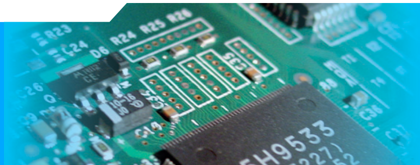
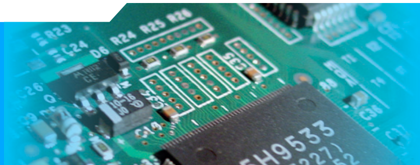
Unit testing

After B. Baudry and B. Combemale



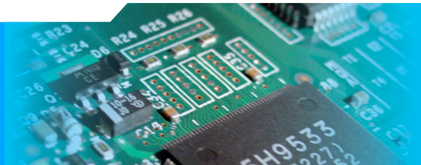
Goals
of unit testing

- Build confidence on the correctness of a class, when used separately
  - this regards the management of its internal state
  - each operation must be tested



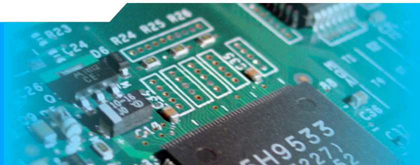
Test and encapsulation

- The oracle needs access to the internal state to check for success or failure
- This goes against encapsulation
- Test points for test probe is a possible answer (by providing a test interface)



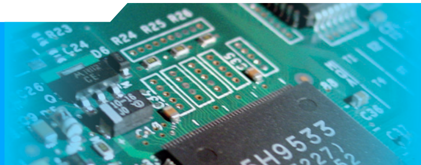
Overall process

- Test initialization operations
  - using retrieve accessors
- Test accessors
  - update then retrieve
- Test service operations



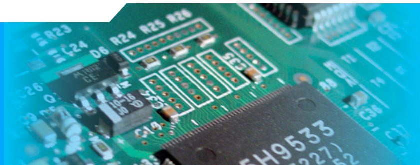
Overall process (cont’d)

- At least one test case for each operation
- Some operations/methods depend on other objects
  - test stubs can be used to solve this



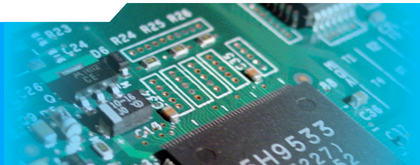
Junit test structure

- A test case is implemented
  - by an operation with an annotation
  - setup (initialisation), calls and oracle are in the method
- A test class groups a set of test operations
  - usually a test class for each tested class

