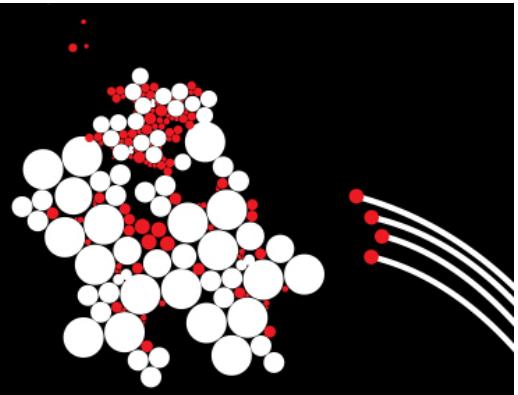
UNIVERSITY OF TWENTE.

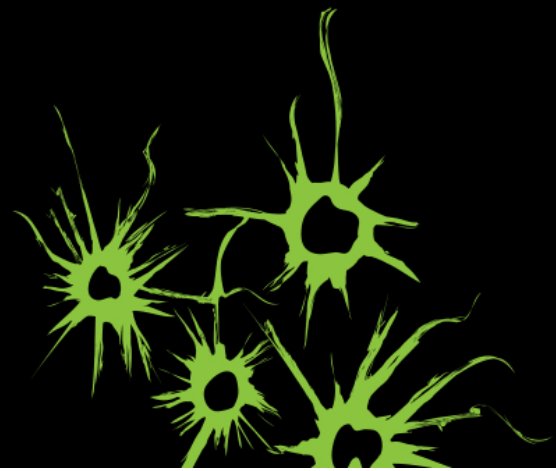


P2.2: CLASS SPECIFICATIONS / TESTING

LUÍS FERREIRA PIRES

201700117-1B MODULE 2: SOFTWARE SYSTEMS

18 NOVEMBER 2019



PROGRAMMING LINE OVERVIEW

Week 1 Values and variables Control flow	**Week 2** Classes and objects Testing	**Week 3** Interfaces and Inheritance Subtyping Security 1
Week 4 Arrays and Lists List implementations Collections	**Week 5** Stream I/O and MVC Exceptions Security 2	**Week 6** Concurrency Project kick-off IDE Tips & Tricks
Week 7 Basic Networking Networking and Multithreading GUIs	**Week 8/9** Advanced Java facilities Test	**Week 10** Project Test resit

CONTENTS

- Class specifications
  - Preconditions, postconditions and invariants
- Programming by contract
- Testing
- Test-driven development

Material

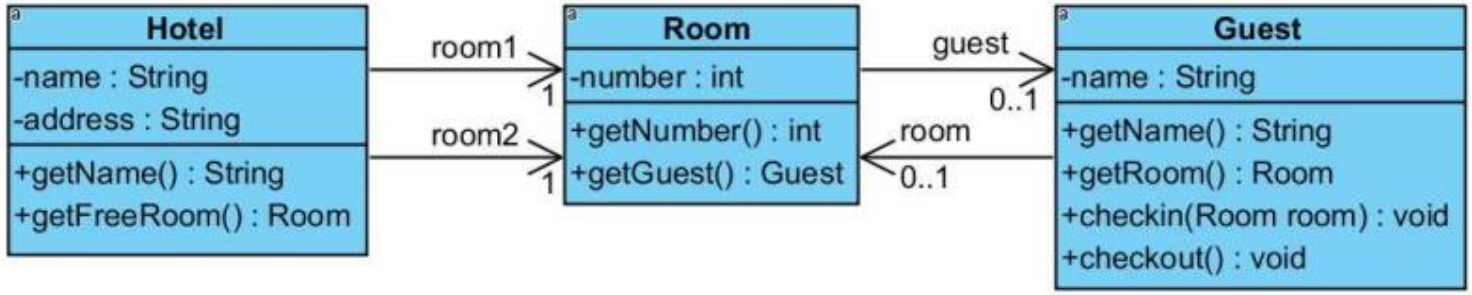
- Appendix A

PROGRAM DESIGN

DISCUSSED IN THE LAST LECTURE

- Consists of classes and their relationships

Example: Hotel Information System



GUEST CLASS

FOR THIS EXAMPLE

- Instance variables: `name`, `room`
- Constructor: `Guest (String name)`
- Getters for `name` and `room`
- No setters!
- `checkin (Room room)` and `checkout()` methods to assign this guest to a room and remove the room, respectively

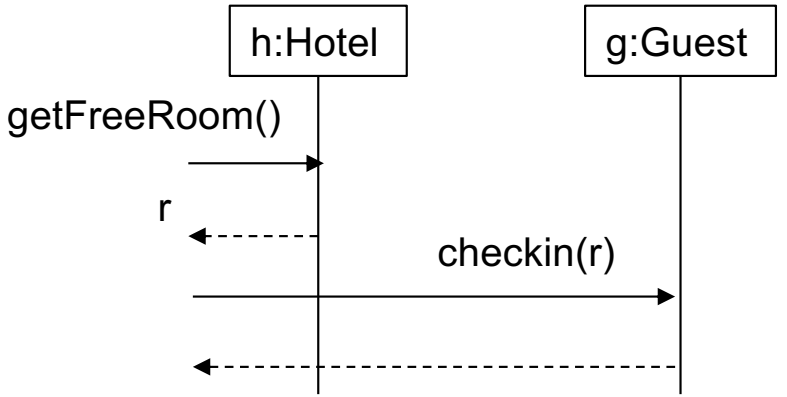

```java
/**
 * Hotel guest with a name and possibly a hotel room.
 * @author Arend Rensink
 */
public class Guest {

    private String name;
    private Room room;

    public Guest(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public Room getRoom() {
        return this.room;
    }

    public boolean checkin(Room room) {
        boolean result = false;
        if (this.room == null && room.getGuest() == null) {
            room.setGuest(this);
            this.room = room;
            result = true;
        }
        return result;
    }

    public boolean checkout() {
        boolean result = false;
        if (this.room != null) {
            this.room.setGuest(null);
            this.room = null;
            result = true;
        }
        return result;
    }
```

OBJECT INTERACTIONS

Assumption: all instance variables are `private`!

