Dependency Injection and Test Driven Development

Silvano Luciani

venda

22/02/11

Dependency Injection and TDD

- DI = Design pattern or principle
- TDD = Programming practice

- Using DI we write code easier to test
- Writing tests we are driven to use DI to make our code easier to test

Content

- Small bits of theory
  - Different forms of Dependency Injection
  - Basic model of TDD
  - Common types of tests
- Case study inspection
  - Code analysis of a sample application

What is Dependency Injection?

An application of Inversion of Control principle.

venda

What is Inversion of Control?

- An abstract principle
- Software designed so that reusable generic code controls the execution of problem-specific code.
- Reusable generic code and problem-specific code can be developed independently.

No IoC

```
package  main;

sub  extractRecords {
    my  $self = shift;
    # db handlers
    my  $dbhMy = DBI->connect(
        'DBI:mysql:database=EXTERNAL_SOURCE',
        'root',
        ''
    );
    my  $dbhLite = DBI->connect(
        'DBI:SQLite:database=EXTERNAL_SOURCE',
        'root',
        ''
    );
    # code to get the data from a MySQL db
    my  $sth = $dbhMy->prepare("SELECT foo, bar FROM table WHERE baz=?");
    $sth->execute( $baz );
    $results = $sth->fetchall_hashref();
    foreach my $record (@$results) {
        # code to adapt the format
        $record->{source}        = 'external';
        $record->{created_at}    = time();
        ##
        # code to save the record to my db ( SQLite )
        ##
    }
}
```

venda

Template Method

```
package main;

sub templateRecords {
    my $self = shift;
    my @records = $self->queryRecords();
    foreach my $record (@records) {
        my $output = $self->templateRecord($record);
        $self->saveRecord($output);
    }
}

sub queryRecords {
    # abstract
}

sub templateRecord {
    # abstract
}

sub saveRecord {
    # abstract
}
```

Delegation

```perl
package main;

sub  importRecords {

    my  $self = shift;

    my  $externalCollection  = ExternalCollection->new();
    my  $recordFormatter     = RecordFormatter->new();
    my  $recordCollection    = RecordCollection->new();

    my  @records = $externalCollection->queryRecords();
    foreach my  $record (@records) {
        my  $output = $recordFormatter->formatRecord($record);
        $recordCollection->saveRecord($output);
    }

}
```

What is a Dependency

- With Delegation come dependencies
- Our class now needs instances of the following three classes to perform its tasks:
  - ExternalCollection
  - RecordFormatter
  - RecordCollection

What is Dependency Injection?

- Object Dependencies are problem-specific code.
- We 'Invert The Control' creating the dependencies outside the class that is consuming the dependencies.

Constructor vs Setter Injection

```
# Constructor Injection
my  $instance = main->new(
     externalCollection  => $externalCollection,
     recordFormatter     => $recordFormatter,
     recordCollection    => $recordCollection,
);



# Setter Injection
my  $instance = main->new();

$instance->setExternalCollection($externalCollection);
$instance->setRecordFormatter($recordFormatter);
$instance->setRecordCollection($recordCollection);
```

2nd Round

Constructor:

- After the constructor method, the object is usable
- Better for required dependencies
- Circular dependencies

Setter:

- No circular dependencies
- Better for optional or dynamic dependencies
- Can be hard to determine when the object is ready to use
- Remember to set the dependencies

Block Injection

- We Inject a block of code (subroutine) that returns an object
- Not famous because in Java it's not possible
- More versatile, handy for handling complex initialisation code

