Acceptance Testing: Mockito, Cucumber, Romeo, and Juliet

Sébastien Mosser
21.03.2018, QGL

pictures: sxc.hu

Product: “Smart Bar”

https://github.com/CodingDojoPolytech/cucumber-mockito-shakespeare

Feature: Cocktail Ordering

As Romeo, I want to offer a drink to Juliet, so that we can “discuss”.

Minimal (~Viable) Scenario

Creating an empty order

```
public class OrderingCocktailTest {

  private Order order;

  @Test
  public void empty_order_by_default() {
    order = new Order();
    order.declareOwner(“Romeo”);
    order.declareTarget(“Juliette");
    List<String> cocktails =  order.getCocktails();
    assertEquals(0, cocktails.size());
  }
}
```

Test as specification: acceptance criteria

```java
@Test
public void empty_order_by_default() {
  order = new Order();
  order.declareOwner(“Romeo”);
  order.declareTarget(“Juliette");
  List<String> cocktails =  order.getCocktails();
  assertEquals(0, cocktails.size());
}
```

But who’ll write that spec ?

How can a non-tech people write acceptance specs ?

```
Given
  Romeo who wants to buy a drink

When
  an order is declared for Juliette

Then
  there is 0 cocktails in the order
```

(this.language = Gherkin)

Given **Romeo** who wants to buy a drink
When an order is declared for **Juliette**
Then there is **0** cocktails in the order

```
@Test
public void empty_order_by_default() {
  order = new Order();
  order.declareOwner(“Romeo”);
  order.declareTarget(“Juliette");
  List<String> cocktails =  order.getCocktails();
  assertEquals(0, cocktails.size());
}
```

```
Given Romeo who wants to buy a drink
When  an order is declared for Juliette
Then  there is 0 cocktails in the order
```

Mapping

```
@Test
public void empty_order_by_default() {
  order = new Order();
  order.declareOwner(“Romeo”);
  order.declareTarget(“Juliette");
  List<String> cocktails =  order.getCocktails();
  assertEquals(0, cocktails.size());
}
```

```java
public class CocktailStepDefinitions {

  private Order order;

  @Given("Romeo who wants to buy a drink")
  public void romeo_who_wants_to_buy_a_drink() {
    order = new Order();
    order.declareOwner(“Romeo”);
  }

  @When("an order is declared for Juliette")
  public void an_order_is_declared_for_juliette() {
    order.declareTarget(“Juliette");
  }

  @Then("There is 0 cocktails in the order")
  public void there_is_no_cocktails_in_the_order() {
    List<String> cocktails =  order.getCocktails();
    assertEquals(0, cocktails.size());
  }
}
```

POJO + Annotations

```
public class CocktailStepDefinitions {

   private Order order;

   @Given("Romeo who wants to buy a drink")
   public void romeo_who_wants_to_buy_a_drink() {
      order = new Order();
      order.declareOwner(“romeo”);
   }

   @When("an order is declared for Juliet")
   public void an_order_is_declared_for_juliette() {
      order.declareTarget(“juliet”);
   }

   @Then("there is 0 cocktails in the order")
   public void there_is_n_cocktails_in_the_order(int n) {
      List<String> cocktails =  order.getCocktails();
      assertEquals(0, cocktails.size());
   }
}
```

```
Scenario: Creating an empty order
  Given Romeo who wants to buy a drink
  When  an order is declared for Juliette
  Then  there is 0 cocktails in the order
```

Cocktail.feature

POJO + Annotations

```
public class CocktailStepDefinitions {

   private Order order;

   @Given("Romeo who wants to buy a drink")
   public void romeo_who_wants_to_buy_a_drink() {
      order = new Order();
      order.declareOwner(“romeo”);
   }

   @When("an order is declared for Juliet")
   public void an_order_is_declared_for_juliette() {
      order.declareTarget(“juliet”);
   }

   @Then("there is 0 cocktails in the order")
   public void there_is_n_cocktails_in_the_order(int n) {
      List<String> cocktails =  order.getCocktails();
      assertEquals(0, cocktails.size());
   }
}
```

```
Scenario: Creating an empty order
  Given Romeo who wants to buy a drink
  When  an order is declared for Juliette
  Then  there is 0 cocktails in the order
```