- When a program is running, objects exist that are instances of the classes defined in the program design
- Objects can only interact by calling each other's methods!

```
Hotel h = new Hotel("Fawlty Towers");
Guest g = new Guest("Major Gowen");
g.checkin(h.getFreeRoom());
```



CLASS SPECIFICATIONS

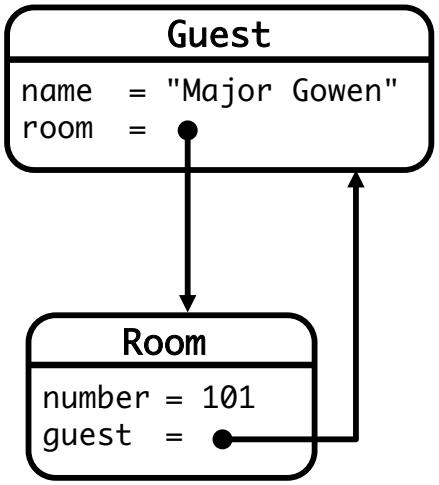
PRECONDITIONS, POSTCONDITIONS AND CLASS INVARIANTS

For each class and each method the program designer must specify the conditions for objects of this class to work properly!

- Preconditions: Which conditions should hold beforehand for a method to work correctly once is called?
- Postconditions: What conditions are satisfied once the method has worked correctly?
- (Class) Invariants: What are the conditions that must always hold in an object of a class?

GUEST CLASS SPECIFICATION

- Method `checkin(Room room)`
  - Precondition: Room is not `null` (`room != null`)
  - Postcondition: Guest related to room is this guest
    (`room.getGuest() == this`)
- Class Invariant: If room attribute is not `null` then the guest related to the room is this guest
  (`room != null ==> room.getGuest() == this`)



COUNTER CLASS EXAMPLE

```
public class Counter {

    /**
     * @invariant value >= 0
     */
    private int value;

        public Counter() {
            value = 0;
}
        public int getValue() {
            return value;
        }

        public void setValue(int value) {
            this.value = value;
        }

        public void increment() {
            value++;
        }
}
```

user defined JavaDoc tag!

PRECONDITION

- Defined using method parameters and/or object conditions that can be defined using `public` methods

Question

- Given the `Counter` invariant (`value >= 0`), what should be the precondition of `setValue(int value)`?

Answer

- `@requires value >= 0`

user defined JavaDoc tag!

PRECONDITION

RESPONSIBILITY OF THE CALLER

Suppose we add to class `Counter`

- A constant *MAX* to define the maximum value of the counter
- A method `reset()` that can only be called when the counter has reached *MAX*
  - Precondition: `@requires getValue() == Counter.MAX;`
- Caller must ensure this precondition!
- Method implementation can rely on this (no need to check!)

ANOTHER EXAMPLE

PRECONDITION AS A RESPONSIBILITY OF THE CALLER

```
public class Lock {
    private int code;
    private boolean open;

    public Lock(int code) { }

    public boolean isOpen() { return open; }

    public void close() { open = false; }

    public void enterDigit(int digit) { }
}
```

- Class invariant

  `0 <= code && code <= 999`

- `enterDigit` precondition

  `@requires 0 <= digit && digit <= 9`

- If we define like as precondition, Caller must ensure this!
- Lock implementation does not have to check it!

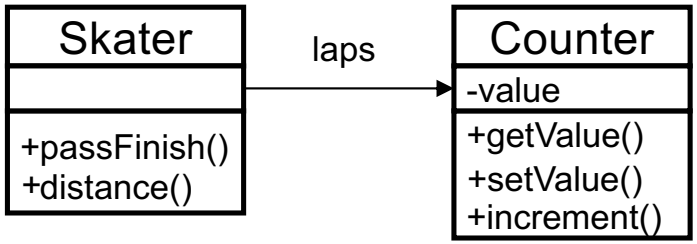
POSTCONDITION

RESPONSIBILITY OF THE CALLED OBJECT (IMPLEMENTER)

```
public class Skater {

    private Counter laps;

    public static final int LAPSIZE = 600;

    public void passFinish() {
        laps.increment();
    }

    public int distance() {
        return laps.getValue() * LAPSIZE;
    }
}
```

What is the postcondition of `increment()`?

```
@ensures getValue() == old.getValue() + 1
```



- What should be the postcondition of `getValue()`?
- It should return a positive number!
- Postcondition

```
@ensures result >= 0
```

yet another user defined JavaDoc tag!

