Testing and Profiling

Goals of Testing

- Understand system quality
- Provide information for decisions
  - E.g. release-readiness
- Bug finding/preventing
  - Beware: Testing shows the **presence**, not the absence of bugs. (Dijkstra)

7 Testing Principles

7 Testing Principles

1 Only presence of bugs can be shown

2 Exhaustive testing practically impossible

3 Test in the early development phases

4 Defect clustering

- Most defects relate to a small number of components

5 Pesticide paradox

- Efficiency of testing decreases when re-executed
- Every methodology misses some problems

6 Testing is context-dependent

7 Absence-of-errors fallacy

- Error-free test execution does not mean error-free program

7 Testing Principles

1 Only presence of bugs can be shown

2 Exhaustive testing practically impossible

3 Test in the early development phases

4 Defect clustering

- Most defects relate to a small number of components

5 Pesticide paradox

- Efficiency of testing decreases when re-executed
- Every methodology misses some problems

6 Testing is context-dependent

7 Absence-of-errors fallacy

- Error-free test execution does not mean error-free program

7 Testing Principles

1 Only presence of bugs can be shown

2 Exhaustive testing practically impossible

3 Test in the early development phases

4 Defect clustering

- Most defects relate to a small number of components

5 Pesticide paradox

- Efficiency of testing decreases when re-executed
- Every methodology misses some problems

6 Testing is context-dependent

7 Absence-of-errors fallacy

- Error-free test execution does not mean error-free program

7 Testing Principles

1 Only presence of bugs can be shown

2 Exhaustive testing practically impossible

3 Test in the early development phases

4 Defect clustering

- Most defects relate to a small number of components

5 Pesticide paradox

- Efficiency of testing decreases when re-executed
- Every methodology misses some problems

6 Testing is context-dependent

7 Absence-of-errors fallacy

- Error-free test execution does not mean error-free program

7 Testing Principles

1 Only presence of bugs can be shown

2 Exhaustive testing practically impossible

3 Test in the early development phases

4 Defect clustering

- Most defects relate to a small number of components

5 Pesticide paradox

- Efficiency of testing decreases when re-executed
- Every methodology misses some problems

6 Testing is context-dependent

7 Absence-of-errors fallacy

- Error-free test execution does not mean error-free program

7 Testing Principles

1 Only presence of bugs can be shown

2 Exhaustive testing practically impossible

3 Test in the early development phases

4 Defect clustering

- Most defects relate to a small number of components

5 Pesticide paradox

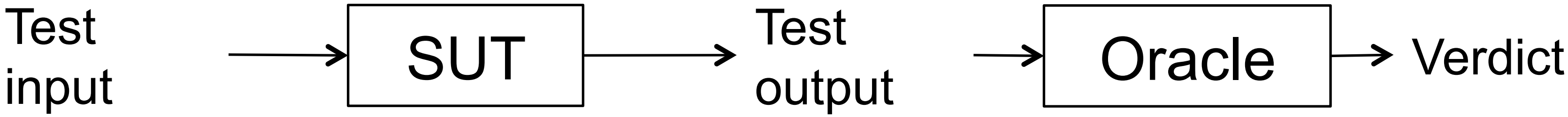
- Efficiency of testing decreases when re-executed
- Every methodology misses some problems

6 Testing is context-dependent

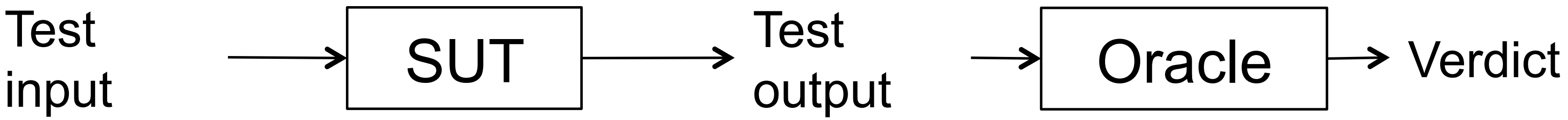
7 Absence-of-errors fallacy

- Error-free test execution does not mean error-free program

Basics

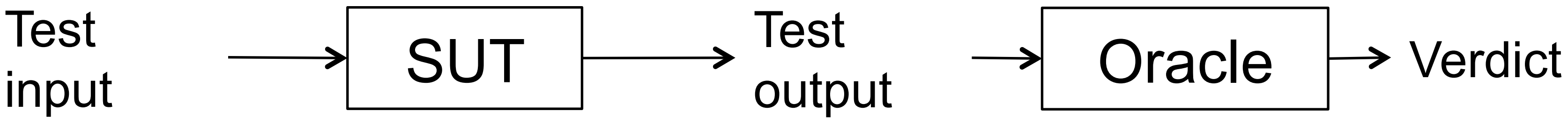


Basics



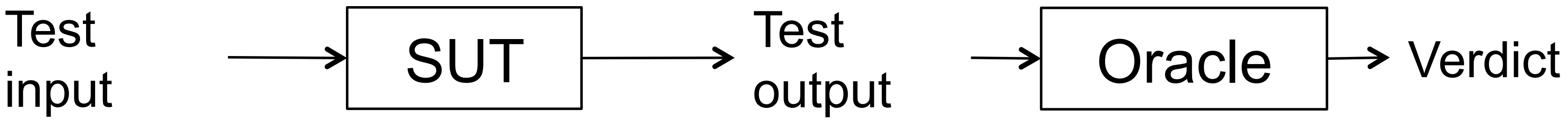
- Test case
  - Input values, preconditions, expected results and postconditions
- Test suite
- Oracle
  - Compares real and expected outputs
- Verdict
  - Pass, Fail, Inconclusive, Error
- Testing != debugging

Basics


Test input → SUT → Test output → Oracle → Verdict

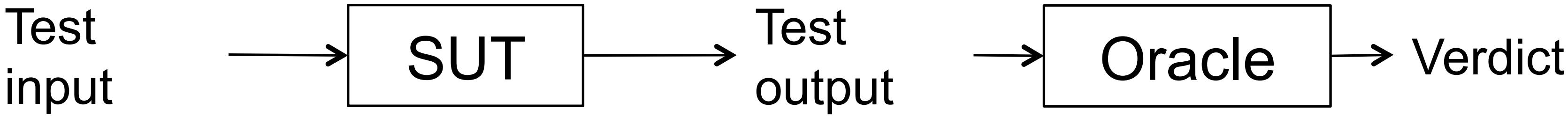
- Test case
  - Input values, preconditions, expected results and postconditions
- Test suite
- Oracle
  - Compares real and expected outputs
- Verdict
  - Pass, Fail, Inconclusive, Error
- Testing != debugging

Basics



- Test case
  - Input values, preconditions, expected results and postconditions
- Test suite
- Oracle
  - Compares real and expected outputs
- Verdict
  - Pass, Fail, Inconclusive, Error
- Testing != debugging

Basics


Test input

SUT

Test output

Oracle

Verdict

- Test case
  - Input values, preconditions, expected results and postconditions
- Test suite
- Oracle
  - Compares real and expected outputs
- Verdict
  - Pass, Fail, Inconclusive, Error
- Testing != debugging

Problems, Challenges

- Test selection
  - How to select test inputs
- Exit criteria
  - When is testing finished
- Oracle
  - How to define a good test oracle
- Testability
  - How easy is to test the system?
    - Observability
    - Controllability

Problems, Challenges

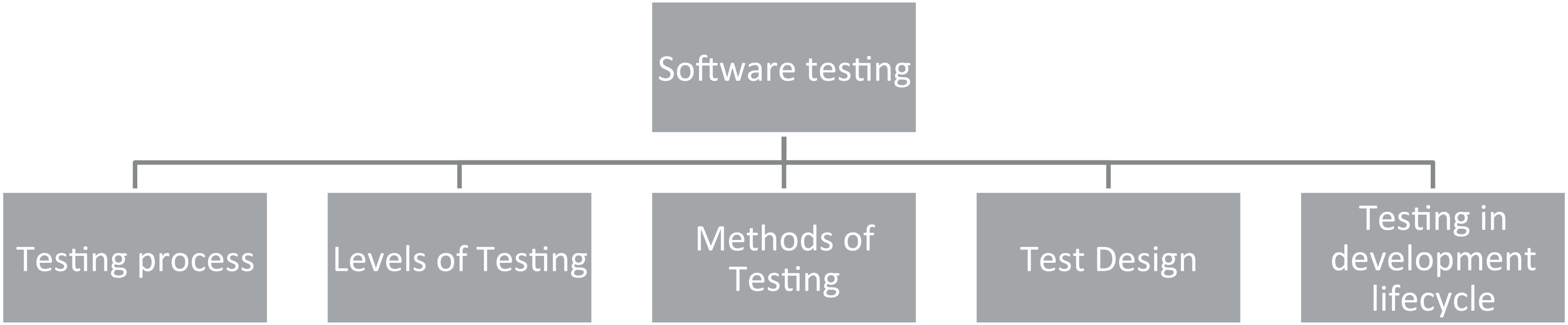
- Test selection
  - How to select test inputs
- Exit criteria
  - When is testing finished
- Oracle
  - How to define a good test oracle
- Testability
  - How easy is to test the system?
    - Observability
    - Controllability

Problems, Challenges

- Test selection
  - How to select test inputs
- Exit criteria
  - When is testing finished
- Oracle
  - How to define a good test oracle
- Testability
  - How easy is to test the system?
    - Observability
    - Controllability

Problems, Challenges

- Test selection
  - How to select test inputs
- Exit criteria
  - When is testing finished
- Oracle
  - How to define a good test oracle
- Testability
  - How easy is to test the system?
    - Observability
    - Controllability



Software testing

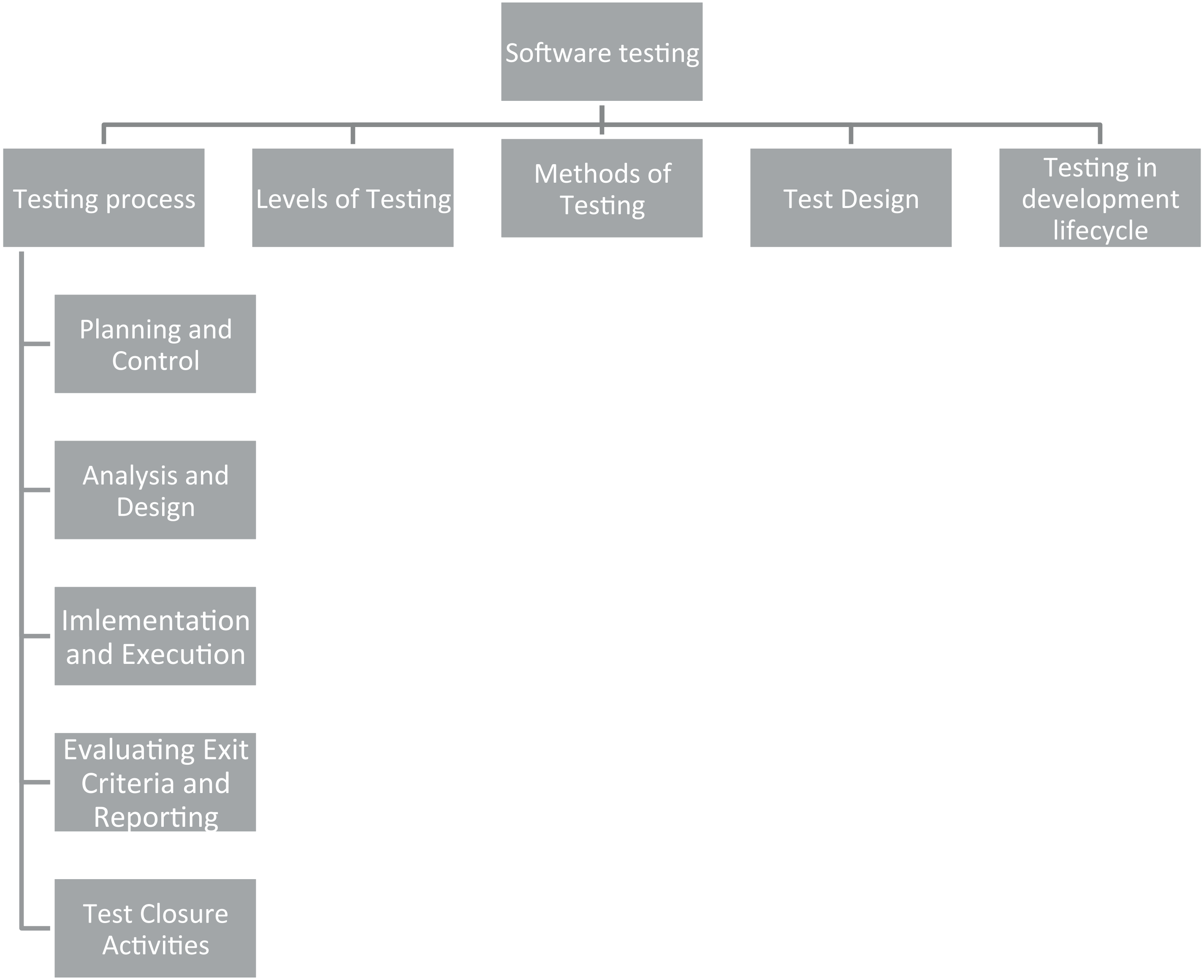
Testing process

Levels of Testing

Methods of Testing

Test Design

Testing in development lifecycle



Software testing

Testing process

Levels of Testing

Methods of Testing

Test Design

Testing in development lifecycle

Planning and Control

Analysis and Design

Imlementation and Execution

Evaluating Exit Criteria and Reporting

Test Closure Activities

Testing Strategy

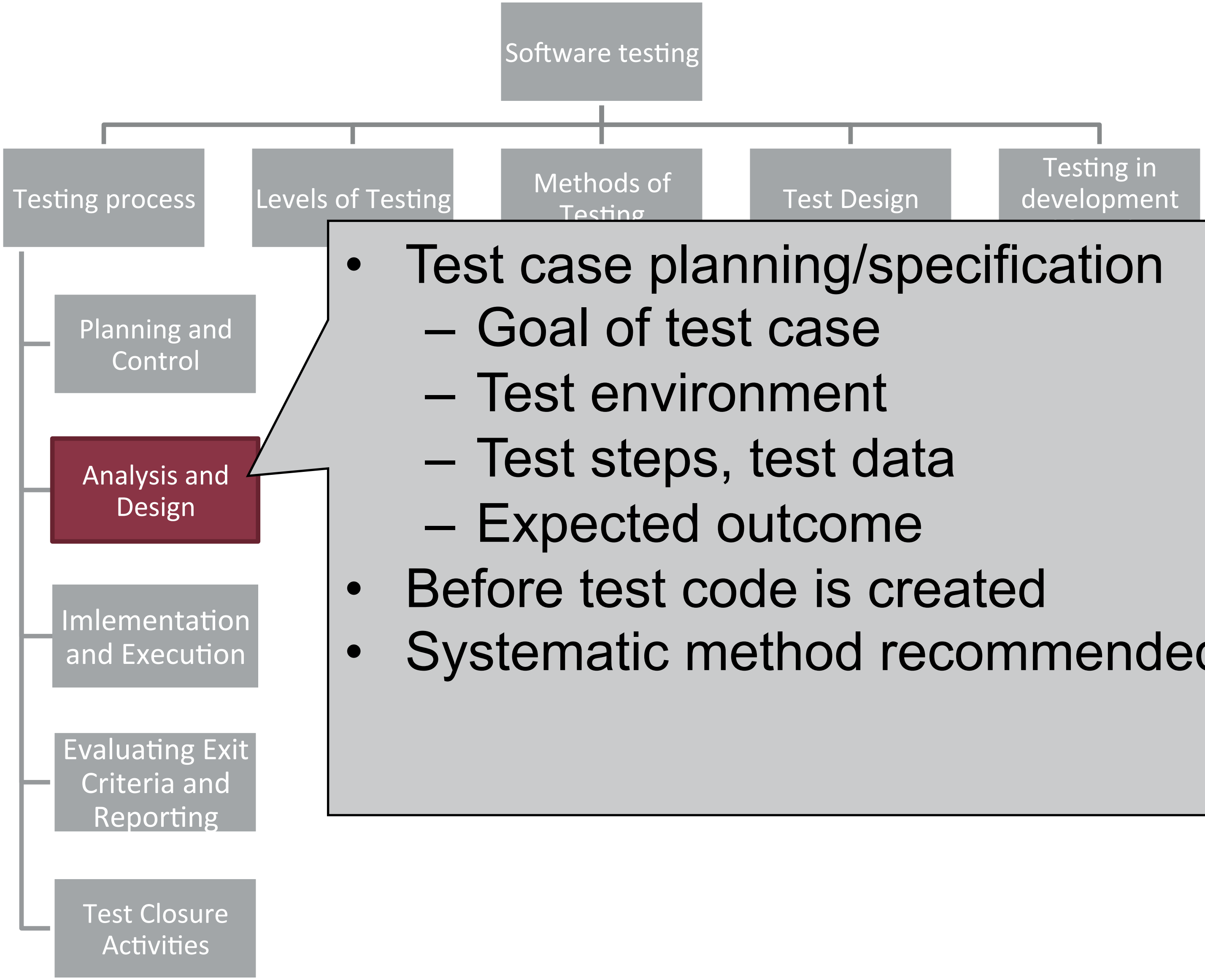
- General policies
  - Methodology
  - Test types
  - Test tools
  - Who tests
  - Exit criteria
  - Testing documentation
  - …

Testing Strategy

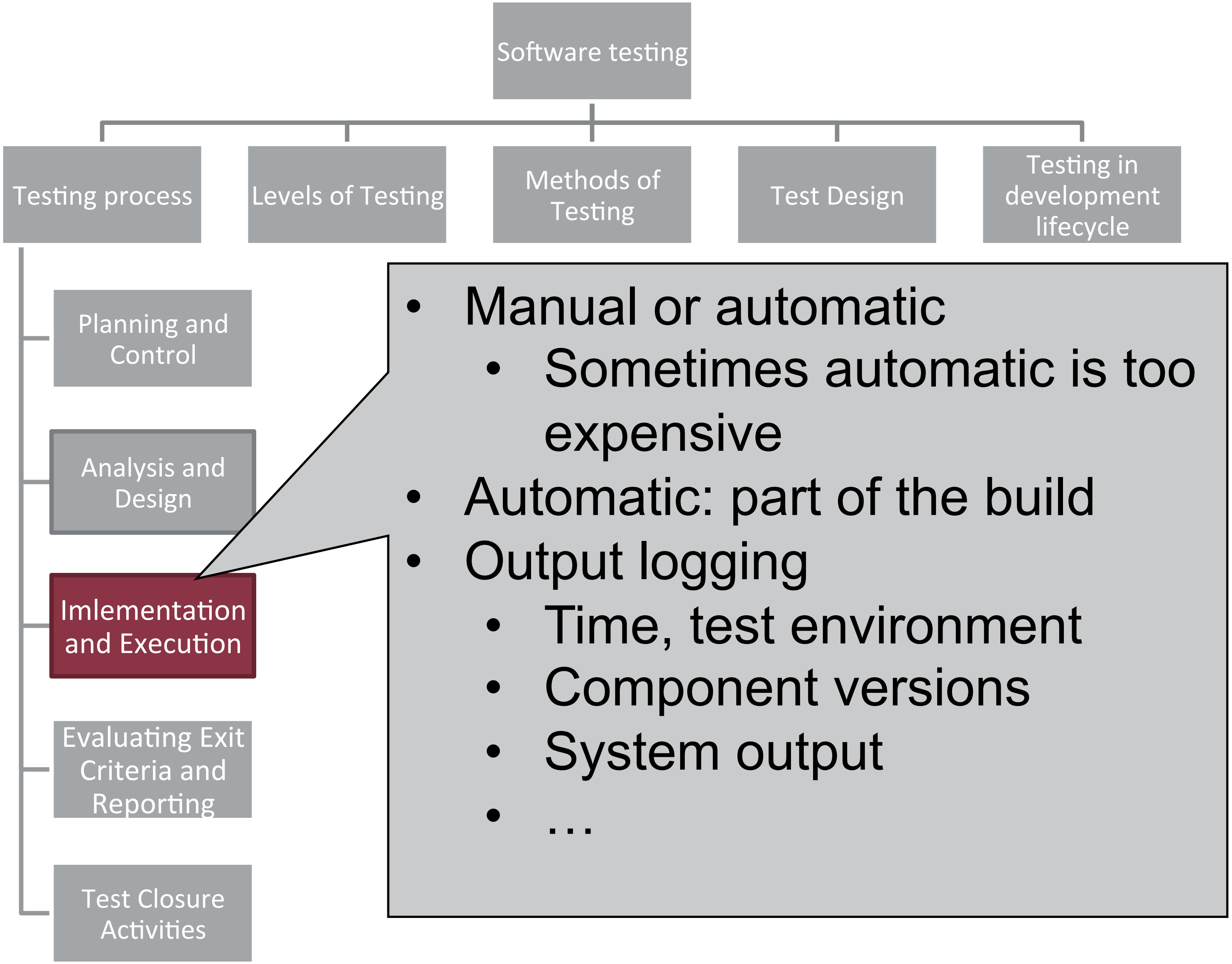
- General policies
  - Methodology
  - Test types
  - Test tools
  - Who tests
  - Exit criteria
  - Testing documentation
  - …
- E.g.:
  - Extreme programming
  - Module & system
  - JUnit & GUI Tester
  - Developers and test team
  - 90% code coverage & 100% use case coveragy

Test Suite Evaluation

- Coverage
  - Code
  - Specification
- Output distribution
- Cost!