Cocktail.feature

cucumber

POJO + Annotations

```java
public class CocktailStepDefinitions {

    private Order order;

    @Given("Romeo who wants to buy a drink")
    public void romeo_who_wants_to_buy_a_drink() {
        order = new Order();
        order.declareOwner("romeo");
    }

    @When("an order is declared fo
    public void an_order_is_declar
        order.declareTarget("juliet
    }

    @Then("there is 0 cocktails in the order")
    public void there_is_n_cocktails_in_the_order(int n) {
        List<String> cocktails =  order.getCocktails();
        assertEquals(0, cocktails.size());
    }
}
```

Test Results	433ms
Feature: Cocktail Ordering	433ms
Scenario: Creating an empty order	167ms
Given Romeo who wants to buy a drink	162ms
When an order is declared for Juliette	0ms
Then there is 0 cocktails in the order	5ms

Good Practice

G/W/T acceptances tests described in your business features in Jira

What about Tom & Jerry?

What about Tom & Jerry?

```
Given Tom who wants to buy a drink
When  an order is declared for Jerry
Then  there is 0 cocktails in the order
```

```
Given Romeo who wants to buy a drink
When  an order is declared for Juliette
Then  there is 0 cocktails in the order
```

What about Tom & Jerry?

```
Given Tom who wants to buy a drink
When  an order is declared for Jerry
Then  there is 0 cocktails in the order
```

```
@Given("Romeo who wants to buy a drink")
public void romeo_who_wants_to_buy_a_drink() { … }

@When("an order is declared for Juliette")
public void an_order_is_declared_for_juliette() { … }

@Given("Tom who wants to buy a drink")
public void tom_who_wants_to_buy_a_drink() { … }

@When("an order is declared for Jerry")
public void an_order_is_declared_for_jerry() { … }

@Then("There is 0 cocktails in the order")
public void there_is_no_cocktails_in_the_order() { … }
```

What about Tom & Jerry?

```
Given Tom who wants to buy a drink
When  an order is declared for Jerry
Then  there is 0 cocktails in the order
```

```
@Given("Romeo who wants to buy a drink")
public void romeo_who_wants_to_buy_a_drink() { … }

@When("an order is declared for Juliette")
public void an_order_is_declared_for_juliette() { … }

@Given("Tom who wants to buy a drink")
public void tom_who_wants_to_buy_a_drink() { … }

@When("an order is declared for Jerry")
public void an_order_is_declared_for_jerry() { … }

@Then("There is 0 cocktails in the order")
public void there_is_no_cocktails_in_the_order() { … }
```

Duplication!

What about Tom & Jerry?

Tom

Jerry

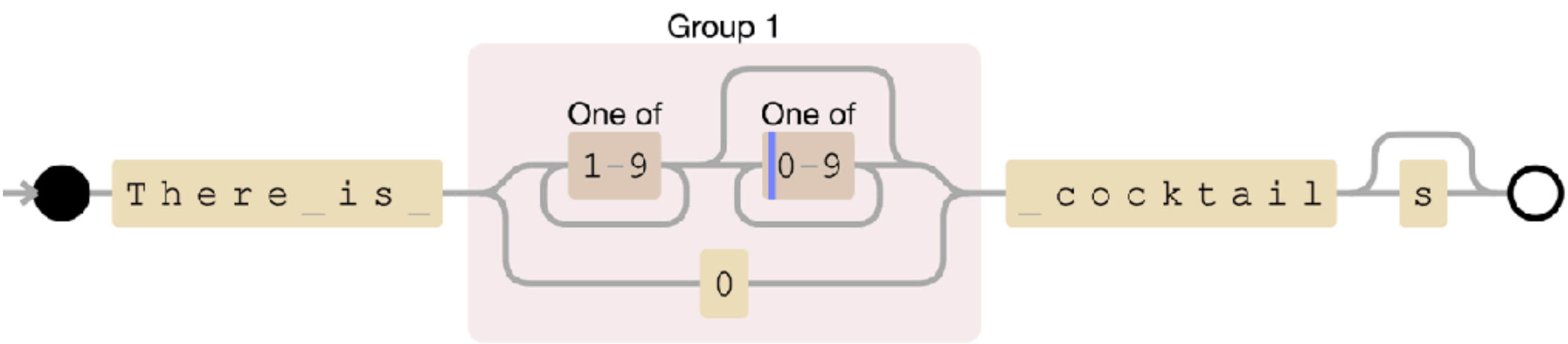
```
Given aString who wants to buy a drink
When  an order is declared for aString
Then  there is n cocktails in the order
```

Romeo

0

Juliet

Regular Expressions!



^There is ([1-9]+[0-9]*|0) cocktails?$

Language digression

A **regular** language is recognised by a **regular** expression.

Un langage **rationnel** est reconnu par une expression ...