DEFINITION OF CLASS INVARIANT

- An class invariant is defined in the scope of a class (not a method)
- General meaning: a property that always holds

- Two usage scenarios
  1. Can refer to internal state of the object (useful for implementer)
  2. Can also serve as documentation of the behaviour of a class (useful for caller)

CLASS INVARIANTS

- Properties that hold for every internally reachable state of an object
  - Define allowed values for instance and static variables
  - Methods rely on these properties

Example

- Public invariant: refers to publicly visible methods

```
@invariant getValue() >= 0
```

- Private invariant: refers to internal state, not visible to caller

```
@invariant value >= 0
```

BOOLEAN CONDITIONS

- Preconditions, postconditions and invariants are all boolean conditions
- You can write them informally (in text) in this module, but the Java notation for booleans is recommended when possible
  - `result`, `this` and `old.<method>` refer to the return of a method, this object and the value before calling the method, respectively
  - Basic logical operators (`&&`, `||`, `!=`) and implication (`==>`)
  - Universal and existential quantification (`forall x:<type> in ...` , `exists x:<type> in ...`, respectively) may also have to be defined!

JML: language to formally define these conditions (advanced topic)

PROGRAMMING BY CONTRACT

PROGRAMMING DISCIPLINE WITH PRE-, POSTCONDITIONS AND INVARIANTS

- If caller respects preconditions then method implementation guarantees postconditions
- Class invariant shows that implementation ensures postconditions

Problem: Can the caller (client) be trusted?

- Question: What if caller does not respect the precondition?
  - Method will not guarantee postcondition and/or invariant
  - Next methods may be called while invariant is violated
  - Program does not behave properly, error hard to find

ANSWER 1: TRUST CLIENT

- Client will always respect preconditions

Consequences

- No special precautions necessary!

- Justified when client and object (class) are developed together

ANSWER 2: GENERATE ERROR MESSAGE

- Client will not always respect preconditions
- When this happens, program should stop, but in controlled manner

Consequences

- Implementation checks (some) preconditions
- `assert` precondition: stop program when precondition not respected
- Especially useful to make sure internal invariants always hold

needs to be enabled in the JVM

- Applicable to larger programs

ANSWER 3: DEFENSIVE PROGRAMMING

- Client will make mistakes (might be on purpose)
- Program should not fail

Consequences

- Implementation checks all preconditions, and if a precondition is not respected take appropriate emergency solution
  - Set default values, throw exceptions
  - Postcondition and invariant always respected
- Useful for critical applications

ANSWER 4: CHECK OR VERIFY

Runtime Checking

- Automatically insert precondition and postcondition checks during execution

Static Checking

- Construct formal proof that
  - Preconditions hold at every method call
  - Postconditions hold at every method exit
  - Invariants are always maintained

Advanced, requires appropriate tooling

Not considered further in this module

PROGRAMMING BY CONTRACT

ESSENCE OF OBJECT-ORIENTATION

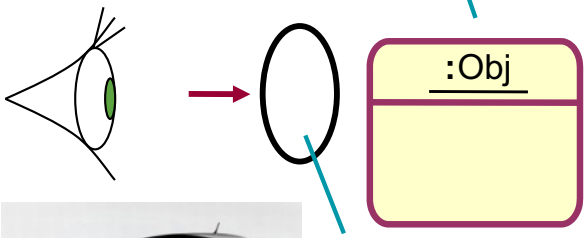
implementation
(how)

Specification

- Methods have preconditions that caller (client) must respect
- Then methods should guarantee that postconditions are satisfied

Implementation

- Class invariants and method implementation guarantee that postconditions are satisfied



specification
(what)

PROGRAMMING BY CONTRACT AND APIS

- APIs (Application Programming Interfaces) follow the programming by contract principles, sometimes in a less systematic way

return is valid double value

postcondition →

precondition →

s != null && proper format

valueOf

```
public static Double valueOf(String s) throws NumberFormatException
```

Returns a `Double` object holding the `double` value represented by the argument string `s`.

If `s` is `null`, then a `NullPointerException` is thrown.

Leading and trailing whitespace characters in `s` are ignored. Whitespace is removed as if by the `String.trim()` method; that is, both ASCII space and control characters are removed. The rest of `s` should constitute a *FloatValue* as described by the lexical syntax rules:

FloatValue:

$Sign_{opt}$ `NaN`

Sign `Infinity`

CONTENTS

- Class specifications
  - Preconditions, postconditions and invariants
- Programming by contract
- Testing
- Test-driven development

