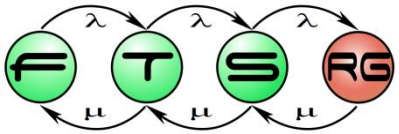
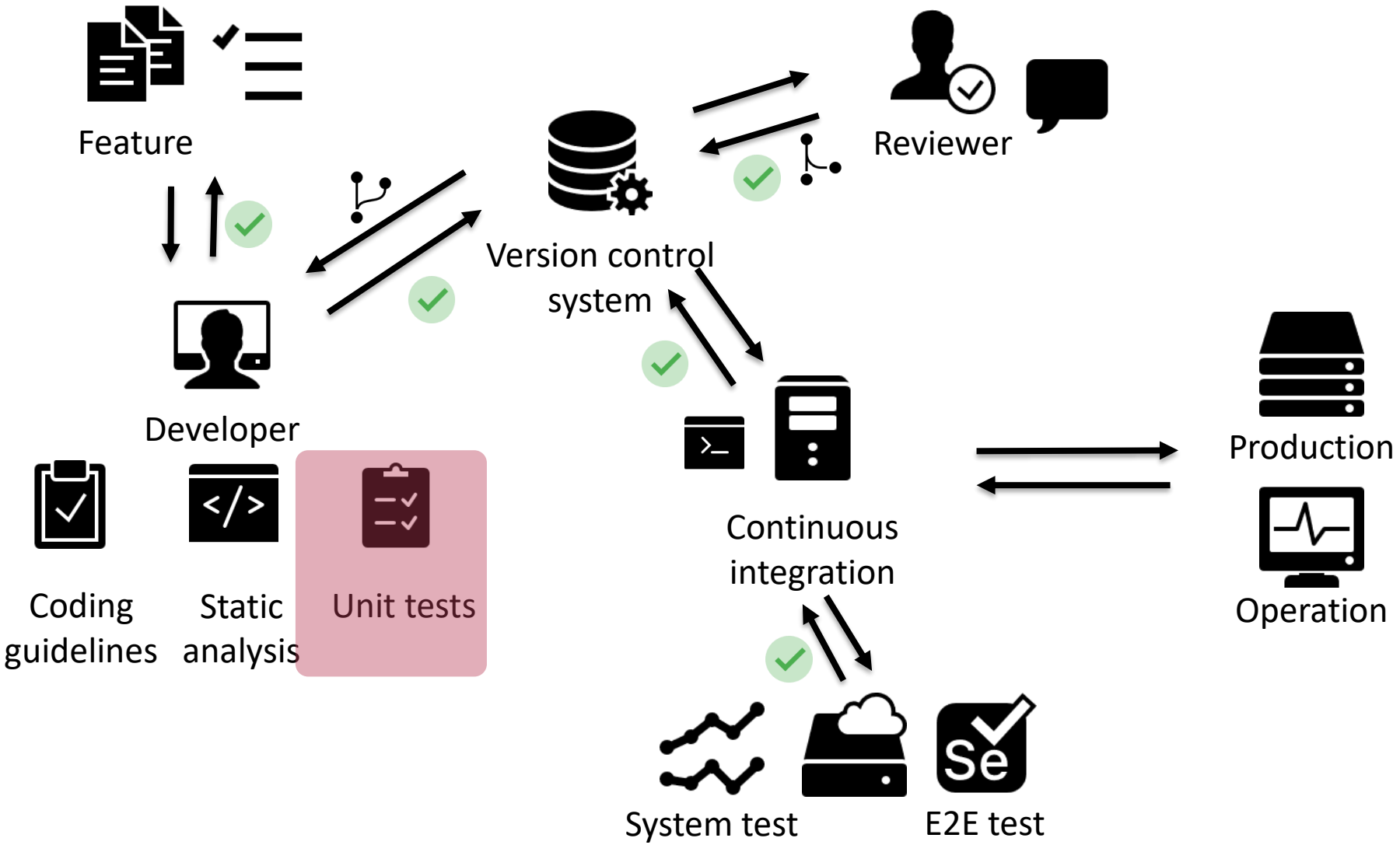
# Unit testing