- Test case planning/specification
  - Goal of test case
  - Test environment
  - Test steps, test data
  - Expected outcome
- Before test code is created
- Systematic method recommended



Software testing

Testing process

Levels of Testing

Methods of Testing

Test Design

Testing in development lifecycle

Planning and Control

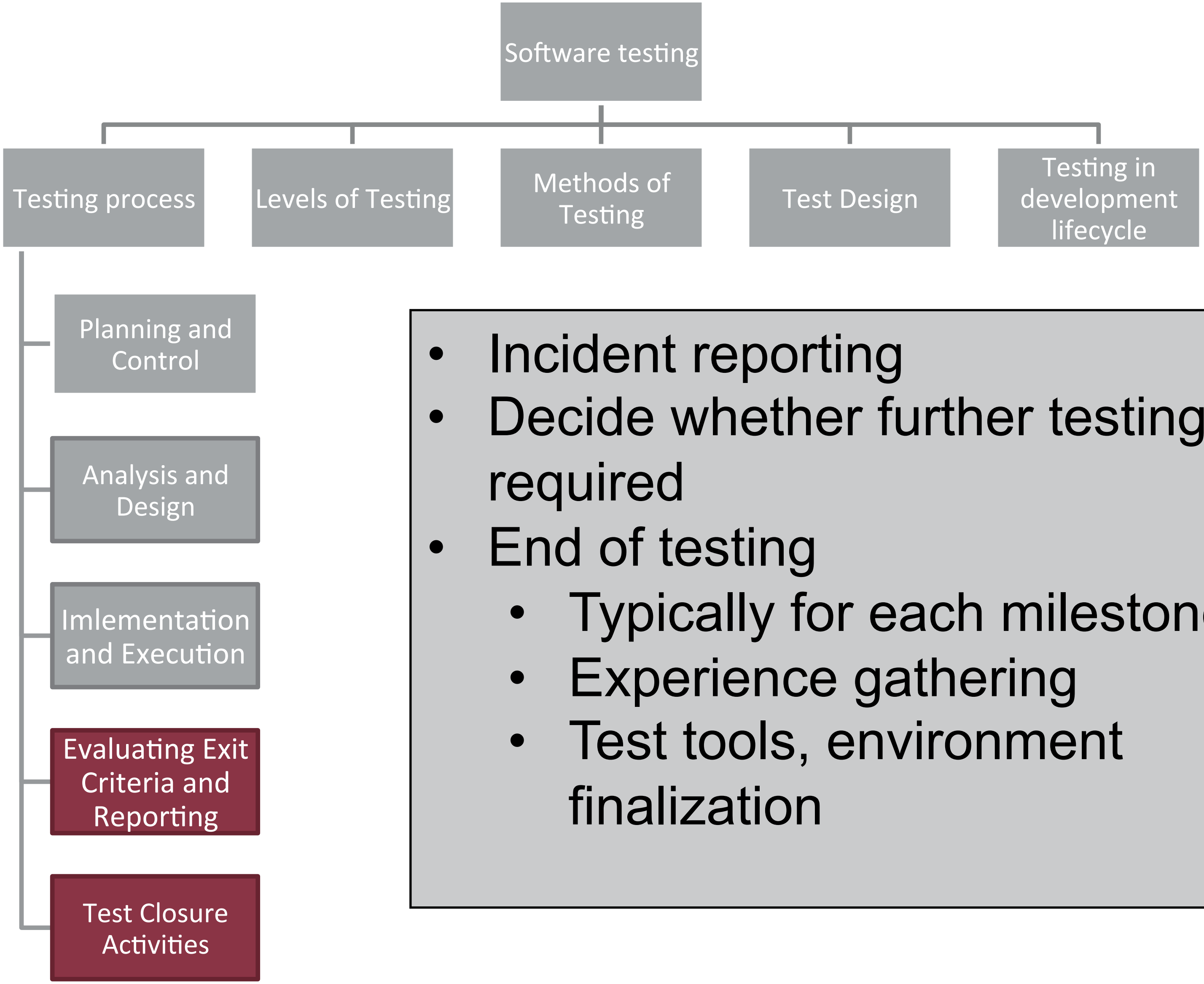
Analysis and Design

Imlementation and Execution

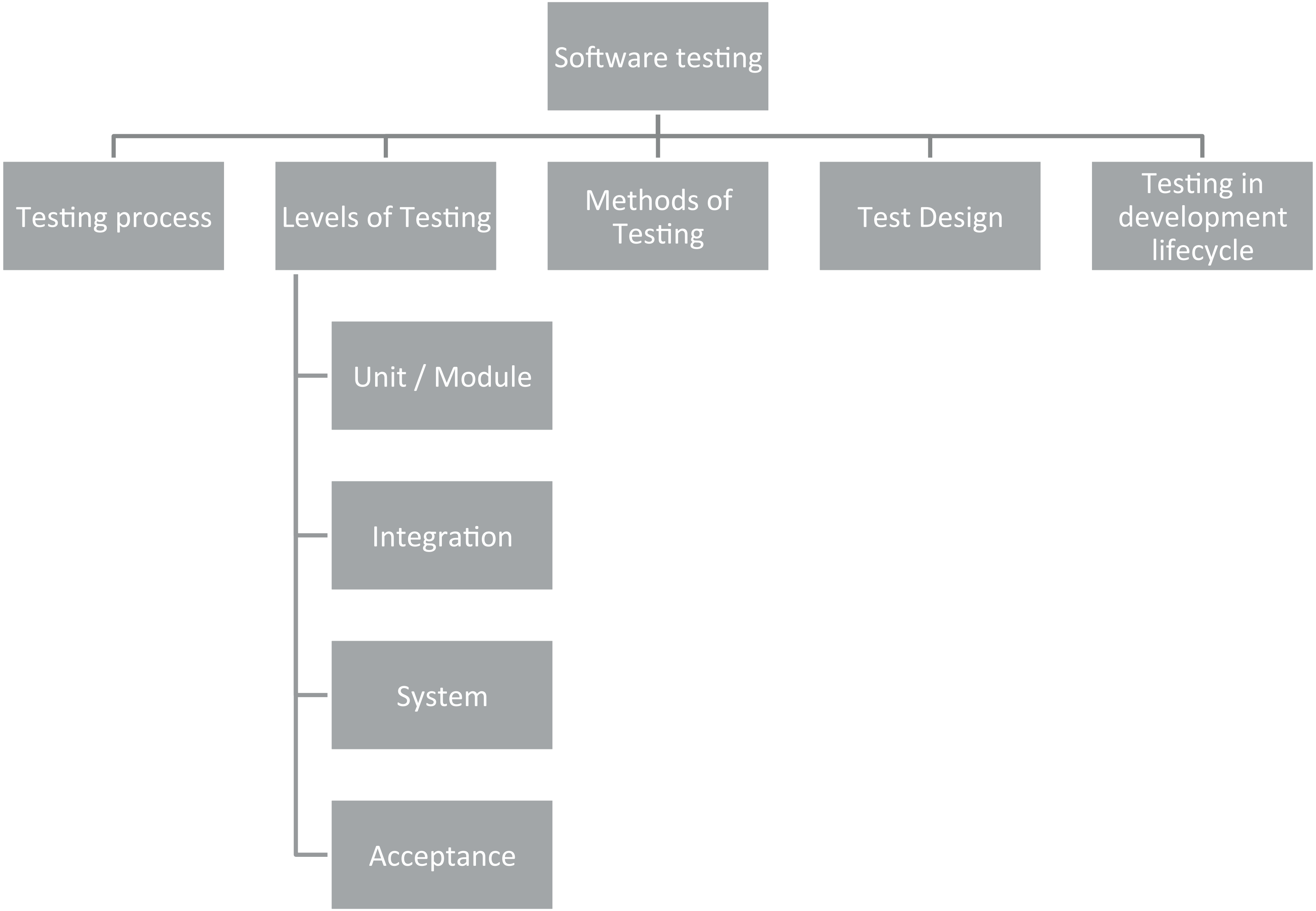
Evaluating Exit Criteria and Reporting

Test Closure Activities

- Manual or automatic
  - Sometimes automatic is too expensive
- Automatic: part of the build
- Output logging
  - Time, test environment
  - Component versions
  - System output
  - …



- Incident reporting
- Decide whether further testing required
- End of testing
  - Typically for each milestone
  - Experience gathering
  - Test tools, environment finalization


