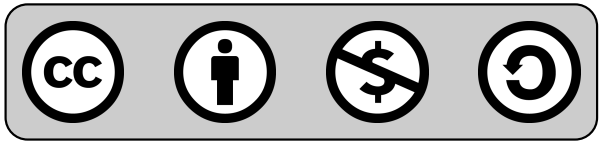
IN323 Software Engineering

Test Driven Development

Christophe Garion

DMIA – ISAE

License CC BY-NC-SA 3.0



Some CS problems. . .

Ariane 5 first test fly: 38 billions francs

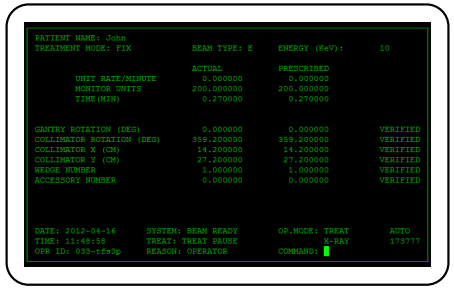
- reusing Ariane 4 fly software
- flight domains not identical
- overflow when converting a 64 bits float to 16 bits integer:

```
P_M_DERIVE(T_ALG.E_BH) := UC_16S_EN_16NS
  (TDB.T_ENTIER_16S ((1.0/C_M_LSB_BH) *
  G_M_INFO_DERIVE(T_ALG.E_BH)));
```

Lions, Jacques-Louis et al. (1996).
Flight 501 Failure.
Report.
European Space Agency.
http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf.

Some CS problems. . .

Therac-25 radiation therapy machine



- at least 6 patients were given massive doses of radiation
- race condition with a one-byte counter overflowing
- software written in assembly language

Leveson, Nancy G. and Clark S. Turner (1993).
"An investigation of the Therac-25 accidents".
In: **IEEE Computer** 26.7,
Pp. 18–41.

Some CS problems. . .

Patriot missile failed to intercept Scud on 02/25/91 in Saudi Arabia

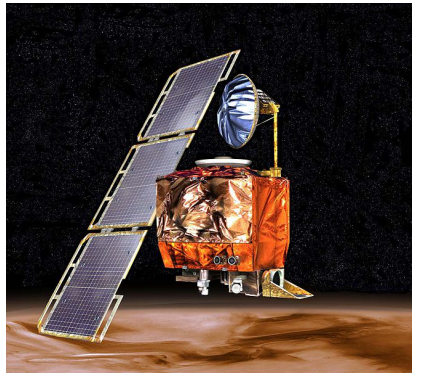
- 28 soldiers killed
- internal clock has drifted by 1/3 of a second
- resulting miss error: 600m
- roundoff error in assemby code

Skeel, Robert (1992).
"Roundoff Error and the Patriot Missile".
In: **SIAM News** 25.4,
P. 11.
http://mate.uprh.edu/~pnm/notas4061/patriot.htm.

Some CS problems. . .

Mars Climate Orbiter space probe disintegrated in Mars atmosphere

- using non SI units: lbf×s instead of N×s
- correctly specified in the contract between NASA and Lockheed



Stephenson, Arthur G. et al. (1999).
Mars Climate Orbiter Mishap Investigation Board – Phase I Report.
Report.
NAtional Space Agency.
`ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf`.

Some CS problems. . .

USS Yorktown was the testbed for Smart Ship program

- crew member entered a 0 value in a database field
- divide-by-zero error
- all machines down on the network
- ship's propulsion system out for more than 2h

Slabodkin, Gregory (1999).
Software glitches leave Navy Smart Ship dead in the water.

`http://gcn.com/Articles/1998/07/13/Software-glitches-leave-Navy-Smart-Ship-dead-in-the-water.aspx`.

Some CS problems... or not!

Common misconception

That's computer scientists fault, blablabla.

Not really, those problems may be have been triggered by humans, system engineering errors etc.

But of course, they all involve computers! Not the same ☺

Software correctness

Definition (correctness)

A software is **functionnaly correct** wrt a specification if the software outputs from given inputs respects its specification.

How to ensure software correctness?

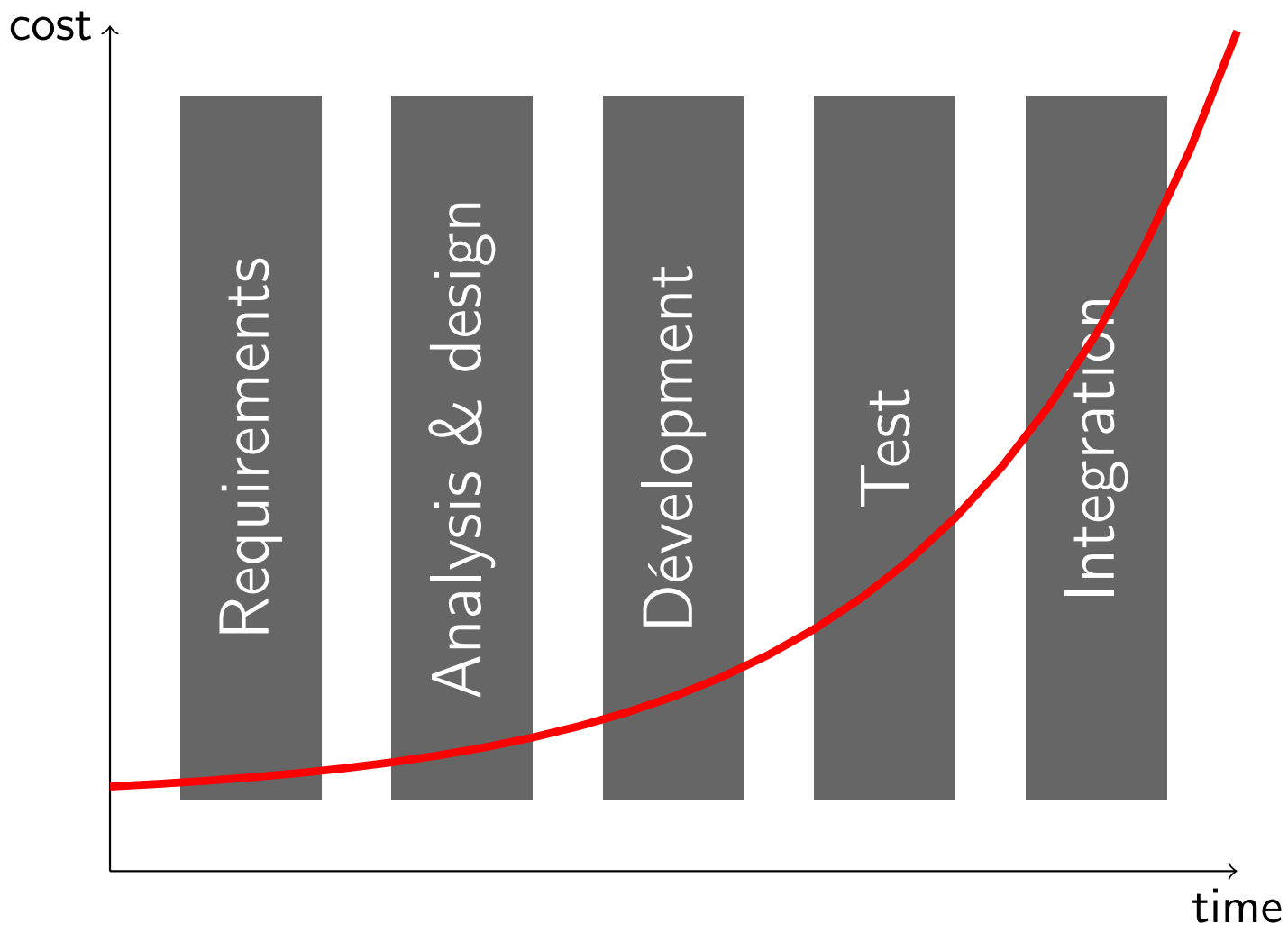
- using complete **tests**
  - ➥ e.g., writing unit tests for each method of a class
- using **formal specifications**
  - ➥ mathematical language
  - ➥ unambiguous specifications
  - ➥ **[IN324] Software Validation**

N.B.

