 - users use old and new version arbitrarily
 - backward compatibility is essential
 - no downtime, if no problems occur
- Steps:
 - replace one node with new version
 - monitor health of new node
 - repeat until all nodes are replaced



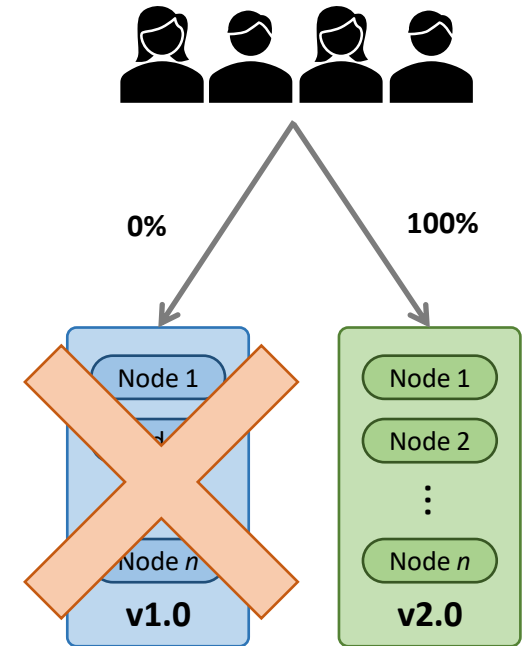
Blue/Green Deployment

- aka Red/Black, Staged
 - infrastructure consists of two identical environments
 - one is used for production and one for testing
 - at deployment both environments are swapped
 - rather costly as infrastructure is duplicated
- Steps:
 - test new version in identical test environment
 - test environment becomes new production environment
 - old production environment becomes test environment for next version



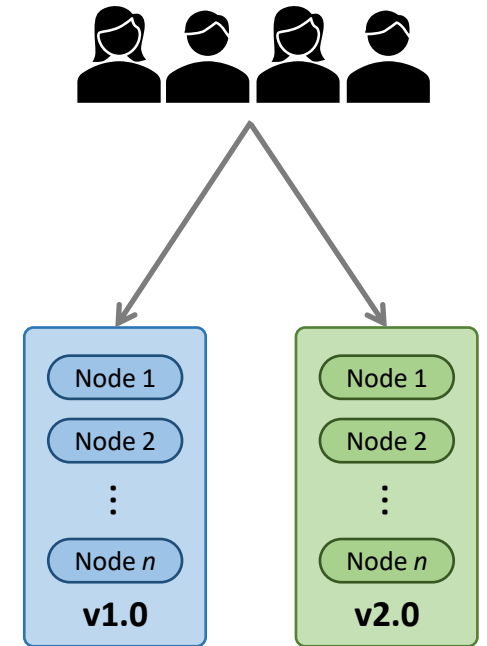
Canary Deployment

- aka Ringed, Silent Launch, A/B Testing
 - gradually increase the number of users who get the new version
 - monitor new version under real-life conditions
 - new version serves as a test instance
 - split users between old and new version by traffic percentage
 - other criteria can also be used for splitting (e.g., location, device, user group, etc.)
- Steps:
 - create new environment for new version
 - provide new version only to a small number of users
 - monitor how new version performs
 - gradually increase number of users
 - decommission old version, after all users have been shifted



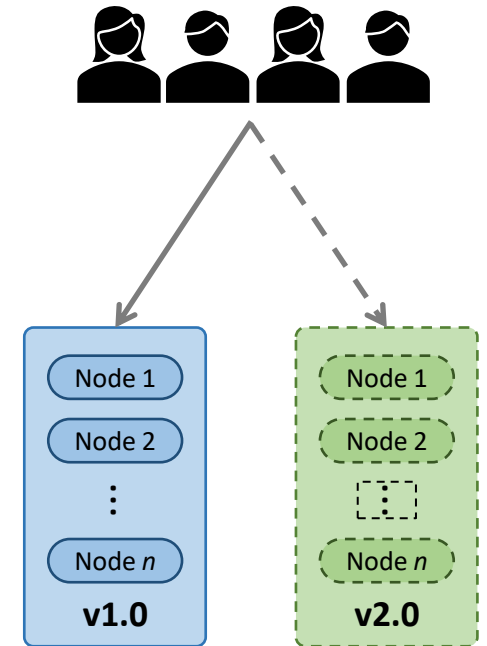
Versioned Deployment

- Parallel Deployment of Different Versions
 - new version is offered side by side with old version
 - versions are distinguished by their routes
 - e.g., very useful for different API versions
- Steps:
 - deploy new version alongside of old version
 - adjust resource capacities as users shift to new version



Shadow Deployment

- Shadow Environment for Testing
 - new version receives copy of real-world data
 - response from new version is ignored
 - enables testing with real-world data
 - infrastructure is duplicated
- Steps:
 - create new environment for new version
 - duplicate all traffic and route it to new version



Deployment Strategies Summary

Strategy	No Downtime	Real Traffic	Costs	Rollback Time	User Impact	Complexity
Reckless	✗	✗	✓	✗	✗	✓
Rolling	✓	✗	✓	✗	✓	✓
Blue/Green	✓	✗	✗	✓	✓	✓
Canary	✓	✓	✓	✓	✓	~
Versioned	✓	✗	~	✓	✓	~
Shadow	✓	✓	✗	✓	✓	✗

