**SW품질관리및테스팅 과제**

* Junit을 이용한 테스트 케이스 작성



**과목명** | SW품질관리및테스팅

**담당교수** | 문봉교 교수님

**학과** | 컴퓨터공학과

**학번** | 2016112182

**이름** | 최고운

**제출일자** | 2020.06.22

**0. Before**

|  |
| --- |
| private CoffeeMaker coffeeMaker; String name; String price; String amtCoffee; String amtMilk; String amtSugar; String amtChocolate; Recipe recipe1;  @Before public void setUp() throws Exception{  coffeeMaker = new CoffeeMaker();   name = "name1";  price = "1000";  amtCoffee = "1";  amtMilk = "1";  amtSugar = "1";  amtChocolate = "1";   recipe1 = new Recipe();  recipe1.setName(name);  recipe1.setPrice(price);  recipe1.setAmtCoffee(amtCoffee);  recipe1.setAmtMilk(amtMilk);  recipe1.setAmtSugar(amtSugar);  recipe1.setAmtChocolate(amtChocolate); } |

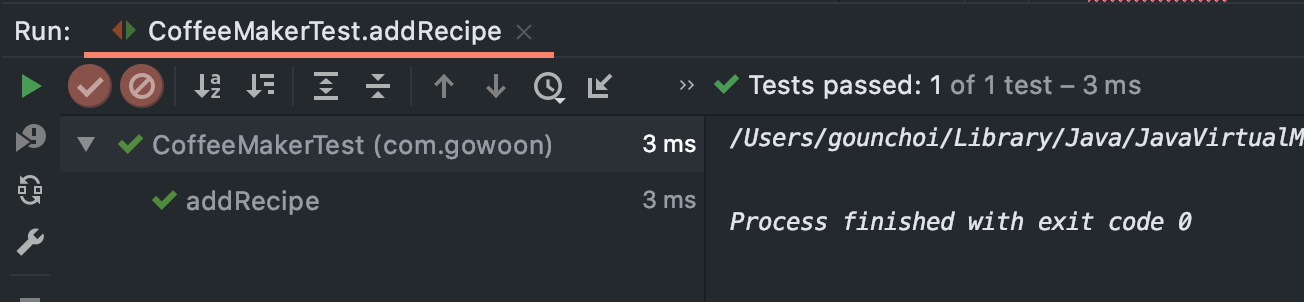
아래의 모든 테스트에서 사용될 CoffeeMaker 객체를 생성하고 기본적으로 사용될 Recipe 객체를 하나 생성하는 코드를 @Before로 작성해주었습니다.

**1. addRecipeTest**

- 성공 테스트

|  |
| --- |
| @Test public void addRecipe() {  // addRecipe 성공 테스트  *assertTrue*(coffeeMaker.addRecipe(recipe1));   // 값 일치 테스트  Recipe recipe = coffeeMaker.getRecipes()[0];  *assertEquals*(name, recipe.getName());  *assertEquals*(Integer.*parseInt*(price), recipe.getPrice());  *assertEquals*(Integer.*parseInt*(amtCoffee), recipe.getAmtCoffee());  *assertEquals*(Integer.*parseInt*(amtMilk), recipe.getAmtMilk());  *assertEquals*(Integer.*parseInt*(amtSugar), recipe.getAmtSugar());  *assertEquals*(Integer.*parseInt*(amtChocolate), recipe.getAmtChocolate()); } |

Recipe가 성공적으로 추가되어 ture가 반환되고, 추가된 레시피가 추가한 레시피의 속성과 일치하는 것까지 확인이 가능하다.



- 이미 존재하는 레시피 추가시 실패

|  |
| --- |
| @Test public void addRecipeAlreadyExistFail() {  // 이미 존재하는 레시피 추가 실패 테스트  *assertTrue*(coffeeMaker.addRecipe(recipe1));  *assertTrue*(coffeeMaker.addRecipe(recipe1)); } |

이미 존재하는 recipe를 또 추가하는 경우 recipe가 추가되지 않아 false를 반환하는 것을 확인할 수 있다. Recipe의 addRecipe함수를 보면 added여부를 true, false로 반환하는 것이므로 원시코드를 수정할 필요는 없다.