Software testing

Testing process

Levels of Testing

Methods of Testing

Test Design

Testing in development lifecycle

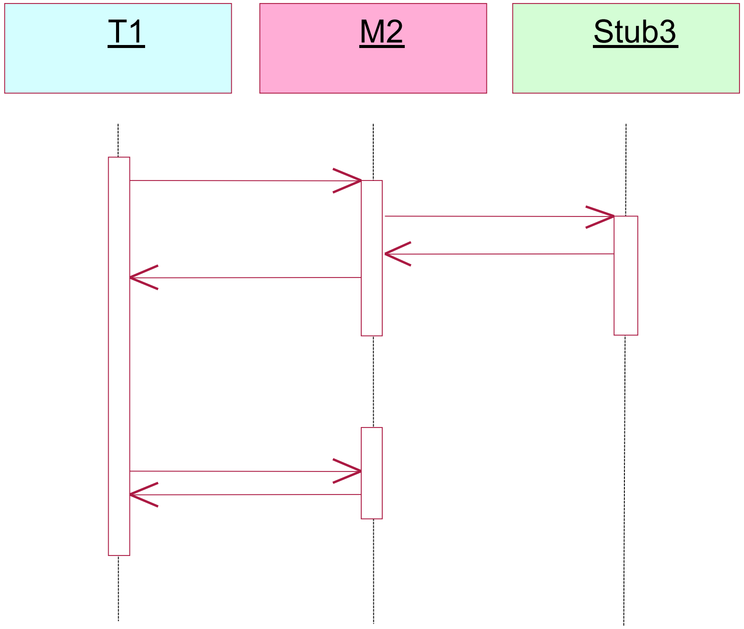
Unit / Module

Integration

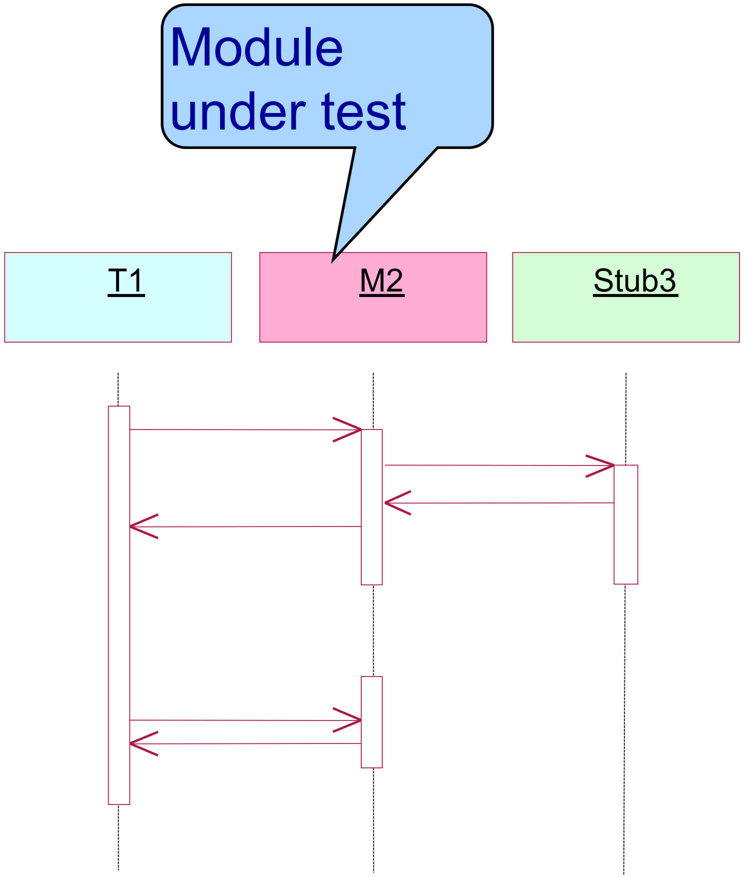
System

Acceptance

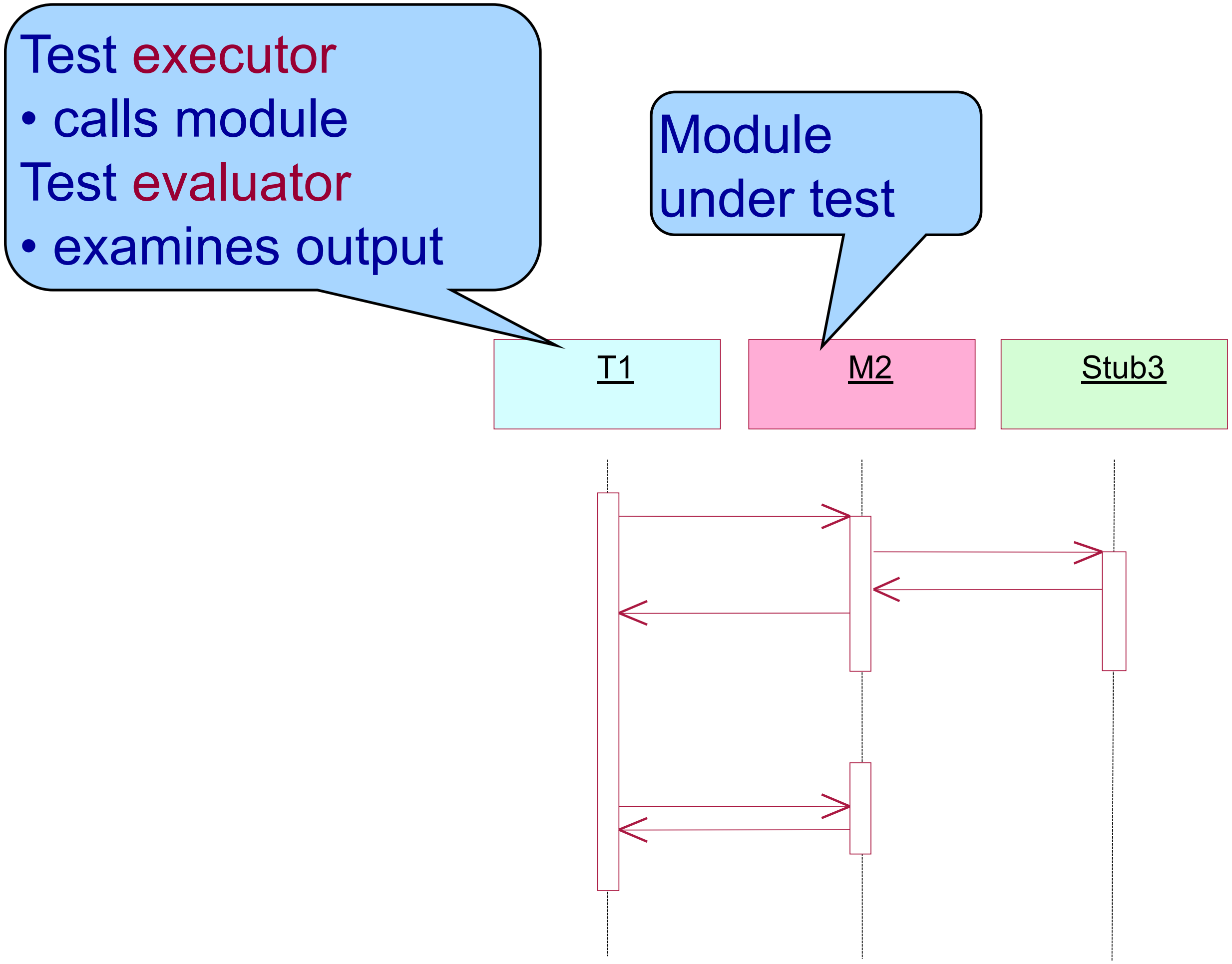
Module Testing



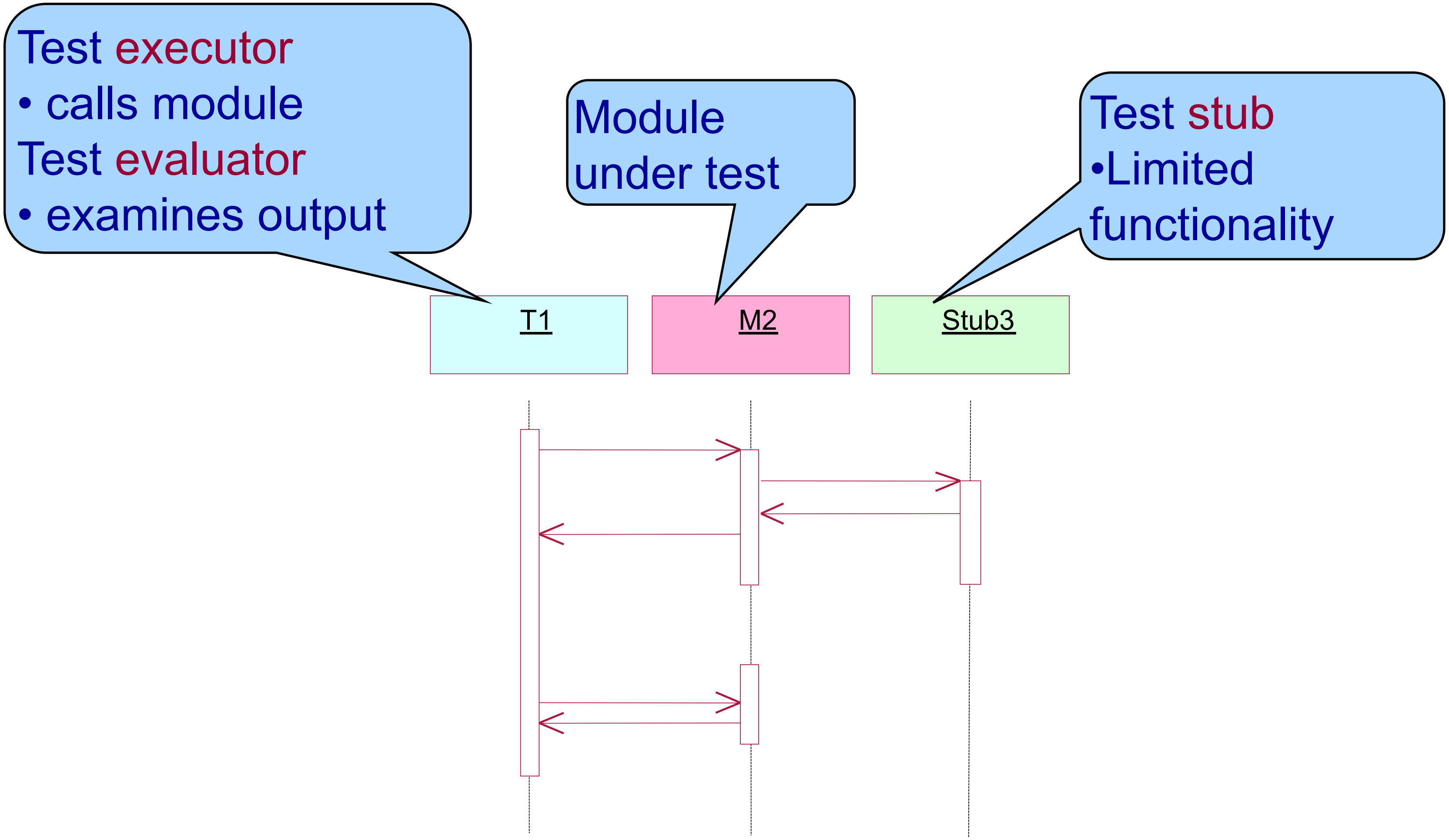
Module Testing



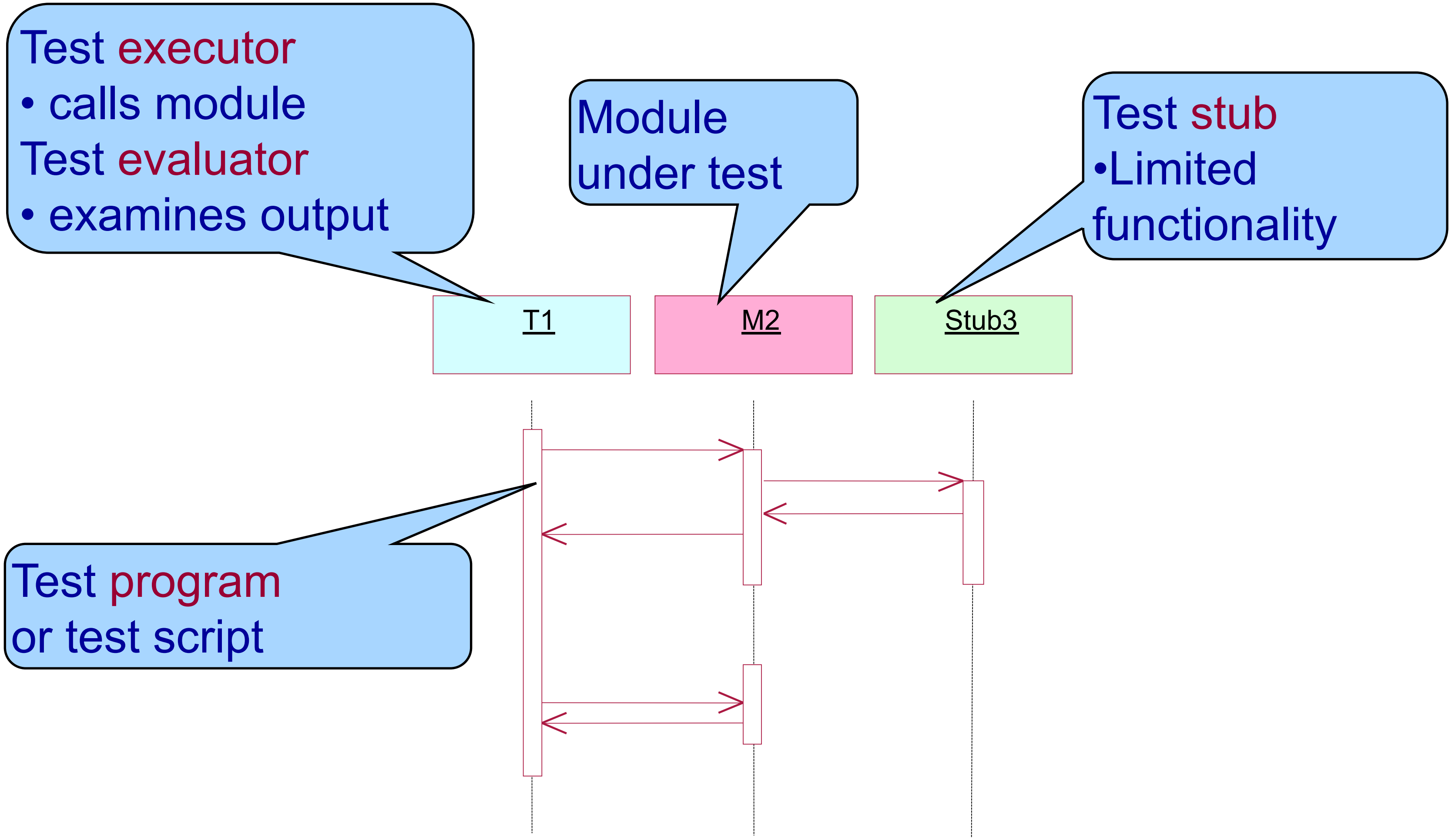
Module Testing



Module Testing



Module Testing



Test executor
- calls module

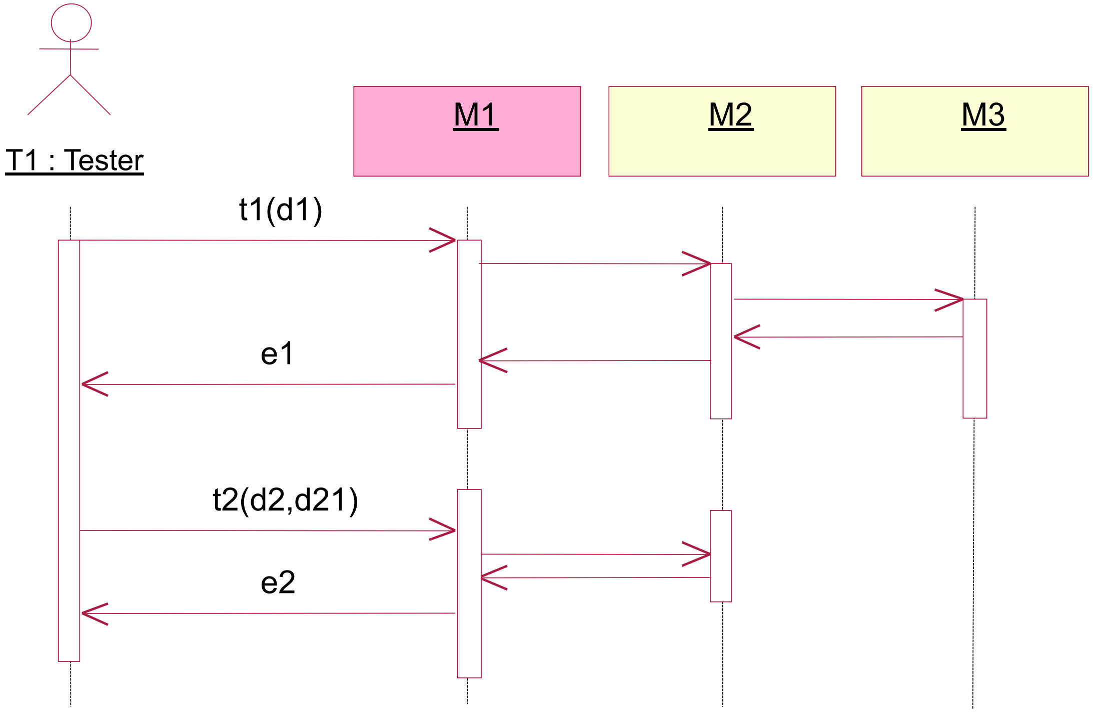
Test evaluator
- examines output

Module under test

Test stub
- Limited functionality

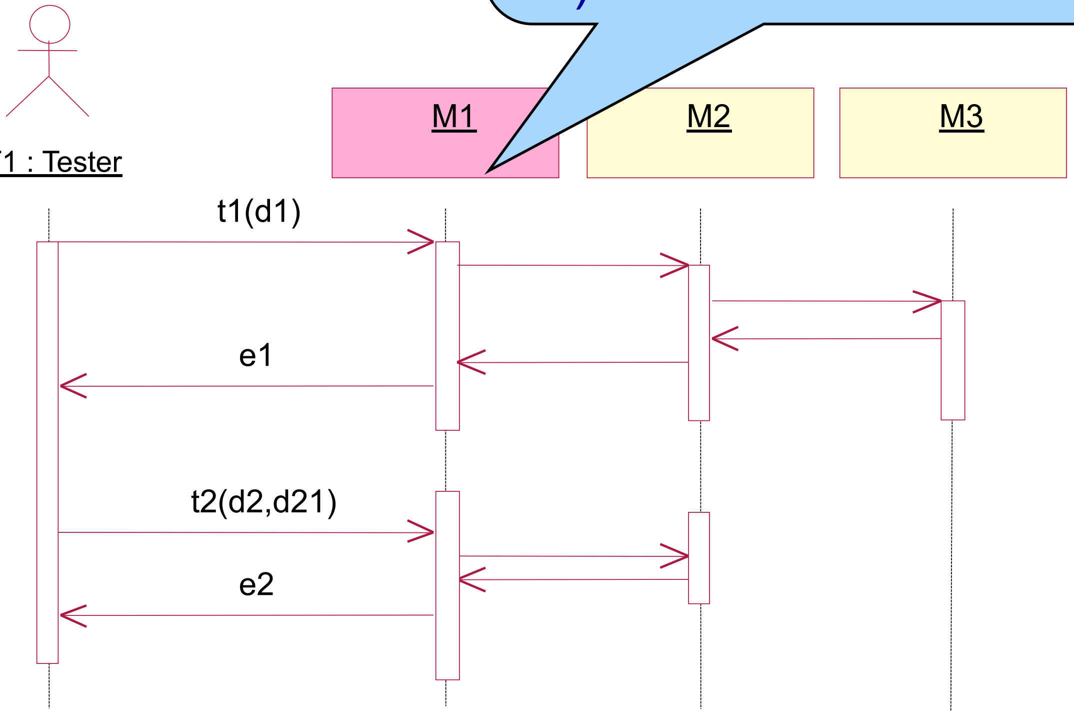
Test program or test script

Integration Testing



Integration Testing

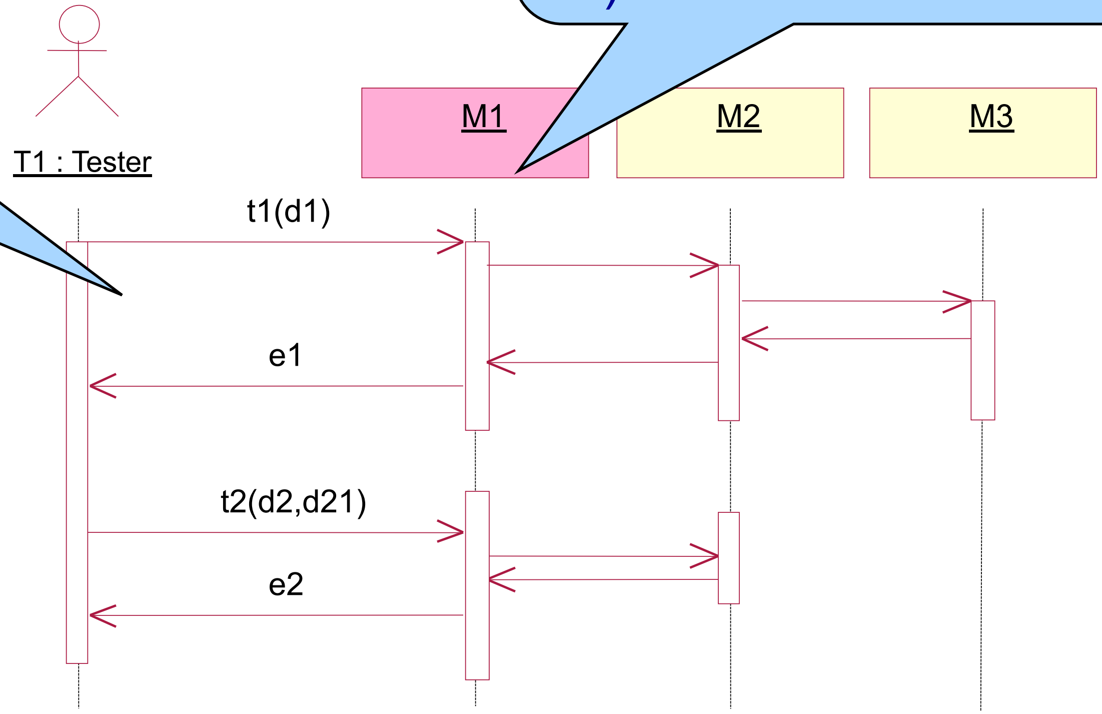
System under test
- Consists of multiple modules (here M1, M2, M3)



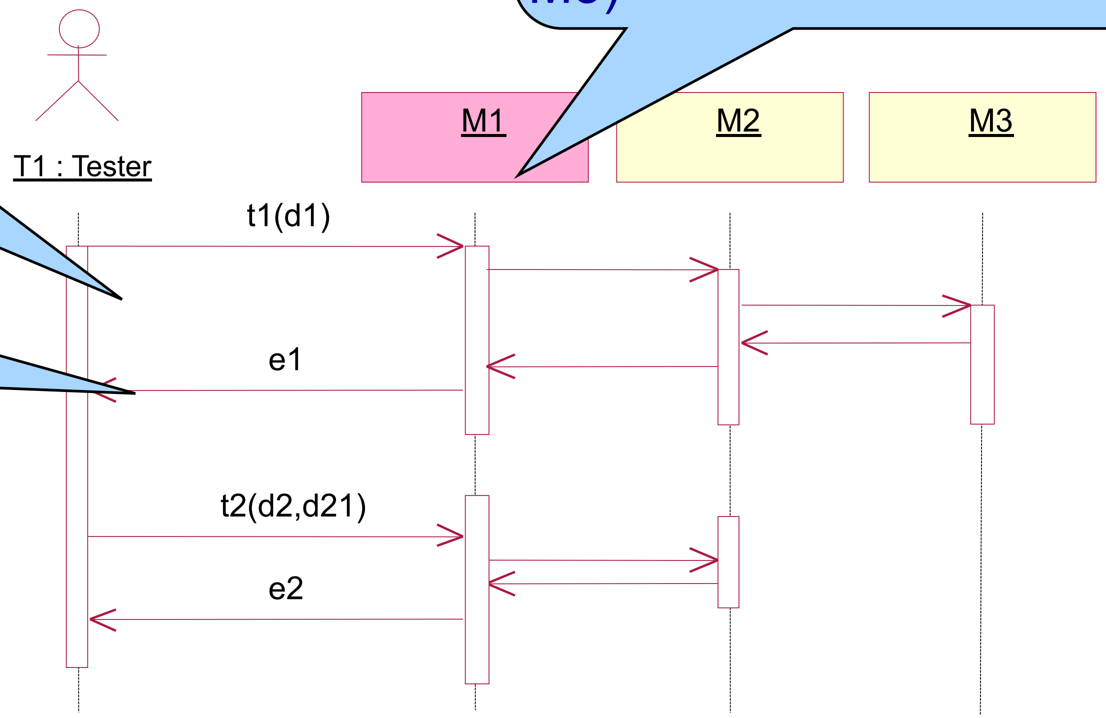
Integration Testing

Test 1
- test input
- expected output

System under test
- Consists of multiple modules (here M1, M2, M3)



Integration Testing



System under test
- Consists of multiple modules (here M1, M2, M3)

Test 1
- test input
- expected output

Test result
- evaluation

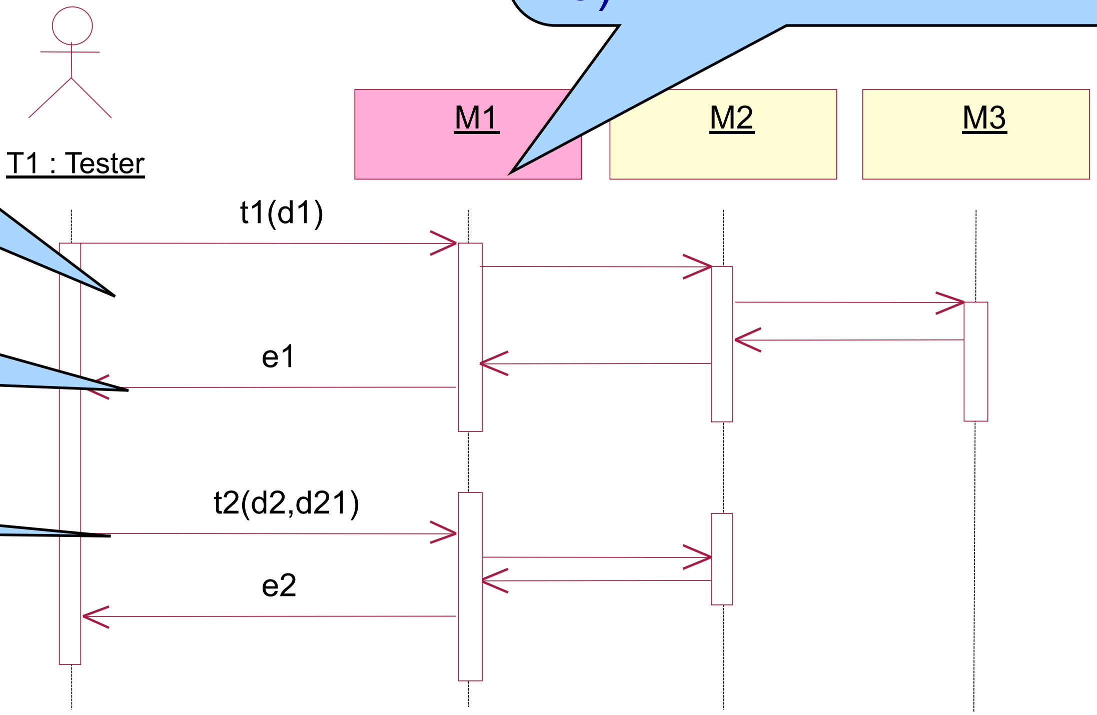
Integration Testing

System under test
- Consists of multiple modules (here M1, M2, M3)

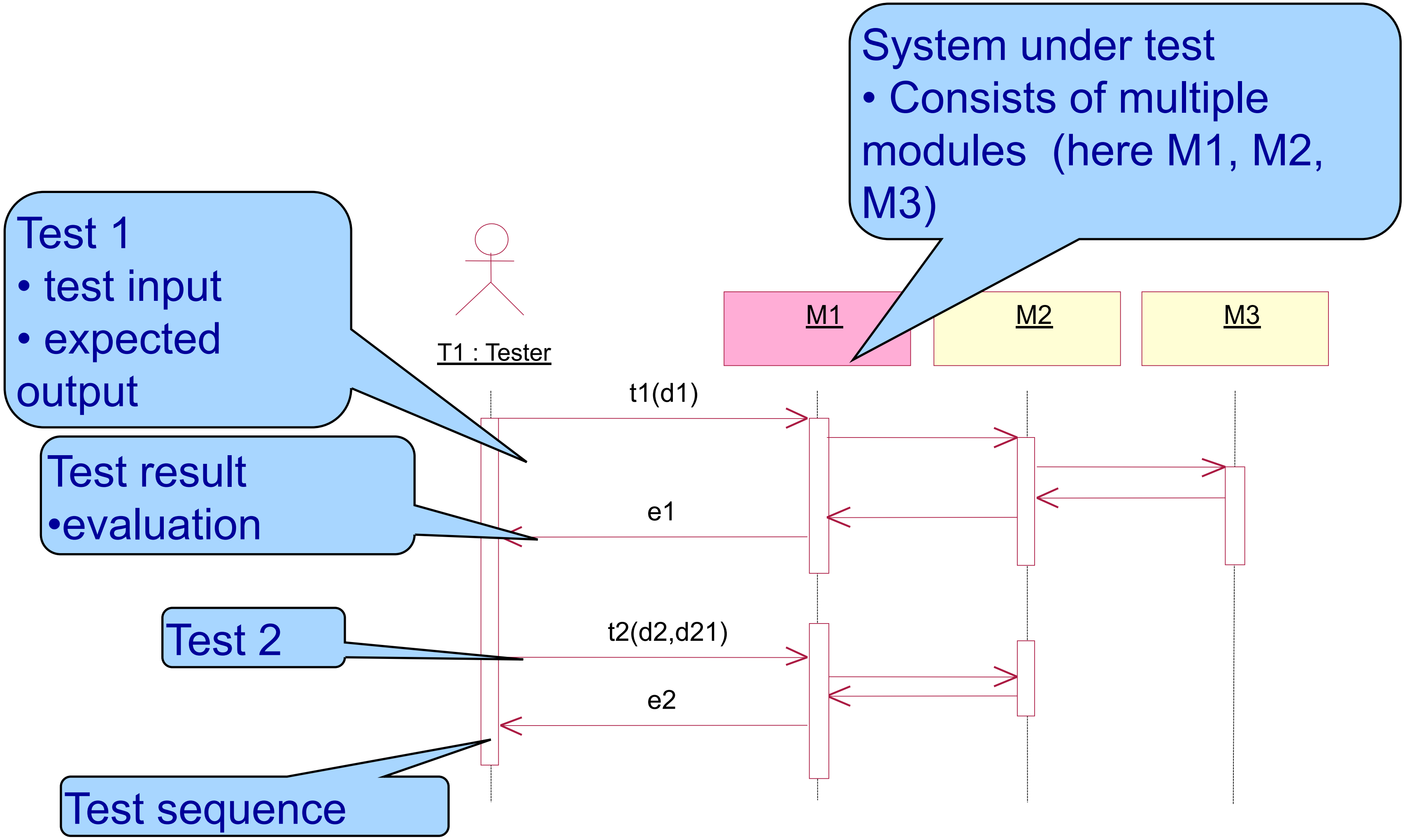
Test 1
- test input
- expected output

Test result
- evaluation

Test 2



Integration Testing



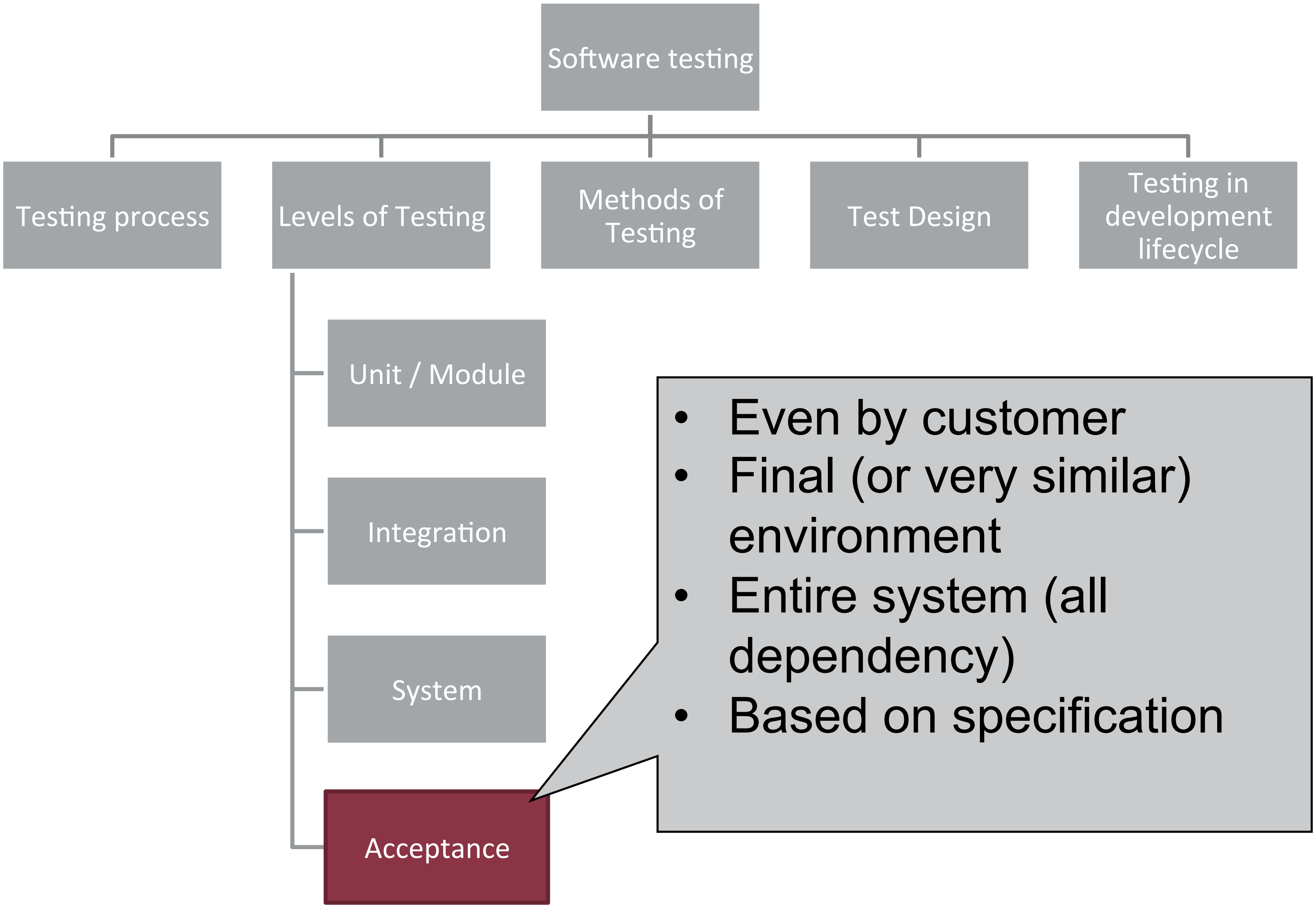
System under test
- Consists of multiple modules (here M1, M2, M3)