These two approaches are not exclusive.

Testing in the software lifecycle

Classicaly, developers use the « develop-test-debug » cycle:



Different kinds of tests

Two test families

- **black-box** tests: internal structure of application is not known (**functional**)
- **white-box** tests: tests internal structures of application

Different kinds of tests

Differents granularities for tests

- **unit** tests: basic components of the system.
- **integration** tests: interaction between the application modules.
- **fonctional** tests: correctness of the application wrt a specification (+ performance, security etc.)
- **system integration** tests: test the system in production
- **user-acceptance** tests: final users validate (or not) the application.
- **non-regression** tests: verifying that a change in a part of the application does not impact the rest of the system.

Properties of good tests

Hunt, A. and D. Thomas (2003).
Pragmatic Unit Testing in Java with Junit.
Pragmatic Programmers.
Pragmatic Bookshelf.

Properties of good tests

A-TRIP properties

- **automatic**: automatic invocation and verification
- **thorough**: test everything that may break
- **repeatable**: every test should be able to run over and over again producing the same result
- **independent**: tests must be independent from the environment. Test one feature at a time
- **professionnal**: test must have the same quality that the code

Some references


Beck, K. (2002).
Test driven development: by example.
Addison-Wesley Professional.


Fowler, M. (1999).
Refactoring: Improving the Design of Existing Code.
Addison-Wesley Professional.


Binder, R. V. (1999).
Testing object-oriented systems: models, patterns and tools.

Addison-Wesley Professional.

Some references


Hunt, A. and D. Thomas (2003).
Pragmatic Unit Testing in Java with Junit.
Pragmatic Programmers.
Pragmatic Bookshelf.


Grenning, James W. (2011).
Test Driven Development for Embedded C.
Pragmatic Programmers.
Pragmatic Bookshelf.


Meszaros, Gerard (2007).
xUnit Test Patterns; Refactoring Test Code.
Addison-Wesley.

Plan

Outline

Outline

TFD: principles

When testing?

Principle (*test-first development*)

Write a test that **must fail** before writing or changing application code.

Test that must fail?

After writing/changing application code, the test must pass.

TFD: principles

When testing?

Principle (*test-first development*)

Write a test that **must fail** before writing or changing application code.

Advantages

- **testable software**: each functionality has an associated test
- **coherent**: if it is difficult to write a test, the design is not good...
- **trust**: how can you trust a code that has not been tested?
- **rhythm**: you know what to do after having written your test
- **documentation**: tests are documentation for code

Red/Green/Refactor principle

What are the associated activities for developers?

Beck, K. (2002).
Test driven development: by example.
Addison-Wesley Professional.

Red/Green/Refactor principle

What are the associated activities for developers?

Red

- write a test that verifies (part of) the behaviour of the code you want to write
- this test **must** fail!

Red/Green/Refactor principle

What are the associated activities for developers?

Green

- write the **minimum** code necessary to pass the test
- do not make big changes to your design/code...
- but your code may be "unreadable" now...

Red/Green/Refactor principle

What are the associated activities for developers?

Refactor

- modify code aspect **without changing its behaviour**
- code more **readable**, more **maintainable** and more **extensible**

Summary

Dev. activities for adding/changing code

1. write the test **justifying the code**
2. the test must fail
3. add/change code as few as possible
4. the test must pass
5. refactor
6. the test must pass

Continuous integration

- verify that your version passes tests
- integrate
- verify that all tests pass

Outline

Red: the Right-BICEP principle

Principle

The **Right**-BICEP!

Principle (Right-BICEP)

Right: are the outputs **correct**?

Those tests are often the **easiest** to write.
They are sometimes inclued in the **specifications** ...
Otherwise, you have to answer the following question: « if the program executes correctly, how do I know it? »

Do not hesitate to use **test files** if data is too big.

Red: the Right-BICEP principle

Principle

The Right-**B**ICEP!

Principle (Right-BICEP)

Boundary conditions: verify that the boundary conditions are **CORRECT** (cf. next slides)

Red: the Right-BICEP principle

Principle

The Right-B**I**CEP!

Principle (Right-BICEP)

Inverse Relationships: verify inverse relationships

Examples:

- $(\sqrt{x})^2 = x$ (with an bounded error)
- looking for an information in a database: use a SGBD routine

Red: the Right-BICEP principle

Principle

The Right-BI**C**EP!

Principle (Right-BICEP)

Cross-check: use other tools/algorithms to verify

Example: use a classical/library routine to compute square root

Red: the Right-BICEP principle

Principle

The Right-BIC**E**P!

Principle (Right-BICEP)

Error conditions: force error conditions

- invalid parameters
- network errors
- lack of memory
- system load

Mock objects are sometimes needed (cf. dedicated section).

Red: the Right-BICEP principle

Principle

The Right-BICE**P**!

Principle (Right-BICEP)

Performance

Not the performance itself, but for instance the growing of input size etc.

Boundary conditions

How to verify **boundary conditions** (conditions in which the application may have a non nominal behaviour)?

Principle (CORRECT)

Conformance: should the value respect a given **format**?

Some typical examples:

- email adress (`c.garion@isae.fr`), what happens if there is no `@` symbol (look at RFC822...)?
- a csv file, what happens if there is no header?

Boundary conditions

How to verify **boundary conditions** (conditions in which the application may have a non nominal behaviour)?

Principle (CORRECT)

Ordering: is the data set **ordered**?

Examples:

- what happens if the searched value is in first position, in last position?
- what happens if the data is not ordered as expected?

Boundary conditions

How to verify **boundary conditions** (conditions in which the application may have a non nominal behaviour)?

Principle (CORRECT)

Range: are there some **min and max bounds** for the value?

Examples:

- a value represents degrees, you should encapsulate the value in a class instead of using `int` value
- interdependence between values
- use **invariants**

Boundary conditions

How to verify **boundary conditions** (conditions in which the application may have a non nominal behaviour)?

Principle (CORRECT)

Reference: does the code use an **external element**?

- for OO, this essentially concerns class state
- **preconditions** and **postconditions**

Boundary conditions

How to verify **boundary conditions** (conditions in which the application may have a non nominal behaviour)?

Principle (CORRECT)

Existence: does the value **exist**?

What happens if:

- the value is **`null`**, 0, an empty string?
- network is not available?
- expected file does not exist?

Boundary conditions

How to verify **boundary conditions** (conditions in which the application may have a non nominal behaviour)?

Principle (CORRECT)

Cardinality: is the **number** of values correct?

Classical example: « fence post errors »
Test with:

- 0 values
- 1 value
- more than 1 value

Boundary conditions

How to verify **boundary conditions** (conditions in which the application may have a non nominal behaviour)?

Principle (CORRECT)

Time: does everything happen in the **right order**? At the **right time**? With correct **correct deadlines**?

- scheduling (cf. IN322)
- absolute time
- concurrency problems (cf. IN325)

Red: when do you stop?

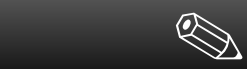
Principle

The tests you write should be small steps to understand your code.