Container

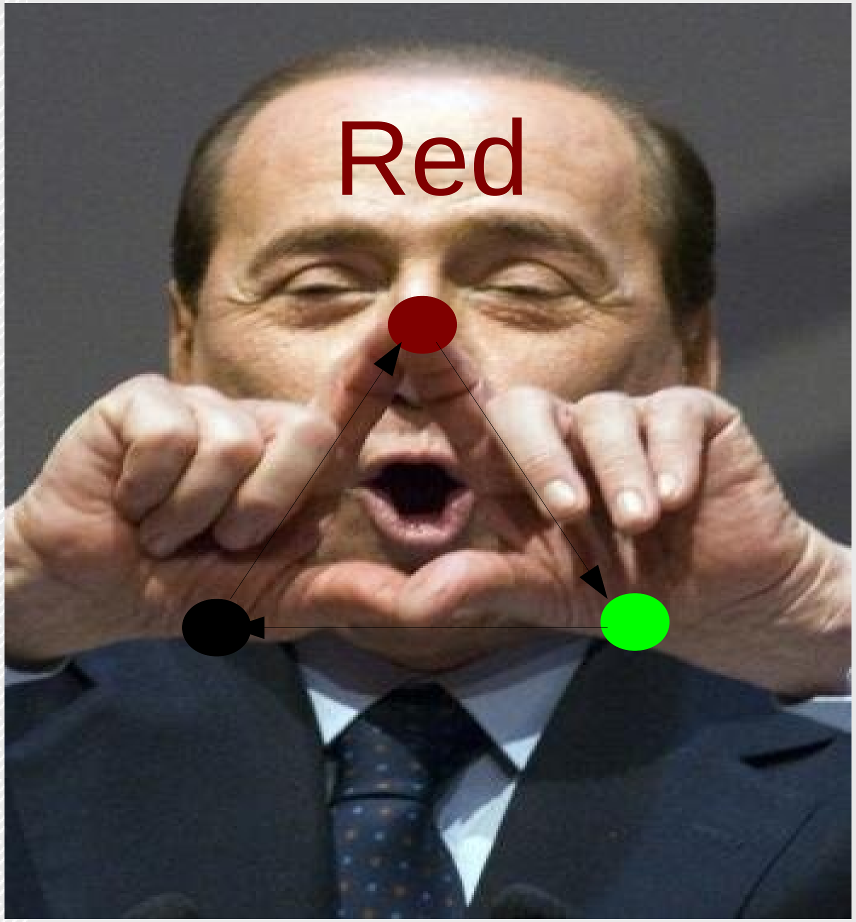
- Control of objects creation
- Include satisfaction of dependencies
- When we need an instance of a class, we ask the Container to create the instance and wire the dependencies
- Using Environments we can easily change the behaviour of the system (very useful for integration and functional tests)

What is TDD?

What is TDD?

22/02/11

What is TDD?



Refactor

Green

22/02/11

What is TDD?

- Write the test for a functionality and execute the test to check that it is failing
- Write the amount of code needed for the test to pass
- Refactor code:
    - Remove duplications
    - Improve the design with tests as a safety net (never modify test and code at the same time!)

Example: 1st iteration

```
sub  test_add {
     is(add(5,5), 10, "Correct result for addition");
}
sub  add {
    return 10;
}
```

Example: 1st iteration

```
sub  test_add {
     is(add(5,5), 10, "Correct result for addition");
}

sub  add {
    return 10;
}
```

Example: 2nd iteration

```
sub  test_add {

     is(add(5,5), 10, "Correct result for addition");
     is(add(3,8), 11, "Correct result for addition");

}

sub  add {

    my ($a,$b) = @_;
    return  $a + $b;

}
```

Why TDD?

- Drive us towards 'good' design principles:
  - Loose coupling
  - Program to an interface, not an implementation
  - Composition over Inheritance
  - Delegation
  - You ain't gonna need it