Material

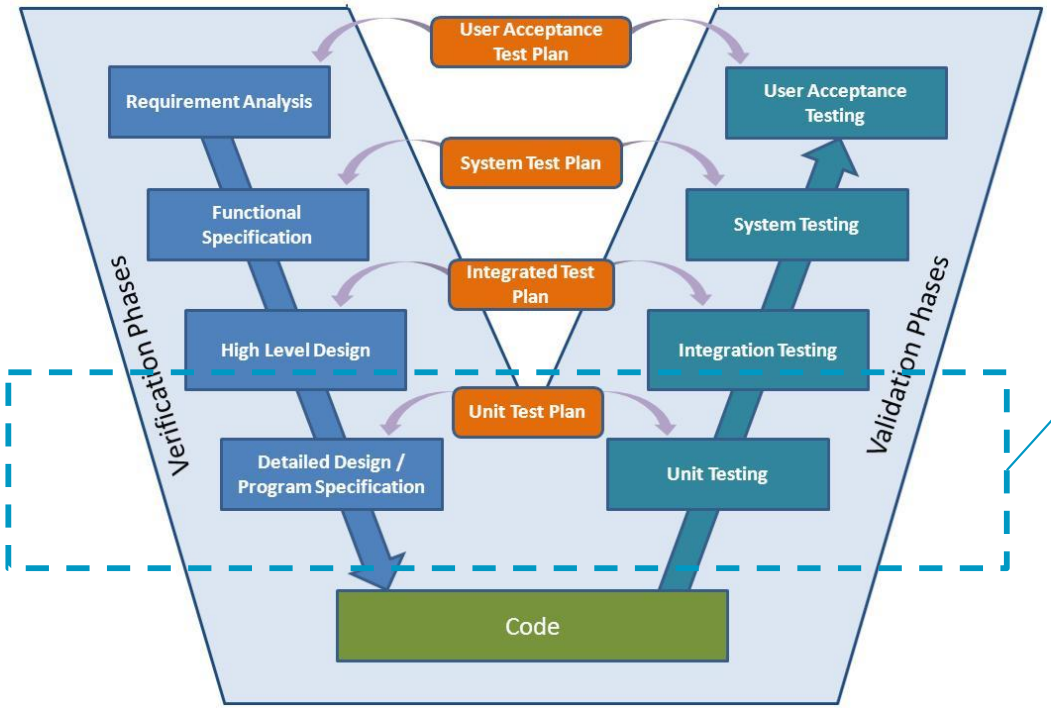
- Appendix A

TESTING PURPOSES

- Check that system behaves as expected
  - Gives correct results
  - Does not crash
- Typical questions
  - What is expected?
  - What are the correct results?
  - How to cover all circumstances? (Tesla's white truck)

KINDS OF TESTS: THE V-MODEL

DESIGN AND TESTING HAND-IN-HAND



Our realm in this lecture

Source: CrackMBA, http://crackmba.com/v-shaped-model/

KINDS OF TESTS

- System test, typically performed by testing team
  - Complete system, final configuration
  - Test functional behavior, e.g., use cases
- Unit test, typically performed by developer of unit
  - Class, method, etc. in isolation
  - Test unit's behavior
- Other tests: Acceptance, Performance, Stress, Platform, Usability, Integration

TEST PLAN

Goal

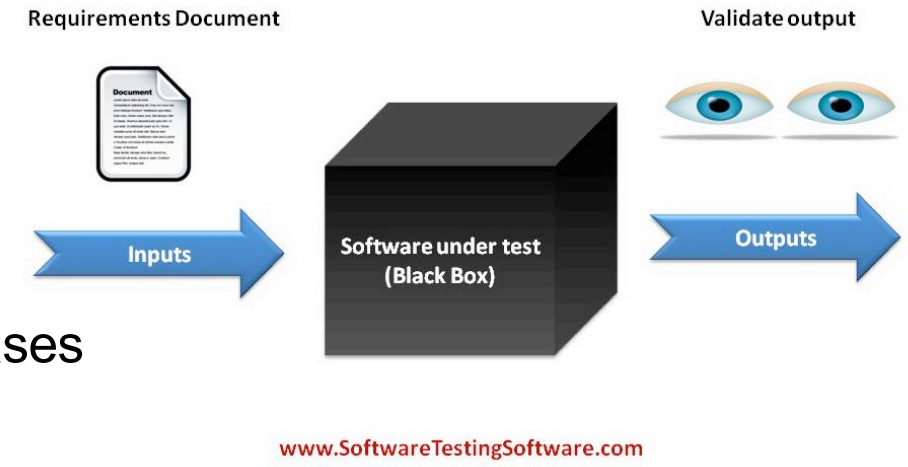
- Systematically test whether the system or unit
  - Behaves as expected if used in specified way

Contents

- List of execution scenarios → test cases
- How should it be carried out
- What is the expected result

TEST PLAN: SYSTEM TESTING

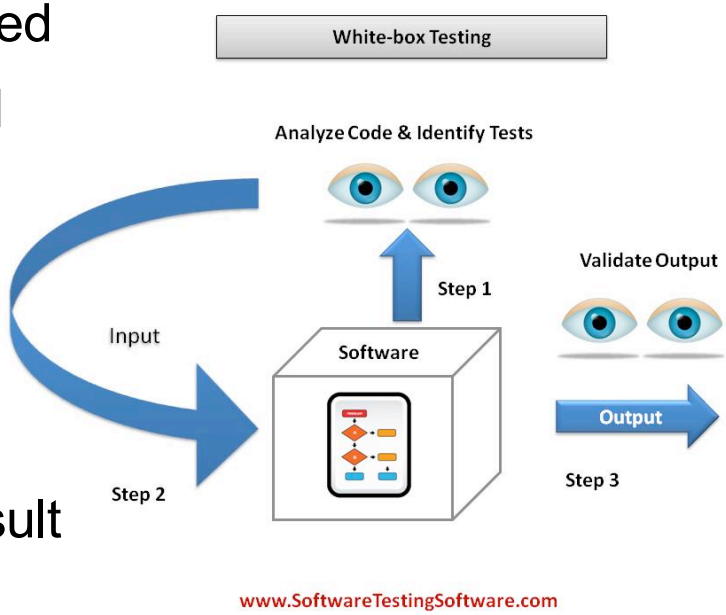
- Use cases to derive test cases
- To consider
  - Alternative flows in use cases
  - Different values used in use cases
  - Additional requirements
- System treated as black box
  - Internal structure is unknown
  - Validation by observing behavior



TEST PLAN: UNIT TESTING

For the programming project, you will have to perform system and unit tests!

- One test plan for each class to be tested
  - Multiple test cases for each method
- Consider method contracts (preconditions and postconditions)
- Treat method as white box
  - Internal structure is known
  - Validation by inspecting method result and/or instance/class variables



HOW TO TEST THIS CLASS?