Some principles:

- every module should have its test class/group
- in OO, simple getters and setters are not tested (but...)
- should you test private features: no *a priori*...
- interaction tests simplified via *mock objects*
- do not test the tests!
- use **code coverage** tools

An example in Java



Stack.java

```
public interface Stack {
    /**
     * Return and remove the most recent item from
     * the top of the stack.
     * @throws StackEmptyException if the stack is empty
     */
    public String pop() throws StackEmptyException;

    /**
     * Add an item to the top of the stack.
     */
    public void push(String item);
```

An example in Java

Stack.java

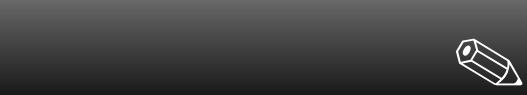
```
    /**
     * Return but do not remove the most recent
     * item from the top of the stack.
     * @throws StackEmptyException if the stack is empty
     */
    public String top() throws StackEmptyException;

    /**
     * Returns true if the stack is empty.
     */
    public boolean isEmpty();
}
```

Question

What are the tests you should write for this interface?

Some (unexhaustive) tests for `Stack`



- for an empty stack, `isEmpty` should return **`true`** and `pop` and `top` should raise exceptions
- from an empty stack, `push` a string and verify that `top` returns this string several times. Verify also that `isEmpty` returns **`false`**
- calling `pop` should remove the string and you should verify that this is the same object as the `push` parameter. Calling `pop` raises an exception
- rewrite those tests by adding several strings. Verify that you get the strings in the right order
- if you `push` **`null`**, calling `pop` returns **`null`**
- the stack can be used after some exceptions have been thrown

Outline

Green: what code do you write?

You have written a test **that fails**, you must write code that **passes the test**.

Principle (green)

- no big changes
- write just the necessary code to pass the test...
- ...even if it seems stupid

Fake It ('till You Make It)

The test fails? Return a **constant** ...

When the test passes, transform the constant in an expression using variables.

Exemple:

```
assertEquals(new MyDate("28.2.11"), new MyDate("1.03.11").yesterday());
```

Code:

MyDate.java

Fake It ('till You Make It)

The test fails? Return a **constant** ...

When the test passes, transform the constant in an expression using variables.

Exemple:

```
assertEquals(new MyDate("28.2.11"), new MyDate("1.03.11").yesterday());
```

Code:

MyDate.java

```
public MyDate yesterday() {
  // fake it!
  return new MyDate("28.2.11");
}
```

Fake It ('till You Make It)

The test fails? Return a **constant** ...

When the test passes, transform the constant in an expression using variables.

Exemple:

```
assertEquals(new MyDate("28.2.11"), new MyDate("1.03.11").yesterday());
```

Code:

MyDate.java

```
public MyDate yesterday() {
  // duplication between code and test
  return new MyDate(new MyDate("1.3.11").days() - 1);
}
```

Fake It ('till You Make It)

The test fails? Return a **constant** ...

When the test passes, transform the constant in an expression using variables.

Exemple:

```
assertEquals(new MyDate("28.2.11"), new MyDate("1.03.11").yesterday());
```

Code:

MyDate.java

```
public MyDate yesterday() {
  // duplication between code and test
  return new MyDate(this.days() - 1);
}
```

Green: some patterns

Triangulate

Use two or more examples to find the code to write.

Obvious implementation

If you know (or think you know...) the implementation, write it!

Beware to red bars...

One to Many

When you work with a collection, code first without collection, and then use a collection.

You should use the Isolate Change refactoring.

Green: a (magical) example...

Compute the Fibonacci sequence...

$f(0) = 0$

```
return 0;
```

$f(1) = 1$ et $f(2) = 1$

```
if (n == 0) { return 0; }
return 1;
```

$f(3) = 2$

```
if (n == 0) { return 0; }
if (n <= 2) { return 1; }
return 1 + 1;
```

```
if (n == 0) { return 0; }
if (n <= 2) { return 1; }
return f(n-1) + f(n-2);
```

Outline

Refactoring: definition

Fowler, M. (1999).
Refactoring: Improving the Design of Existing Code.
Addison-Wesley Professional.

— .**Refactoring home page**.
`http://www.refactoring.com`.

Refactoring: definition

Definition (refactoring)

Disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

When writing code, you can choose the easiest solution, because you have methods and techniques to make your code evolve to a more general or complex solution.

Idea: apply sequentially elementary code transformations.

You never do big changes in your code.

N.B.

Unit tests are important, as they guarantee that there is no error.

Refactoring: example

Consider the following code:

```
public void readConfigs() {

  open system config file
  read data

  open user config file
  read data
}
```

Refactoring: an example

Apply two times the "extract method" refactoring:

```
public void readConfigs() {
  ArrayList v1 = readSystemConfig();
  ArrayList v2 = readUserConfig();
}

public ArrayList readSystemConfig() {
  open system config file
  read data
}

public ArrayList readUserConfig() {
  open user config file
  read data
}
```

Refactoring: an example

Apply “parametrize method”:

```
public void readConfigs() {
  ArrayList v1 = readConfig("sys.cfg");
  ArrayList v2 = readUserConfig();
}

public ArrayList readConfig(String nom) {
  open config file nom
  read data
}

public ArrayList readUserConfig() {
  open user config file
  read data
}
```

Refactoring: an example

Eliminate the second method:

```
public void readConfigs() {
  ArrayList v1 = readConfig("sys.cfg");
  ArrayList v2 = readConfig("user.cfg");
}

public ArrayList readConfig(String filename) {
  open file filename
  read data
}
```

You can now overload `readConfig` to read from an URL for instance.

Which refactorings?

In **Refactoring: Improving the Design of Existing Code** , there are 72 refactorings!

For TDD, the most important are:

- Reconcile Differences
- Isolate Change
- Migrate Data
- Extract Method
- Inline Method
- Extract Interface
- Move Method
- Method Object
- Add Parameter
- Method Parameter to Constructor Parameter