Testing Layers

Functional

Unit

Usually

“A piece of this, a piece of that”
K-Hos from Bethnal Green™

Testing Layers again

Usually again

Functional

Integration

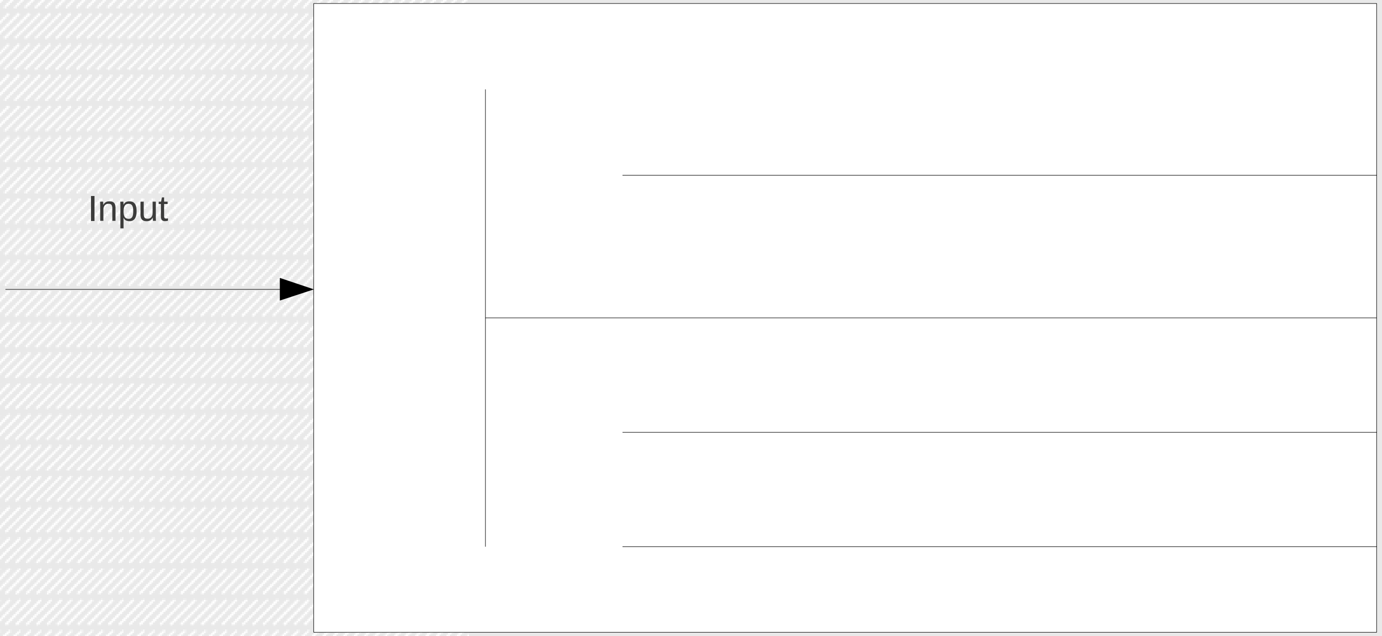
Unit

22/02/11

Unit Test

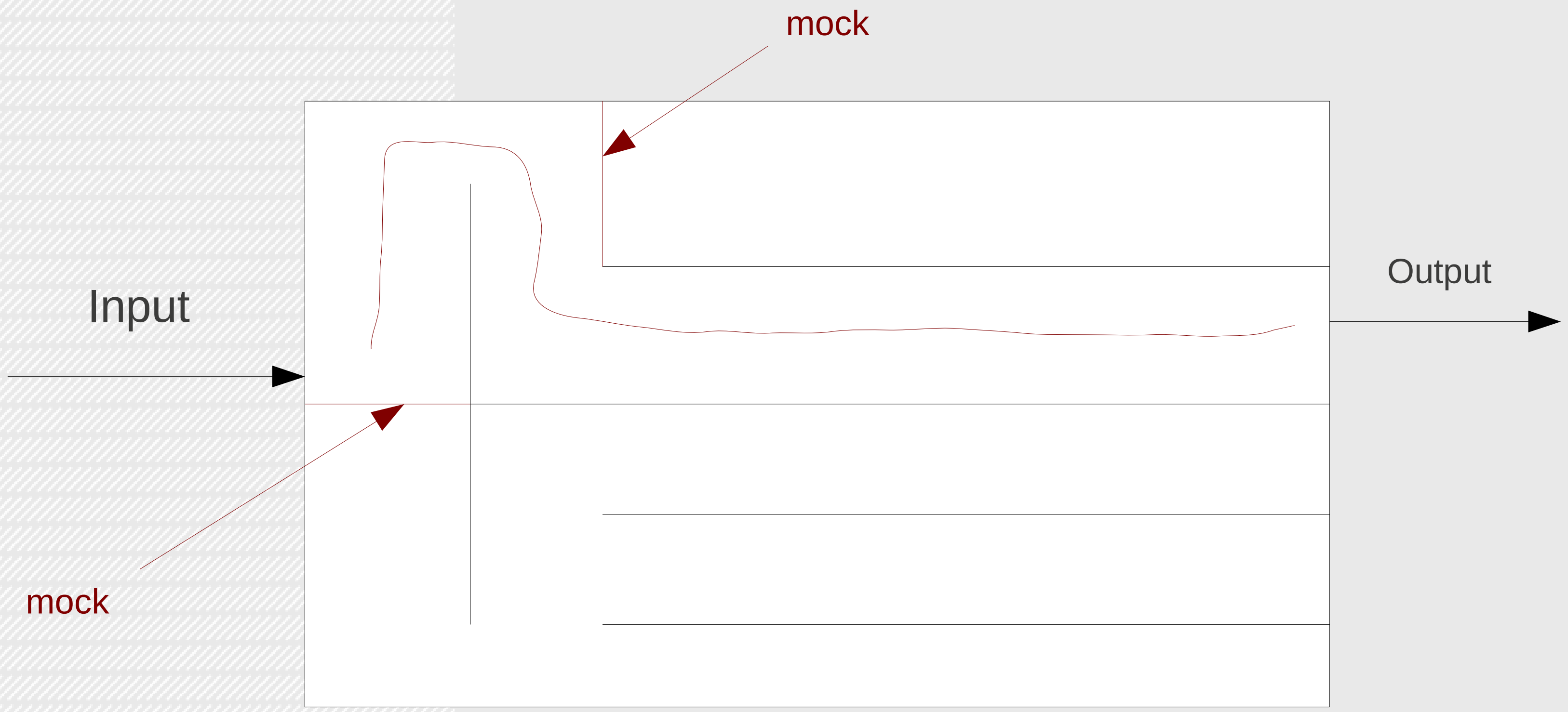
- Individual units of source code tested
- A unit is the smallest testable part
- White Box test
- All the dependencies are mocked
- Written by developers