Test 1
- test input
- expected output

Test result
•evaluation

Test 2

Test sequence



Software testing

Testing process

Levels of Testing

Methods of Testing

Test Design

Testing in development lifecycle

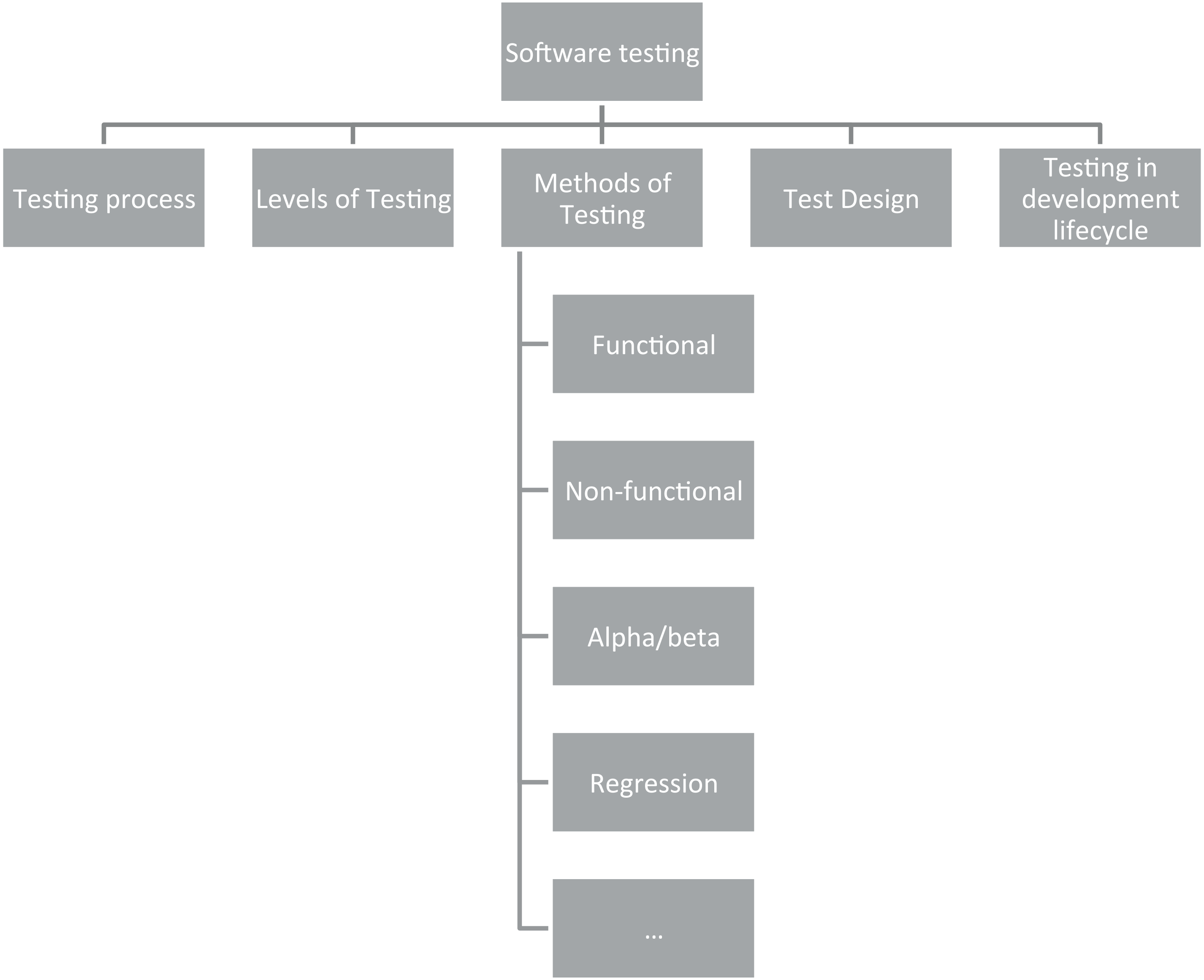
Unit / Module

Integration

System

Acceptance

- Even by customer
- Final (or very similar) environment
- Entire system (all dependency)
- Based on specification



Software testing

Testing process

Levels of Testing

Methods of Testing

Test Design

Testing in development lifecycle

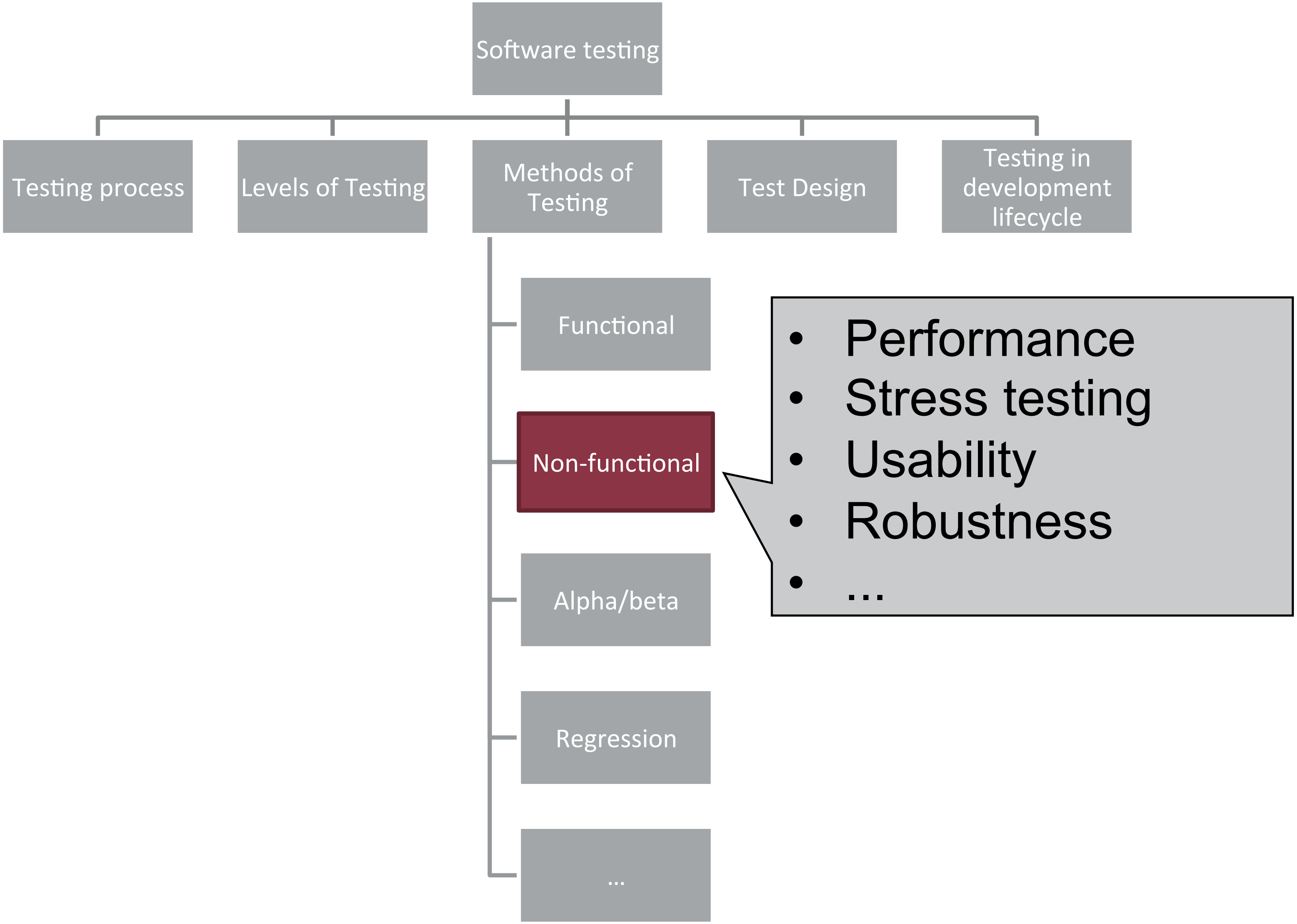
Functional

Non-functional

Alpha/beta

Regression

…



Software testing

Testing process

Levels of Testing

Methods of Testing

Test Design

Testing in development lifecycle

Functional

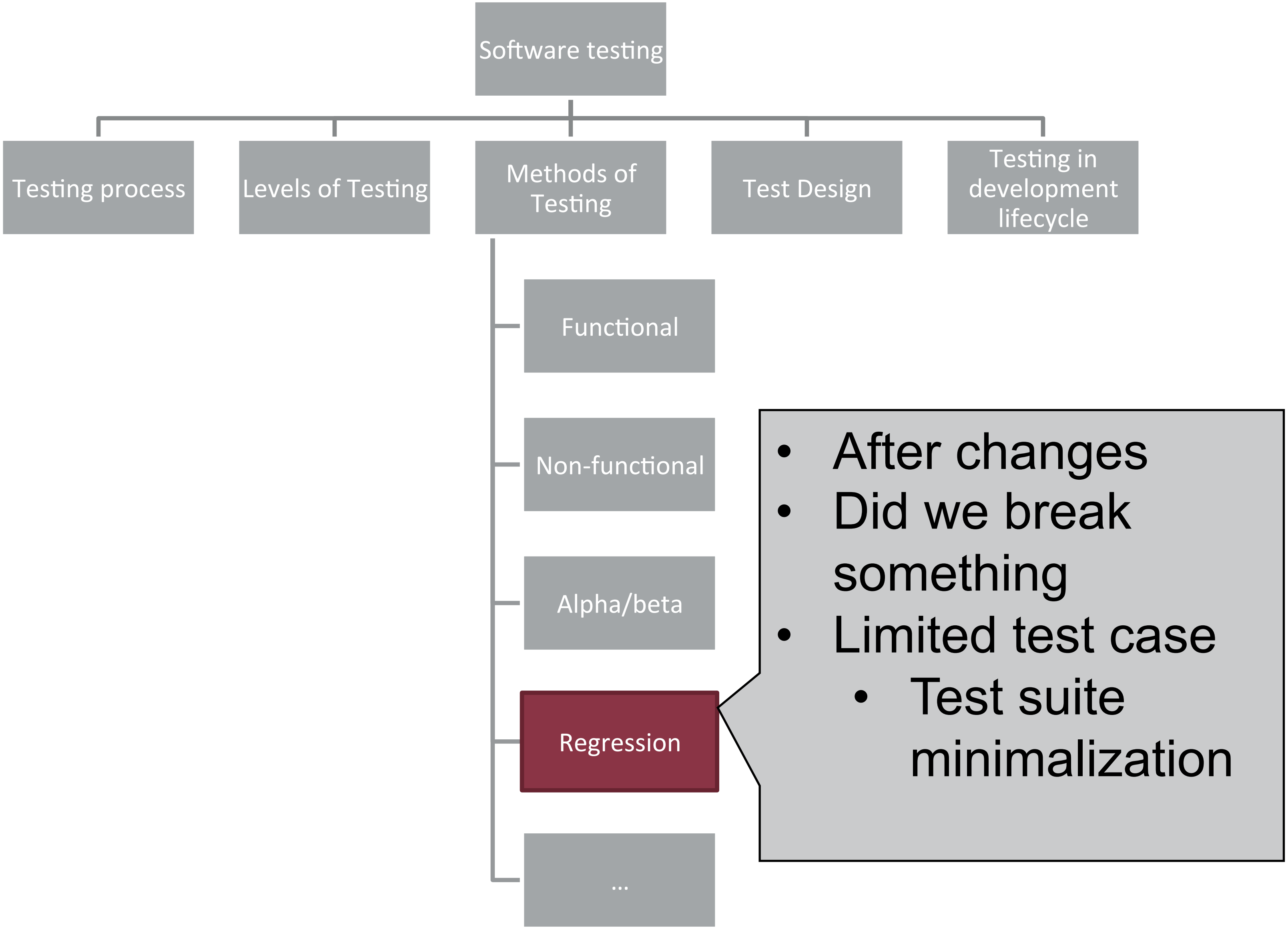
Non-functional

Alpha/beta

Regression

…

- Performance
- Stress testing
- Usability
- Robustness
- …



Software testing

Testing process

Levels of Testing

Methods of Testing

Test Design

Testing in development lifecycle

Functional

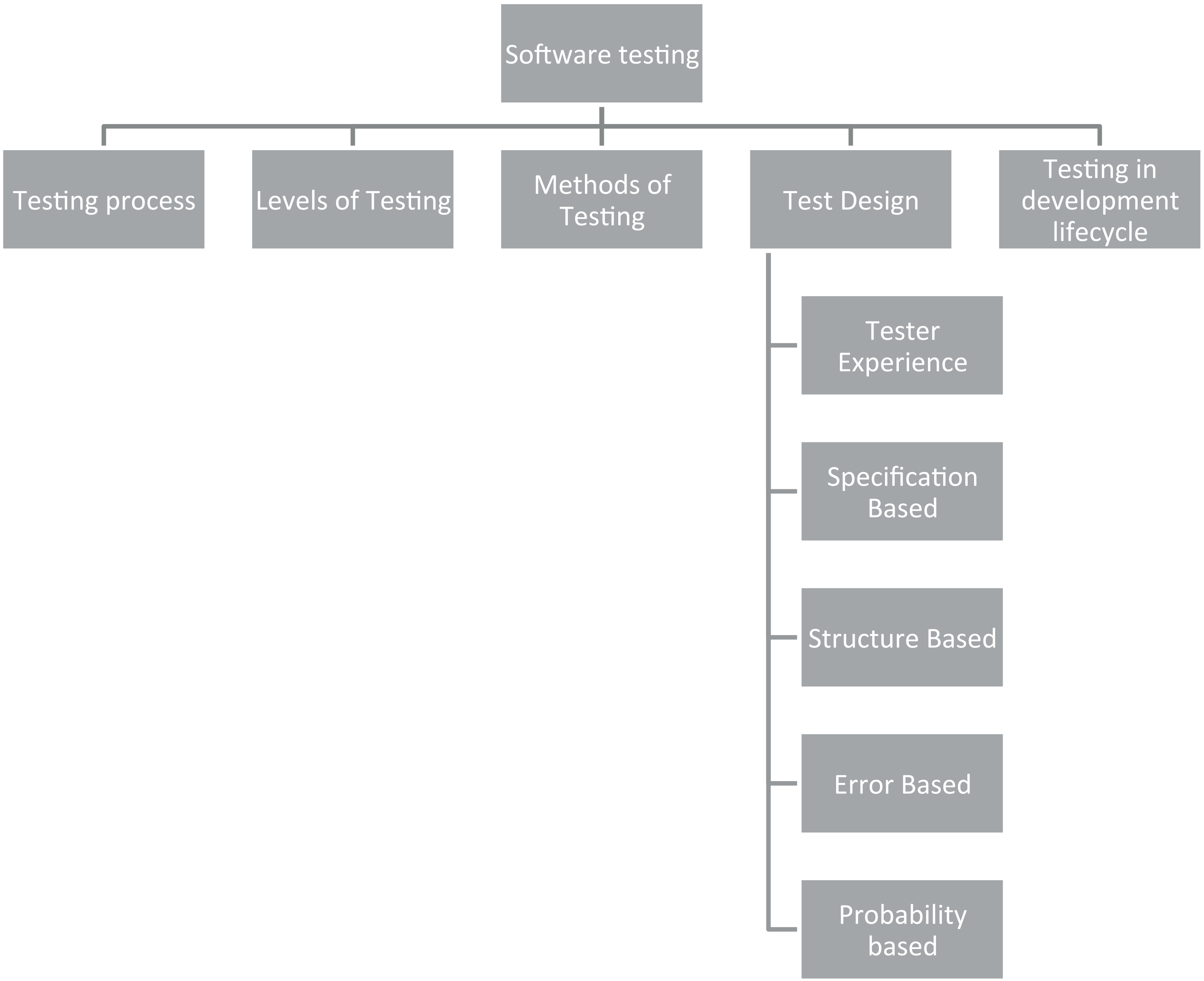
Non-functional

Alpha/beta

Regression

…

- After changes
- Did we break something
- Limited test case
  - Test suite minimalization