Outline

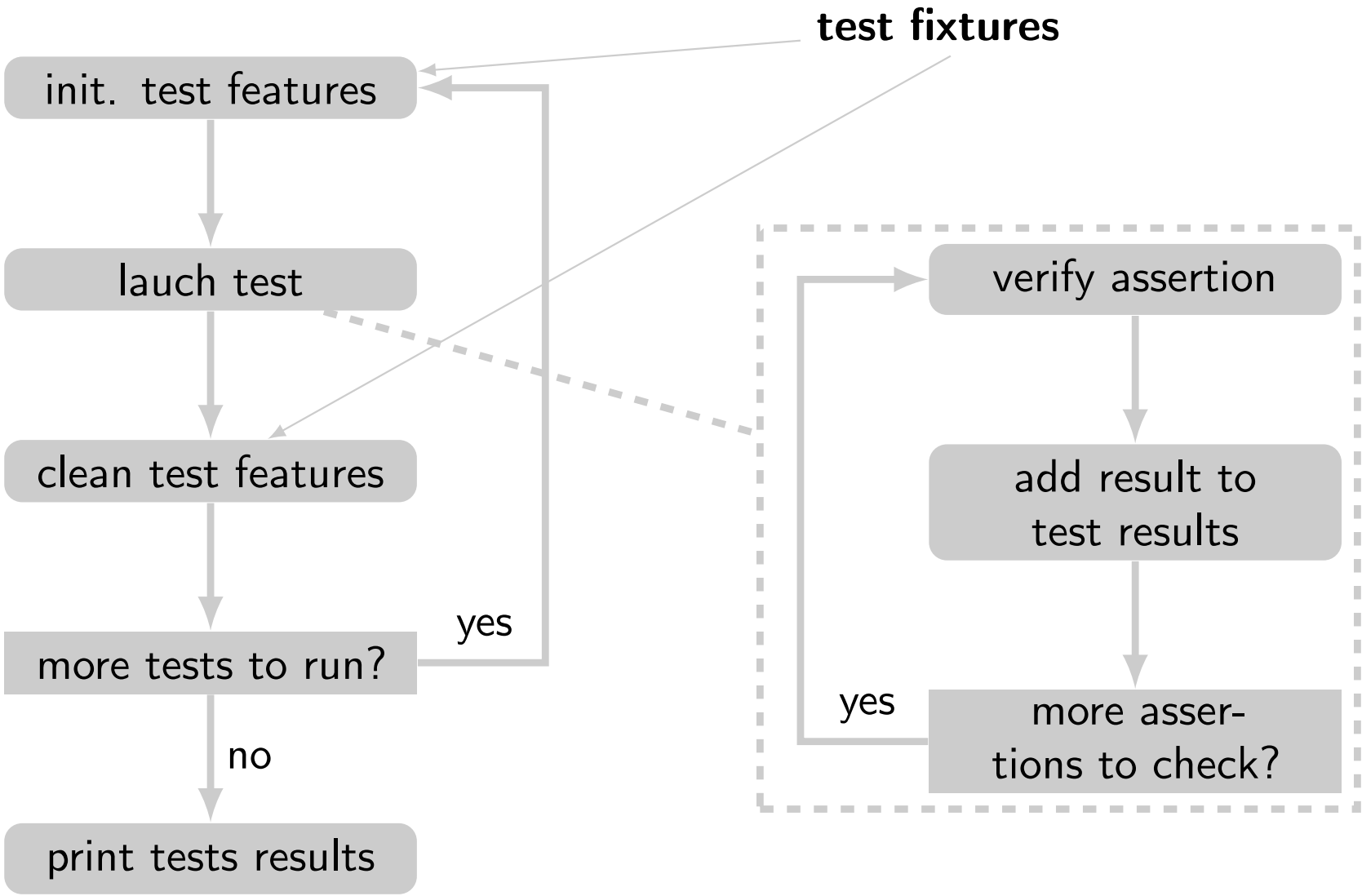
Expected qualities for a test framework

Link, J. (2004).
Unit testing in Java.
Elsevier Science.

Properties of a test framework

- A-TRIP properties
- the test programming language should be the same as the application programming language
- tests sources and application sources should be easily separated
- tests should be independent from each other
- you should be able to create tests suites easily runnable
- you should be able to easily verify that a test passes or fails

Lifecycle of a test framework



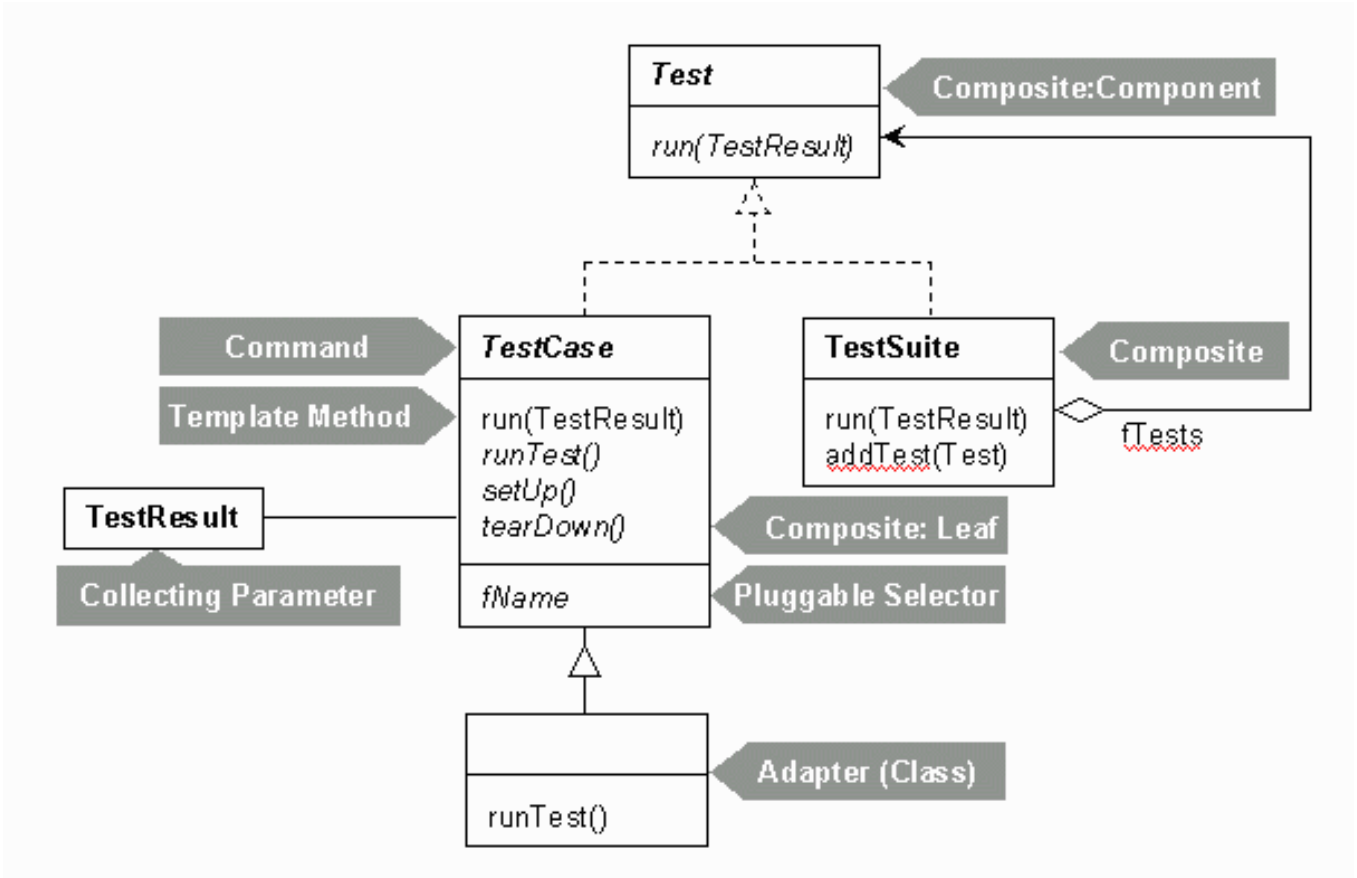
Some vocabulary

- **FUT**: Function Under Test
- **OUT**: Object Under Test
- **test fixtures**: initializing and cleaning the FUT/OUT and other needed parameters/objects
- **assertion**: a **logical** sentence, e.g. the string should be equal to this one, this float should be equal to this one with a certain error etc.
- **test suite**: a sequence of unit tests to perform. For instance, put the unit tests for a compilation unit (a class for instance) in a test group.

Writing your own test framework?

You can write your own test framework, using macros for instance in C (cf. **Test Driven Development for Embedded C**) or reflection/annotations in Java, but it is not so easy ☺

For instance, look at JUnit basic architecture (taken from `http://junit.org`):



Outline

The Unity framework

Unity is a unit test framework written in C. It uses a macro system to define tests, tests groups etc.

Karlesky, Mike, Mark Vandervoord, and Greg Williams (2013). **Unity**. http://throwtheswitch.org/white-papers/unity-intro.html.