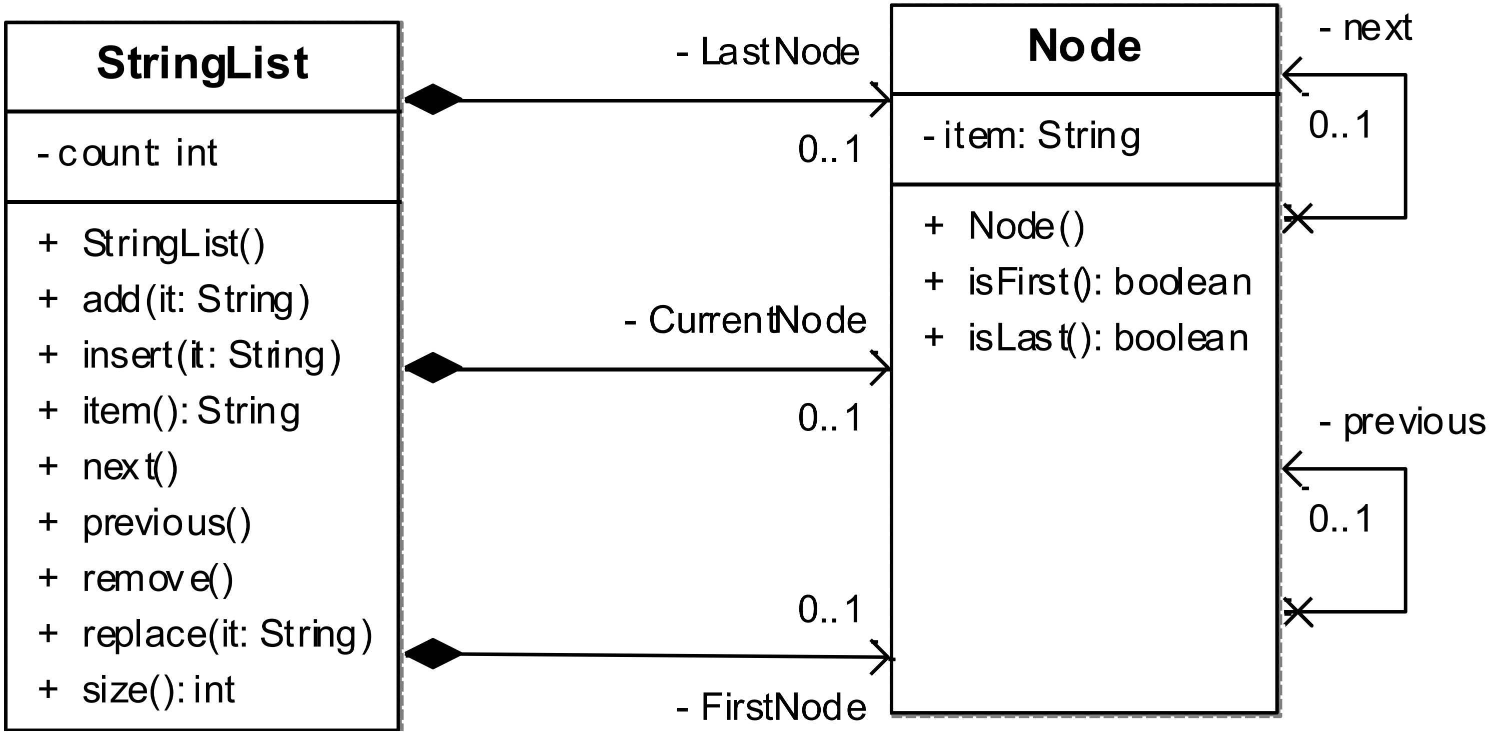


Structure for a test method

- Initialisation
  - to set the object’s internal state in a state suitable for the test
  - JUnit provides a system to factorise the initialisation code: the @BeforeEach annotation
- A call to the object’s operation
- An oracle statement
  - to compare the call’s result/effect with the expected result/effect



Example



StringList

- count: int

+ StringList()
+ add(it: String)
+ insert(it: String)
+ item(): String
+ next()
+ previous()
+ remove()
+ replace(it: String)
+ size(): int

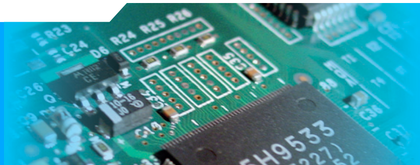
Node

- item: String

+ Node()
+ isFirst(): boolean
+ isLast(): boolean

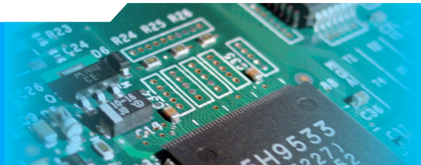
Example (cont'd)

- Nine operations
  - therefore at least nine test cases and therefore **at least** 9 test operations
- No access to the attributes
  - count, lastNode, currentNode, , firstNode
  - white box testing



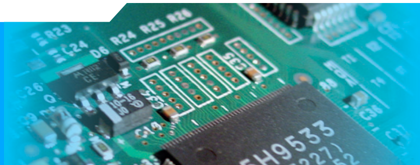
The JUnit 5 framework

- Goals
  - to provide elementary services to ease test case definition
  - to automatically look for, execute and store test case execution results, including with varying parameters
  - to ease systematic testing and prevent regression



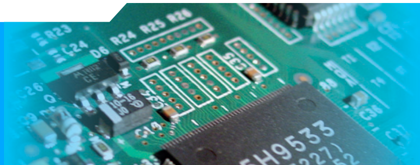
JUnit v5

- Packages
  - import org.junit.jupiter.api.Test;
  - import org.junit.jupiter.api.DisplayName;
- Test cases
  - any operation (not necessarily public)
  - annotated with @Test, @ParameterizedTest, @RepeatedTest, @TestFactory or @TestTemplate



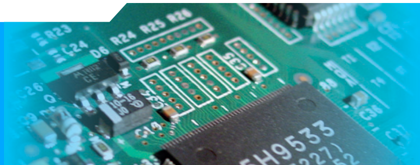
Before/after

- Setup operations are tagged
  - use @BeforeEach
  - import org.junit.jupiter.api.BeforeEach;
  - execution order undefined if several setup operations
- Same logic for clean up using @AfterEach



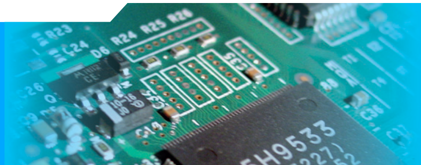
@BeforeAll and @AfterAll

- An operation annotated with @BeforeAll
  - is run before execution of the test case sequence
- Mutatis mutandis for @AfterAll



Parameters

- See the Length example on the ACO2018 gitlab project
  - https://gitlab.istic.univ-rennes1.fr/plouzeau/ACO2018/tree/master/Length/src/fr/istic/nplouzeau/length