(et tant qu'on y est une **librairie** c'est là où on achète des livres, alors qu'une **bibliothèque** ...)

```java
public class CocktailStepDefinitions {

   private Order order;

   @Given("^(.*) who wants to buy a drink$")
   public void someone_who_wants_to_buy_a_drink(String romeo) {
      order = new Order();
      order.declareOwner(romeo);
   }

   @When("^an order is declared for (.*)$”)
   public void an_order_is_declared_for_someone(String juliette) {
      order.declareTarget(juliette);
   }

   @Then("^there is (\\d+) cocktails in the order$")
   public void there_is_n_cocktails_in_the_order(int n) {
      List<String> cocktails =  order.getCocktails();
      assertEquals(n, cocktails.size());
   }
}
```

File Cocktail.feature

```
Scenario: Creating an empty order
  Given Romeo who wants to buy a drink
  When  an order is declared for Juliette
  Then ther
        there is <number> cocktails in the order
```

Automated completion

What about Laziness?

What about Laziness?

```
Given Tom who wants to buy a drink
When  an order is declared for Jerry
Then  there is 0 cocktails in the order
```

```
Given Romeo who wants to buy a drink
When  an order is declared for Juliette
Then  there is 0 cocktails in the order
```

What about Laziness?

```
Given Romeo who wants to buy a drink
When  an order is declared for Juliet
  And  a message saying “Ciao!" is added
Then the ticket must say “From R to J: Ciao!”
```

```
Given Romeo who wants to buy a drink
When  an order is declared for Tom
  And  a message saying “Hey!" is added
Then the ticket must say “From R to T: Hey!”
```

Background & Outline

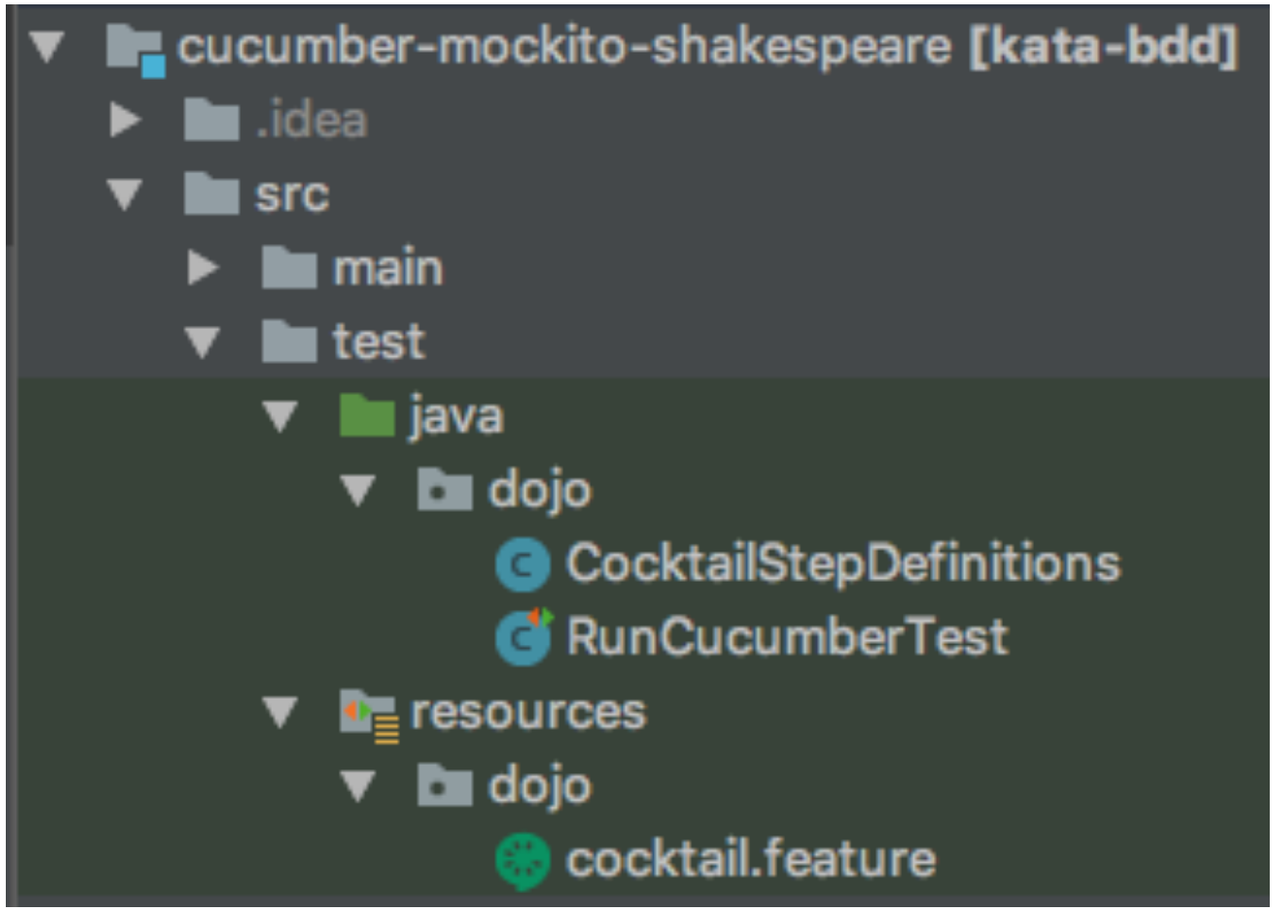
```
Background:
  Given Romeo who wants to buy a drink

Scenario Outline: Sending a message with an order
  When  an order is declared for <to>
    And  a message saying "<msg>" is added
  Then the ticket must say "<expected>"

  Examples:
    | to        | msg    | expected              |
    | Juliette | Ciao! | From T to J: Ciao! |
    | Tom       | Hey!   | From R to T: Hey!   |
    # …
```

