# Código Fuente y Pruebas de Unidad

## Iteración 1:

#### Taskcard 3 Vista alta, modificar y baja vendedor

Código del archivo AdministrarVendedorController.java

/\*\*

\* Acción que se ejecuta al presionar el botón agregar

\* Se pasa a la pantalla alta vendedor

\*/

**public** **void** agregarAction(ActionEvent event) {

//Se llama a la pantalla alta vendedor

cambiarmeAScene(AltaVendedorController.***URLVista***, ***URLVista***);

}

/\*\*

\* Acción que se ejecuta al presionar el botón modificar

\* Se pasa a la pantalla modificar vendedor

\*/

**public** **void** modificarAction(ActionEvent event) {

//Se comprueba que efectivamente haya un vendedor seleccionado

Vendedor vendedor = tablaVendedores.getSelectionModel().getSelectedItem();

**if**(vendedor == **null**){

**return**;

}

//Se llama a la pantalla de modificar vendedor

ModificarVendedorController controlador = (ModificarVendedorController) cambiarmeAScene(ModificarVendedorController.***URLVista***, ***URLVista***);

controlador.setVendedor(vendedor);

}

/\*\*

\* Acción que se ejecuta al presionar el botón eliminar

\* Se muestra una ventana emergente para confirmar la operación

\*/

**public** **void** eliminarAction(ActionEvent event) {

//Se comprueba que efectivamente haya un vendedor seleccionado

Vendedor vendedor = tablaVendedores.getSelectionModel().getSelectedItem();

**if**(vendedor == **null**){

**return**;

}

//Se pregunta al usuario si quiere eliminar el vendedor

VentanaConfirmacion ventana = presentador.presentarConfirmacion("Eliminar vendedor", "Está a punto de eliminar al vendedor.\n¿Desea continuar?", **this**.stage);

**if**(ventana.acepta()){

**try**{

//Se llama a la lógica para eliminar el inmueble y se recibe el resultado de las validaciones

ResultadoEliminarVendedor resultado = coordinador.eliminarVendedor(vendedor);

**if**(resultado.hayErrores()){

StringBuilder stringErrores = **new** StringBuilder();

//Procesamiento de errores de la lógica

**for**(ErrorEliminarVendedor err: resultado.getErrores()){

**switch**(err) {

//Por el momento no hay errores que se puedan mostrar

}

}

//Se muestran los errores devueltos por la vista

presentador.presentarError("No se pudo eliminar el vendedor", stringErrores.toString(), stage);

}

**else**{

//Si no hubo errores

//Se quita el inmueble de la vista

tablaVendedores.getItems().remove(vendedor);

//Se muestra una notificación de que se eliminó correctamente el inmueble

presentador.presentarToast("Se ha eliminado al vendedor " + vendedor.getNombre() + " con éxito", stage);

}

//Se muestran las excepciones, en caso de que ocurra alguno

} **catch**(PersistenciaException e){

presentador.presentarExcepcion(e, stage);

} **catch**(Exception e){

presentador.presentarExcepcionInesperada(e,stage);

}

}

}

Prueba de unidad del archivo AdministrarInmuebleControllerTest.java

/\*\*

\* Prueba el método agregarVendedor()

\* Se comprueba que se llame a la pantalla correspondiente cuando se presiona el boton agregar

\*

\* **@throws** Throwable

\*/

@Test

**public** **void** testAgregarVendedor() **throws** Throwable {

//Se crean los mocks necesarios

ScenographyChanger scenographyChangerMock = *mock*(ScenographyChanger.**class**);

AltaVendedorController altaVendedorControllerMock = *mock*(AltaVendedorController.**class**);

CoordinadorJavaFX coordinadorMock = *mock*(CoordinadorJavaFX.**class**);

//Se setea lo que debe devolver el mock cuando es invocado por la clase a probar

*when*(scenographyChangerMock.cambiarScenography(*any*(String.**class**), *any*())).thenReturn(altaVendedorControllerMock);

//Controlador a probar;

AdministrarVendedorController administrarVendedorController = **new** AdministrarVendedorController() {

@Override

**public** **void** inicializar(URL location, ResourceBundle resources) {

**this**.coordinador = coordinadorMock;

**this**.setScenographyChanger(scenographyChangerMock);

**super**.inicializar(location, resources);

}

@Override

**protected** **void** setTitulo(String titulo) {

}

};

//Los controladores de las vistas deben correrse en un thread de JavaFX

ControladorTest corredorTestEnJavaFXThread = **new** ControladorTest(AdministrarVendedorController.***URLVista***, administrarVendedorController);

administrarVendedorController.setStage(corredorTestEnJavaFXThread.getStagePrueba());