Assertions available in Unity

There are lots of assertions available in Unity (see `unity.h`):

- **`TEST_ASSERT_EQUAL_INT`**`(expected, actual)`
- **`TEST_ASSERT_EQUAL_INT32`**`(expected, actual)`
- **`TEST_ASSERT_INT_WITHIN`**`(delta, expected, actual)`
- **`TEST_ASSERT_EQUAL_STRING`**`(expected, actual)`
- **`TEST_ASSERT_EQUAL_INT16_ARRAY`**`(expected, actual, n_elts)`
- **`TEST_ASSERT_DOUBLE_WITHIN`**`(delta, expected, actual)`
- **`TEST_ASSERT_EQUAL_INT_MESSAGE`**`(expected, actual, message)`
- **`TEST_FAIL_MESSAGE`**`(message)`
- `...`

Writing a simple test with Unity

unity_basics.c

```c
#include "unity.h"

void setUp(void) {
    printf("I am called before each test\n");
}

void tearDown(void) {
    printf("I am called after each test\n");
}

void test_silly1(void) {
    TEST_ASSERT_EQUAL_INT(2, 2);
    TEST_ASSERT_EQUAL_INT_MESSAGE(2, 1, "You idiot!");
    TEST_ASSERT_INT_WITHIN(2, 5, 4);
}

void test_silly2(void) {
    TEST_FAIL_MESSAGE("Not yet implemented!");
}
```

Writing a simple test with Unity

unity_basics.c

```
22 static void runAllTests(void) {
23     RUN_TEST(test_silly1, TEST_LINE_NUM);
24     RUN_TEST(test_silly2, TEST_LINE_NUM);
25 }
26
27 int main(int argc, char * argv[]) {
28     return UnityMain(argc, argv, runAllTests);
29 }
```

Writing a simple test with Unity

result

```
Unity test run 1 of 1
I am called before each test
:13:test_silly1:FAIL: Expected 2 Was 1. You idiot!
I am called after each test
I am called before each test
:18:test_silly2:FAIL: Not yet implemented!
I am called after each test

-----------------------
2 Tests 2 Failures 0 Ignored
FAIL
```

Using unity_fixture.h

unity_fixtures.c

```
 1 #include "unity_fixture.h"
 2
 3 TEST_GROUP(my_group);
 4
 5 static int something_i_want_to_use;
 6
 7 TEST_SETUP(my_group) {
 8     printf("I am called before each test\n");
 9 };
10
11 TEST_TEAR_DOWN(my_group) {
12     printf("I am called after each test\n");
13 };
14
15 TEST(my_group, silly1) {
16     TEST_ASSERT_EQUAL_INT(2, 2);
17     TEST_ASSERT_EQUAL_INT_MESSAGE(2, 1, "You idiot!");
18     TEST_ASSERT_INT_WITHIN(2, 5, 4);
19 };
```

Using unity_fixture.h

unity_fixtures.c

```
21 TEST(my_group, silly2) {
22     TEST_FAIL_MESSAGE("Not yet implemented!");
23 };
24
25 TEST_GROUP_RUNNER(my_group) {
26     RUN_TEST_CASE(my_group, silly1);
27     RUN_TEST_CASE(my_group, silly2);
28 };
29
30 static void runAllTests(void) {
31     RUN_TEST_GROUP(my_group);
32 }
33
34 int main(int argc, char * argv[]) {
35     return UnityMain(argc, argv, runAllTests);
36 }
```

Outline

The CppUTest framework

CppUTest is a C/C++ based xUnit test framework written in C++ but available for C. It also includes a mocking library (cf. section on mocking).

CppUTest contributors (2013).
CppUTest.
http://cpputest.github.io/.

N.B.

Beware, you have to compile your tests using g++.

Assertions available in CppUTest

Assertions available in CppUTest:

- `CHECK(boolean condition)`
- `CHECK_TEXT(boolean condition, text)`
- `CHECK_EQUAL(expected, actual)`
- `STRCMP_EQUAL(expected, actual)`
- `LONGS_EQUAL(expected, actual)`
- `BYTES_EQUAL(expected, actual)`
- `POINTERS_EQUAL(expected, actual)`
- `DOUBLES_EQUAL(expected, actual, tolerance)`
- `FAIL(text)`

N.B.

Not as complete as Unity...

Writing a simple test with CppUTest

cpp_basics.cpp

```
 1 #include "CppUTest/TestHarness.h"
 2 #include "CppUTest/CommandLineTestRunner.h"
 3
 4 extern "C" {
 5   // if you need to include, define something
 6   // for C, e.g. include header files
 7 #include <stdio.h>
 8 }
 9
10 static int something_i_want_to_use;
11
12 TEST_GROUP(my_group) {
13     void setup() {
14         printf("I am called before each test\n");
15     };
16
17     void teardown() {
18         printf("I am called after each test\n");
19     };
20 };
```

Writing a simple test with CppUTest

cpp_basics.cpp

```
22 TEST(my_group, silly1) {
23     LONGS_EQUAL(2, 2);
24     LONGS_EQUAL(2, 1);
25 };
26
27 TEST(my_group, silly2) {
28     FAIL("Not yet implemented!");
29 };
30
31 int main(int ac, char** av) {
32     return CommandLineTestRunner::RunAllTests(ac, av);
33 }
```

Writing a simple test with CppUTest

result

```
I am called before each test

cpp_basics.cpp:28: error: Failure in TEST(my_group, silly2)
        Not yet implemented!

I am called after each test
.I am called before each test

cpp_basics.cpp:24: error: Failure in TEST(my_group, silly1)
        expected <2 0x2>
        but was  <1 0x1>

I am called after each test
.
Errors (2 failures, 2 tests, 2 ran, 2 checks, 0 ignored, 0 filtered out, 0 ms)
```

Unity or CppUTest?

Unity

- written in C, no compilation/makefile problems
- lots of useful assertions
- generation of mocks less easy than with CppUTest

CppUTest

- written in C++, benefits from C++ capacities compared to pure C
- can be used to unit test C++ applications
- less assertions that Unity
- memory leak detection with macros overloading `free`/`malloc`
- better automatic mock support

Other (promising?) unit testing frameworks for C

Check contributors (2013).
Check – a unit testing framework for C.
http://check.sourceforge.net/.

Schneider, Andreas (2013).
cmocka – a unit testing framework for C with mock objects.

http://www.cmocka.org/.

Outline

A toy example...

Point.java

```
public class Point {

    private double x;
    private double y;

    public Point(double x_, double y_) {
        this.x = x_;
        this.y = y_;
    }

    public double getX() {
        return this.x;
    }

    public double getY() {
        return this.y;
    }

    public void translate(double dx, double dy) {
        this.x += dx;
        this.y += dx;
    }
}
```

JUnit presentation

JUnit is a Java unit testing framework.

JUnit Team (2013).
JUnit.
http://www.junit.org.

For each class to test, build a test class whose architecture is:

PointTest.java

```
import org.junit.Before;
import org.junit.After;
import org.junit.Test;
import static org.junit.Assert.*;

public class PointTest {


}
```

Assertions available in Junit

Basic assertions are written using static methods of the `Assert` class:

- `assertEquals(String expected, String actual)`
- `assertEquals(double exp, double actual, double delta)`
- `assertEquals(int expected, int actual)`
- `assertFalse(boolean condition)`
- `assertTrue(boolean condition)`
- `assertNotNull(Object object)`
- `assertNull(Object object)`
- `assertSame(Object expected, Object actual)`
- `assertNotSame(Object expected, Object actual)`
- ...

All those methods can take an first argument of type `String` representing a message to print when the assertion fails.

Assertions available in Junit

Advanced assertions use the `assertThat` method that uses a `Matcher` object.
`Matcher` objects are found in the `org.hamcrest.CoreMatchers` and `org.junit.matchers.JUnitMatchers` packages.