System Health Monitoring

- Health Monitoring is Essential for Automated Deployment
 - health of the system has to be checked continuously during/after deployment
 - any problems or errors must be detected and reported as early as possible
 - if the new version does not operate as expected, it has to be rolled back
- Collect, Visualize, and Analyze Logs and Metrics
 - logs and metrics should be collected and aggregated at a single point
- Logs vs. Metrics
 - logs are textual messages created by the system during operation
 - metrics are numerical values which describe the system's performance
 - metrics can be compared with thresholds to trigger alerts

Prometheus

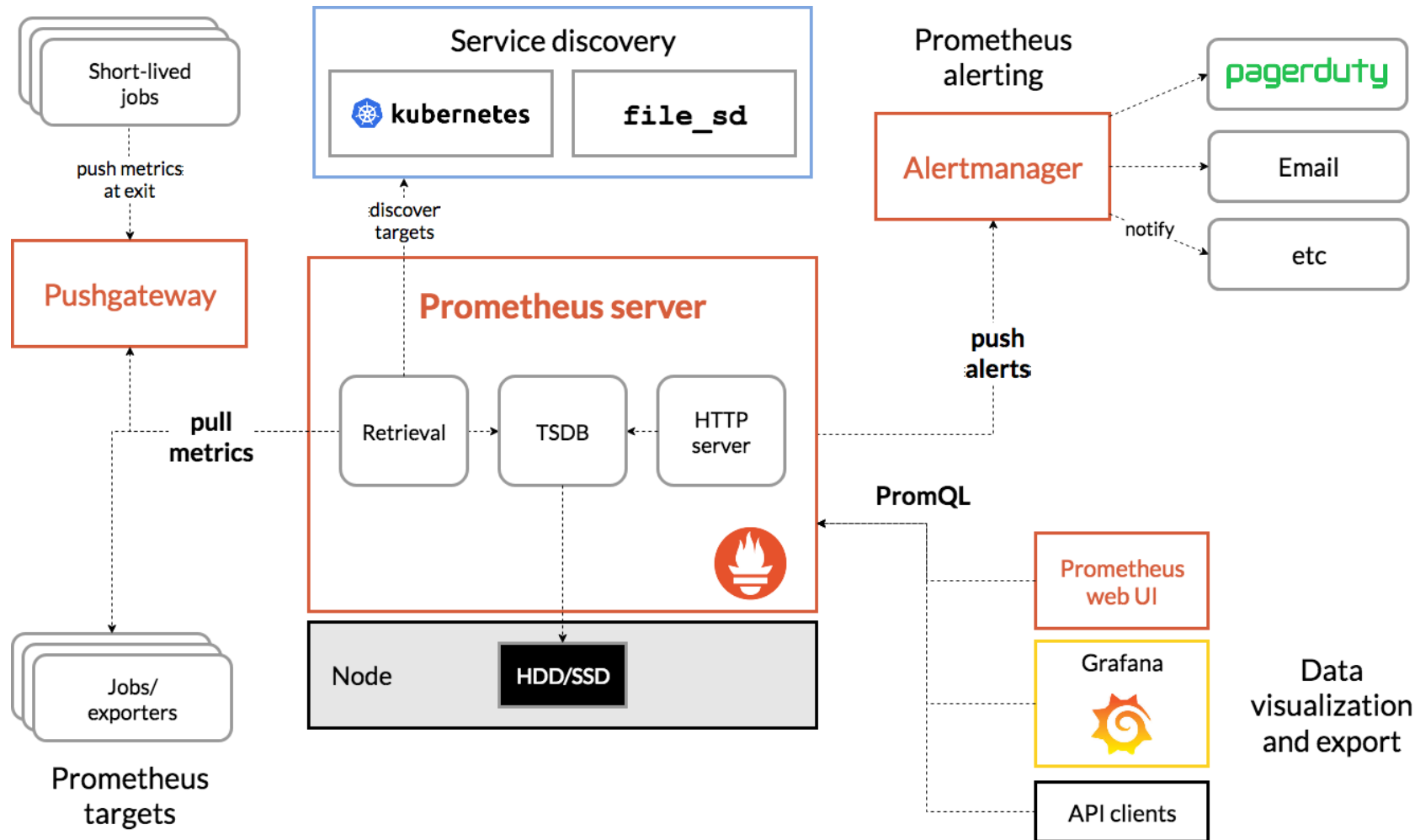
<https://prometheus.io>



- What is Prometheus?
 - open-source systems monitoring and alerting toolkit
 - collects and stores metrics in a time series database
- Features
 - multi-dimensional data model with metrics and labels
 - PromQL query language
 - targets are statically configured or dynamically discovered
 - metrics are pulled over HTTP
 - intermediary gateway also supports pushing

Prometheus

<https://prometheus.io>



Grafana

<https://grafana.com>



- What is Grafana?
 - open-source analytics and interactive visualization web application
 - provides dashboards for data retrieved from various data sources
- Features
 - custom dashboards and dashboard templates
 - lots of predefined dashboards and a very active community
 - support of many different data sources (e.g., Prometheus, InfluxDB, Elasticsearch, MySQL, MS SQL Server, MongoDB, Splunk, ...)
 - authentication, permission management, and multi-tenancy
 - data transformations and annotations

Grafana

<https://grafana.com>



<https://play.grafana.org>