Software testing

Testing process

Levels of Testing

Methods of Testing

Test Design

Testing in development lifecycle

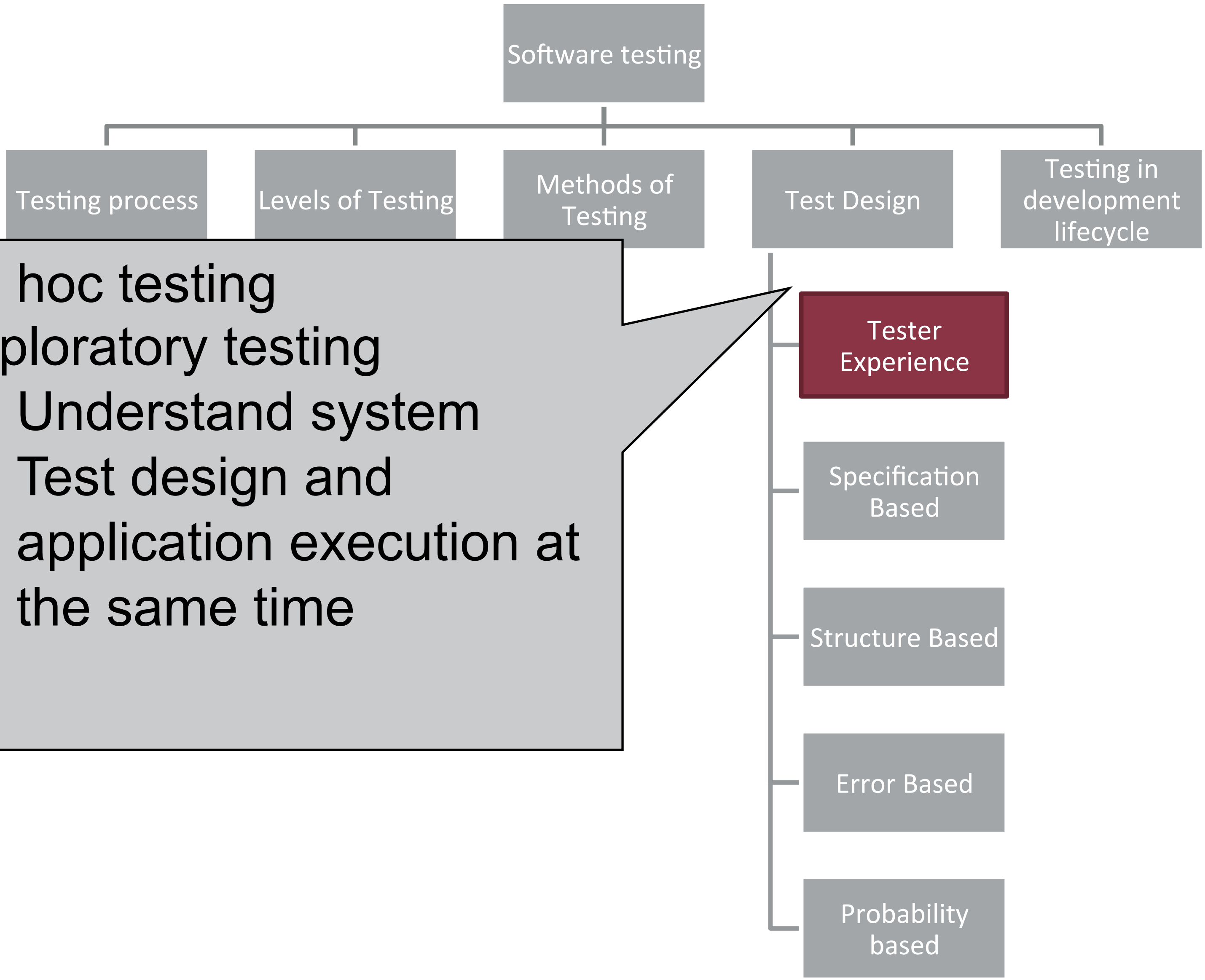
Tester Experience

Specification Based

Structure Based

Error Based

Probability based



Software testing

Testing process

Levels of Testing

Methods of Testing

Test Design

Testing in development lifecycle

Tester Experience

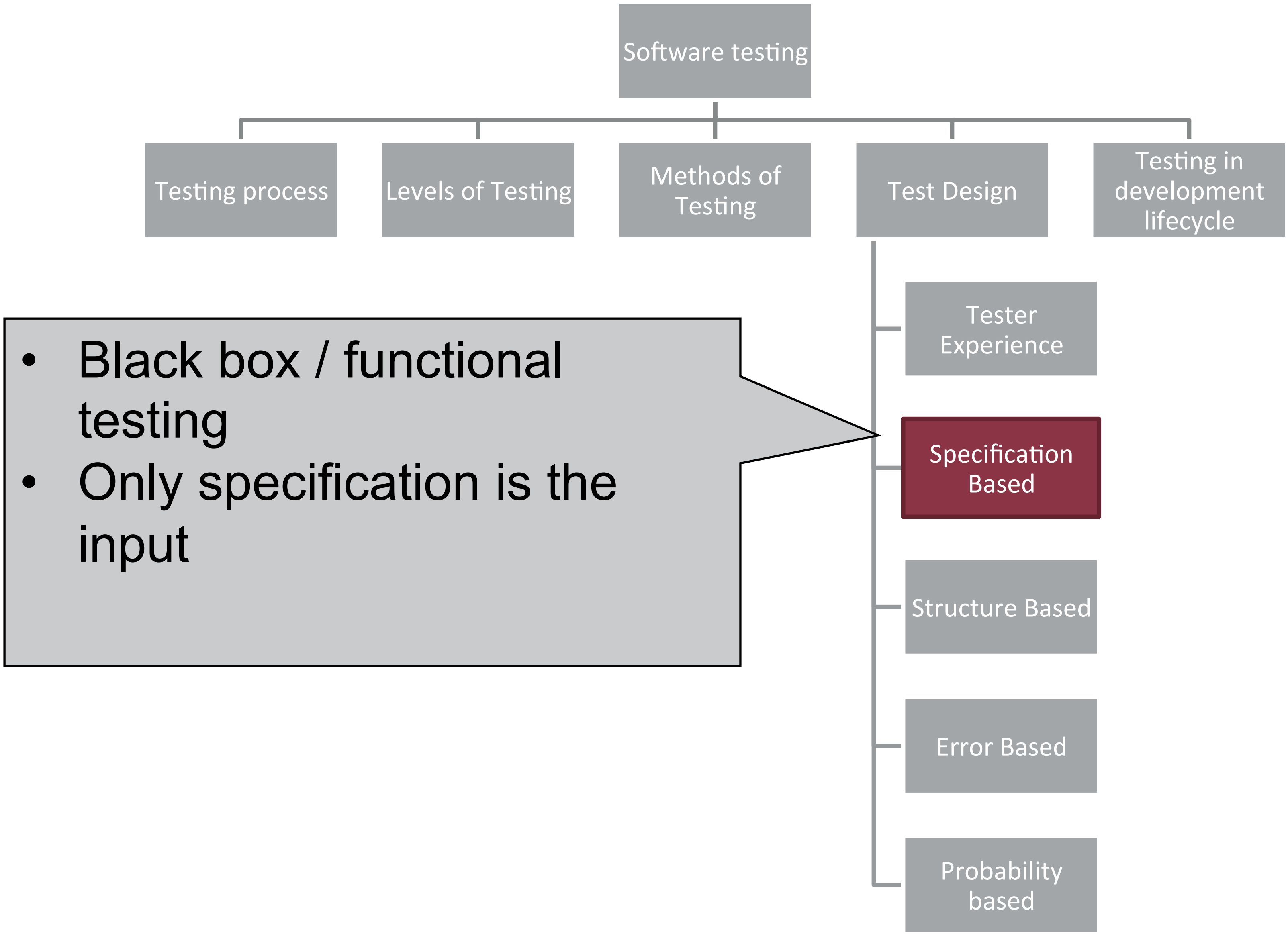
Specification Based

Structure Based

Error Based

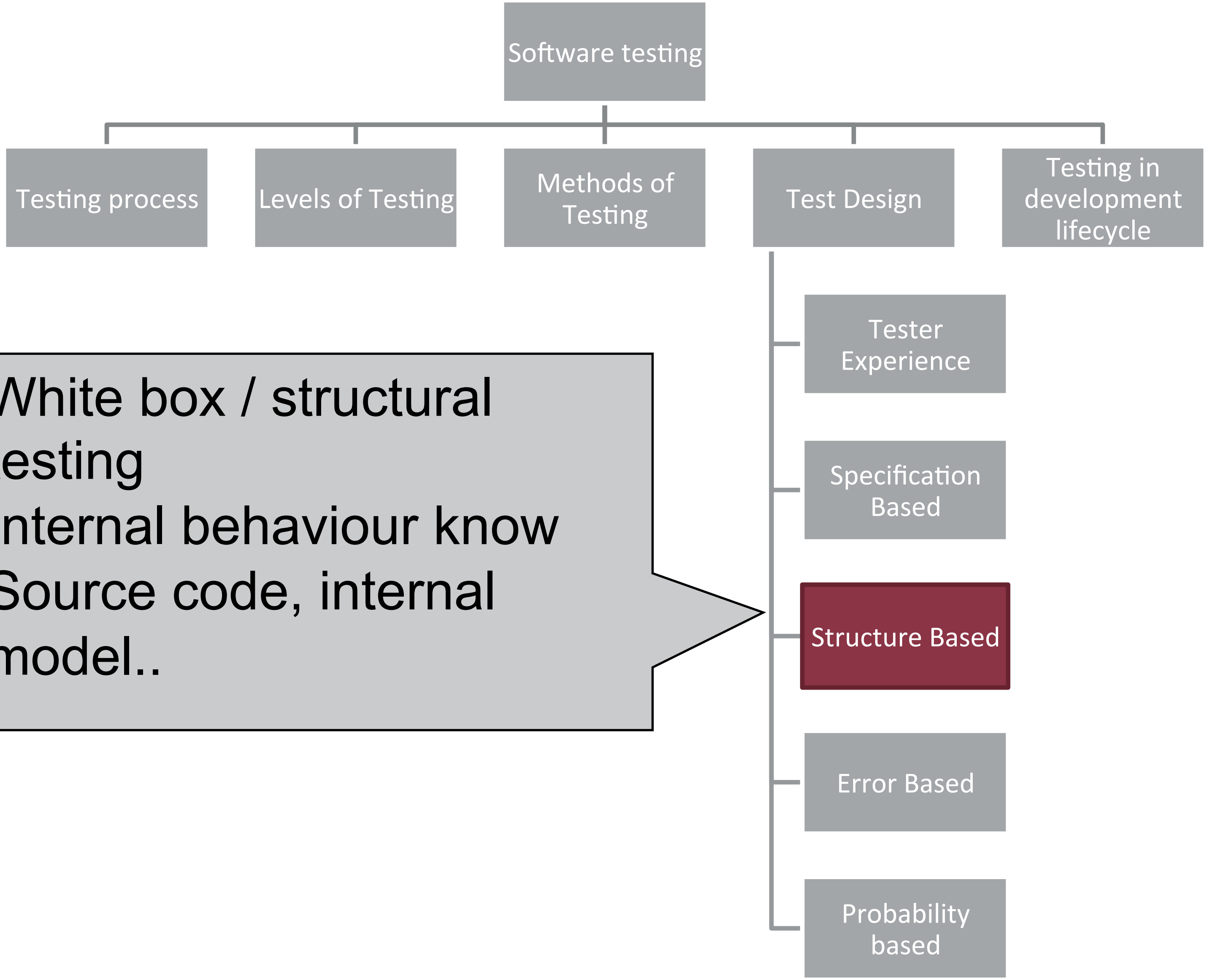
Probability based

- Ad hoc testing
- Exploratory testing
  - Understand system
  - Test design and application execution at the same time

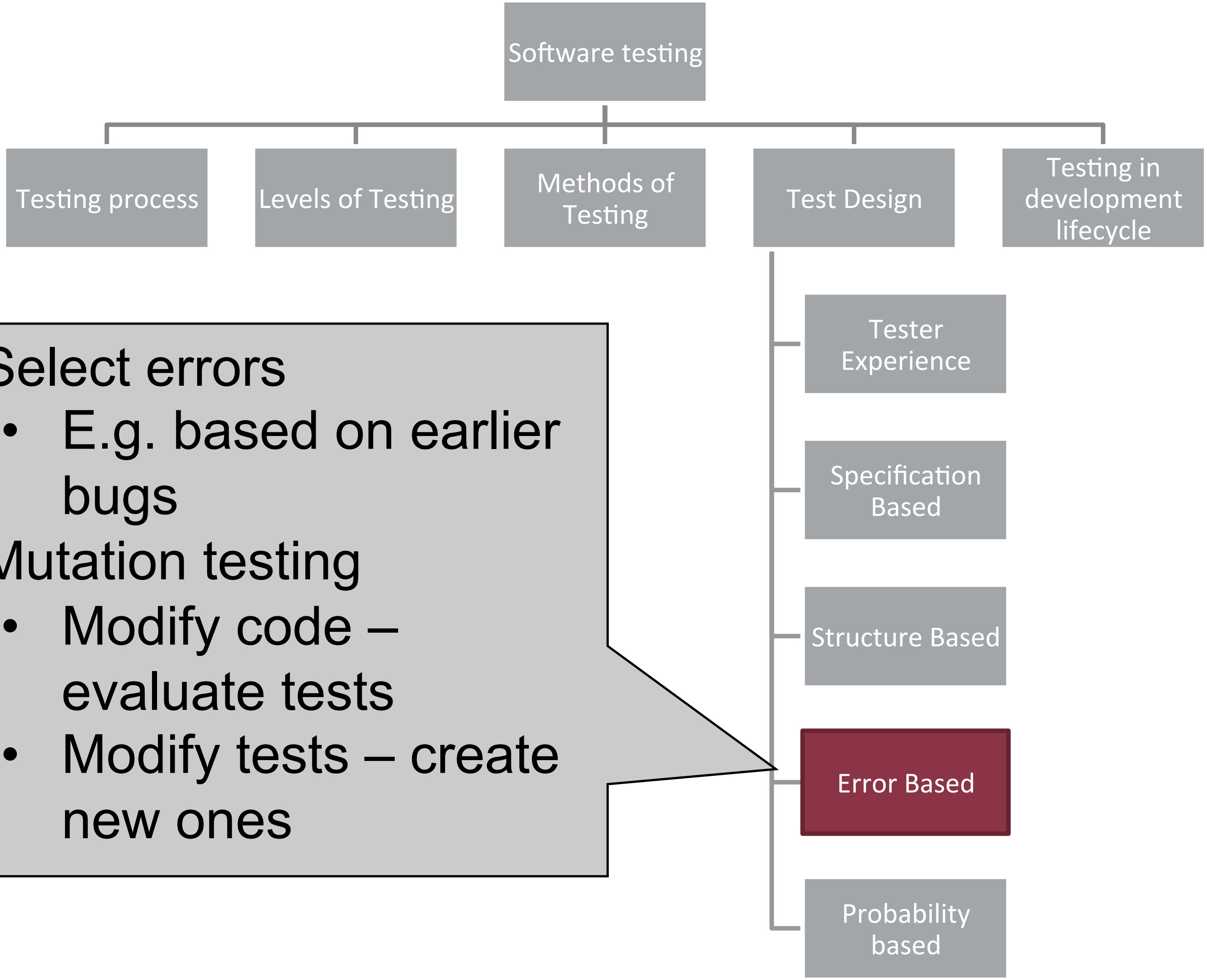


- Software testing
  - Testing process
  - Levels of Testing
  - Methods of Testing
  - Test Design
    - Tester Experience
    - Specification Based
    - Structure Based
    - Error Based
    - Probability based
  - Testing in development lifecycle

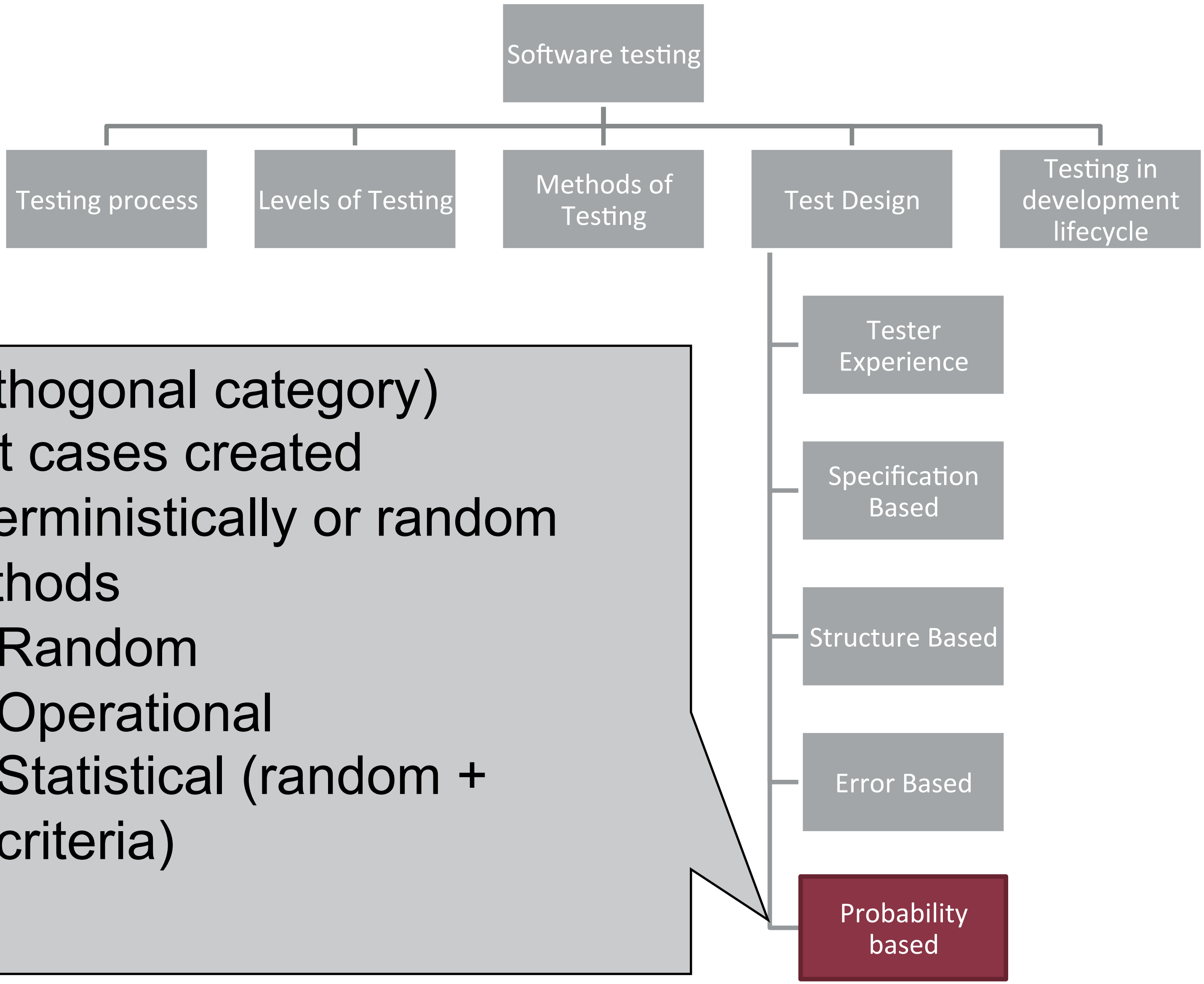
- Black box / functional testing
- Only specification is the input



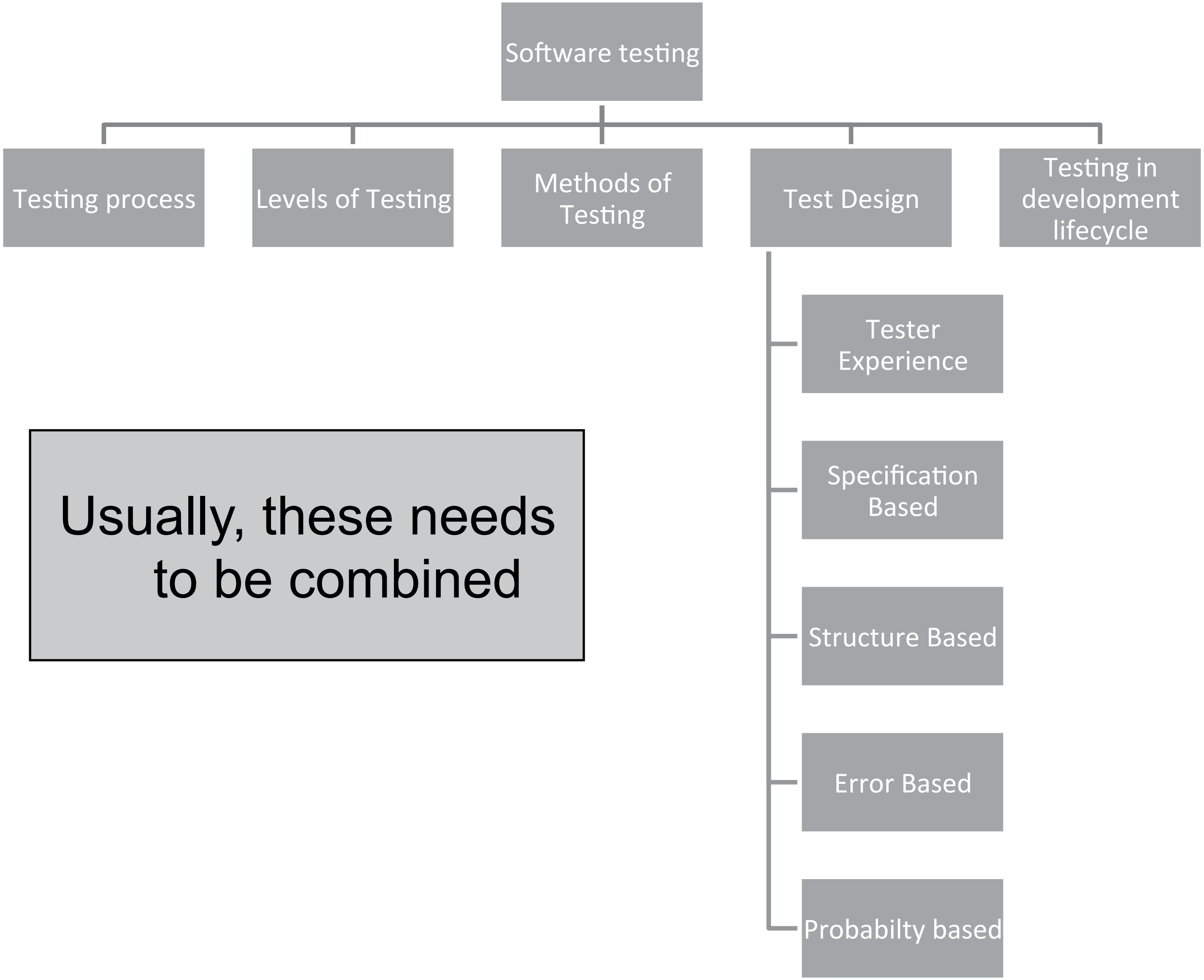
- White box / structural testing
- Internal behaviour know
- Source code, internal model..



- Select errors
  - E.g. based on earlier bugs
- Mutation testing
  - Modify code – evaluate tests
  - Modify tests – create new ones

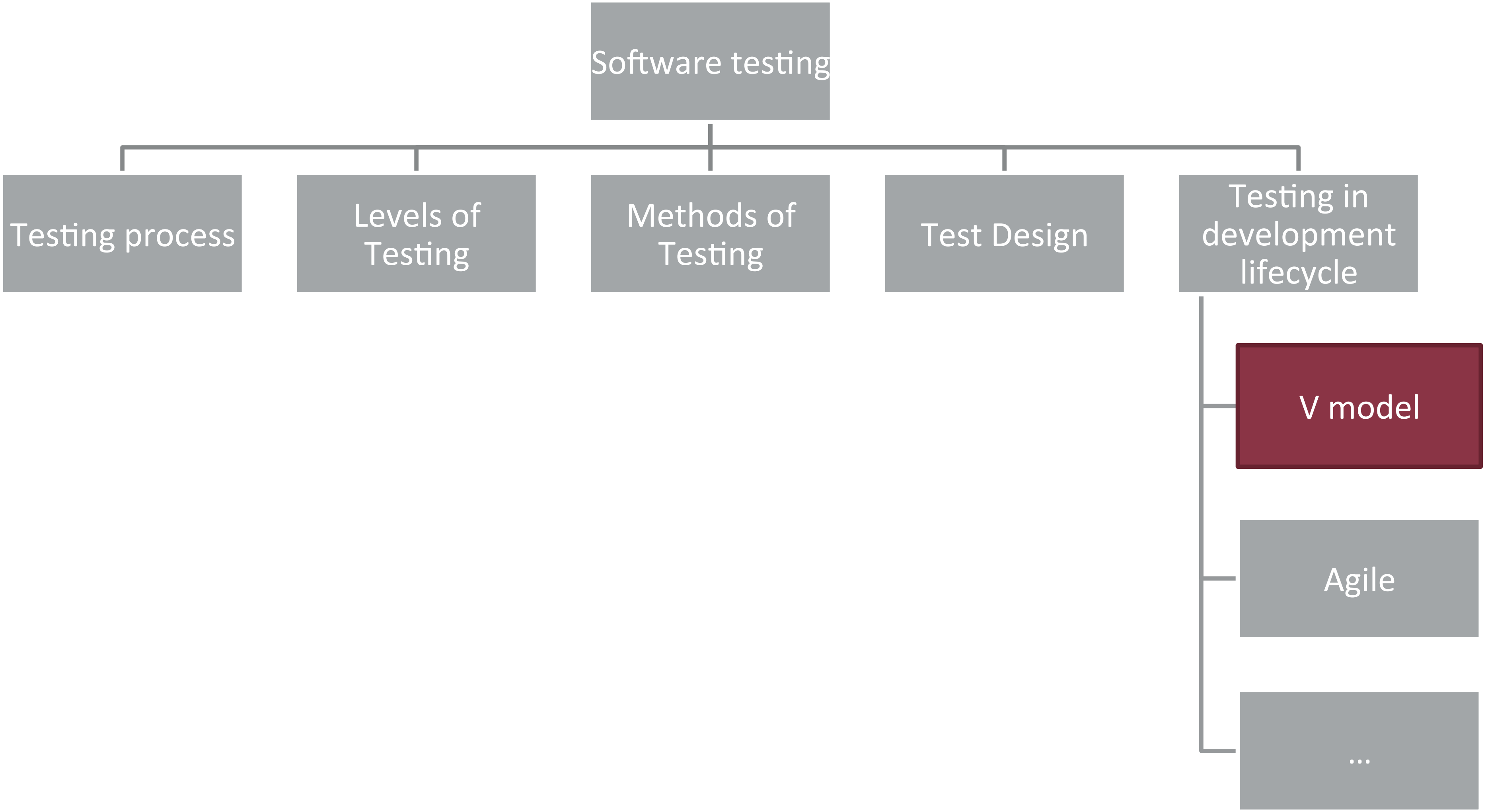


- (Orthogonal category)
- Test cases created deterministically or random
- Methods
  - Random
  - Operational
  - Statistical (random + criteria)