Examples of matchers:

- `org.hamcrest.CoreMatchers.allOf`
- `org.hamcrest.CoreMatchers.anyOf`
- `org.hamcrest.CoreMatchers.equalTo`
- `org.hamcrest.CoreMatchers.not`
- `org.hamcrest.CoreMatchers.sameInstance`
- `org.hamcrest.CoreMatchers.startsWith`
- `org.junit.Assert.assertThat`
- `org.junit.matchers.JUnitMatchers.both`
- `org.junit.matchers.JUnitMatchers.containsString`
- `org.junit.matchers.JUnitMatchers.everyItem`
- `org.junit.matchers.JUnitMatchers.hasItems`

JUnit: writing a test

A test method is JUnit is prefixed by the annotation **@Test**.
All methods prefixed by **@Test** are used in the tests suite defined by the test class.

Example:

PointTest.java

```
public class PointTest {

    @Test public void testTranslate() {
        Point p = new Point(1,2);

        p.translate(3,6);

        assertEquals(4.0, p.getX(), 0.0);
        assertEquals(8.0, p.getY(), 0.0);
    }
}
```

Test fixtures with JUnit

Test fixtures in JUnit are represented by methods prefixed by the **@Before** and **@After** annotations.

PointTest.java

```
public class PointTest {

    private Point p;
    private static final double EPS = 10E-9;

    @Before public void setUp() {
        this.p = new Point(1,2);
    }
```

There are also `@BeforeClass` and `@AfterClass` annotations.

A simple test class for Point

PointTest.java

```
import org.junit.Before;
import org.junit.After;
import org.junit.Test;
import static org.junit.Assert.*;

public class PointTest {

    private Point p;
    private static final double EPS = 10E-9;

    @Before public void setUp() {
        this.p = new Point(1,2);
    }

    @Test public void testTranslateBasic() {
        p.translate(3,6);
        assertEquals(4.0, p.getX(), EPS);
        assertEquals(8.0, p.getY(), EPS);
    }
```

A simple test class for `Point`

PointTest.java

```
    @Test public void testTranslateNullVector() {
        p.translate(0,0);
        assertEquals(1.0, p.getX(), EPS);
        assertEquals(2.0, p.getY(), EPS);
    }

    @Test public void testTranslateBack() {
        Point pold = new Point(p.getX(), p.getY());
        p.translate(2,3);
        p.translate(-2,-3);
        assertEquals(pold.getX(), p.getX(), EPS);
        assertEquals(pold.getY(), p.getY(), EPS);
    }
}
```

Running tests in JUnit

To "execute" `PointTest`, use the JUnit runner `org.junit.runner.JUnitCore`:

shell

```
[tof@suntof]~ $ java org.junit.runner.JUnitCore PointTest
JUnit version 4.11
...E
Time: 0,016
There was 1 failure:
1) testTranslateBasic(PointTest)
java.lang.AssertionError: expected:<8.0> but was:<5.0>
        at org.junit.Assert.fail(Assert.java:88)
        at org.junit.Assert.failNotEquals(Assert.java:743)
        at org.junit.Assert.assertEquals(Assert.java:494)
        ...

FAILURES!!!
Tests run: 3,  Failures: 1
```

Managing polymorphism

You can write Stack test class:

StackTest.java

```
public class StackTest {

    private Stack myStack;

    @Test public void testNewStack() {
        assertTrue(this.myStack.isEmpty());
    }
    ...
}
```

How to test StackTab?

StackTab.java

```
public class StackTab implements Stack {

    private String[] tab;
    ...
}
```

Liskov, JUnit and factory method

StackTest.java

```
abstract public class StackTest {

    private Stack myStack;

    public abstract Stack createStack();

    @Before public void setUp() {
        this.myStack = this.createStack();
    }

    @Test public void testNewStack() {
        assertTrue(this.myStack.isEmpty());
    }
    ...
}
```

Liskov, JUnit and factory method

StackTabTest.java

```
public class StackTabTest extends StackTest {

    public Stack createStack() {
        return new StackTab();
    }
}
```

And Liskov's principle is verified without effort...

Exceptions and “performance”

```
@Test(expected=java.lang.ArithmeticException.class)
public void divideByZero() throws ArithmeticException {
    21 / 0;
}

@Test(timeout=2000)
public void testPerfoButNotRealistic() {
    ...
}
```

Tests suite

You can launch several tests using a tests suite:

AllStacksTest.java

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

@RunWith(Suite.class)
@Suite.SuiteClasses({ stack.StackTabTest.class,
                      stack.StackLinkedListTest.class })

public class AllStacksTest {

}
```

Test suites are particularly useful for non-regression tests.

Parametrized tests

How to write a test using several input/output pairs?

ParamTest.java

```
public class ParamTest {

    private Class clzz;
    private String name;

    @Test
    public void verifyHierarchies() throws Exception {
        Class superClass = clzz.getSuperclass();
        assertEquals(superClass.toString(), name);
    }

}
```

Parametrized tests: constructor

ParamTest.java

```
import org.junit.*;
import static org.junit.Assert.*;
import org.junit.runners.Parameterized;
import static org.junit.runners.Parameterized.*;
import org.junit.runner.RunWith;
import java.util.*;

@RunWith(Parameterized.class)
public class ParamTest {

    private Class clzz;
    private String name;

    public ParamTest(Class clzz, String name) {
        this.clzz = clzz;
        this.name = name;
    }
```

Parametrized tests: generation

ParamTest.java

```
    @Test
    public void verifyHierarchies() throws Exception {
        Class superClass = clzz.getSuperclass();
        assertEquals(superClass.toString(), name);
    }

    @Parameters
    public static Collection hierarchyValues() {
        Object[][] param = new Object[][] {
            {Vector.class, "class java.util.AbstractList"},
            {String.class, "class java.lang.Object"} };

        ArrayList<Object[]> a = new ArrayList<Object[]>();
        a.add(param[0]);
        a.add(param[1]);

        return a;
    }
}
```

Want to do some combinatorics?

TheoriesTest.java

```
import static org.junit.Assume.assumeTrue;
import org.junit.experimental.theories.*;
import org.junit.runner.RunWith;

@RunWith(Theories.class)
public class TheoriesTest {

    @DataPoint
    public static String a = "a";

    @DataPoint
    public static String b = "bb";

    @DataPoint
    public static String c = "ccc";

    @Theory
    public void stringTest(String x, String y) {
        assumeTrue(x.length() >= 1);

        System.out.println(x + " " + y);
    }
}
```

OK, but I have different types...

TheoriesTypesTest.java

```
import static org.junit.Assume.assumeTrue;
import org.junit.experimental.theories.*;
import org.junit.runner.RunWith;

@RunWith(Theories.class)
public class TheoriesTypesTest {

    @DataPoints
    public static String[] a = { "a", "bb", "ccc" };

    @DataPoints
    public static Integer[] j = { 1, 2, 3 };

    @Theory
    public void doubleTest(String x, Integer y) {
        assumeTrue(x.length() > 1);

        System.out.println(x + " " + y);
    }
}
```

I want different test data for `String`

AllNames.java

```
import java.lang.annotation.*;
import org.junit.experimental.theories.*;

@Retention(RetentionPolicy.RUNTIME)

@ParametersSuppliedBy(NameSupplier.class)
public @interface AllNames {}
```

I want different test data for `String`

NameSupplier.java