**Zoltan Micskei**

**Budapest University of Technology and Economics**
**Fault Tolerant Systems Research Group**



# Overview



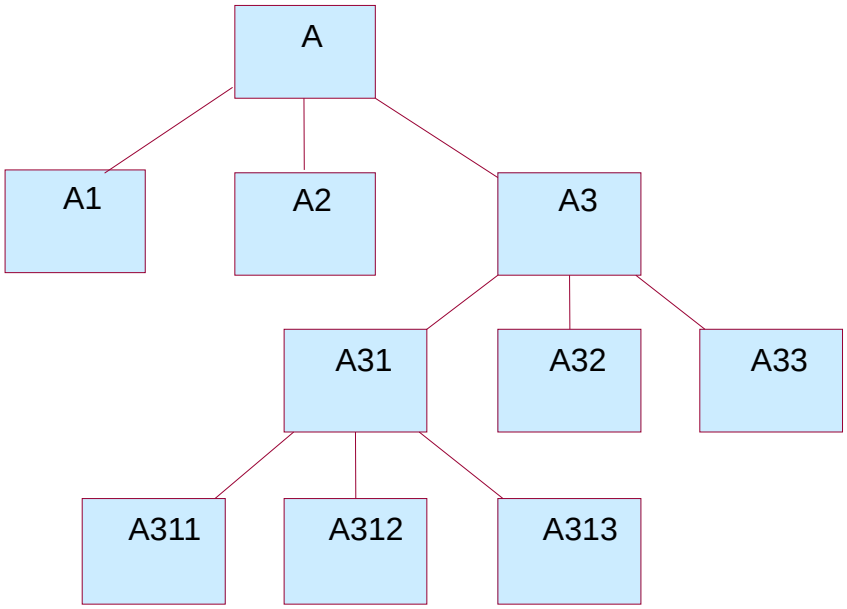

Icons: icons8.com

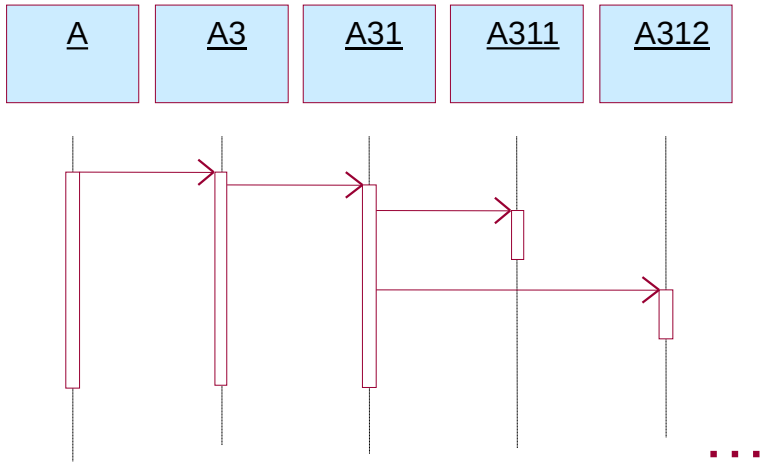
# Learning outcomes

- Explain characteristics of good unit tests (K2)
- Write unit tests using a unit testing framework (K3)

# Module / unit testing

- Module / unit:
  - Logically separable part
  - Well-defined interface
  - Can be: method / class / package / component…
- Call hierarchy (ideal case):



# Why do we need unit testing?

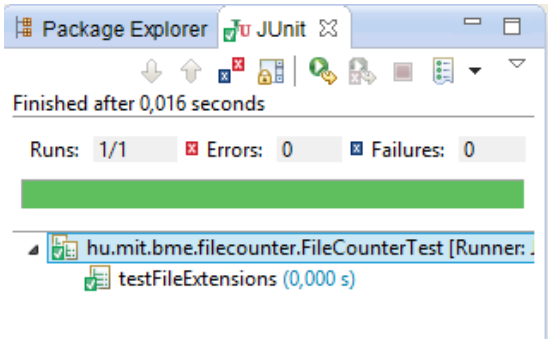
- Goal: Detect and fix defects during development (lowest level)
  - Can integrate later already tested modules
  - Developer of the unit can fix the defect fastest

- Units can be tested separately
  - Manage complexity
  - Locate defects more easily, fix is cheaper
  - Gives confidence for performing changes

- Characteristics of unit tests
  - Checks a well-defined functionality
  - Defines a “contract” for the unit
  - Can be used as an example
  - (Not necessarily automatic)