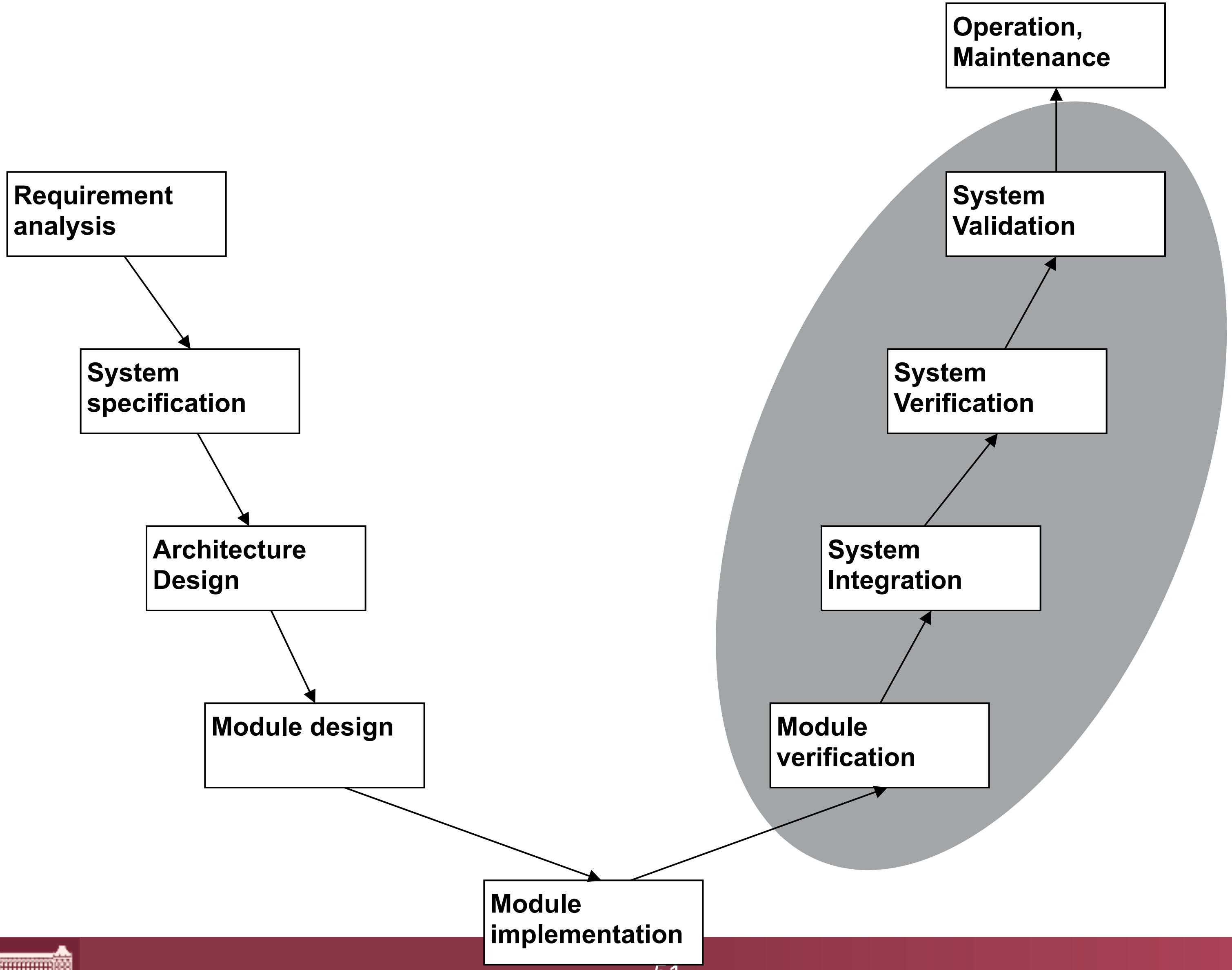


- Software testing
  - Testing process
  - Levels of Testing
  - Methods of Testing
  - Test Design
    - Tester Experience
    - Specification Based
    - Structure Based
    - Error Based
    - Probabilty based
  - Testing in development lifecycle

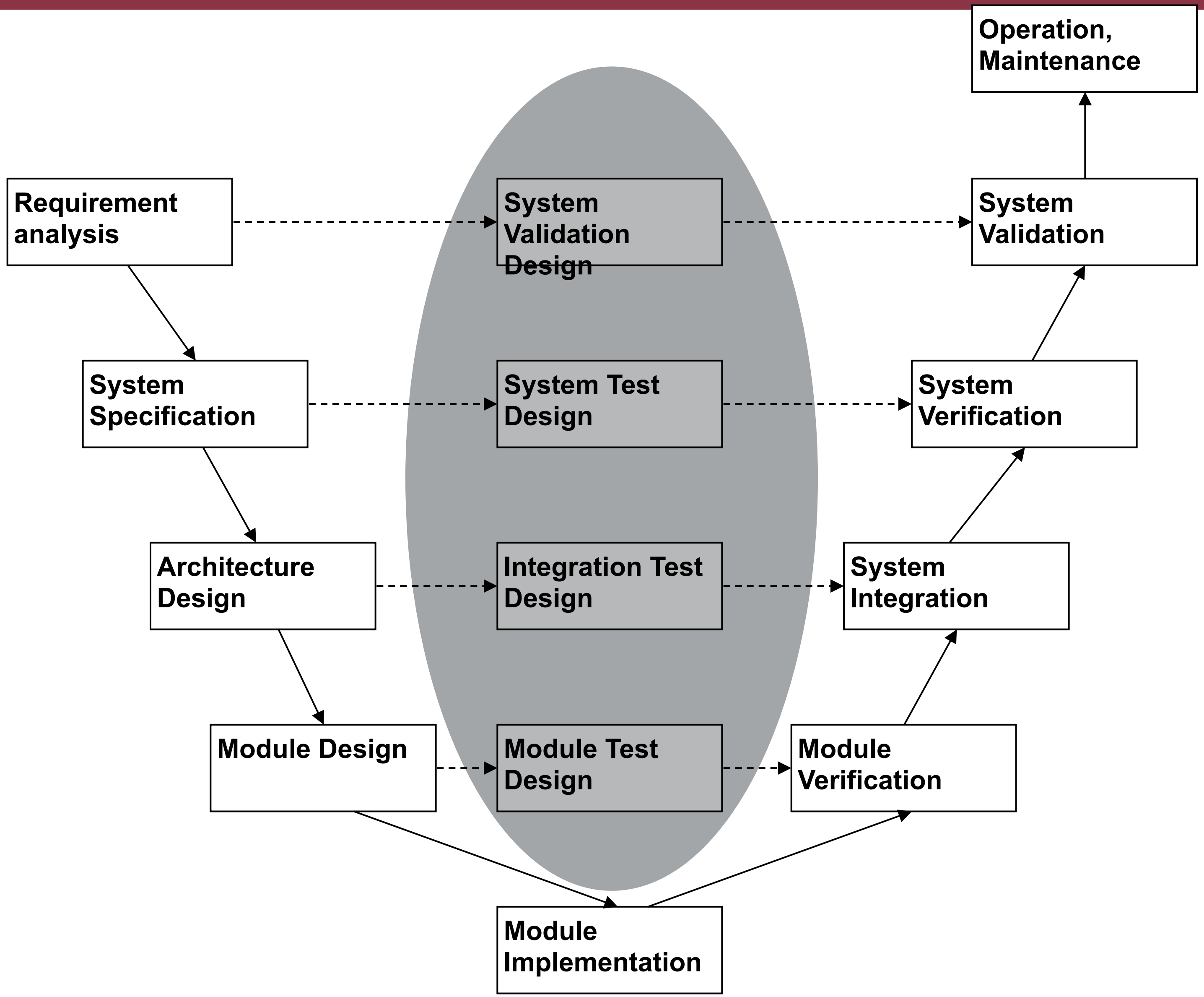
Usually, these needs to be combined

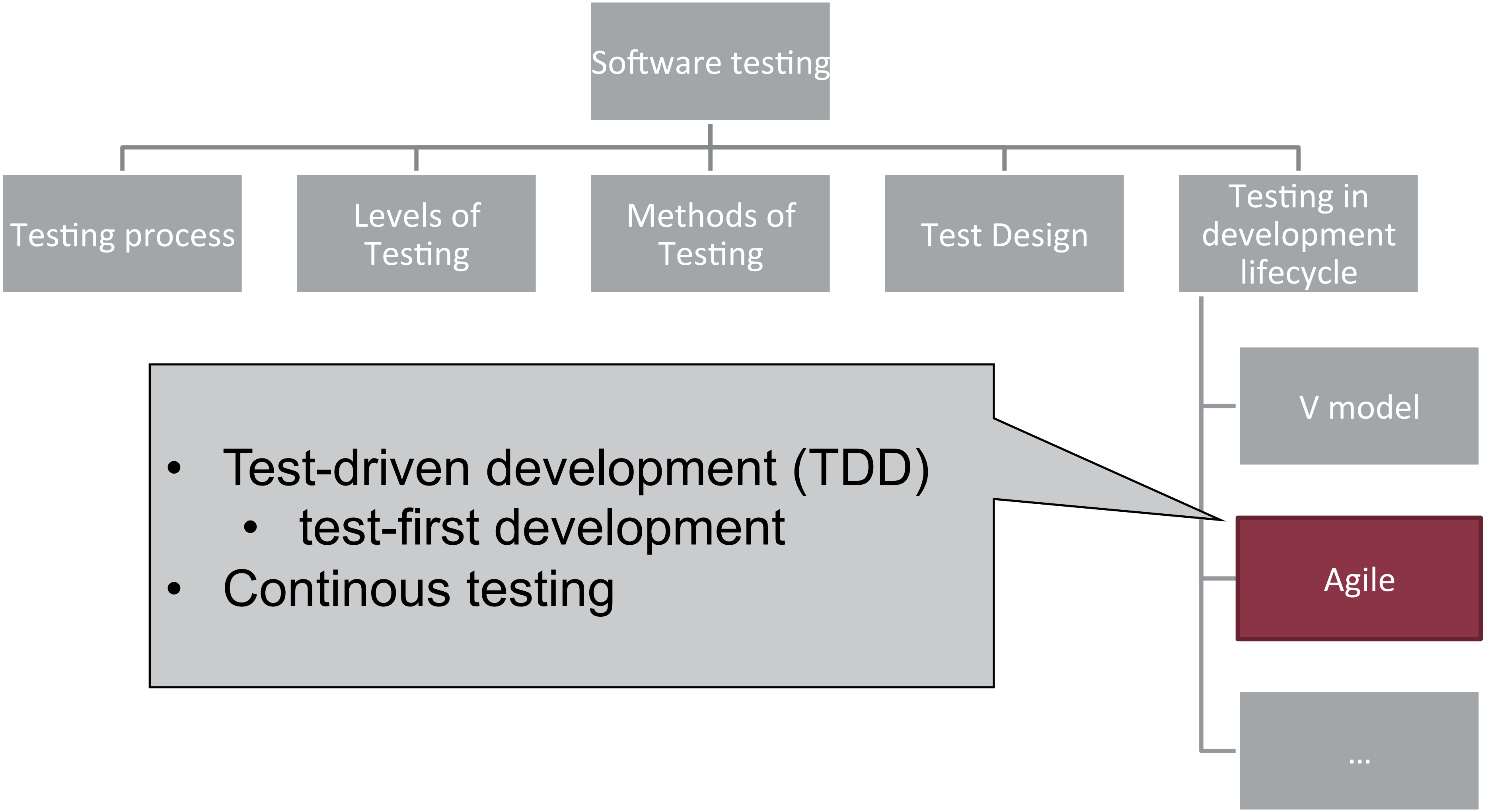

Software testing
Testing process
Levels of Testing
Methods of Testing
Test Design
Testing in development lifecycle
V model
Agile
...

Testing in the V-model



Test Design in the V-model





- Test-driven development (TDD)
  - test-first development
- Continous testing

Testing in Practice

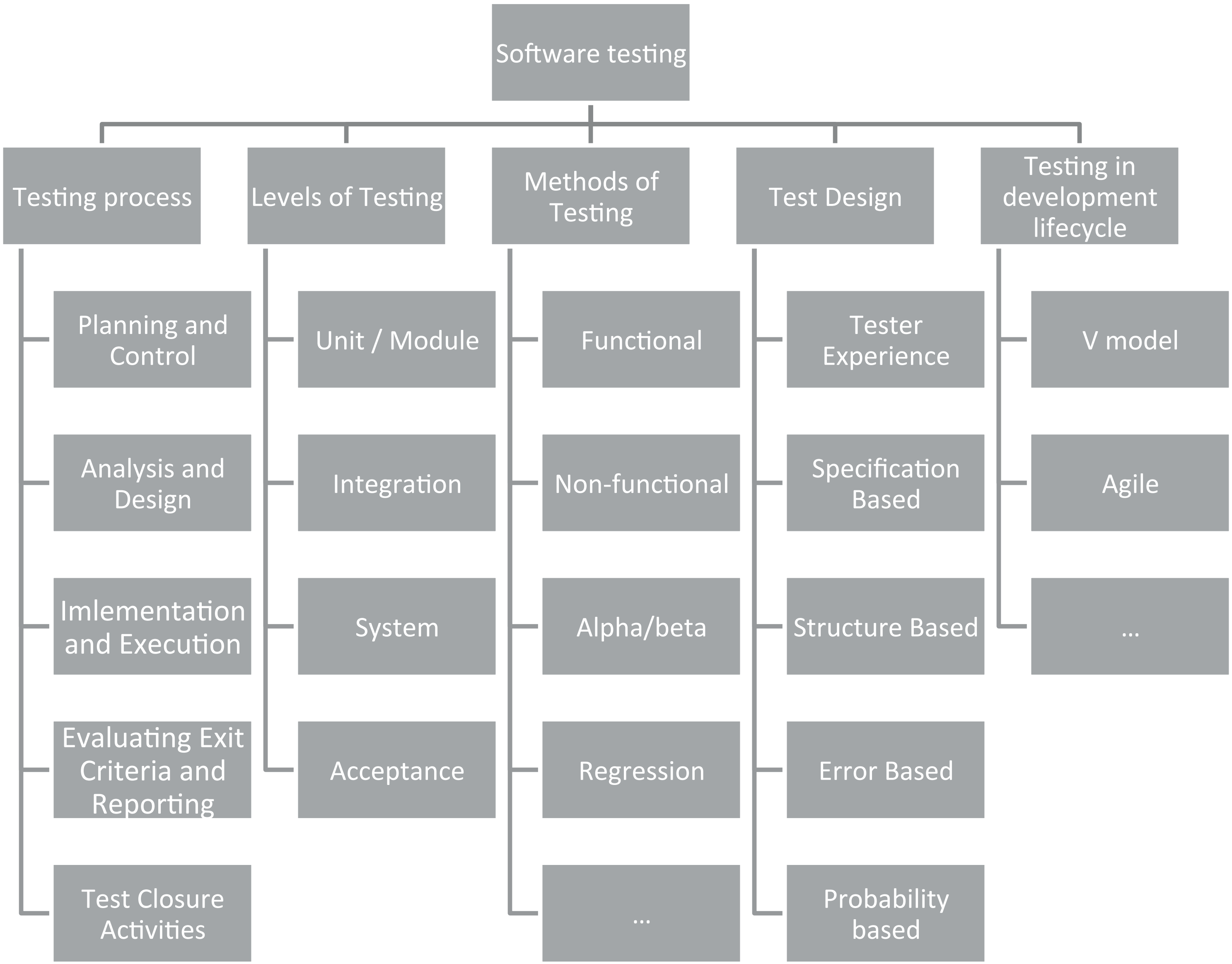
Testing in Practice

- Testing requires more than 50% of total development cost!
  - Test data generation
  - Test code creation
  - Test execution
  - Result evaluation

- Subtasks automatizable
  - Based on: e.g. models
    - class diagram: module interfaces
      → test controller and test stub generation
    - state machines: cooperation between modules
    - → test sequence generation

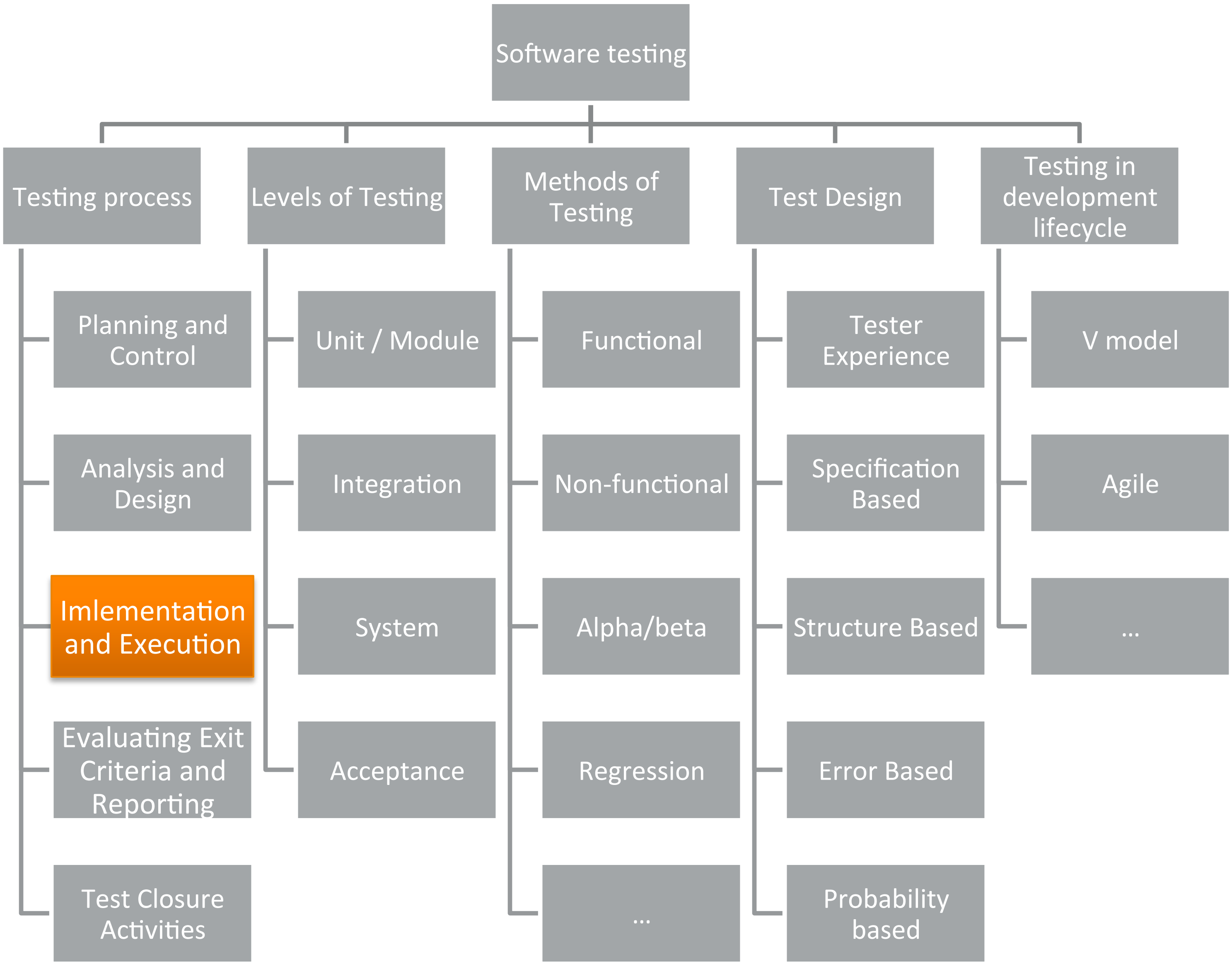
Further reading

- International Software Testing Qualifications Board (ISTQB), URL: http://istqb.dedicated.adaptavist.com/
  - ISTQB Glossary of Testing Terms
  - Foundation Level Syllabus (2010)
  - Even in Hungarian: http://www.hstqb.com/index.php?title=Downloads
- IEEE, Software Engineering Body of Knowledge (SWEBOK), URL: http://www.computer.org/portal/web/swebok/
  - Chapter 5: Software Testing
- IEEE, Software and Systems Engineering Vocabulary (SE VOCAB), URL: http://pascal.computer.org/sev_display/
  - Searchable set of definitions



- Software testing
  - Testing process
    - Planning and Control
    - Analysis and Design
    - Imlementation and Execution
    - Evaluating Exit Criteria and Reporting
    - Test Closure Activities
  - Levels of Testing
    - Unit / Module
    - Integration
    - System
    - Acceptance
  - Methods of Testing
    - Functional
    - Non-functional
    - Alpha/beta
    - Regression
    - …
  - Test Design
    - Tester Experience
    - Specification Based
    - Structure Based
    - Error Based
    - Probability based
  - Testing in development lifecycle
    - V model
    - Agile
    - …