```
import org.junit.experimental.theories.*;
import java.util.*;

public class NameSupplier extends ParameterSupplier {

    @Override
    public List getValueSources(ParameterSignature signature) {
        ArrayList result = new ArrayList();

        result.add(PotentialAssignment.forValue("Garion", "Garion"));
        result.add(PotentialAssignment.forValue("Hugues", "Hugues"));
        result.add(PotentialAssignment.forValue("Siron", "Siron"));

        return result;
    }
}
```

I want different test data for `String`

TheoriesSupplierTest.java

```
import static org.junit.Assume.assumeTrue;
import org.junit.experimental.theories.*;
import org.junit.runner.RunWith;

@RunWith(Theories.class)
public class TheoriesSupplierTest {

    @Theory
    public void testNames(@AllFirstNames String x,
                          @AllNames String y) {
        System.out.println(x + " " + y);
    }
}
```

Conclusion on test frameworks

Most of test frameworks are currently using the ideas developed for the xUnit testing frameworks, particularly JUnit:

- test fixtures
- powerful assertions
- tests easy to write/run
- automatic collection of results
- ...

But some difficulties remain, for instance testing concurrent programs.

Outline

Doubles and mock objects

When developing an application, you may

- not have access to a part of the application (notyet developed)
- use "expensive" technologies, e.g. network
- not want to use production systems, e.g. databases

How then to test your code?

To solve this problem, you can use **doubles** (like in movie production):

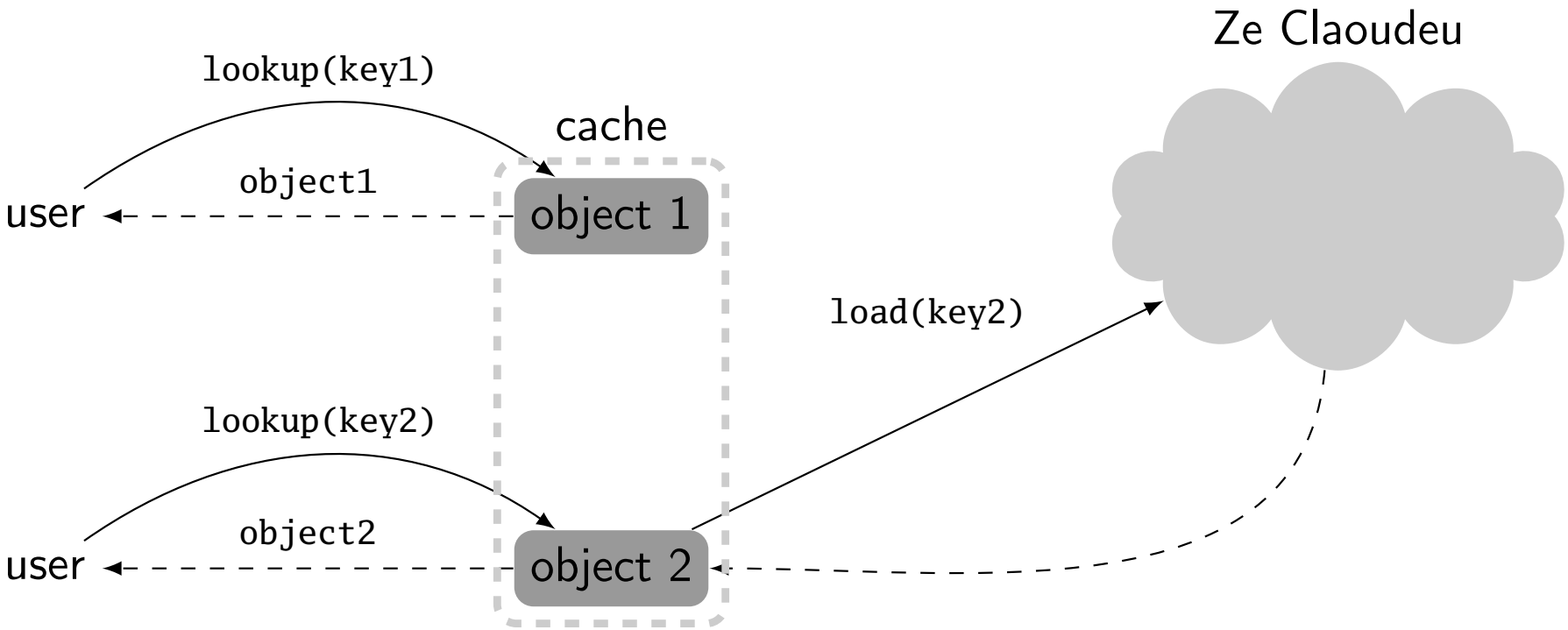
- **dummy**: only here for compilation purposes
- **fake**: functional but simplified
- **stubs**: return some predefined values for testing purpose
- **mocks**: a debug replacement for a real world object/function. Mocks are preprogrammed from a specification

Example used during the lecture

Example used: a cache system, where "objects" are referenced by keys.

Freeman, Steve et al. (2004).
"Mock roles, not objects".
In: **Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004**.
Ed. by John M. Vlissides and Douglas C. Schmidt.
ACM Press,
Pp. 236–246.
`http://jmock.org/oopsla2004.pdf`.

Example used during the lecture



Developing the cache without the loader

We want to develop the cache system (`Cache` object or `lookup` function) without the loader. To simplify, we use strings as keys and objects.

loader.h

```
#ifndef LOADER_H
#define LOADER_H

char* load(char *key);

#endif
```

Loader.java

```
public interface Loader {
    public String load(String key);
}
```

Outline

CMock presentation

CMock is a mocking framework for C using Unity.
It uses Ruby to generate C source code.

Karlesky, Mike, Mark Vandervoord, and Greg Williams (2013). **CMock intro**. http://throwtheswitch.org/white-papers/cmock-intro.html.

shell

```
[tof@suntof]~ $ ruby /usr/local/lib/cmock/lib/cmock.rb
   include/loader.h
Creating mock for loader...
```

Generated header file

mocks/Mockloader.h

```
/* AUTOGENERATED FILE. DO NOT EDIT. */
#ifndef _MOCKLOADER_H
#define _MOCKLOADER_H

#include "loader.h"

void Mockloader_Init(void);
void Mockloader_Destroy(void);
void Mockloader_Verify(void);




#define load_ExpectAndReturn(key, cmock_retval) load_CMockExpectAndReturn(__LINE__
void load_CMockExpectAndReturn(UNITY_LINE_TYPE cmock_line, char* key, char* cmock_

#endif
```

The cache source (with errors ☺)

cache.c

```c
#include "cache.h"
#include "loader.h"

char* lookup(char* key) {
    return load(key);
}
```

Using the mock in tests

cache_tests.c

```
 1 #include "unity_fixture.h"
 2 #include "Mockloader.h"
 3 #include "cache.h"
 4
 5 TEST_GROUP(cache);
 6
 7 static char *key1 = "key1";
 8 static char *key2 = "key2";
 9 static char *object1 = "object1";
10 static char *object2 = "object2";
11
12 TEST_SETUP(cache) {
13     Mockloader_Init();
14 };
15
16 TEST_TEAR_DOWN(cache) {
17     Mockloader_Verify();
18     Mockloader_Destroy();
19 };
```

Using the mock in tests

cache_tests.c

```
21 TEST(cache, load_object_not_cached) {
22     load_ExpectAndReturn(key1, object1);
23
24     TEST_ASSERT_EQUAL_STRING(object1, lookup(key1));
25 };
26
27 TEST(cache, load_objects_not_cached) {
28     load_ExpectAndReturn(key1, object1);
29     load_ExpectAndReturn(key2, object2);
30
31     TEST_ASSERT_EQUAL_STRING(object1, lookup(key1));
32 };
33
34 TEST(cache, load_object_cached) {
35     load_ExpectAndReturn(key1, object1);
36
37     TEST_ASSERT_EQUAL_STRING(object1, lookup(key1));
38     TEST_ASSERT_EQUAL_STRING(object1, lookup(key1));
39 };
```

Test results

test results

```
Unity test run 1 of 1
..tests/cache_tests.c:27:TEST(cache, load_objects_not_cached):FAIL:
   Function 'load' called less times than expected.
.tests/cache_tests.c:34:TEST(cache, load_object_cached):FAIL:
   Function 'load' called more times than expected.