Unit Test



22/02/11

Unit Test



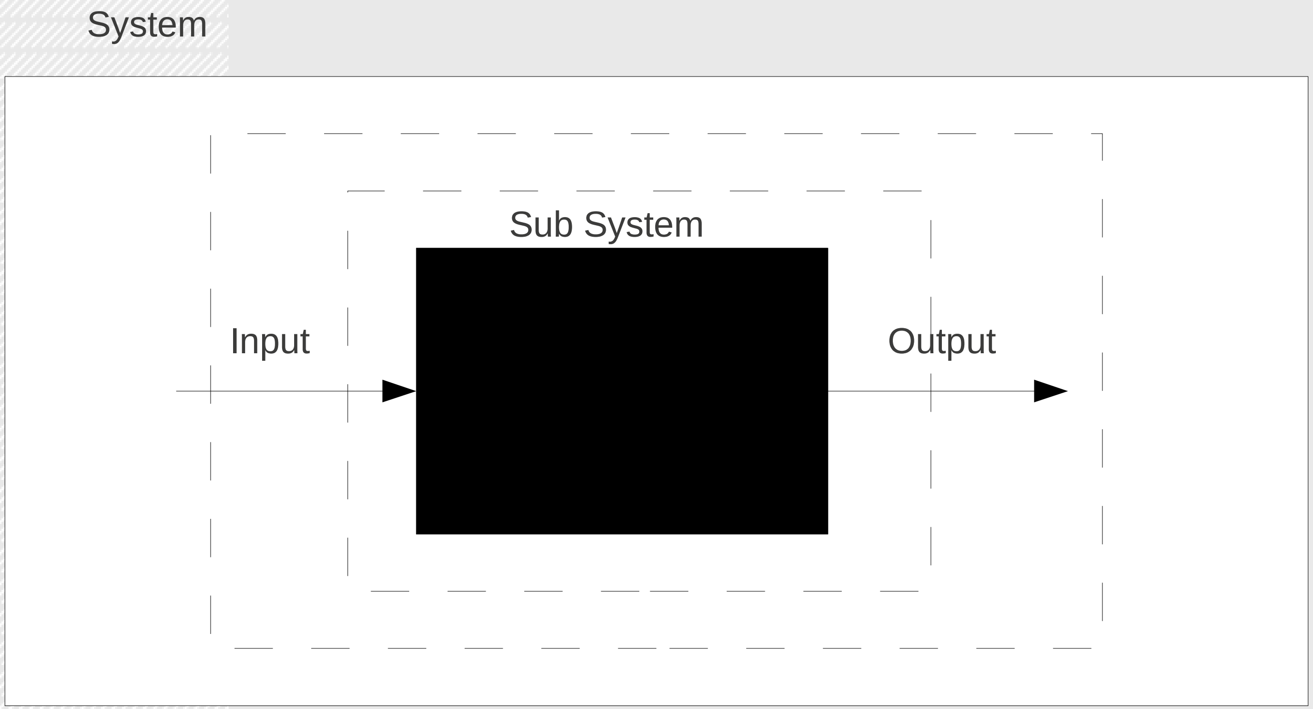
venda

Integration Test

- Individual software modules are combined and tested as a group (subsystem)
- Partially White Box, partially Black Box
- We can mock some objects to position ourself in a certain starting condition
- Written by developers, but with the right tools QA can help