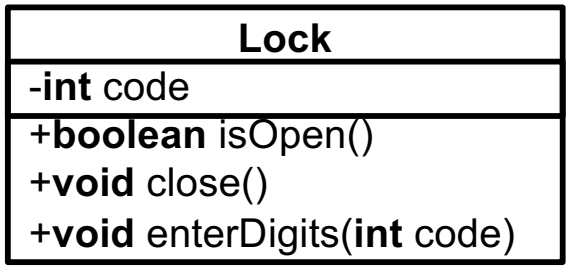
Lock
-**int** code
+**boolean** isOpen() +**void** close() +**void** enterDigits(**int** code)

TEST CASES

FOUR CASES

1. Attempt to open (calling `enterDigits`) in open state
   - Postcondition: `isOpen()` returns `true`
2. Attempt to open (calling `enterDigits`) in closed state
   - Postcondition: `isOpen()` returns `true` if and only if code is correct
3. Attempt to close (calling `close`) in open state
   - Postcondition: `isOpen()` returns `false`
4. Attempt to close in closed state
   - Postcondition: `isOpen()` returns `false`

Not so trivial!



TEST CASE 2: OPEN IN CLOSED STATE

- Test case 2 can be split up. For example, for 3 digits
  - 1000 possibilities for `this.code`
  - 1000 possibilities `int code` passed to `enterDigits`
  - 1,000,000 combinations!

- Do we really have to test all these alternatives?

TEST CASE 2: OPEN IN CLOSED STATE

- Do we have to test for all possible (combinations of) values?
  - No! Select representative values
- What is representative?
  - Group values for which the unit under test behaves the same
  - Pick samples from group: arbitrary value and boundaries (max/min)
- Lock example
  - Arbitrary value (123) and boundary values for `code` and `enterDigits` (0 and 999) reduce this to 9 combinations!

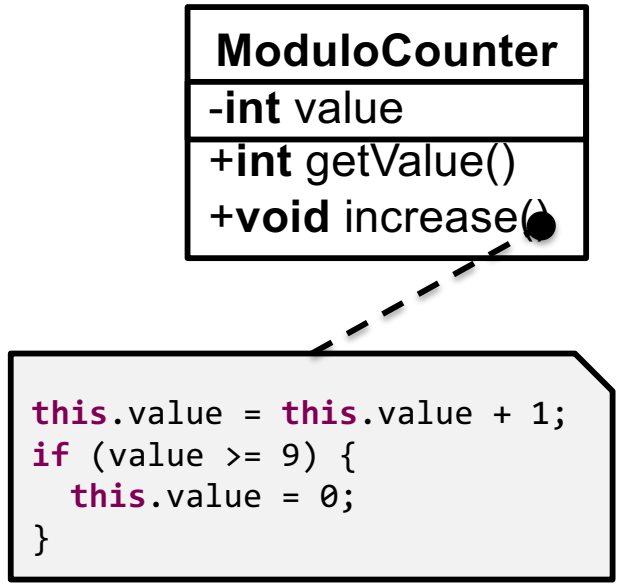
BOUNDARY CASES

- We often make mistakes at boundaries, the so called off-by-one errors

Example: `ModuloCounter` class, method increase

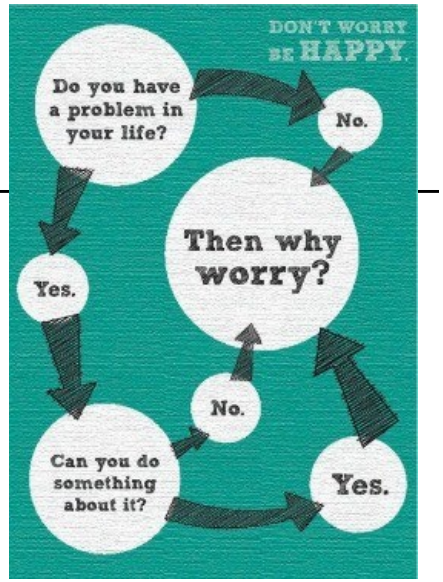
  - Count from 0 up to 9 modulo 10
  - When value is 9, increase returns to 0

- Cases to test
  - Invoke increase 0 times, 9 times, 10 times (boundaries), and invoke 5 times (arbitrary)



Off-by-one error!
Jumps from 8 to 0

TESTING FOR UNEXPECTED SITUATIONS



- Primarily test for correct behaviour
  - Input within specified bounds
  - Actions occur in the expected order (“happy flow”)
- Sometimes we want to test robustness
  - Nothing bad happens when input is wrong
  - For instance, for data entered through the user interface

TESTING FOR ROBUSTNESS

- Can only be properly tested if expected behaviour for wrong input is specified, by, for example
  - Using default values in case of wrong input
  - Throwing exceptions
- Warning!
  - Behaviour outside precondition encourages sloppiness
  - Programmers can start to rely on default behaviour

FROM THEORY TO PRACTICE: JUNIT

JUNIT 5

- JUnit is a library consisting of classes and annotations to facilitate the implementation of unit tests (typically to test a class)
- Current version: JUnit 5 (also called Jupiter), also supports JUnit 4
- JUnit 5 is already installed in Eclipse, but you must add it to the Build Path of your project in order to be able to use it
- New JUnit 5 test classes can be generated with an Eclipse wizard
  - For more information on how to set up JUnit and generate test cases see the manual!

