Cómo desarrollar una prueba JUnit para bibliotecas en APX

Cuando se crea una biblioteca en APX, de forma predeterminada se genera un paquete de prueba, que contiene clases para ejecutar pruebas unitarias.

En el directorio src\test\java, hay un paquete como com.bbva.uuaa.lib.r001, que contiene una clase denominada UUAAR001Test en la que se definen las pruebas unitarias (JUnit), donde:

- UUAA.- Identificar el código de la aplicación
- R001.- Identificar el resto del código en la biblioteca
- Test.- Es un sufijo que indica que se trata de una prueba

De forma predeterminada, se creará una prueba que iniciará la biblioteca sin ningún parámetro de entrada.

En la forma "tradicional" de hacer las pruebas, se utiliza el contexto de Spring, que utiliza una serie de Mocks, que permiten burlarse parcialmente de la arquitectura para la ejecución de las pruebas. El problema es que estas clases vienen en la carpeta Test, por lo que si la interfaz de los módulos simulados cambia, TODAS nuestras pruebas fallan. Además, se burla en exceso, lo que hace imposibles las pruebas unitarias y las hace excesivamente lentas (recuerda que hay que buscar pruebas unitarias para tomar siempre lo menos posible).

Por ejemplo, si utilizamos una librería con JDBC, creamos una fábrica jdbc, que nos permite burlarnos de JDBC, donde tenemos que poner nuestras consultas. El principal problema es que TODOS usan esta clase, por lo que si en nuestra biblioteca tenemos 50 llamadas diferentes desde jdbc, tendremos que tener en este punto los 50 mocks, por lo que perdemos el control de todo el contexto dentro de nuestra clase de prueba, ya que todo está cargado desde los contextos simulados.