venda

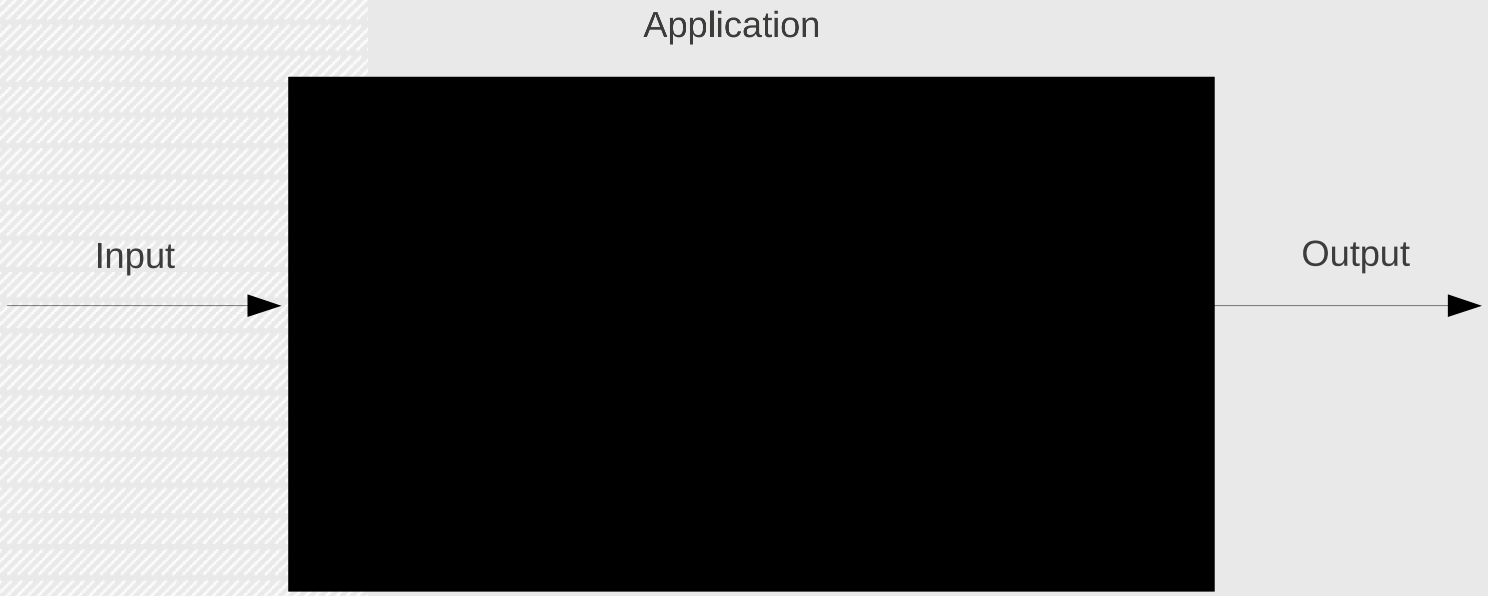
Integration Test



Functional Test

- Specifications of the software are tested
- Black Box
- We can use testing environments (i.e. db), but we should not mock anything
- QA, with help from the developers

Functional Test



Sample Application

- A command line perl application to help an Estate Agency for students to determine the days in which the tenants must pay the rent and the bills
- The rent is due the 1st day of the month if it is a weekday, otherwise the next monday.
- The bills are due the 15th day of the month if it is a weekday, otherwise the previous friday.

Input

- Student Name and Expiration (index 1 – 12)
- Csv Input: a line for each student, first value is the name, second value is the expiration
- MySql Input: a table named 'Students' with two columns, name and expiration

Output

- Name of the student and for every month until the expiration month (included) :
  - Month name (January, February … December)
  - Rent Payment Day
  - Bills Payment Day
- Csv Output: a line with the name of the student and then one line for each of the month
- Yaml Output: array of arrays
- The user can specify the name of the output file

Some of the modules

- Moose
- Test::Class and Test::More
- Test::Cukes
- Class::MOP and Moose::Meta::*
- Bread::Board