텍스트이(가) 표시된 사진

자동 생성된 설명

- 레시피 설정시에 타입이 일치하지 않는 실패

|  |
| --- |
| @Test public void addRecipeInvalidTypeFail() throws RecipeException {  // 가격과 커피, 우유, 설탕, 초콜릿의 양을 설정할 때 타입을 맞추지 않아 RecipeException 발생  Recipe recipe2 = new Recipe();  recipe1.setName("name2");  recipe1.setPrice("a");  recipe1.setAmtCoffee("-1");  recipe1.setAmtMilk("a");  recipe1.setAmtSugar("a");  recipe1.setAmtChocolate("a"); } |

Price, amtCoffee, amtMilk, amtSugar, amtChocolate은 내부에서 integer형태로 변환되어 저장되므로 문자나 음수같은 적절하지 않는 타입을 넣는 경우 RecipeException이 발생한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

- RecipeBook의 페이지를 초과하여 Recipe를 추가하는 경우

|  |
| --- |
| @Test public void addRecipeOutOfBoundFail() throws RecipeException {  // recipeBook의 사이즈 초과시 실패 테스트  Recipe recipe3 = new Recipe();  recipe3.setName(name);  recipe3.setPrice(price);  recipe3.setAmtCoffee(amtCoffee);  recipe3.setAmtMilk(amtMilk);  recipe3.setAmtSugar(amtSugar);  recipe3.setAmtChocolate(amtChocolate);  Recipe recipe4 = new Recipe();  recipe4.setName(name);  recipe4.setPrice(price);  recipe4.setAmtCoffee(amtCoffee);  recipe4.setAmtMilk(amtMilk);  recipe4.setAmtSugar(amtSugar);  recipe4.setAmtChocolate(amtChocolate);  Recipe recipe5 = new Recipe();  recipe4.setName(name);  recipe4.setPrice(price);  recipe4.setAmtCoffee(amtCoffee);  recipe4.setAmtMilk(amtMilk);  recipe4.setAmtSugar(amtSugar);  recipe4.setAmtChocolate(amtChocolate);   *assertTrue*(coffeeMaker.addRecipe(recipe3));  *assertTrue*(coffeeMaker.addRecipe(recipe4));  *assertTrue*(coffeeMaker.addRecipe(recipe5)); } |

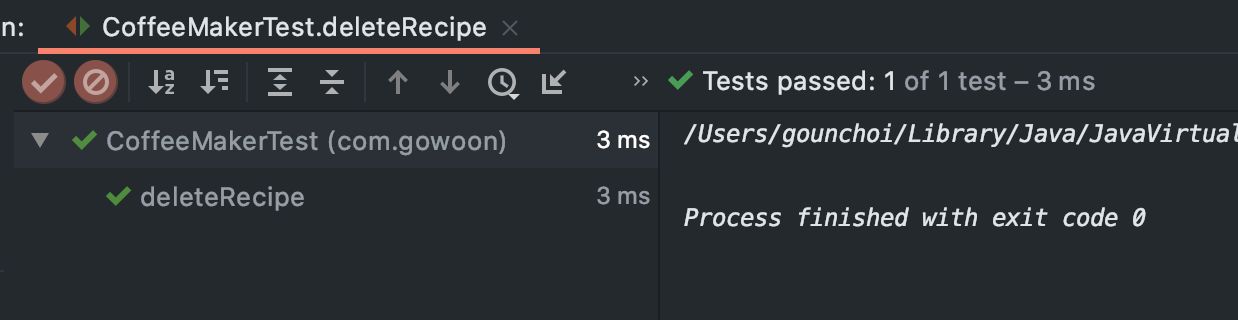
테스트시에 레시피를 너무 많이 생성해야하므로 RecipeBook의 PAGE\_NUM을 테스트의 편의를 위해 임의로 조금 줄여주고 테스트를 진행하였다. PAGE\_NUM을 넘게 레시피를 추가하는 경우 added가 true로 할당되지 못하고 false로 반환된다. 이 역시 추가되지 않아 false를 반환하는 것으로 원시코드 수정은 필요하지 않다.