Para acelerar este proceso y ahorrar tiempo, es mejor usar solo Mockito. Un ejemplo, tener este código:

```
paquete com.bbva.qwai.lib.r001.impl;

importe com.bbva.qwai.dto.customer.CustomerDTO;
importe com.bbva.qwai.lib.r001.QWAIR001;
importe com.bbva.qwai.lib.r002.QWAIR002;

importar org.slf4j.Logger;
importar org.slf4j.LoggerFactory;

importar java.util.ArrayList;
importar java.util.Date;
importar java.util.HashMap;
importar java.util.List;
```

```
importar java.util.Map;

 la clase pública  QWAIR001Impl extiende QWAIR001Abstract {
  Logger LOGGER final estático privado = LoggerFactory.getLogger(QWAIR001.class);
cadena final estática privada  CONSULTA_VACIA = "QWAI00841000";
cadena final estática privada  CAMPO_INCORRECTO = "QWAI00841001";
cadena final estática privada  ERROR_INSERT = "QWAI00841002";
 QWAIR002 privado qwaiR002;

	@Override
	public List<CustomerDTO> executeGet(String documentType, int paginationKey, int
pagSize) {
		List<Map<String, Object>> mapCustomers;
		List<CustomerDTO> dtoCustomers = null;
if ((documentType == null) || (documentType.length() ==  0)) {
			mapCustomers = jdbcUtils.pagingQueryForList("sql.select_all", paginationKey,
pagSize);
		} else {
			String codDocType = qwaiR002.executeDtoToBdDocType(documentType);
si(! "".equals(codDocType)){
				mapCustomers = jdbcUtils.pagingQueryForList("sql.select_codtipident",
paginationKey, pagSize, codDocType);
			}
			else{
				LOGGER.warn("Campo de entrada incorrecto {}",documentType);
this.addAdvice(CAMPO_INCORRECTO,"documentType");
				return dtoCustomers;
			}
		}
		if (mapCustomers.isEmpty()) {
			LOGGER.warn("No hay registros en la consulta");
this.addAdvice(CONSULTA_VACIA);
		} else {
			dtoCustomers = nuevo ArrayList<>();
			for (Map<String, Object> map : mapCustomers) {
				CustomerDTO dto = mapToDTOCustomer(mapa);
				dtoCustomers.add(dto);
			}
			LOGGER.info("Se obtuvieron " + dtoCustomers.size() + " clientes");
		}
		return dtoCustomers;
	}

	@Override
public boolean executePost(CustomerDTO dto, String user, String entity) {
		// Validacion de campos de entrada por sus valores
		resultado booleano = false;
		if (comparadorCampos(dto)) {
int resultado = this.jdbcUtils.update("sql.insert_customer", dtoToMapCustomer(dto, usuario,
entidad));
			if (resultado == 1) {
				LOGGER.info("Cliente registrado en DB");
				resultado = verdadero;
			} else {
				LOGGER.error("No se registro el cliente en BD");
this.addAdvice(ERROR_INSERT);
```

```
                }
            }
            resultado de devolución ;
        }

        private CustomerDTO mapToDTOCustomer(Map<String, Object> map) {
            CustomerDTO dto = nuevo CustomerDTO();
dto.setNationality(map.get("COD_PAISOALF").toString());
dto.setIdentityDocumentType(qwaiR002.executeBdToDtoDocType(map.get("COD_TIPIDENT"
).toString()));
dto.setIdentityDocumentNumber(map.get("COD_DOCUMPS").toString());
dto.setPersonalTitle(qwaiR002.executeBdToDtoPersTitle(map.get("COD_TRATANOR").toStri
ng()));
dto.setFirstName(map.get("DES_NOMBFJ").toString());
dto.setLastName(map.get("DES_APELLUNO").toString() + " " + map.get("DES_APELLDOS"));
dto.setMaritalStatus(qwaiR002.executeBdToDtoMaritalSt(map.get("COD_CECIVI").toString()))
;
dto.setBirthDate((Date) map.get("FEC_FNACIF"));
dto.setGenderId(qwaiR002.executeBdToDtoGender(map.get("XTI_SEXO").toString()));
dto.setCustomerId((String) map.get("COD_PERSCTPN"));
            devolver dto;
        }

        private Map<String, Object> dtoToMapCustomer(CustomerDTO dto, String user, String
entity) {
            Map<String, Object> map = nuevo HashMap<>();
map.put("codPaisoalf", dto.getNationality());
map.put("codTipident", qwaiR002.executeDtoToBdDocType(dto.getIdentityDocumentType()));
map.put("codDocumps", dto.getIdentityDocumentNumber());
map.put("codTratanor", qwaiR002.executeDtoToBdPersTitle(dto.getPersonalTitle()));
map.put("desNombfj", dto.getFirstName());
map.put("desApelluno", dto.getLastName().split(" ")[0]);
map.put("desApelldos", dto.getLastName().split(" ")[1]);
map.put("codCecivi", qwaiR002.executeDtoToBdMaritalSt(dto.getMaritalStatus()));
map.put("fecFnacif", dto.getBirthDate());
map.put("xtiSexo", qwaiR002.executeDtoToBdGender(dto.getGenderId()));
map.put("codPersctpn", dto.getCustomerId());
map.put("audUsuario", usuario);
map.put("codEntalfa", entidad);
 mapa de retorno;
        }

        comparador booleano privadoCampos(CustomerDTO dto) {
            resultado booleano = false;
            String docType =
qwaiR002.executeDtoToBdDocType(dto.getIdentityDocumentType());
            String genderId = qwaiR002.executeDtoToBdGender(dto.getGenderId());
            String personalTitle = qwaiR002.executeDtoToBdPersTitle(dto.getPersonalTitle());
            String maritalStatus = qwaiR002.executeDtoToBdMaritalSt(dto.getMaritalStatus());
            if ("".equals(docType)) {
this.addAdvice(CAMPO_INCORRECTO, "documentType");
            } else if ("".equals(genderId)) {
this.addAdvice(CAMPO_INCORRECTO, "genderId");
            } else if ("".equals(personalTitle)) {
this.addAdvice(CAMPO_INCORRECTO, "personalTitle");
            } else if ("".equals(maritalStatus)) {
```

```
this.addAdvice(CAMPO_INCORRECTO, "maritalStatus");
            } else {
                  resultado = verdadero;
            }
            resultado de devolución ;
      }

      public void setQwaiR002(QWAIR002 qwaiR002) {
this.qwaiR002 = qwaiR002;
      }
}
```