(templating)

- Scenario Outline: Sending a message with an order 1ms
  - Examples: 1ms
    - Scenario: Line: 20 0ms
      - Given Romeo who wants to buy a drink 0ms
      - When an order is declared for Juliette 0ms
      - And a message saying "Wanna chat?" is added 0ms
      - Then the ticket must say "From Romeo to Juliette 0ms
    - Scenario: Line: 21 1ms
      - Given Romeo who wants to buy a drink 0ms
      - When an order is declared for Jerry 1ms
      - And a message saying "Hei!" is added 0ms
      - Then the ticket must say "From Romeo to Jerry: H 0ms

Technical Details

```
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-java</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-junit</artifactId>
    <scope>test</scope>
</dependency>
```




```
@RunWith(Cucumber.class)
public class RunCucumberTest { }
```

```
Feature: Cocktail Ordering

  As Romeo, I want to offer a drink to Juliette so that we can discuss together (and maybe more).

    Background:
      Given Romeo who wants to buy a drink
      When  an order is declared for Juliette


    Scenario: Creating an empty order
      Then  there is 0 cocktails in the order

    Scenario Outline: Sending a message with an order
      When  an order is declared for <to>
        And  a message saying "<message>" is added
      Then the ticket must say "<expected>"

      Examples:
        | to        | message      | expected                          |
        | Juliette  | Wanna chat?  | From Romeo to Juliette: Wanna chat? |
        | Jerry     | Hei!         | From Romeo to Jerry: Hei!         |
      # ...


    Scenario: Offering a mojito to Juliette
      When a mocked menu is used
        And the mock binds #42 to mojito
        And a cocktail #42 is added to the order
      Then there is 1 cocktails in the order
        And  the order contains a mojito

    Scenario: Paying the mojito offered to Juliette
      When a mocked menu is used
        And the mock binds #42 to $10
        And a cocktail #42 is added to the order
        And Romeo pays his order
      Then the payment component must be invoked 1 time for $10

    Scenario: Not paying the empty bill
      When Romeo pays his order
      Then the payment component must be invoked 0 time for $0
```

cocktail.feature

Run Feature: cocktail

Done: Scenario:

- Test Results 460ms
  - Feature: Cocktail Ordering 460ms
    - Scenario: Creating an empty order 176ms
      - Given Romeo who wants to buy a drink 172ms
      - Then there is 0 cocktails in the order 4ms
    - Scenario Outline: Sending a message with an order 1ms
      - Examples: 1ms
        - Scenario: Line: 20 0ms
          - Given Romeo who wants to buy a drink 0ms
          - When an order is declared for Juliette 0ms
          - And a message saying "Wanna chat?" is ad 0ms
          - Then the ticket must say "From Romeo to J 0ms
        - Scenario: Line: 21 1ms
          - Given Romeo who wants to buy a drink 0ms
          - When an order is declared for Jerry 1ms
          - And a message saying "Hei!" is added 0ms
          - Then the ticket must say "From Romeo to J 0ms
    - Scenario: Offering a mojito to Juliette 249ms
      - Given Romeo who wants to buy a drink 0ms
      - When a mocked menu is used 214ms
      - And the mock binds #42 to mojito 32ms
      - And a cocktail #42 is added to the order 3ms
      - Then there is 1 cocktails in the order 0ms
      - And the order contains a mojito 0ms
    - Scenario: Paying the mojito offered to Juliette 34ms
      - Given Romeo who wants to buy a drink 0ms
      - When a mocked menu is used 0ms
      - And the mock binds #42 to $10 0ms
      - And a cocktail #42 is added to the order 2ms
      - And Romeo pays his order 21ms
      - Then the payment component must be invoked 1 11ms
    - Scenario: Not paying the empty bill 0ms
      - Given Romeo who wants to buy a drink 0ms
      - When Romeo pays his order 0ms
      - Then the payment component must be invoked 0 0ms

```
Testing started at 21:36 ...
/Library/Java/JavaVirtualMachines/jdk1.8.0_91.jdk/Contents/Home/bin/java
objc[32006]: Class JavaLaunchHelper is implemented in both /Library/Java/

6 Scenarios (6 passed)
25 Steps (25 passed)
0m0.468s


Process finished with exit code 0
```

IDE Integration

Mocks?

```
Scenario: Offering a mojito to Juliette

  Given
    Romeo who wants to buy a drink
  When
    an order is declared for Juliette
  And
    a cocktail #42 is added to the order
  Then
    there is 1 cocktail in the order
  And
    the order contains a mojito
```

```
Scenario: Offering a mojito to Juliette

  Given
    Romeo who wants to buy a drink
  When
    an order is declared for Juliette
  And
    a cocktail #42 is added to the order
  Then
    there is 1 cocktail in the order
  And
    the order contains a mojito
```

```java
public class CocktailStepDefinitions {

   private Order order;
   private Menu menu;

   @Before
   public void a_mocked_menu_is_used(){
      menu = mock(Menu.class);
      order.useMenu(menu);
      when(menu.getPrettyName(42)).thenReturn("mojito");
   }

   @When("^a cocktail #(\\d+) is added to the order$")
   public void a_cocktail_C_is_added_to_the_order(int C) {
      order.addCocktail(C);
   }

   @Then("^the order contains a (.*)")
   public void the_order_contains_a_given_cocktail(String givenCocktail) {
      assertTrue(order.getCocktails().contains(givenCocktail));
   }
}
```

Controlling the mock from the feature

```
Scenario: Paying the mojito offered to Juliette
  Given Romeo who wants to buy a drink
  When  an order is declared for Juliette
    And a mocked menu is used
    And the mock binds #42 to mojito
    And the mock binds #42 to $10
    And a cocktail #42 is added to the order
    And Romeo pays his order
  Then the payment component must be invoked 1 time for $10
```

Good or Bad Idea?

Controlling the mock from the feature

```
Scenario: Paying the mojito offered to Juliette
  Given Romeo who wants to buy a drink
  When  an order is declared for Juliette
    And a mocked menu is used
    And the mock binds #42 to mojito
    And the mock binds #42 to $10
    And a cocktail #42 is added to the order
    And Romeo pays his order
  Then the payment component must be invoked 1 time for $10
```

Good or Bad Idea?

```java
public class CocktailStepDefinitions {

    private Order order;
    private Menu menu;
    private Payment paypal;

    @When("^a mocked menu is used$")
    public void a_mocked_menu_is_used(){
        menu = mock(Menu.class);
        order.useMenu(menu);
    }

    @When("^the mock binds #(\\d+) to ([^\\$]*)$")
    public void the_mock_binds_Id_to_Cocktail(int id, String cocktail) {
        when(menu.getPrettyName(id)).thenReturn(cocktail);
    }

    @When("^the mock binds #(\\d+) to \\$(\\d+)$")
    public void the_mock_binds_Id_to_Price(int id, int price) {
        when(menu.getPrice(id)).thenReturn(price);
    }

    @When("^Romeo pays his order$")
    public void romeo_pays_his_order() {
        paypal = mock(Payment.class);
        Cashier.processOrder(paypal, order);
    }

    @Then("^the payment component must be invoked (\\d+) time for \\$(\\d+)")
    public void the_payment_component_must_be_invoked_N_times(int n, int amount){
         verify(paypal, times(n)).performPayment(amount);
    }
}
```

Alessandro Baffa 👨‍💻🗼🍕
@alebaffa

Abonné

Mockito is one of the most dangerous tools for a Java developer. So powerful and time saving, if well used. A curse, if otherwise.

À l'origine en anglais

05:21 - 17 févr. 2018

Tweeter votre réponse