**2. deleteRecipeTest**

- 성공 테스트

|  |
| --- |
| @Test public void deleteRecipe() {  // deleteRecipe 성공 테스트  coffeeMaker.addRecipe(recipe1);  *assertEquals*(name, coffeeMaker.getRecipes()[0].getName());  *assertEquals*(name, coffeeMaker.deleteRecipe(0));  *assertEquals*("", coffeeMaker.getRecipes()[0].getName()); } |

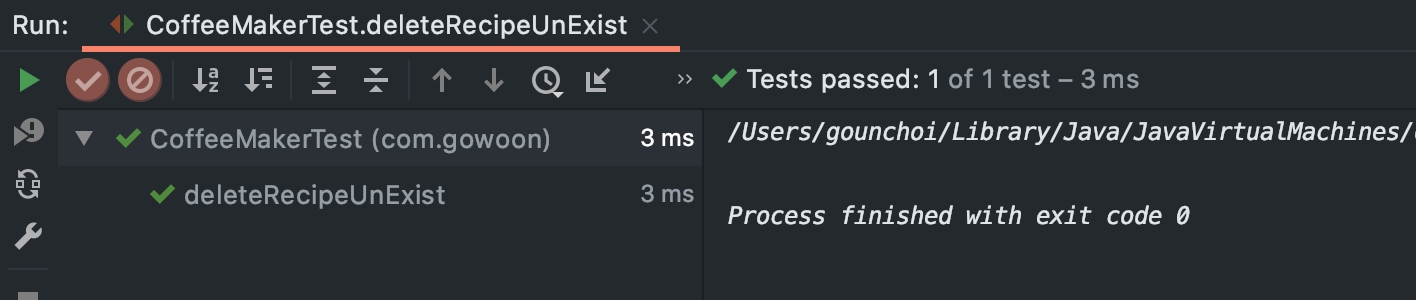
Recipe를 추가한 후 name을 확인해 정상적으로 레시피가 추가된 것을 확인한 후, 삭제를 진행하여 정상적으로 삭제가 진행되고 삭제된 recipe name이 반환되는 것을 확인할 수 있다. 삭제는 해당 recipe를 새로운 텅 빈 recipe로 할당해주므로 다시 해당 recipe의 name을 확인하면 초기값인 “”가 반환되는 것을 확인할 수 있다.



- 존재하지 않는 레시피 삭제 테스트

|  |
| --- |
| @Test public void deleteRecipeUnExist(){  // 존재하지 않는 Recipe delete시 null 반환  *assertEquals*(null, coffeeMaker.deleteRecipe(0)); } |

Delete에서 고려할만한 테스트 케이스로 존재하지 않는 recipe를 삭제하는 경우를 테스트한 결과 null을 반환하는 것을 확인할 수 있다. 원시코드에서 분기 처리를 통해 존재하지 않는 recipe에 대한 예외처리가 진행된 것이므로 원시코드를 수정할 필요는 없다.