-----------------------
3 Tests 2 Failures 0 Ignored
FAIL
```

Beware, order of expectations is important.

Outline

CppUTest mocking capacities

CppUTest has support for building mocks, with no code generation.
It uses C++ capacities to simplify the user interaction with the mocking code.

CppUTest contributors (2013).
CppUMock Manual.
http://cpputest.github.io/mocking_manual.html.

Beware (again), you have to compile as C++ code!

Using the mock in tests

cache_tests.cpp

```
 1 #include "CppUTest/TestHarness.h"
 2 #include "CppUTestExt/MockSupport.h"
 3
 4 extern "C"
 5 {
 6 #include "CppUTestExt/MockSupport_c.h"
 7 #include "cache.h"
 8 #include "loader.h"
 9
10 char *load(char *key) {
11     mock_c()->actualCall("load")
12             ->withStringParameters("key", key);
13     return (char *) mock_c()->returnValue().value.stringValue;
14 }
15 }
```

Using the mock in tests

cache_tests.cpp

```
17 static char *key1 = (char *) "key1";
18 static char *key2 = (char *) "key2";
19 static char *object1 = (char *) "object1";
20 static char *object2 = (char *) "object2";
21
22 TEST_GROUP(cache) {
23
24     void setup() {
25
26     }
27
28     void teardown() {
29         mock().checkExpectations();
30         mock().clear();
31     }
32 };
```

Using the mock in tests

cache_tests.cpp

```
34 TEST(cache, load_object_not_cached) {
35     mock().expectOneCall("load")
36           .withParameter("key", (char *) key1)
37           .andReturnValue((char *) object1);
38
39     STRCMP_EQUAL(object1, lookup(key1));
40 };
41
42 TEST(cache, load_objects_not_cached) {
43     mock().expectOneCall("load")
44           .withParameter("key", (char *) key1)
45           .andReturnValue((char *) object1);
46     mock().expectOneCall("load")
47           .withParameter("key", (char *) key2)
48           .andReturnValue((char *) object2);
49
50     STRCMP_EQUAL(object1, lookup(key1));
51     STRCMP_EQUAL(object1, lookup(key1));
52 };
```

Using the mock in tests

cache_tests.cpp

```
54 TEST(cache, load_object_cached) {
55     mock().expectOneCall("load")
56           .withParameter("key", (char *) key1)
57           .andReturnValue((char *) object1);
58
59     STRCMP_EQUAL(object1, lookup(key1));
60     STRCMP_EQUAL(object1, lookup(key1));
61 };
```

Test results

test results

```
tests/cache_tests.cpp:54: error: Failure in TEST(cache, load_object_cached)
        Mock Failure: Unexpected additional (2th) call to function: load
        EXPECTED calls that did NOT happen:
                <none>
        ACTUAL calls that did happen (in call order):
                load -> char* key: <key1>

.
tests/cache_tests.cpp:42: error: Failure in TEST(cache, load_objects_not_cached)
        Mock Failure: Unexpected parameter value to parameter "key" to function "l
        EXPECTED calls that DID NOT happen related to function: load
                load -> char* key: <key2>
        ACTUAL calls that DID happen related to function: load
                load -> char* key: <key1>
        ACTUAL unexpected parameter passed to function: load
                char* key: <key1>

..
Errors (2 failures, 3 tests, 3 ran, 6 checks, 0 ignored, 0 filtered out, 1 ms)
```

Outline

mockito presentation

Mockito is a simple and easy to use mocking framework for Java.

Faber, Szczepan and mockito contributors (2013).
mockito.
http://code.google.com/p/mockito/.

The Cache class...

Cache.java

```
public class Cache {
    private Loader loader;

    public Cache(Loader loader_) {
        this.loader = loader_;
    }

    public String lookup(String key) {
        return this.loader.load(key);
    }
}
```

Initializing the mock for testing

CacheTest.java

```
 1 import org.junit.*;
 2 import static org.junit.Assert.*;
 3 import org.mockito.*;
 4 import static org.mockito.Mockito.*;
 5
 6 public class CacheTest {
 7
 8     private String key1 = "key1";
 9     private String key2 = "key2";
10     private String object1 = "object1";
11     private String object2 = "object2";
12
13     private Cache myCache;
14     private Loader myMock;
15
16     @Before public void setUp() {
17         myMock = mock(Loader.class);
18         myCache = new Cache(myMock);
19
20         when(myMock.load(key1)).thenReturn(object1);
21         when(myMock.load(key2)).thenReturn(object2);
22     }
```

Using the mock in tests

CacheTest.java

```
27      @Test public void testLoadObjectNotCached() {
28          assertEquals(object1, myCache.lookup(key1));
29
30          verify(myMock).load(key1);
31          verifyNoMoreInteractions(myMock);
32      }
33
34      @Test public void testLoadObjectsNotCached() {
35          assertEquals(object1, myCache.lookup(key1));
36          assertEquals(object2, myCache.lookup(key2));
37
38          InOrder inOrder = inOrder(myMock);
39
40          inOrder.verify(myMock).load(key2);
41          inOrder.verify(myMock).load(key1);
42      }
43
44      @Test public void testLoadObjectCached() {
45          assertEquals(object1, myCache.lookup(key1));
46          assertEquals(object1, myCache.lookup(key1));
47
48          verify(myMock, times(1)).load(key1);
49      }
```

Test results (extract)

test results

```
There were 2 failures:
1) testLoadObjectsNotCached(CacheTest)
org.mockito.exceptions.verification.VerificationInOrderFailure:
Verification in order failure
Wanted but not invoked:
loader.load("key1");
-> at CacheTest.testLoadObjectsNotCached(CacheTest.java:41)
Wanted anywhere AFTER following interaction:
loader.load("key2");
-> at Cache.lookup(Cache.java:9)

...

2) testLoadObjectCached(CacheTest)
org.mockito.exceptions.verification.TooManyActualInvocations:
loader.load("key1");
Wanted 1 time:
-> at CacheTest.testLoadObjectCached(CacheTest.java:48)
But was 2 times. Undesired invocation:
-> at Cache.lookup(Cache.java:9)
```

Powerful features of mockito

- verification modes: method called n times, never called, at least once, at least/most n times, custom...
- fine parameters matching
- timeout management
- spying on real objects
- useful annotations (not presented here)

See also JMockit.

JMockit contributors (2013).
JMockit.
http://code.google.com/p/jmockit/.

Pros/cons of mocks

Pros

- better granularity
- you focus on the class to test
- tests are repeatable
- mocks in tests are **specifications** for the real objects/functions

Cons

- difficult to use for testing interaction errors
- API must be stable!

Principle (behaviour driven development)

Focus on the expected behavior of objects/methods, not on exhaustive tests.

Outline

Conclusion

TDD is a good programming practise: it give the programmer a **rhythm** and he can **trust** her/his code.

xUnit testing frameworks exist for lots of languages: HTTP, Ruby, Python, Scala, ...
You will also find testing frameworks for container load, GUI, performance, ...

An important notion not presented here is code coverage (see J. Hugues' lecture on the subject).