# Unit test frameworks

- Run unit tests frequently
  - During development (e.g. refactoring)
  - → Need to be fast
- Good tool support
  - Frameworks (JUnit, xUnit, TestNG, …)
  - Support in IDE (Eclipse, VS, …)
- Usually simple functionality
  - Define test sequences and checks
  - Run tests
  - Display results (red-green)



# Example: simple JUnit test

```
public class ListTest{
    List list; // SUT

    @Before public void setUp(){
        list = new List();
    }

    @Test public void add_EmptyList_Success(){
        list.Add(1);
        assertEquals(1, list.getSize());
    }
}
```

Preparing test

Calling SUT

Checking

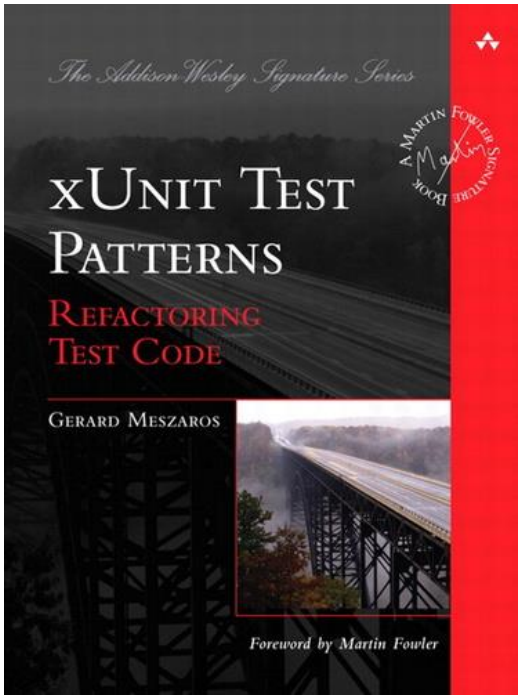
# JUnit annotations (excerpt)

| Annotation | Description |
|---|---|
| `@Test` | Defining a test method |
| `@Before` | Executed before each test, e.g. setting environment |
| `@After` | Executed after each test |
| `@BeforeClass` | Executed only once before all tests<br>(caution, tests will be not independent!) |
| `@AfterClass` | Executed after all tests, e.g. global cleanup |
| `@Ignore` | Do not execute given test |
| `@Test(expected=IllegalArgument Exception.class)` | Test passes if code throws this Exception |
| `@Test(timeout=100)` | Limits execution time of test |

Others: creating Suite, Category, Parametrized…

# How to write good unit tests?

1. Following guidelines
2. Using test patterns
3. Avoiding test smells




Gerard Meszaros. 2006. *xUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

http://xunitpatterns.com

# Good unit tests (guidelines!)

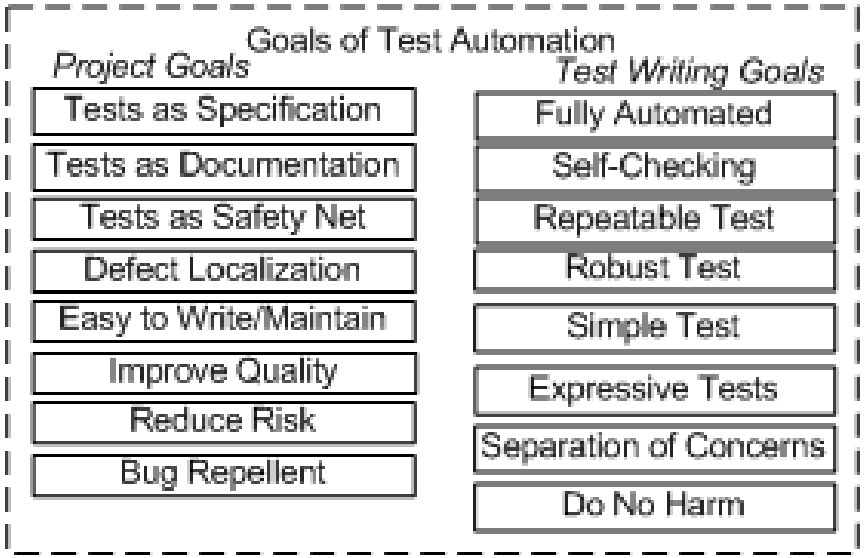
- Simple, reliable
  - No complex logic (e.g. loops, try/catch)
- One test does one thing
  - (not necessarily one assert statement)
- This is also quality code
  - No duplicated code
  - Easy to understand and maintain
- Tests are independent
- Checks are not overspecified
- …