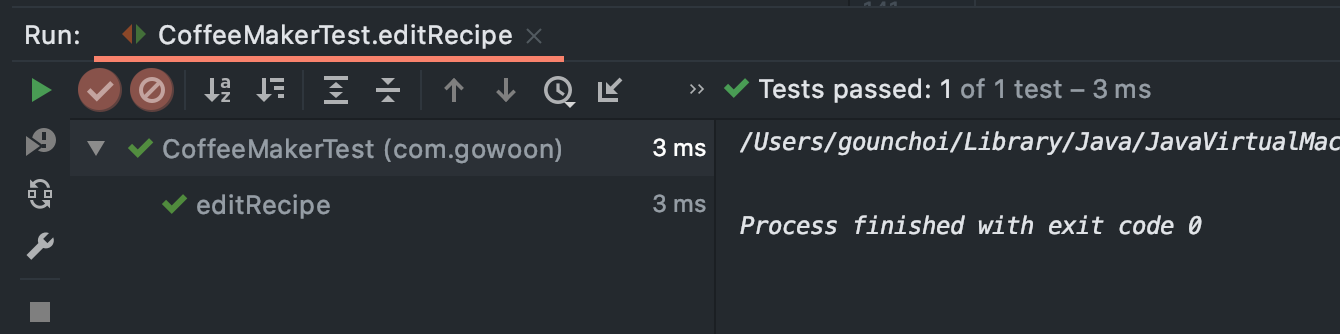


**3. editRecipeTest**

- 성공 테스트

|  |
| --- |
| @Test public void editRecipe() throws RecipeException {  // editRecipe 성공 테스트  Recipe recipe2 = new Recipe();  recipe2.setName("name2");  recipe2.setPrice("2000");   coffeeMaker.addRecipe(recipe1);  coffeeMaker.addRecipe(recipe2);   Recipe[] recipes = coffeeMaker.getRecipes();   *assertEquals*(1000,recipes[0].getPrice());  *assertEquals*("name1",coffeeMaker.editRecipe(0, recipe2));  *assertEquals*(2000,recipes[0].getPrice()); } |

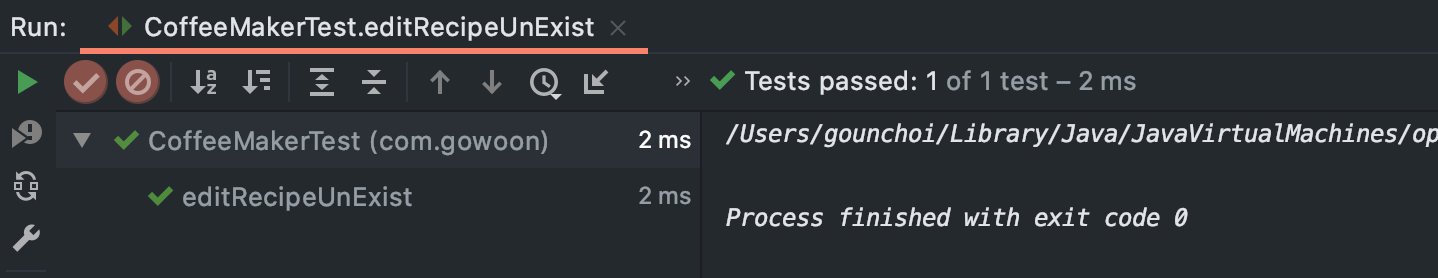
Recipe 두 개를 추가한 후에 첫 번째 레시피를 두 번째 레시피로 수정해주었다. 수정 전에는 가격이 1000이고, 수정을 진행하면서 name1 레시피가 수정된 것, 이후에는 가격이 2000으로 수정된 것까지 확인 가능하다.



- 존재하지 않는 레시피 수정 테스트

|  |
| --- |
| @Test public void editRecipeUnExist(){  // 존재하지 않는 recipe 수정시 null 반환  *assertEquals*(null, coffeeMaker.editRecipe(0, recipe1)); } |

Edit에서 고려할만한 테스트 케이스로 존재하지 않는 recipe를 수정하는 경우를 테스트한 결과 null을 반환하는 것을 확인할 수 있다. 원시코드에서 분기 처리를 통해 존재하지 않는 recipe에 대한 예외처리가 진행된 것이므로 원시코드를 수정할 필요는 없다.