USING JUNIT 5

RECOMMENDATION: ONE JUNIT TEST CLASS FOR EACH CLASS

Use Java annotations to tell JUnit what to do with the methods

- `@BeforeEach: public void` method executed before every test case, typically called `setup()`
  - Sets up test context that every test method starts with, so that different tests do not interfere → initialisation!
- `@Test: public void` method that implements a test case
  - Individual success/failure reported when running the test class
  - Method name typically `test<Something>()`

@TEST METHOD

- What should be in the test method body?
  - Call methods of unit under test, according to the test case scenario
  - Call assertions (e.g., `assertEquals(expectedValue,actualValue)`) to check outcome
  - Assertions are static methods defined in package `org.junit.jupiter.api.Assertions`

JUNIT EXAMPLE

MODULOCOUNTER CLASS WITH STUB METHOD IMPLEMENTATIONS

```
public class ModuloCounter {
    /**
     * Creates a new ModuloCounter, initialised to 0
     */
    public ModuloCounter() {
    }

    /**
     * Returns the current value of the counter
     */
    public int getValue() {
        return -1; // -1 is “impossible” for a ModuloCounter
    }

    /**
     * Increments the current value by one, resetting to 0
     * in case value 10 would have been reached
     */
    public void increase() {
    }

}
```

JUNIT 5 EXAMPLE

MODULOCOUNTERTEST JUNIT CLASS

```
class ModuloCounterTest {

    private ModuloCounterCorrect counter;

    @BeforeEach
    void setUp() throws Exception {
        counter = new ModuloCounterCorrect();
    }

    @Test
    public void testZero() {
        assertEquals(0, counter.getValue());
    }

    @Test
    public void testNine() {
        for (int i = 0; i < 9; i++) {
            counter.increase();
        }
        assertEquals(9, counter.getValue());
    }
}
```

Setup method (`@BeforeEach` annotation)

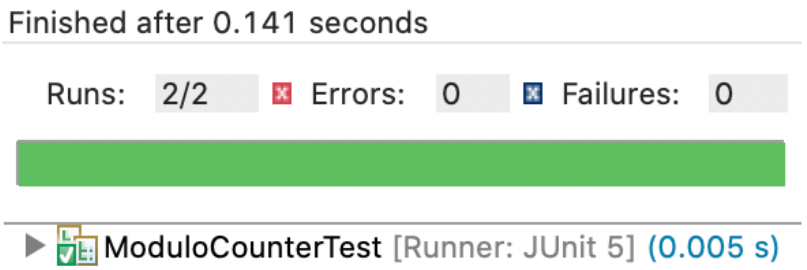
Test case: increase zero times (`@Test` annotation)

Test case: increase `counter` nine times, reinitialised in `setup()`

JUNIT EXAMPLE

MODULOCOUNTER WITH A SERIOUS BUG

```
public class ModuloCounter {
    private int value = 0;

    /**
     * Creates a new ModuloCounter, initialised to 0
     */
    public ModuloCounter() {
    }

    /** Returns the current value of the count
    public int getValue() {
        return value;
    }

    /**
     * Increments the current value by one, resetting to 0
     * in case value 10 would have been reached
     */
    public void increase() {
        value++; // will it pass our test?
    }
}
```

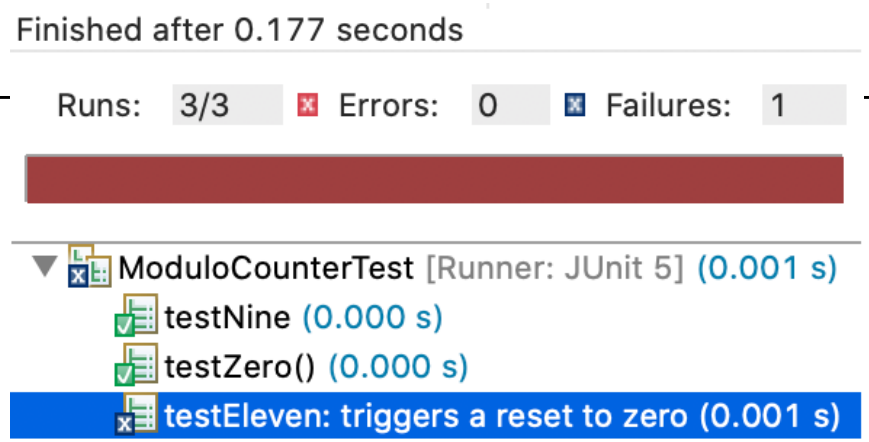


We need to improve our tests!!!

JUNIT EXAMPLE

ADDITIONAL TEST CASE

```
@Test
@DisplayName("testNine")
public void testNine() {
    for (int i = 0; i < 9; i++) {
        counter.increase();
    }
    assertEquals(9, counter.getValue());
}

@Test
@DisplayName("testEleven: triggers a reset to zero")
public void testEleven() {
    for (int i = 0; i < 11; i++) {
        counter.increase();
    }
    assertEquals(1, counter.getValue());
}
```



Test cases:
1. increase nine times
2. increase it *eleven* times, which should trigger the “resetting to zero” action

RELATION WITH POSTCONDITIONS

- Postconditions of method `increase()`

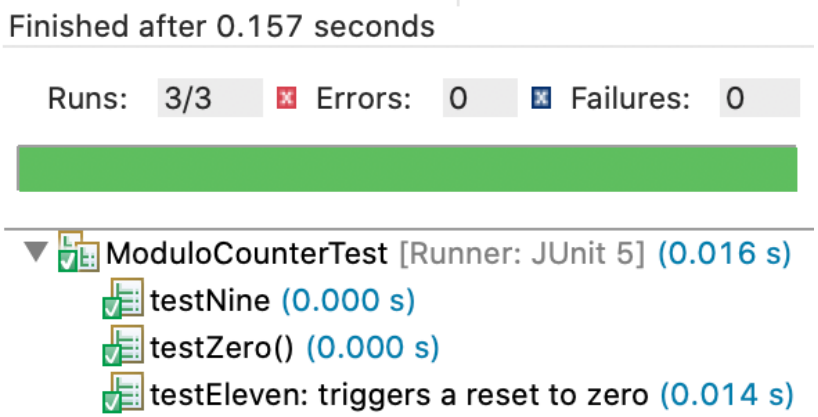
```
@ensures old.getValue() < 9 ==> getValue() == old.getValue() + 1
@ensures old.getValue() == 9 ==> getValue() == 0
```