The JUnit Framework



- Software testing
  - Testing process
    - Planning and Control
    - Analysis and Design
    - Imlementation and Execution
    - Evaluating Exit Criteria and Reporting
    - Test Closure Activities
  - Levels of Testing
    - Unit / Module
    - Integration
    - System
    - Acceptance
  - Methods of Testing
    - Functional
    - Non-functional
    - Alpha/beta
    - Regression
    - …
  - Test Design
    - Tester Experience
    - Specification Based
    - Structure Based
    - Error Based
    - Probability based
  - Testing in development lifecycle
    - V model
    - Agile
    - …

JUnit

- JUnit
  - Very common Java test framework
  - Original authors: Erich Gamma and Kent Beck
  - Multiple test executors
    - Command line
    - Simple GUI
    - IDE integrated
- JUnit 4
  - Uses Java 1.5 features, e.g. annotations
  - Completely different API than JUnit 3

Simple JUnit Test

- Java annotated methods
  - Similar to Eclipse 4 API

- Simple test cases:
  - At least one method annotated with `@org.junit.Test`
    - Annotated methods are the concrete test cases
    - Output validation:
      - Static methods of org.junit.Assert.*
      - E.g. assertEquals(expected, actual)

Simple JUnit Test

```
package hu.bme.mit.junit.example;

import org.junit.Test;
import junit.framework.TestCase;

public class ListTest {
    @Test
    public void testAddToEmptyList(){
        MyList l = new MyList();
        l.add(1);

        org.junit.Assert.assertEquals(1, l.getSize());
    }
}
```

Simple JUnit Test

```java
package hu.bme.mit.junit.example;


import org.junit.Test;
import junit.framework.TestCase;

public class ListTest {
   @Test
   public void testAddToEmptyList(){
        MyList l = new MyList();
        l.add(1);

        org.junit.Assert.assertEquals(1, l.getSize());
   }
}
```

Test method

Simple JUnit Test

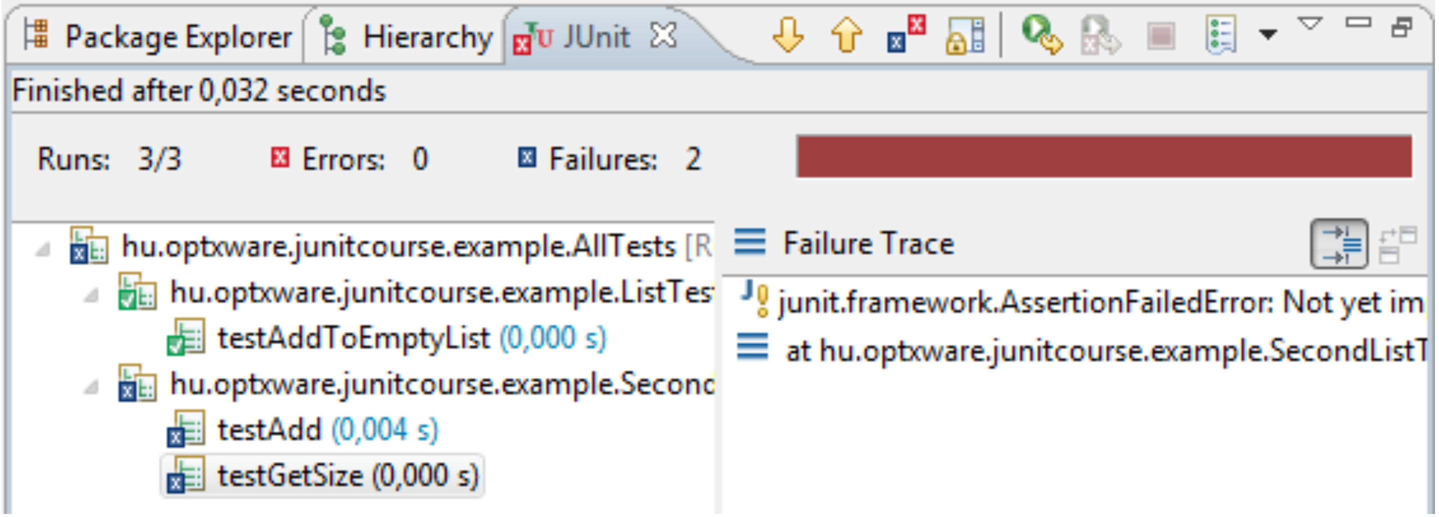
```java
package hu.bme.mit.junit.example;


import org.junit.Test;
import junit.framework.TestCase;

public class ListTest {
    @Test
    public void testAddToEmptyList(){
        MyList l = new MyList();
        l.add(1);

        org.junit.Assert.assertEquals(1, l.getSize());
    }
}
```

Test method

Validation

JUnit Test Execution in Eclipse

- Select class containing tests
- Run As -> JUnit test
- Results:
  - Colored by output
    - Ok, Error, Failure



Finished after 0,032 seconds

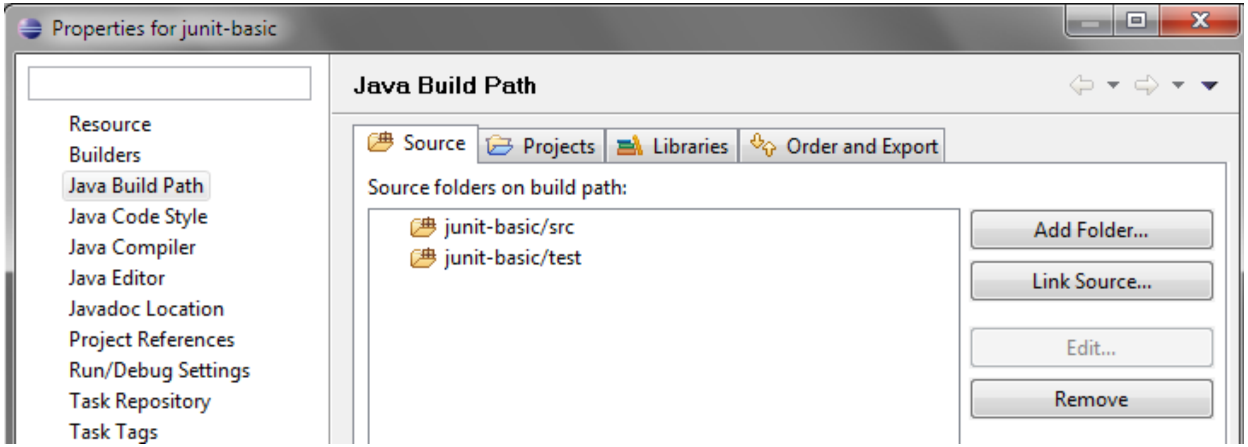
Runs: 3/3 Errors: 0 Failures: 2

```
hu.optxware.junitcourse.example.AllTests [R
  hu.optxware.junitcourse.example.ListTes
    testAddToEmptyList (0,000 s)
  hu.optxware.junitcourse.example.Seconc
    testAdd (0,004 s)
    testGetSize (0,000 s)
```

```
junit.framework.AssertionFailedError: Not yet im
at hu.optxware.junitcourse.example.SecondListT
```

Using JUnit in Eclipse - Preparations

- Convention: separate source folder for tests
  - Named: test



Properties for junit-basic

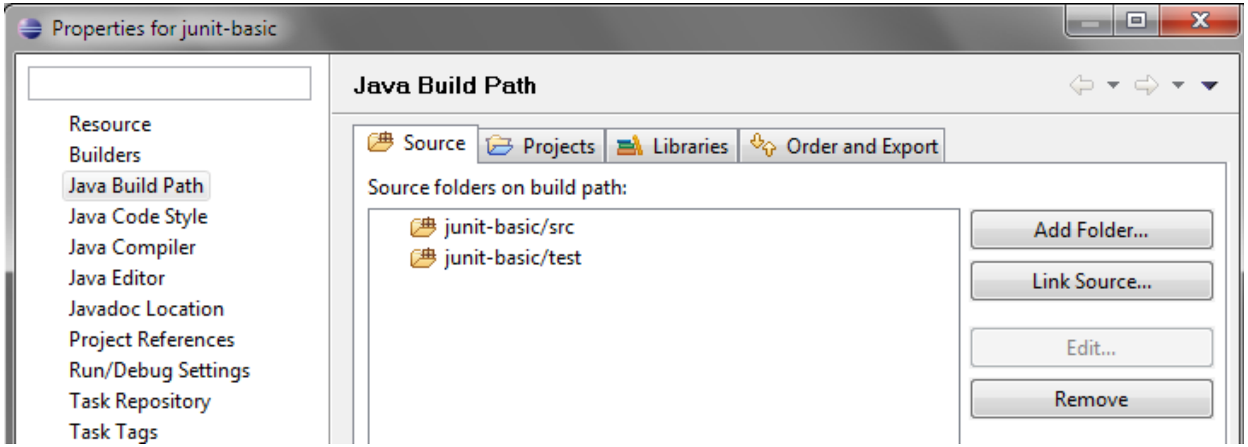
Source folders on build path:

junit-basic/src
junit-basic/test

- JUnit Library in classpath

Using JUnit in Eclipse - Preparations

- Convention: separate source folder for tests
  - Named: test



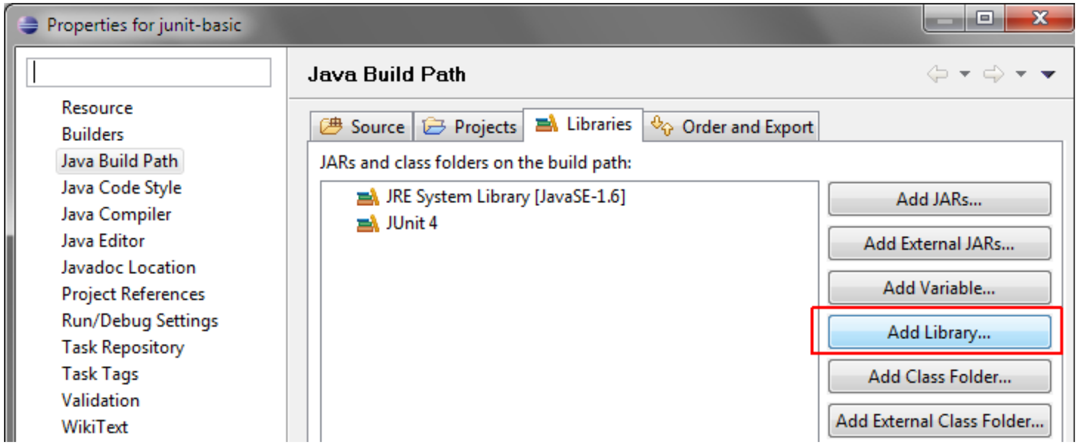
Properties for junit-basic

Java Build Path

Source folders on build path:

junit-basic/src
junit-basic/test

- JUnit Library in classpath



Properties for junit-basic

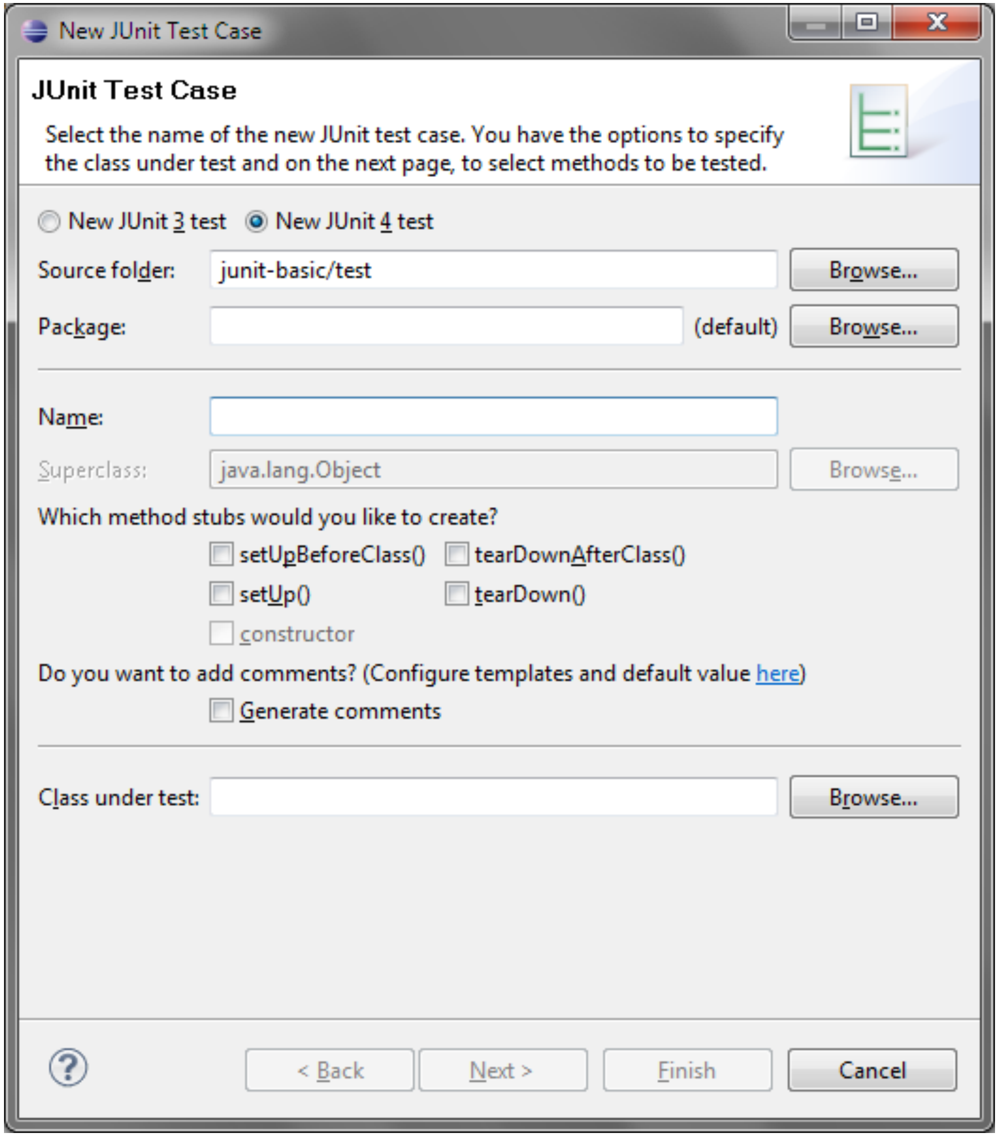
Java Build Path

JARs and class folders on the build path:

JRE System Library [JavaSE-1.6]
JUnit 4

JUnit Test Creation

- Test Class Name
  - «Unit_name»Test
  - «Unit_name»Tests
- Select class under test
- Helper methods
  - „test fixture”
  - setup & teardown



New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

○ New JUnit 3 test ● New JUnit 4 test

Source folder: junit-basic/test

Package: (default)

Name:

Superclass: java.lang.Object

Which method stubs would you like to create?

□ setUpBeforeClass() □ tearDownAfterClass()

□ setUp() □ tearDown()

□ constructor

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Class under test:

Test fixture

- Prepares environment for tests:
    - May be shared between tests
    - Beware for interdependent tests!
- Once per testing process
    - @BeforeClass, @AfterClass
- Before and after each test case
    - @Before, @After

Test fixture - Example

```
public class ListTests {

     List emptyList;

    @Before
    public void setUp(){
        emptyList = Collections.EMPTY_LIST;
    }

    @After
    public void tearDown() {
        emptyList = null;
    }

     @Test
     public void testEmptyList() {
        assertEquals("Empty list should have 0 elements",
                0, emptyList.size());
     }

}
```

Assertions

- Automatic validation of test cases
  - Provide one per test case
  - Unless it is harder to find concrete issue
- Static methods of org.junit.Assert
  - assertEquals(expected,actual)
  - assertFalse(boolean)
  - assertTrue(boolean)
  - assertNull(object)
  - assertNotNull(object)
  - assertSame(expected,actual)
  - assertNotSame(expected,actual)
  - assertArrayEquals(expecteds, actuals)
  - …

JUnit Annotations

Annotation	Description
@Test public void method()	Defines test method
@Before public void method()	Executes before each test
@After public void method()	Executes after each test
@BeforeClass public void method()	Executes once before all tests
@AfterClass public void method()	Executes once after all tests
@Ignore	Skips the test; use sparingly
@Test(expected=IllegalArgumentException.class)	Test case is succesful if throws the selected exception
@Test(timeout=100)	Limits test execution time

Grouping Test cases

- Test Suites in JUnit 4:
  - @RunWith: define test executor
  - @SuiteClasses: defines members

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
  ListTest.class, VectorTest.class})
public class AllTests {
      // placeholder for the above annotations
}
```

Expected exception

- Evaluate error handling
  - We sometimes expect the trowing of exceptions

```
public class RegularExpressionJUnit4Test {
  private static String zipRegEx = "^\\d{5}([\\-]\\d{4})?$";
  private static Pattern pattern;

  @BeforeClass
  public static void setUpBeforeClass() throws Exception {
   pattern = Pattern.compile(zipRegEx);
  }

  @Test(expected=IndexOutOfBoundsException.class)
  public void verifyZipCodeGroupException() throws Exception{
   Matcher matcher = this.pattern.matcher("22101-5051");
   boolean isValid = matcher.matches();
   matcher.group(2);
  }
}
```

Parameterized tests

- Common case: many similar tests
  - only different in parameters
  - test code should not be redundant

- JUnit 4: parameterized tests
  - Separate test code and test data
  - Framework executes the test code with all data

JUnit Parameterized Tests 1/6

- Write parameterless test code

```
@Test
public void testComplexCalculation() throws Exception
{
    Integer r = calc.complexCalculation(a, b);
    assertEquals(res, r);
}
```

- Test data is yet undefined:
  - a, b: input
  - res: expected output

JUnit Parameterized Tests 2/6

- Create „feeder” method
  - Static method, returning a collection of arrays
  - Annotated with `@Parameters`
  - Arrays are used to serialize test data

```
@Parameters
public static Collection<Object[]> complexCalcValue() {
return Arrays.asList(new Object[][] {
      { 1, 1, 12 },
      { -1, 1, -10 },
      { 10, 10, 30 },
      { 2, 2, 14 } });
}
```

JUnit Parameterized Tests 3/6

- Create attributes for single test data:

```
private Integer a;

private Integer b;

private Integer res;
```

JUnit Parameterized Tests 4/6

- JUnit will call the constructor parameters in order:

```
public ParametricTest(Integer a,
Integer b, Integer res) {
   super();
   this.a = a;
   this.b = b;
   this.res = res;
}
```

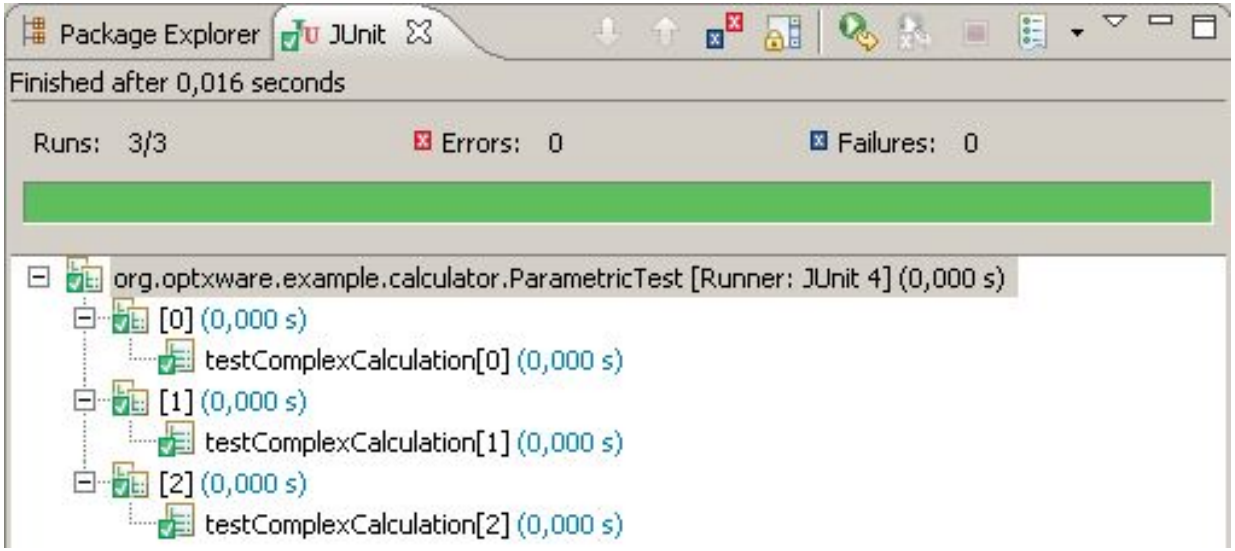
JUnit Parameterized Tests 5/6

- Selected specific test runner for the class
  - `org.junit.runners.Parametrized`

```
@RunWith(Parameterized.class)
public class ParametricTest {
      …
}
```

JUnit Parameterized Tests 6/6

- Execute tests
  - The testComplexCalculation() test executes four times
  - Executes with all values from step 2.