**4. addInventoryTest**

- 성공 테스트

|  |
| --- |
| @Test public void addInventory() {  // addInventory 성공 테스트  *assertEquals*("Coffee: 15\nMilk: 15\nSugar: 15\nChocolate: 15\n", coffeeMaker.checkInventory());  coffeeMaker.addInventory("5","5","5","5");  *assertEquals*("Coffee: 20\nMilk: 20\nSugar: 20\nChocolate: 20\n", coffeeMaker.checkInventory()); } |

위와 같이 성공 테스트를 작성하여 테스트를 하였으나 addInventory()에서 아래와 같이 sugar must be a positive integer를 메세지로 하는 InventoryException이 발생하였다. 양수인 5를 넣어주었으므로 이상하다고 판단하고 원시코드를 확인하자 amtSugar가 0보다 크거나 같다면 반영하고, 그렇지 않은 경우 exception을 발생시켜야 하는데 0보다 작거나 같다고 조건이 되어있어 그 부분을 수정해주었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

|  |
| --- |
| 수정 전 원시코드 |
| public synchronized void addSugar(String sugar) throws InventoryException {  int amtSugar = 0;  try {  amtSugar = Integer.*parseInt*(sugar);  } catch (NumberFormatException e) {  throw new InventoryException("Units of sugar must be a positive integer");  }  if (amtSugar <= 0) {  Inventory.*sugar* += amtSugar;  } else {  throw new InventoryException("Units of sugar must be a positive integer");  } } |
| 수정 후 원시코드 |
| public synchronized void addSugar(String sugar) throws InventoryException {  int amtSugar = 0;  try {  amtSugar = Integer.*parseInt*(sugar);  } catch (NumberFormatException e) {  throw new InventoryException("Units of sugar must be a positive integer");  }  if (amtSugar >= 0) {  Inventory.*sugar* += amtSugar;  } else {  throw new InventoryException("Units of sugar must be a positive integer");  } } |

위와 같이 수정을 완료하고 다시 테스트를 수행하자 성공의 결과를 확인할 수 있었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

- 적절치 않은 타입의 변수를 넣는 경우

|  |
| --- |
| @Test public void addInventoryNegativeInteger() throws InventoryException {  coffeeMaker.addInventory("-5","5","5","5"); } @Test public void addInventoryInvalidType() throws InventoryException {  coffeeMaker.addInventory("a","5","5","5"); } |

위와 같이 음수를 넣거나, 문자를 넣는 등 양의 정수가 아닌 값을 넣는 경우 아래와 같이 Inventory Exception을 발생시킨다. 예외를 인식하고 Exception을 발생시키고 있으므로 원시코드 수정은 필요하지 않다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**5. checkInventoryTest**

- 성공 테스트

|  |
| --- |
| @Test public void checkInventory() throws InventoryException {  *assertEquals*("Coffee: 15\nMilk: 15\nSugar: 15\nChocolate: 15\n", coffeeMaker.checkInventory());  coffeeMaker.addInventory("5","5","5","5");  *assertEquals*("Coffee: 20\nMilk: 20\nSugar: 20\nChocolate: 20\n", coffeeMaker.checkInventory()); } |

위와 같이 inventoryCheck를 수행하면 정상적으로 기본값인 15를 가지고 결과를 반환하는 것을 확인할 수 있다. 이후 inventory값을 증가시키고 다시 확인하면 역시 정상적으로 변경된 값을 반환하는 것을 확인할 수 있다.

**6. makeCoffeeTest**

- 성공 테스트

|  |
| --- |
| @Test public void makeCoffee() {  *assertEquals*("Coffee: 15\nMilk: 15\nSugar: 15\nChocolate: 15\n", coffeeMaker.checkInventory());  coffeeMaker.addRecipe(recipe1);  coffeeMaker.makeCoffee(0, 1000);  *assertEquals*("Coffee: 14\nMilk: 14\nSugar: 14\nChocolate: 14\n", coffeeMaker.checkInventory()); } |

위와 같이 기존에 각 inventory가 초기값인 15임을 확인한 후 recipe1을 추가하고 이로 makeCoffee를 실행하였다. 1000이라는 pay를 정상적으로 지불하였다. 하지만 아래와 같이 실행 결과가 fail이었고, coffee의 inventory가 이상한 것을 확인하고 inventory 클래스의 useIngredients함수에서 coffee 값을 변경하는 코드를 아래와 같이 수정해주었다.