Statement test = **new** Statement() {

@Override

**public** **void** evaluate() **throws** Throwable {

//Método a probar

administrarVendedorController.agregarAction(**null**);

//Se hacen las verificaciones pertinentes para comprobar que el controlador se comporte adecuadamente

Mockito.*verify*(scenographyChangerMock, *times*(1)).cambiarScenography(AltaVendedorController.***URLVista***, **false**);

}

};

//Se corre el test en el hilo de JavaFX

corredorTestEnJavaFXThread.apply(test, **null**).evaluate();

;

}

//Casos de prueba

//listaVendedores, llamaAModificar

/\* 0 \*/ **new** Object[] { listaVendedoresConUnVendedor, 1}, //Si hay un vendedor seleccionado se debe llamar a la pantalla modificar

/\* 1 \*/ **new** Object[] { listaVendedoresVacia, 0} //Si la lista es vacía no se debería llamar a modificar

/\*\*

\* Prueba el método Modificar Vendedor()

\* **@param** listaVendedores

\* vendedores inicialmente en la tabla de vendedores

\* **@param** llamaAModificar

\* indica si se debe llamar a la pantalla modificar vendedor

\* **@throws** Throwable

\*/

@Test

@Parameters

**public** **void** testModificarVendedor(ArrayList<Vendedor> listaVendedores, Integer llamaAModificar) **throws** Throwable {

//Se crean los mocks necesarios

ScenographyChanger scenographyChangerMock = *mock*(ScenographyChanger.**class**);

CoordinadorJavaFX coordinadorMock = *mock*(CoordinadorJavaFX.**class**);

ModificarVendedorController modificarVendedorControllerMock = *mock*(ModificarVendedorController.**class**);

//Se setea lo que deben devolver los mocks cuando son invocados por la clase a probar

*when*(coordinadorMock.obtenerVendedores()).thenReturn(listaVendedores);

*when*(scenographyChangerMock.cambiarScenography(*any*(String.**class**), *any*())).thenReturn(modificarVendedorControllerMock);

//Controlador a probar, se sobreescriben algunos métodos para setear los mocks y setear los datos en la tabla

AdministrarVendedorController administrarVendedorController = **new** AdministrarVendedorController() {

@Override

**public** **void** inicializar(URL location, ResourceBundle resources) {

**this**.coordinador = coordinadorMock;

**this**.setScenographyChanger(scenographyChangerMock);

**super**.inicializar(location, resources);

**this**.tablaVendedores.getSelectionModel().select(0);

}

@Override

**protected** **void** setTitulo(String titulo) {

}

};

//Los controladores de las vistas deben correrse en un thread de JavaFX

ControladorTest corredorTestEnJavaFXThread = **new** ControladorTest(AdministrarVendedorController.***URLVista***, administrarVendedorController);

administrarVendedorController.setStage(corredorTestEnJavaFXThread.getStagePrueba());

Statement test = **new** Statement() {

@Override

**public** **void** evaluate() **throws** Throwable {

//Método a probar

administrarVendedorController.modificarAction(**null**);

//Se hacen las verificaciones pertinentes para comprobar que el controlador se comporte adecuadamente

Mockito.*verify*(scenographyChangerMock, *times*(llamaAModificar)).cambiarScenography(ModificarVendedorController.***URLVista***, **false**);

}

};

//Se corre el test en el hilo de JavaFX

corredorTestEnJavaFXThread.apply(test, **null**).evaluate();

}

//Casos de prueba

//listaVendedores, llamaAPresentadorVentanasPresentarConfirmacion, aceptarVentanaConfirmacion, resultadoEliminarVendedorEsperado, llamaAPresentadorVentanasPresentarError, llamaAPresentadorVentanasPresentarExcepcion, excepcion

/\* 0 \*/ **new** Object[] { listaVendedores, 1, **true**, resultadoSinError, 0, 0, **null** }, //test correcto, se acepta la confirmación y sin errores

/\* 1 \*/ **new** Object[] { listaVendedores, 1, **true**, resultadoSinError, 0, 1, exception }, //la lógica devuelve una excepción

/\* 2 \*/ **new** Object[] { listaVendedores, 1, **false**, resultadoSinError, 0, 0, **null** }, //no se acepta la ventna de confirmación

/\* 3 \*/ **new** Object[] { listaVendedoresVacia, 0, **false**, resultadoSinError, 0, 0, **null** }, //la lista de vendedores es vacía

/\*\*

\* Prueba el método eliminar Vendedor()