La clase de prueba será esta:

```
paquete com.bbva.qwai.lib.r001;

importar java.text.ParseException;
importar java.text.SimpleDateFormat;
importar java.util.ArrayList;
importar java.util.Date;
importar java.util.HashMap;
importar java.util.List;
importar java.util.Map;

importar org.junit.Before;
importar org.junit.Test;
importar org.mockito.InjectMocks;
importar org.mockito.Mock;
importar org.mockito.Mockito;
importar org.mockito.MockitoAnnotations;
importar org.mockito.Spy;
importar org.slf4j.Logger;
importar org.slf4j.LoggerFactory;
import org.springframework.aop.framework.Advised;

importe com.bbva.elara.utility.jdbc.JdbcUtils;
importe com.bbva.qwai.dto.customer.CustomerDTO;
importe com.bbva.qwai.lib.r001.impl.QWAIR001Impl;
importe com.bbva.qwai.lib.r002.QWAIR002;
importe com.bbva.qwai.mock.QWAIR002Mock;

import junit.framework.Assert;

import com.bbva.elara.domain.transaction.Advice;
importe com.bbva.elara.domain.transaction.Context;
import com.bbva.elara.domain.transaction.ThreadContext;

clase pública QWAIR001Test {

      Logger LOGGER final estático privado =
LoggerFactory.getLogger(QWAIR001.class);
```

```
    @InjectMocks
    QWAIR001Impl privado qwaiR001;

    @Spy
  contexto privado;

    @Mock
    privado JdbcUtils jdbcUtils;

    QWAIR002 privado qwaiR002 = nuevo QWAIR002Mock();

    @Before
    public void setUp() arroja Exception {
    MockitoAnnotations.initMocks(esto);
    Establecer el contexto actual para las pruebas de consejos
    ThreadContext.set(contexto);
    getObjectIntrospection();
  }

    private Object getObjectIntrospection() arroja Exception {
        Resultado del objeto = this.qwaiR001;
        if (this.qwaiR001 instanceof Advised) {
            Aconsejado aconsejado = (Aconsejado) this.qwaiR001;
            result = advised.getTargetSource().getTarget();
        }
        resultado de devolución;
    }

    @Test
    public void executeGet(){
        LOGGER.info("Ejecucion del test executeGet");
        this.qwaiR001.setQwaiR002(qwaiR002);

Mockito.doReturn(returnCustomers(null)).when(jdbcUtils).pagingQueryForList("sql.sele
ct_all", 0, 10);
        List<CustomerDTO> clientes = qwaiR001.executeGet(null, 0, 10);
        LOGGER.info("Busqueda realizada en test get");
        List<Advice> adviceList = context.getAdviceList();
        Assert.assertEquals(0, adviceList.size());
        Assert.assertNotNull(clientes);
    }
    @Test
    public void executeGetEmptyString(){
        LOGGER.info("Ejecucion del test executeGetEmptyString");
        this.qwaiR001.setQwaiR002(qwaiR002);

Mockito.doReturn(returnCustomers(null)).when(jdbcUtils).pagingQueryForList("sql.sele
ct_all", 0, 10);
        List<CustomerDTO> clientes = qwaiR001.executeGet("", 0, 10);
        LOGGER.info("Busqueda realizada en test get");
        List<Advice> adviceList = context.getAdviceList();
        Assert.assertEquals(0, adviceList.size());
        Assert.assertNotNull(clientes);
    }
    @Test
```

```
	public void executeGetEmpty(){
		LOGGER.info("Ejecucion del test executeGetEmpty");
		this.qwaiR001.setQwaiR002(qwaiR002);
		Mockito.doReturn(new ArrayList<Map<String,
Object>>()).when(jdbcUtils).pagingQueryForList("sql.select_all", -1, 10);
		List<CustomerDTO> clientes = qwaiR001.executeGet(null, -1, 10);
		LOGGER.info("Busqueda realizada en test get");
		AFIRMAR
	List<Advice> adviceList = context.getAdviceList();
	Assert.assertEquals(adviceList.get(0).getCode(), "QWAI00841000");
	Assert.assertNull(clientes);
	}
	@Test
	public void executeGetQueryParam() {
		LOGGER.info("Ejecucion del test del executeGetQueryParam");
		this.qwaiR001.setQwaiR002(qwaiR002);

Mockito.doReturn(returnCustomers("CURP")).when(jdbcUtils).pagingQueryForList("sql.
select_codtipident", 0, 10,"7");
		List<CustomerDTO> clientes = qwaiR001.executeGet("CURP", 0, 10);
		LOGGER.debug("Cliente encontrado"+customers.get(0));
		LOGGER.info("Busqueda realizada en test get");
		List<Advice> adviceList = context.getAdviceList();
		Assert.assertEquals(0, adviceList.size());
		Assert.assertNotNull(clientes);
	}
}
```

Al usar Mockito, podemos hacer uso de las siguientes anotaciones:

@InjectMocks: Crea instancias de la biblioteca que queremos probar e inyecta todos los simulacros definidos.

@Mock: Esto crea una instancia de una instancia simulada de la clase anotada.

@Spy: Crea una instancia de la biblioteca probada, pero la agrega al contexto de ejecución de Mockito para que podamos tratarla como si fuera un Mock, pero conserva su lógica inicial a menos que se especifique lo contrario.

De esta manera, ganamos control y sobre todo, rapidez en la ejecución de nuestros trajes de prueba.