텍스트이(가) 표시된 사진

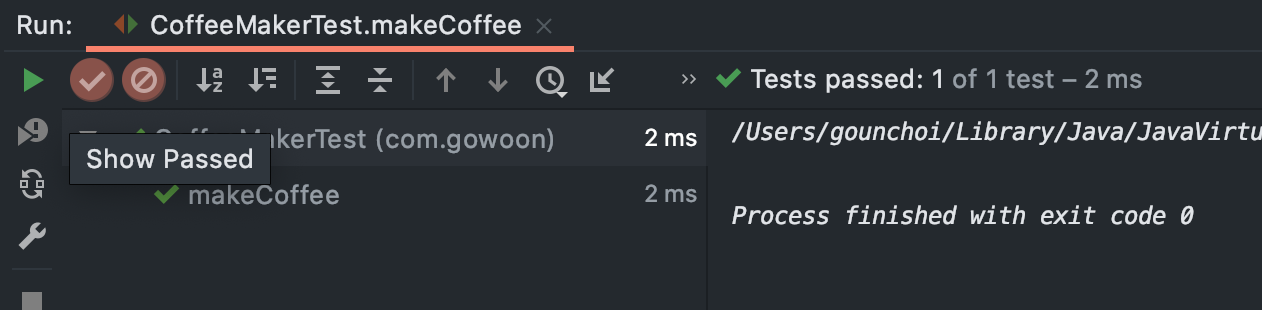
자동 생성된 설명

텍스트, 스크린샷, 모니터, 화면이(가) 표시된 사진

자동 생성된 설명

|  |
| --- |
| 수정 전 원시코드 |
| public synchronized boolean useIngredients(Recipe r) {  if (enoughIngredients(r)) {  Inventory.*coffee* += r.getAmtCoffee();  Inventory.*milk* -= r.getAmtMilk();  Inventory.*sugar* -= r.getAmtSugar();  Inventory.*chocolate* -= r.getAmtChocolate();  return true;  } else {  return false;  } } |
| 수정 후 원시코드 |
| public synchronized boolean useIngredients(Recipe r) {  if (enoughIngredients(r)) {  Inventory.*coffee* -= r.getAmtCoffee();  Inventory.*milk* -= r.getAmtMilk();  Inventory.*sugar* -= r.getAmtSugar();  Inventory.*chocolate* -= r.getAmtChocolate();  return true;  } else {  return false;  } } |

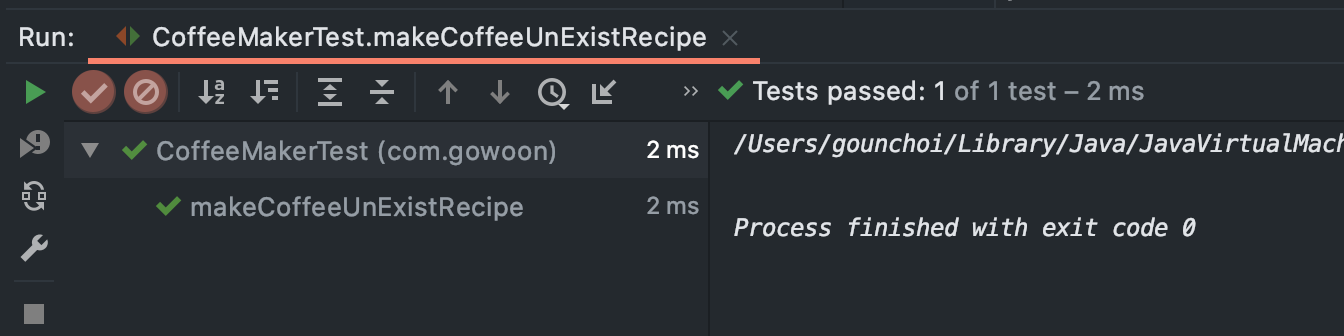
이후 위의 테스트를 다시 실행한 결과 아래와 같이 성공하는 것을 확인할 수 있다.