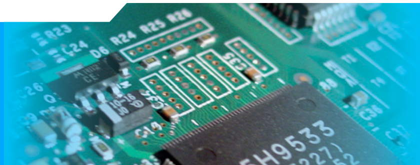


Key points of the Length example

- Interface/class separation
- Javadoc to describe the operations
- @BeforeEach
- Assertions
- Parameterized tests

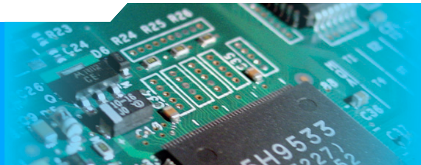
Another example

- See on the ISTIC gitlab:
  - https://gitlab.istic.univ-rennes1.fr/plouzeau/ACO2018/tree/master/StackExample



Key points of the Stack example

- Categories of operations tested
  - initialisation and simple getters
  - retrieve operations
  - update operations
  - robustness test
    - explicitly call operations using invalid preconditions and check for exceptions



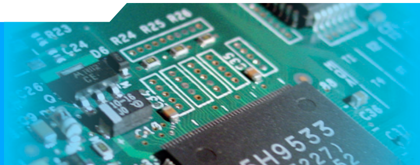
Questions?

- You will use Junit in practical sessions

Writing mock classes

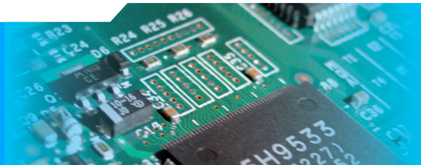
Mock classes

- Mock classes help in unit testing
  - they provide a simulated, controlled environment for the class under test
  - in practice, they are a vital part of the tests as they are a part of the oracle



Example

- For an implementation of the Observer design pattern
  - how does one check that the subject really calls the registered observers?
  - a mock observer is needed, that will record the subject's activity



Writing mocks is boring

- Many pieces of trivial software (stubs)
- And complex pieces (mechanisms for oracle design)
- But fortunately we now have mock generators

Mockito

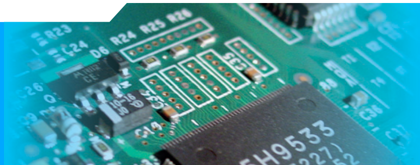
- Mockito is a framework for defining mocks
  - It using Java introspection and intercession to define mock objects at runtime
  - Mockito is really powerful to design complex oracles

A simple Mockito example

- For an Observer design pattern implementation
  - In this example Mockito will generate a fake Observer object
  - This object will report on calls to its update() operation

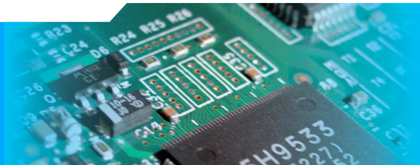
Setting up the mock

- @Test
- public void testUpdateIsCalled()
- Observer<String> fake = Mockito.mock(Observer.class)
- subject.attach(fake)



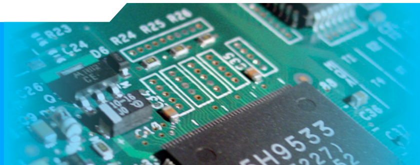
Checking

- // Should trigger one call to the update
- // operation of the fake observer
- subject.setValue("Test")
- Mockito.verify(fake).update(subject)



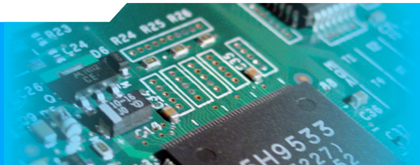
Effect

- If the verification is not successful the surrounding test case will fail



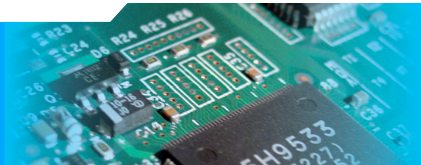
More details on Mockito

- On github
  - https://github.com/mockito/mockito



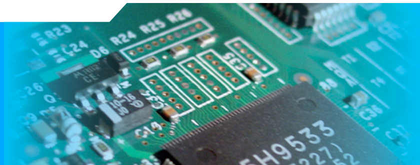
The coverage tool

- Helps you to build test cases so that you leave no line untested
- Very simple to use from an IDE (Eclipse, IntellijIDEA)
- Run it on each test class
  - Then look for red bars in the coverage window for each implementation class tested
  - Do not bother with test class coverage, it is the tested class coverage that counts
- This gives you precious hints about missing test cases



Process and guidelines

- Use
  - SOLID
  - DRY
  - CBD
    - Write preconditions and postconditions
    - Test them (defensive programming)
- TDD: test driven design



Test driven design

- Design the architecture (SOLID)
- Write the specifications of the interface types
    - pre and postconditions
- Write the test cases using JUnit and mockito
- Write the code of the implementation, use sonarlint
- Run JUnit using coverage, correct the implementation and add more test cases to have 100% code coverage