- Can be made to correspond to the test cases
  - Methods `testNine()` and `testEleven()` or `testTen()`, respectively

JUNIT EXAMPLE

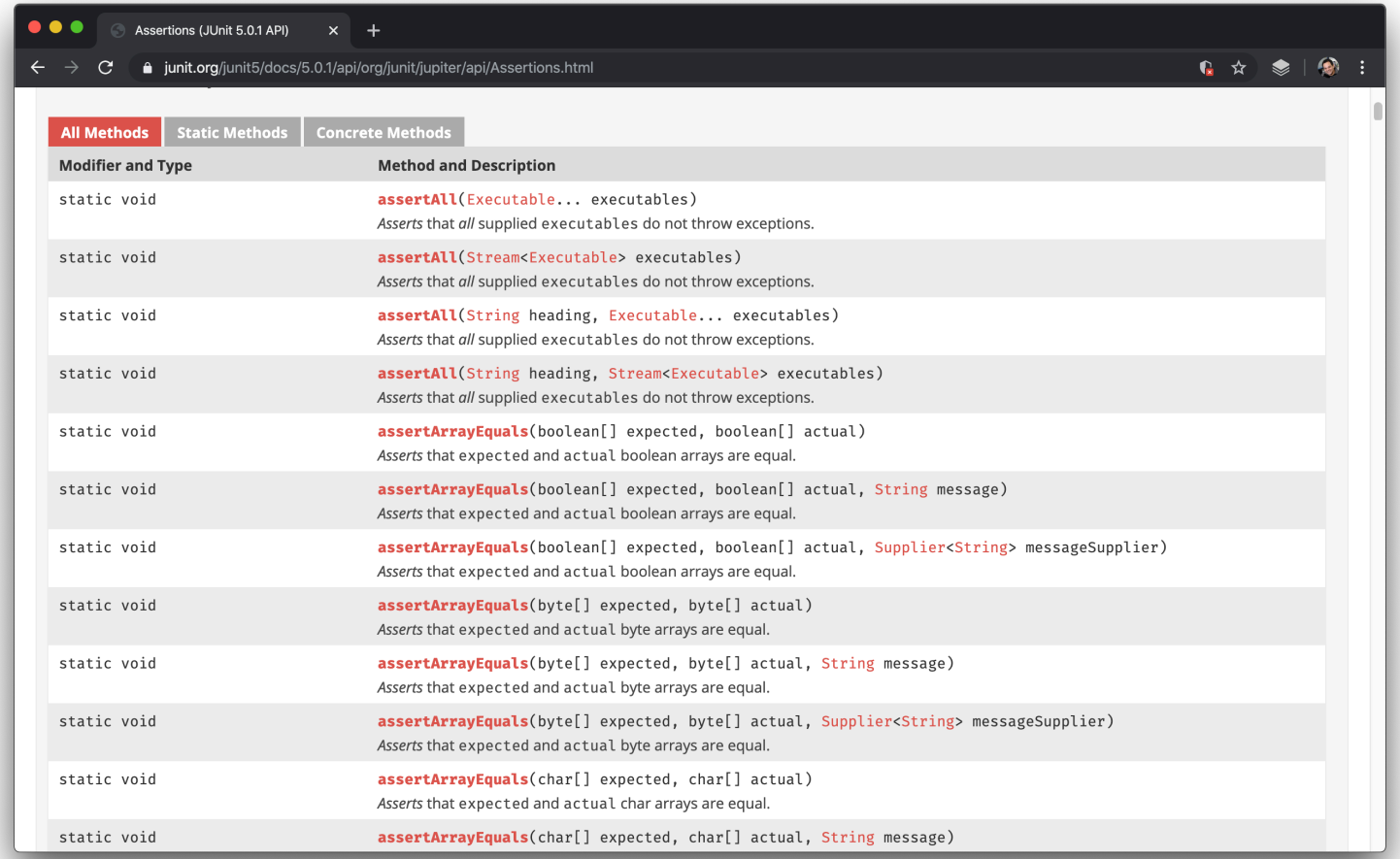
MODULOCOUNTER WITH CORRECT METHODS

```
public class ModuloCounter {
    private int value = 0;

    /** Creates a new ModuloCounter, initialised to 0 */
    public ModuloCounter() {
    }

    /** Returns the current value of the counter */
    public int getValue() {
        return value;
    }

    /**
     * Increments the current value by one, reset
     * in case value 10 would have been reached
     */
    public void increase() {
        if (value == 9) {
            value = 0;
        } else {
            value++;
        }
    }
}
```



JUNIT 5 ASSERTIONS

CHECK THE API!



TESTING AND DEVELOPMENT PROCESS

Approach of this module!