- 존재하지 않는 레시피

|  |
| --- |
| @Test public void makeCoffeeUnExistRecipe(){  *assertEquals*("Coffee: 15\nMilk: 15\nSugar: 15\nChocolate: 15\n", coffeeMaker.checkInventory());  coffeeMaker.addRecipe(recipe1);  coffeeMaker.makeCoffee(1, 1000);  } |

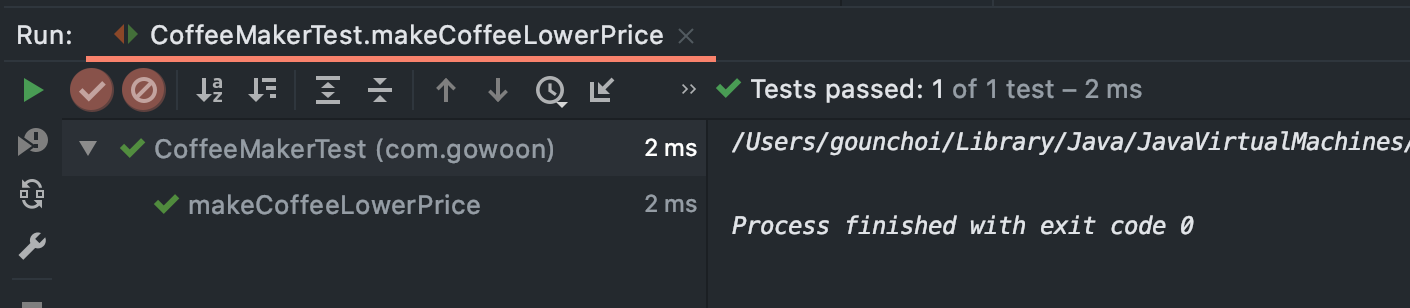
위와 같이 존재하지 않는 레시피로 커피를 만드는 경우 예외를 발생시켜야 하지만 아래와 같이 성공을 반환하는 것을 확인할 수 있다.



- 더 낮은 금액

|  |
| --- |
| @Test public void makeCoffeeLowerPrice(){  *assertEquals*("Coffee: 15\nMilk: 15\nSugar: 15\nChocolate: 15\n", coffeeMaker.checkInventory());  coffeeMaker.addRecipe(recipe1);  coffeeMaker.makeCoffee(0, 500); } |

위와 같이 지불해야하는 금액보다 더 낮은 금액을 지불한 경우 예외를 발생시켜야 하지만 아래와 같이 테스트가 성공하는 걸 확인할 수 있다.



- 거스름 돈

|  |
| --- |
| @Test public void makeCoffeeChange() {  *assertEquals*("Coffee: 15\nMilk: 15\nSugar: 15\nChocolate: 15\n", coffeeMaker.checkInventory());  coffeeMaker.addRecipe(recipe1);  *assertEquals*(200, coffeeMaker.makeCoffee(0, 1200));  *assertEquals*("Coffee: 14\nMilk: 14\nSugar: 14\nChocolate: 14\n", coffeeMaker.checkInventory()); } |

텍스트이(가) 표시된 사진

자동 생성된 설명

위와 같이 1000원의 커피를 만드는 경우 1200원을 낸다면 200원을 change로 정상적으로 반환하는 것을 확인할 수 있다.

7. getRecipesTest

- 성공테스트

|  |
| --- |
| @Test  public void getRecipes() {  coffeeMaker.addRecipe(recipe1);  *assertEquals*(recipe1, coffeeMaker.getRecipes()[0]);  } } |