```
Finished after 0,016 seconds

Runs: 3/3          Errors: 0          Failures: 0

org.optxware.example.calculator.ParametricTest [Runner: JUnit 4] (0,000 s)
  [0] (0,000 s)
    testComplexCalculation[0] (0,000 s)
  [1] (0,000 s)
    testComplexCalculation[1] (0,000 s)
  [2] (0,000 s)
    testComplexCalculation[2] (0,000 s)
```

Categories

- Further categorization, e.g.
  - Execution time
  - Resource requirements

- Categories defined as interfaces

```
    public interface FastTests {}
    public interface SlowTests {}
```

- Inheritance supported

```
interface PerformanceTests extends
SlowTests{}
```

Categories /2

- Use case 1: annotate test classes

```
@Category(FastTests.class)
public class CalculatorTest {}
```

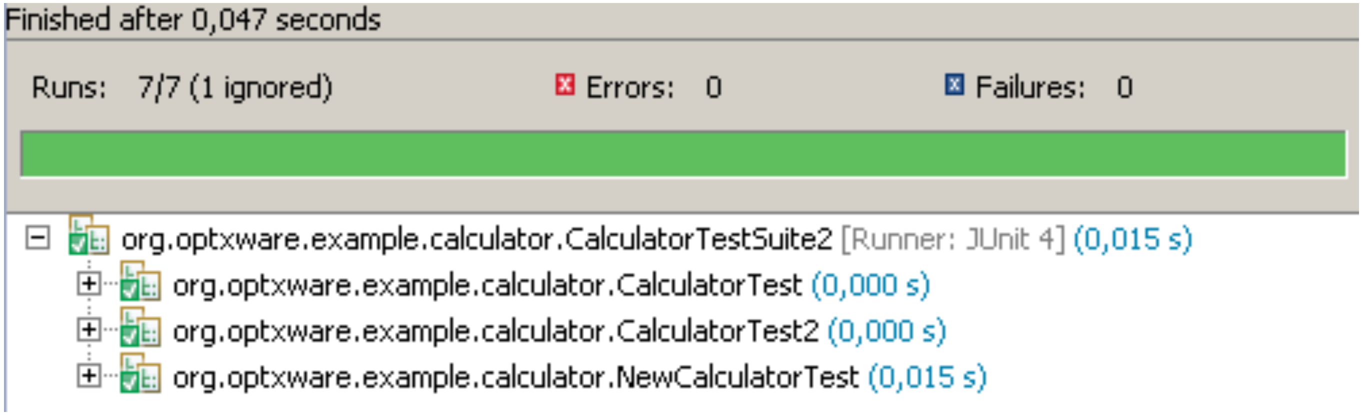
- Use case 2: annotate test cases

```
@Category(SlowTests.class)
public void complexCalc (int a, int
b){}
```

- Execution
  - `@RunWith(Categories.class)`
  - `@IncludeCategory(SlowTests.class)`
  - `@SuiteClasses(`
    `{ CalculatorTest.class,`
    `ParametricTest.class })`
  - `public static class SlowTestSuite {}`

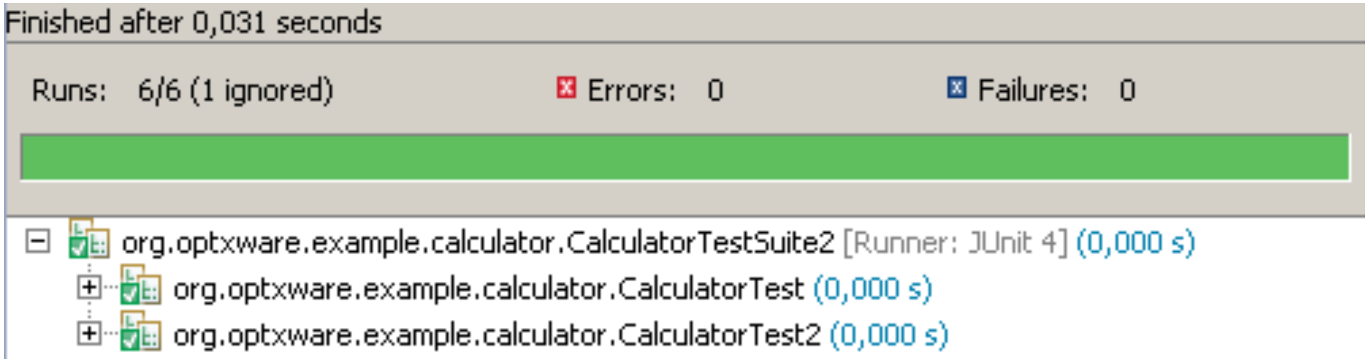
Categories /3

```
@IncludeCategory(FastTests.class)
```



```
Finished after 0,047 seconds
Runs: 7/7 (1 ignored)    Errors: 0    Failures: 0

⊟ org.optxware.example.calculator.CalculatorTestSuite2 [Runner: JUnit 4] (0,015 s)
    ⊞ org.optxware.example.calculator.CalculatorTest (0,000 s)
    ⊞ org.optxware.example.calculator.CalculatorTest2 (0,000 s)
    ⊞ org.optxware.example.calculator.NewCalculatorTest (0,015 s)
```

```
@ExcludeCategory(SlowTests.class)
```



```
Finished after 0,031 seconds
Runs: 6/6 (1 ignored)    Errors: 0    Failures: 0

⊟ org.optxware.example.calculator.CalculatorTestSuite2 [Runner: JUnit 4] (0,000 s)
    ⊞ org.optxware.example.calculator.CalculatorTest (0,000 s)
    ⊞ org.optxware.example.calculator.CalculatorTest2 (0,000 s)
```

Rules

- JUnit extension points
- Base extensions
  - TemporaryFolder
    - For storing temporary files and folders
    - Will be removed after the test execution

```
@Rule
public TemporaryFolder tempFolder = new TemporaryFolder();

File newFile = tempFolder.newFile("myfile.txt");
```

  - ExternalResource
    - External resource that needs to be reset after testing

Rules /2

- ErrorCollector
  - In case of exception don’t stop but continue
  - All exceptions will be shown in the end
- ExpectedException
  - Specifies expected exceptions inside test cases

```
@Rule
public ExpectedException exception = ExpectedException.none();
…
exception.expect(IllegalArgumentException.class);
```

- Timeout
  - Class-level timeout setting

```
@Rule
public MethodRule globalTimeout = new Timeout(20);
```

Theories

- Generalizes connection between input and output
- Simpler structure than parameterized tests
- Class annotated with `@RunWith(Theories.class)`
- Required
  - Data generation method - `@DataPoints`
  - Generated data will be used by test cases as input

```
@DataPoints
public static Integer[] data(){
        return new Integer[] {
                new Integer(10),
                new Integer(17),
                new Integer(-16)
};
```

Theories /2

- Theory
  - Test is annotated with @Theory
  - Must contain (at least) one assertion

```
@Theory
public void addTheory(Integer a, Integer b){
    assumeTrue(a > 0);
    assumeTrue(b > 0);

    assertEquals((a+b), calc.complexAddMethod(a, b));
 }
```

Assumptions

- Describes preconditions for test case
- If Assumption fails, test case still ok

```
assumeTrue(a>0);
assumeTrue(b>0);
```

- Useful for *Theories*
  - Filters invalid input for test case

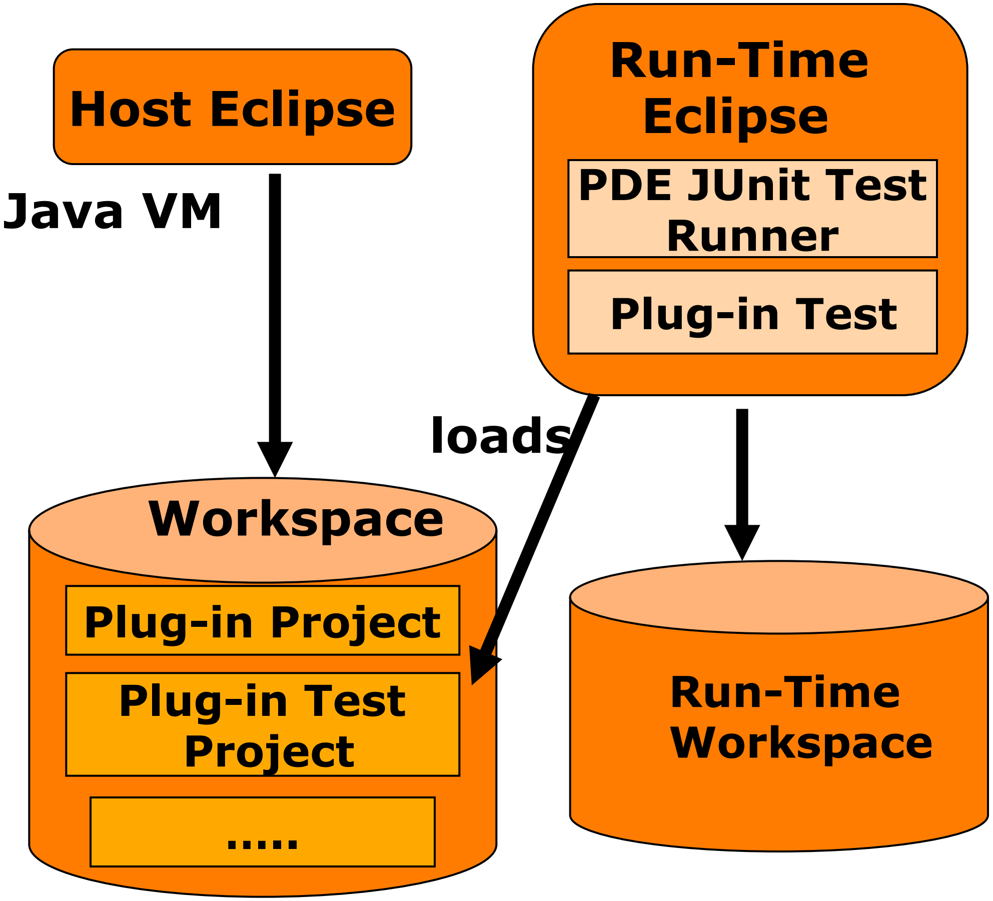
PDE JUnit Tests

JUnit Plug-in Tests

- PDE JUnit
  - Test execution for
    - Eclipse plug-ins
    - OSGi bundles
  - Part of Plug-in Development Environment 3.x/4.x
- Behaves like plain JUnit
- Differences:
  - Custom test runner: starts a new Eclipse instance
  - Similar to runtime workbench
  - Every test is executed in this workbench
    - Full Eclipse API available
    - OSGi classloading in action!

PDE JUnit

- Steps:
  - Starting runtime workbench
  - JUnit TestRunner gets control
  - Tests executed in runtime
  - Runtime workbench stopped

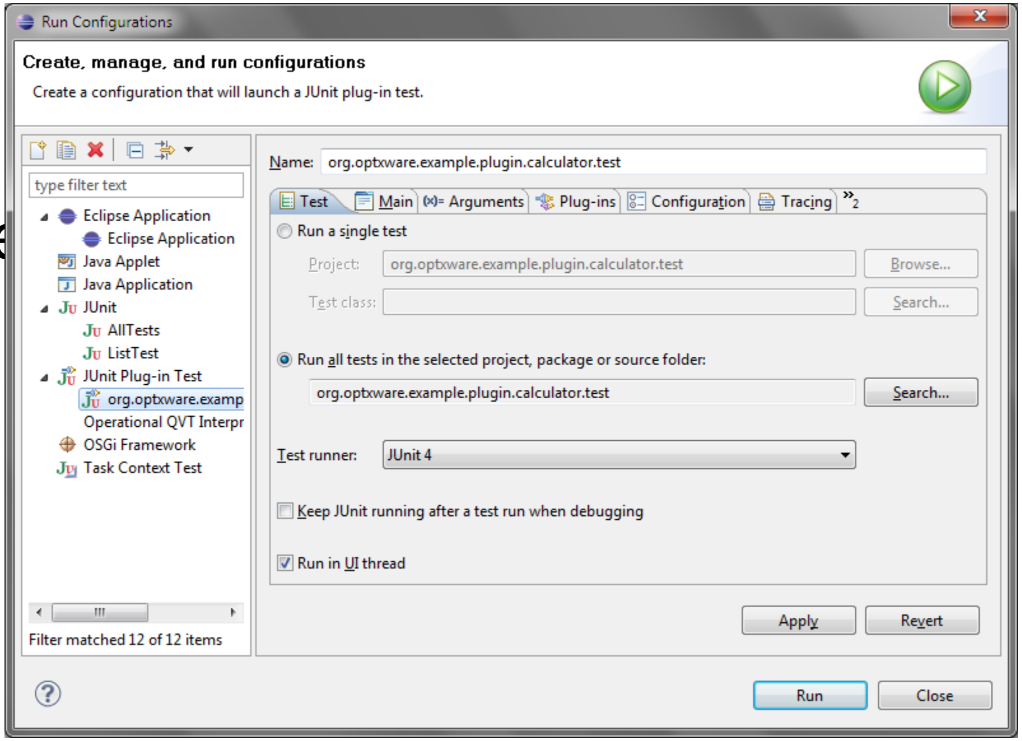


JUnit Plug-in Test Settings

- Test
  - What test to run
- Main
  - Run an application – Headless mode
- Plug-ins
  - What plug-ins to load
- Configuration
  - Clear the configuration area before launch

JUnit Plug-in Test Settings

- Test
  - What test to run
- Main
  - Run an application – He
- Plug-ins
  - What plug-ins to load
- Configuration
  - Clear the configuration



Run Configurations

Create, manage, and run configurations

Create a configuration that will launch a JUnit plug-in test.

type filter text

- Eclipse Application
  - Eclipse Application
- Java Applet
- Java Application
- JUnit
  - AllTests
  - ListTest
- JUnit Plug-in Test
  - org.optxware.examp
- Operational QVT Interpr
- OSGi Framework
- Task Context Test

Filter matched 12 of 12 items

Name: org.optxware.example.plugin.calculator.test

○ Run a single test

Project: org.optxware.example.plugin.calculator.test

Test class:

● Run all tests in the selected project, package or source folder:

org.optxware.example.plugin.calculator.test

Test runner: JUnit 4

□ Keep JUnit running after a test run when debugging

☑ Run in UI thread

Plug-in testing

- Unit tests
  - Can be problematic because of many dependencies
  - Can be mocked if required
- Integration testing
  - More common for PDE JUnit

Test Case Placement Options

Test Case Placement Options

- Separate source folder
  - As Java projects
  - BUT: JUnit dependency for plug-in!
- Plug-in fragment
  - Sees the inside of the host
- Separate plug-in
  - Only public API is available

Test Case Placement Options

- Separate source folder
  - As Java projects
  - BUT: JUnit dependency for plug-in!
- Plug-in fragment
  - Sees the inside of the host
- Separate plug-in
  - Only public API is available

Test Case Placement Options

- Separate source folder
  - As Java projects
  - BUT: JUnit dependency for plug-in!
- Plug-in fragment
  - Sees the inside of the host
- Separate plug-in
  - Only public API is available

Test Case Placement Options

- Separate source folder
  - As Java projects
  - BUT: JUnit dependency for plug-in!
- Plug-in fragment
  - Sees the inside of the host
- Separate plug-in
  - Only public API is available

Headless mode

- Testing without GUI
  - Much faster
  - For UI-independent plug-ins
  - Requires planning in advance

Related Eclipse Projects

- GUI testing
  - SWTBot
    - Supports even GEF-based editors!
  - Jubula
    - Model-based test specification
  - WindowTester Pro
    - Capture-and-playback
    - Previously developed by Instantiations
      - As WindowBuilder Pro

Further Reading

- JUnit, http://www.junit.org/
- Lars Vogel, JUnit – Tutorial, http://www.vogella.de/articles/JUnit/article.html
- Andrew Glover, Jump into JUnit 4, http://www.ibm.com/developerworks/java/tutorials/j-junit4/index.html

Profiling

Problem

- Application
  - Slow, or
  - Requires a lot of memory
- How to fix it?

Profiling

- (Performance) information collection for application
  - Dynamic, runtime techniques

- Tipical information collected:
  - Method execution count (both start and return)
  - Execution times
  - Memory usage
  - Call stack
  - Thread states
  - …

Profiling implementation

- Instrumentation
  - Flagging, logging instructions added
  - Manual / automatic
  - Code / binary / runtime level

- Framework support
  - E.g. Java (Java Virtual Machine Tool Interface), .NET
  - Events, callback methods

- Sampling
  - Periodically looks at state
  - Less intrusive, but less precise
  - HW support possible

Profiling implementation

- Instrumentation
  - Flagging, logging instructions added
  - Manual / automatic
  - Code / binary / runtime level

- Framework support
  - E.g. Java (Java Virtual Machine Tool Interface), .NET
  - Events, callback methods

- Sampling
  - Periodically looks at state
  - Less intrusive, but less precise
  - HW support possible

Profiling implementation

- Instrumentation
  - Flagging, logging instructions added
  - Manual / automatic
  - Code / binary / runtime level
- Framework support
  - E.g. Java (Java Virtual Machine Tool Interface), .NET
  - Events, callback methods
- Sampling
  - Periodically looks at state
  - Less intrusive, but less precise
  - HW support possible

Java Profiler tools

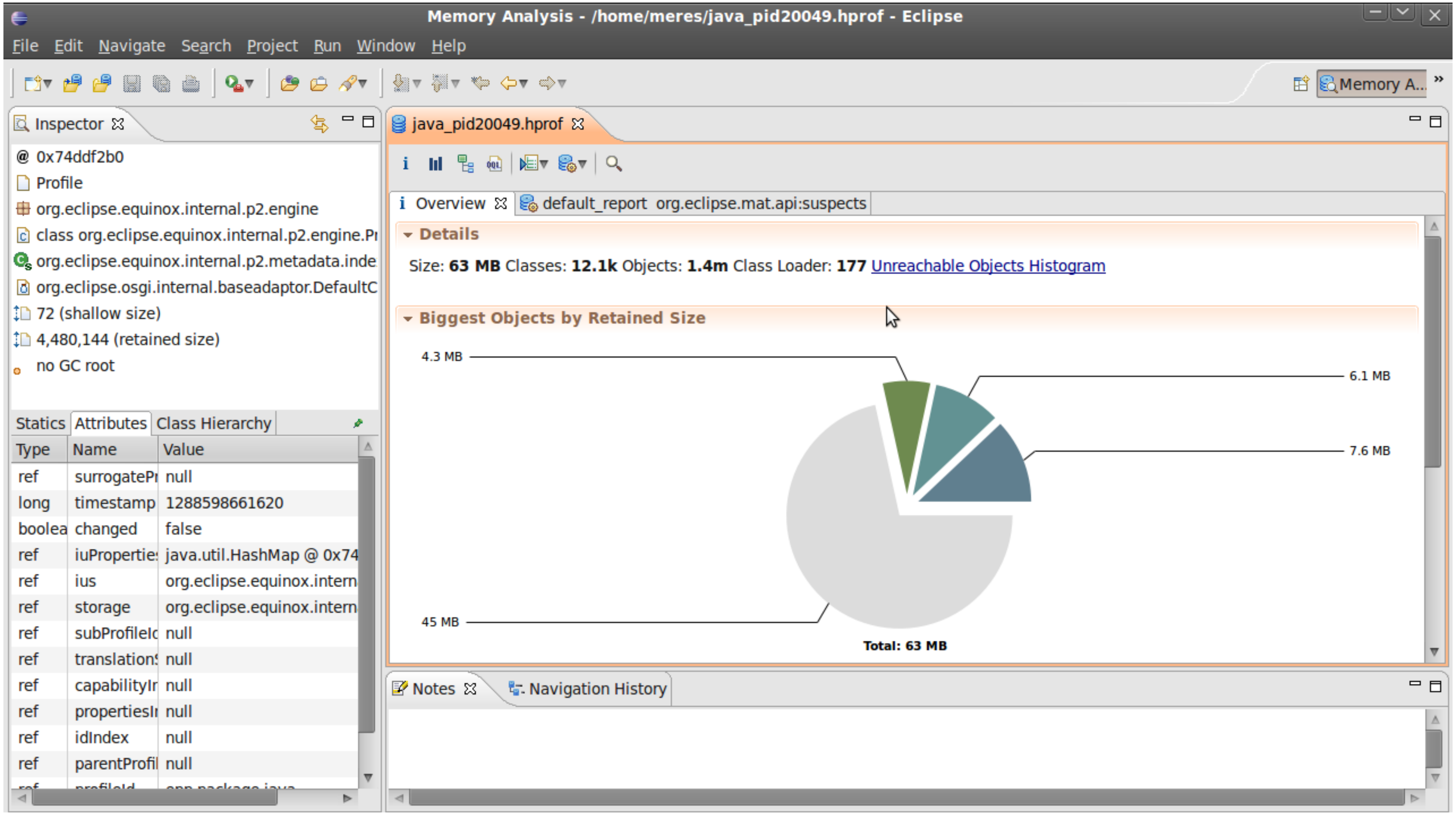
- Multiple profilers available, see
  - http://java-source.net/open-source/profilers

- jvisualvm
  - Based on JDK features
- YourKit Java Profiler
- Quest JProbe
- JIP – Java Interactive Profiler
- Netbeans Profiler
- …
- Eclipse:
  - Memory Analyzer (MAT) – heap analyzer
  - Test & Performance Tools Platform (TPTP)

Memory Analyzer (MAT)

- Heap dump file analysis
  - Can be created by JVM tools
  - Basically, memory map
- Available as RCP application or Eclipse plug-in

Memory Analyzer (MAT)



Memory Analysis - /home/meres/java_pid20049.hprof - Eclipse

@ 0x74ddf2b0
Profile
org.eclipse.equinox.internal.p2.engine
class org.eclipse.equinox.internal.p2.engine.Pr
org.eclipse.equinox.internal.p2.metadata.inde
org.eclipse.osgi.internal.baseadaptor.DefaultC
72 (shallow size)
4,480,144 (retained size)
no GC root

Type	Name	Value
ref	surrogatePr	null
long	timestamp	1288598661620
boolea	changed	false
ref	iuPropertie:	java.util.HashMap @ 0x74
ref	ius	org.eclipse.equinox.intern
ref	storage	org.eclipse.equinox.intern
ref	subProfileIc	null
ref	translation!	null
ref	capabilityIr	null
ref	propertiesIı	null
ref	idIndex	null
ref	parentProfil	null

Details

Size: **63 MB** Classes: **12.1k** Objects: **1.4m** Class Loader: **177** Unreachable Objects Histogram

Biggest Objects by Retained Size

Test & Performance Tools Platform

- Top-level eclipse.org project
- Provides profiling tools
  - Be careful, project is close to dead