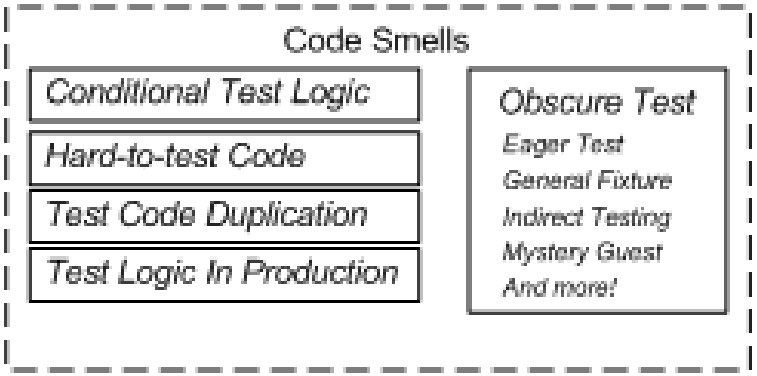
# Coding conventions

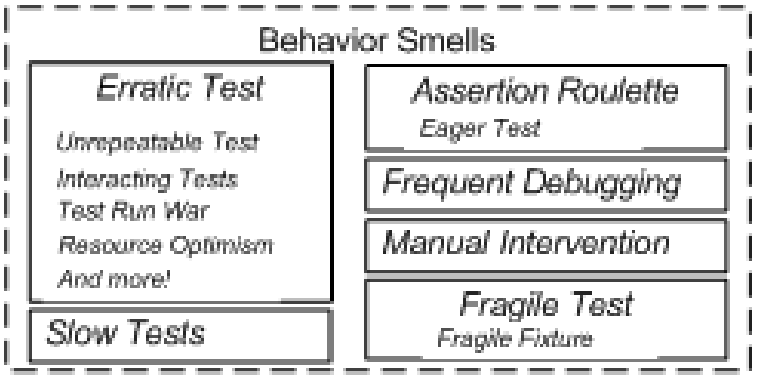
- Name of test class: `[Unit_name]Test`
- Name of test method:
  - **`Method_StateUnderTest_ExpectedBehavior`**
  - **`MethodName_DoesWhat_WhenTheseConditions`**
  - **`[feature being tested]`**
- Structure of test:

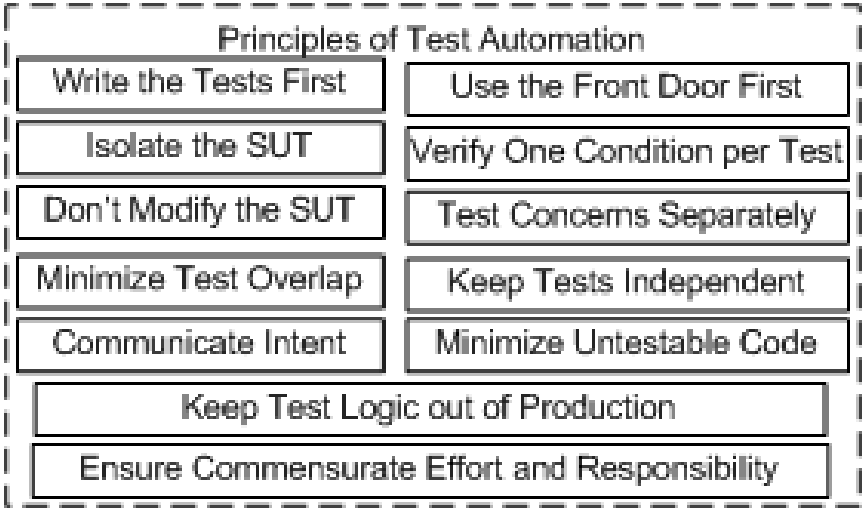
```
// Arrange          // Given
// Act              // When
// Assert           // Then
```