- Iterative process
  - Develop tests even before you start to implement (test-driven development) based on the specifications
  - Perform unit tests immediately after functionality is complete
  - Extending the implementation requires extending test plan
  - Test results lead to further coding
  - Test coverage results lead to extended test plan

TESTING AND DEVELOPMENT PROCESS

- Regression
  - Bugs may be introduced in previously correct code
  - Rerun your tests on a regular basis (regression testing)
  - You will only ever do this if it is automated → JUnit is your friend!
- Philosophies
  - "Code a little, test a little"
  - "Test a little, code a little" (test-driven development, test-first)
- Watch out: Testing can take up to 50 % of development time

TESTING QUALITY
QUALITY OF YOUR TESTS

- "If my code passes all tests, I'm done!" Is this correct?
  - No, of course not
  - You can never test for all possibilities in realistic systems
- How to assure my tests have appropriate quality?
  - Systematically develop tests based on contracts
  - Apply best practices when using JUnit
    - Many test classes, small test cases, meaningful test case name
  - Carry out test reviews

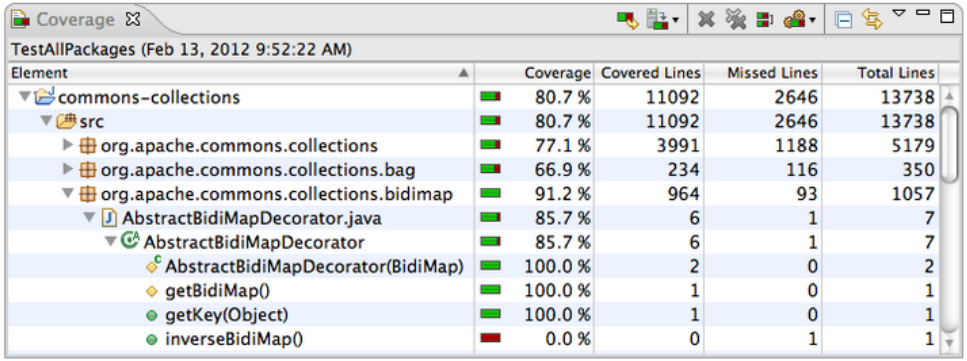
TESTING QUALITY

TESTING COVERAGE

- Measure test coverage! → Emma Eclipse plugin
- Statement coverage
  - Each statement executed at least once
- Branch coverage
  - Each branch traversed (and every entry point taken) at least once
- Path coverage
  - All program paths traversed at least once

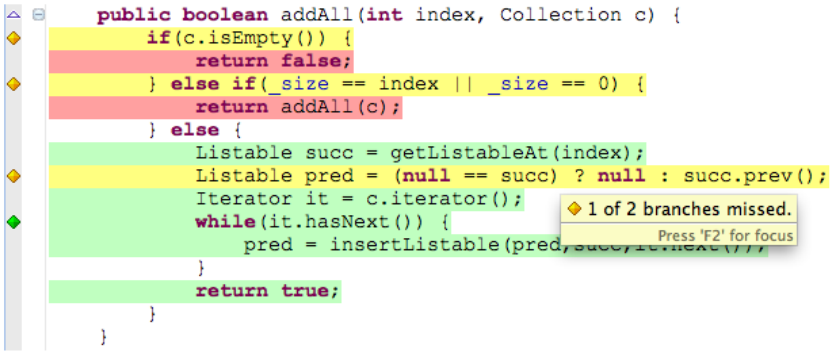
ECLEMMA TOOL (EMMA ECLIPSE PLUGIN)

EXECUTED WITH COVERAGE AS… → JUNIT TEST



Element	Coverage	Covered Lines	Missed Lines	Total Lines
commons-collections	80.7 %	11092	2646	13738
src	80.7 %	11092	2646	13738
org.apache.commons.collections	77.1 %	3991	1188	5179
org.apache.commons.collections.bag	66.9 %	234	116	350
org.apache.commons.collections.bidimap	91.2 %	964	93	1057
AbstractBidiMapDecorator.java	85.7 %	6	1	7
AbstractBidiMapDecorator	85.7 %	6	1	7
AbstractBidiMapDecorator(BidiMap)	100.0 %	2	0	2
getBidiMap()	100.0 %	1	0	1
getKey(Object)	100.0 %	1	0	1
inverseBidiMap()	0.0 %	0	1	1

In addition, EclEmma highlights the execution status directly in the Java source editors: Green lines were fully executed, yellow lines were executed partially only and red lines were not hit at all. Little diamonds symbols in the editor's ruler, on the left to the source code, show the execution status of branches in your code (e.g.. for if and switch statements):



- Green: ok, fully executed
- Yellow: Only partially executed
- Red: Not executed at all

- Colored diamonds on the left show coverage of branches (i.e., if statements)

http://www.eclemma.org

CODE COVERAGE
DIFFICULTIES

- 100% coverage is difficult to achieve, especially for branch or path coverage too many test cases would be needed
- Examples of hard-to-cover code
  - Synthetic code generated by compiler
  - Code that handles exceptional situations
  - Conditions that combine cases (using `&&` and `||`)
  - Code of your test classes

CODE COVERAGE

SOME HINTS

- Try to achieve as high statement coverage as possible (e.g., > 80%)
- Keep testing in mind when you design your code: test-driven design!
- This might involve refactoring (reorganising) your code to make it more easily testable

TAKE HOME MESSAGES

- Behaviour of methods can be (precisely) specified
  - Precondition: what should hold when method is called
  - Postcondition: what implementation guarantees when method finishes
  - Invariant: property that holds throughout life of object
- Specifications can be checked during execution
  - Insert checks manually (e.g., using `assert` statements)
  - Use dedicated tool support

TAKE HOME MESSAGES

- Tests relevant for this module: system tests, unit tests
- Test plan
  - System tests based on use cases
  - Unit tests based on method contracts
- Choose representative values in tests: boundary cases, arbitrary values, exceptional cases
- Use tests systematically: regression tests, coverage, etc.
- Test-driven design → specify, write tests, implement and run tests!