# Guidelines from “xUnit Test Patterns”



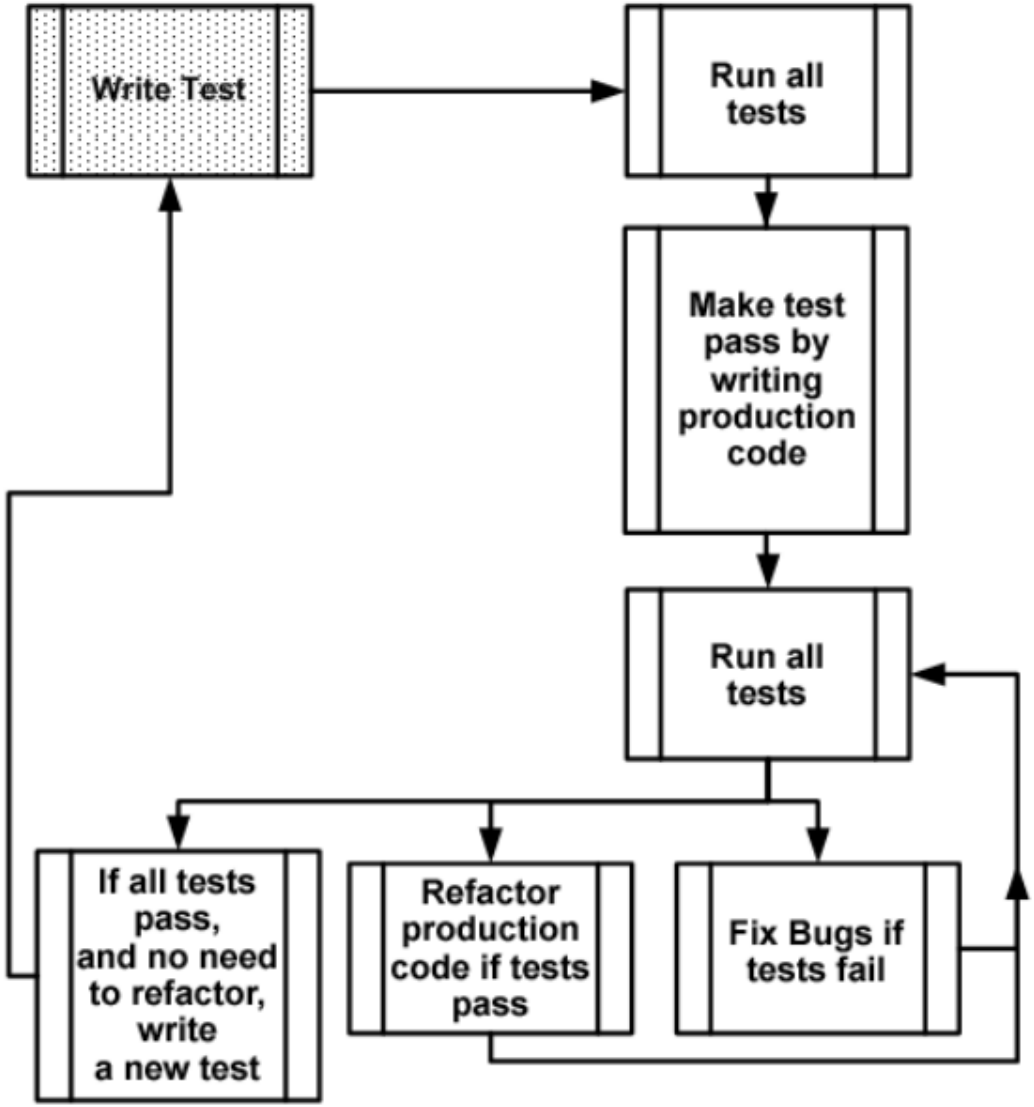





Source: Gerard Meszaros, http://xunitpatterns.com/

# Test-Driven Development (TDD)



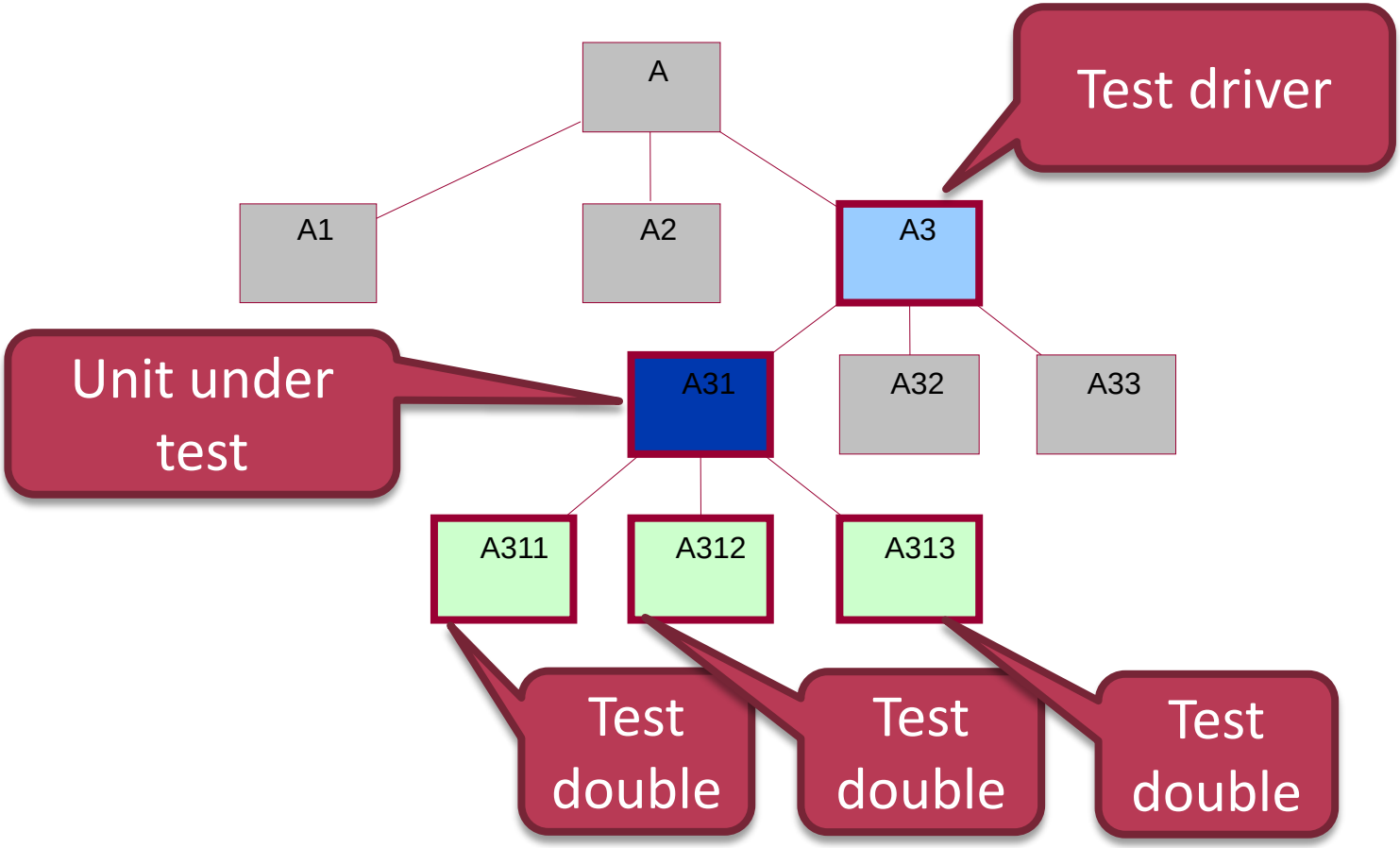
# UNIT ISOLATION

# Learning outcomes

- Explain why and when isolation of dependencies is recommended in unit tests (K2)
- Use an isolation framework to write isolated unit tests (K3)

# Isolation in unit tests

- Tests unit separately in isolation
- Needed: test drivers and test doubles



# Problem: Handling dependencies

- What is a dependency?

  Anything that

  - is collaborating with the SUT,
  - but does not belong to (could not control it)

- Examples:
  - Other modules
  - File system, network call
  - Using current date/time
  - …

# Example: Hard to test code

```
public class PriceService{
    private DataAccess da = new DataAccess();

    public int getPrice(String product)
     throws ProductNotFoundException {
        Integer p = this.da.getProdPrice(product);
        if (p == null)
            throw new ProductNotFoundException();

        return p;
    }
}
```

Hard to isolate, how to specify a test version?

# Example: Making the SUT testable

```
public class PriceService{
    private IDataAccess da;

    public PriceService(IDataAccess da){
        this.da = da;
    }

    public int getPrice(String product)
     throws ProductNotFoundException {
        Integer p = this.da.getProdPrice(product);
        if (p == null)
            throw new ProductNotFoundException();
        return p;
    }
}
```

Passing implementation in constructor

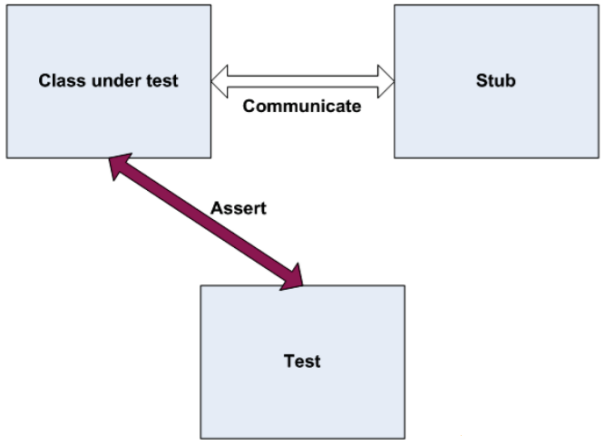
# Example: Unit tests for the SUT

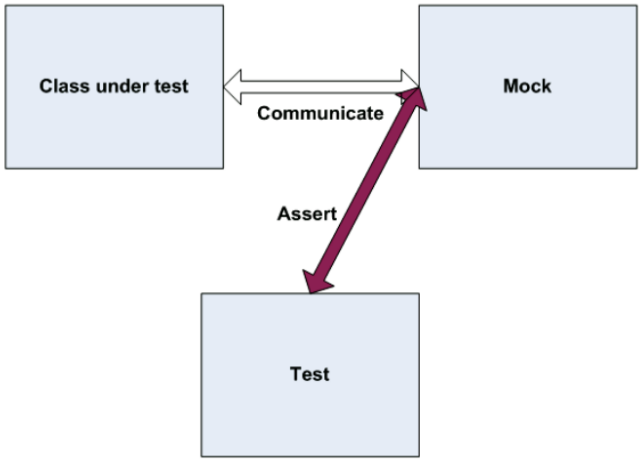
```
public class PriceServiceTest{
    @Before public void init(){
        DataAccessStub das = new DataAccessStub();
        das.add("A100", 50);
        ps = new PriceService(das);
    }

    @Test public void SuccessfulPriceQuery(){
        int p = ps.getPrice("A100");
        assertEquals(50, p);
    }

    @Test(expected = ProductNotFoundException.class)
    public void NotExistingProduct(){
        ps.getPrice("notExists");
    }
}
```

Using a stub in the test

# Stub, Mock, Dummy, Fake… objects

- Many techniques for substituting dependencies
  - Different names (see *xUnit Patterns*)
  - Common name: **Test double**
- Stub
  - Checks the state of the SUT
- Mock
  - Check the interactions of the SUT
- Dummy
  - Not used object (filler)
- Fake
  - Working, but not the real one



# Isolation frameworks

- Tedious to create mocks and stubs
  - Lots of repeated, plumbing code
  - Time-consuming to maintain

- Framework functionality
  - Input: interface or class description
  - Output: generate stub/mock dynamically

- Example:
  - JMock, Mockito, Rhino Mocks, Typemock…

# Example: Using mocks (Mockito)

```
public class PriceServiceTest{
      @Before public void init(){
            mockDA = mock(DataAccess.class);
            ps = new PriceService(mockDA);
      }

      @Test public void SuccessfulPriceQuery(){
            // Arrange
            when(mockDA.getProdPrice("A100")).thenReturn(50);

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

            // Assert
            verify(mockDA, times(1)).getProdPrice("A100")
      }
      …
}
```

Create a compatible test double

Specify rules

Check interactions

# SUMMARY

# Is it worth to create such unit tests?

## Calculations on an example project:

| Stage | Team without tests | Team with tests |
|---|---|---|
| Implementation (coding) | 7 days | 14 days |
| Integration | 7 days | 2 days |
| Testing and bug fixing | Testing, 3 days<br>Fixing, 3 days<br>Testing, 3 days<br>Fixing, 2 days<br>Testing, 1 day<br>Total: 12 days | Testing, 3 days<br>Fixing, 1 day<br>Testing, 1 day<br>Fixing, 1 day<br>Testing, 1 day<br>Total: 8 days |
| Overall release time | 26 days | 24 days |
| Bugs found in production | 71 | 11 |

Source: Roy Osherove, The Art of Unit Testing, 2009.

# Further information

- Roy Osherove: The Art of Unit Testing: With Examples in C#
  - Manning Publications; 2nd edition (2013)
  - URL: http://artofunittesting.com/

- Martin Fowler: Mocks Aren't Stubs,  2007
  - URL: http://martinfowler.com/articles/mocksArentStubs.html
- Martin Fowler: UnitTest, 2014
  - http://martinfowler.com/bliki/UnitTest.html