\* **@param** listaVendedores

\* vendedores inicialmente en la tabla de vendedores

\* **@param** llamaAPresentadorVentanasPresentarConfirmacion

\* indica si se debe llamar a la ventana de presentar confirmación

\* **@param** aceptarVentanaConfirmacion

\* indica si el usuario acepta la ventana de confirmación

\* **@param** resultadoEliminarVendedorEsperado

\* resultado devuelto por la capa lógica

\* **@param** llamaAPresentadorVentanasPresentarError

\* indica si se debe presentar un error

\* **@param** llamaAPresentadorVentanasPresentarExcepcion

\* indica si se debe presentar una excepcion

\* **@param** excepcion

\* excepcion lanzada por la capa lógica

\* **@throws** Throwable

\*/

@Test

@Parameters

**public** **void** testEliminarVendedor(ArrayList<Vendedor> listaVendedores,

Integer llamaAPresentadorVentanasPresentarConfirmacion,

Boolean aceptarVentanaConfirmacion,

ResultadoEliminarVendedor resultadoEliminarVendedorEsperado,

Integer llamaAPresentadorVentanasPresentarError,

Integer llamaAPresentadorVentanasPresentarExcepcion,

PersistenciaException excepcion)

**throws** Throwable {

//Se crean los mocks necesarios

ScenographyChanger scenographyChangerMock = *mock*(ScenographyChanger.**class**);

CoordinadorJavaFX coordinadorMock = *mock*(CoordinadorJavaFX.**class**);

PresentadorVentanas presentadorVentanasMock = *mock*(PresentadorVentanas.**class**);

VentanaError ventanaErrorMock = *mock*(VentanaError.**class**);

VentanaErrorExcepcion ventanaErrorExcepcionMock = *mock*(VentanaErrorExcepcion.**class**);

VentanaConfirmacion ventanaConfirmacionMock = *mock*(VentanaConfirmacion.**class**);

//Se setea lo que deben devolver los mocks cuando son invocados por la clase a probar

*when*(coordinadorMock.obtenerVendedores()).thenReturn(listaVendedores);

*when*(presentadorVentanasMock.presentarError(*any*(), *any*(), *any*())).thenReturn(ventanaErrorMock);

*when*(presentadorVentanasMock.presentarExcepcion(*any*(), *any*())).thenReturn(ventanaErrorExcepcionMock);

*when*(presentadorVentanasMock.presentarConfirmacion(*any*(), *any*(), *any*())).thenReturn(ventanaConfirmacionMock);

*when*(ventanaConfirmacionMock.acepta()).thenReturn(aceptarVentanaConfirmacion);

*doNothing*().when(presentadorVentanasMock).presentarToast(*any*(), *any*());

*when*(coordinadorMock.eliminarVendedor(*any*(Vendedor.**class**))).thenReturn(resultadoEliminarVendedorEsperado);

**if**(excepcion != **null**){

*when*(coordinadorMock.eliminarVendedor(*any*(Vendedor.**class**))).thenThrow(excepcion);

}

//Controlador a probar, se sobreescriben algunos métodos para setear los mocks y setear los datos en la tabla

AdministrarVendedorController administrarVendedorController = **new** AdministrarVendedorController() {

@Override

**public** **void** inicializar(URL location, ResourceBundle resources) {

**this**.coordinador = coordinadorMock;

**this**.setScenographyChanger(scenographyChangerMock);

**this**.presentador = **new** PresentadorVentanas();

**this**.presentador = presentadorVentanasMock;

**super**.inicializar(location, resources);

**this**.tablaVendedores.getSelectionModel().select(0); //se selecciona el primer vendedor de la lista

}

@Override

**protected** **void** setTitulo(String titulo) {

}

};

//Los controladores de las vistas deben correrse en un thread de JavaFX

ControladorTest corredorTestEnJavaFXThread = **new** ControladorTest(AdministrarVendedorController.***URLVista***, administrarVendedorController);

administrarVendedorController.setStage(corredorTestEnJavaFXThread.getStagePrueba());

Statement test = **new** Statement() {

@Override

**public** **void** evaluate() **throws** Throwable {

Integer cantidadVendedoresAntesDeEliminar = administrarVendedorController.tablaVendedores.getItems().size();

//Método a probar

administrarVendedorController.eliminarAction(**null**);

//Se hacen las verificaciones pertinentes para comprobar que el controlador se comporte adecuadamente

Mockito.*verify*(presentadorVentanasMock, *times*(llamaAPresentadorVentanasPresentarConfirmacion)).presentarConfirmacion(*any*(), *any*(), *any*());

Mockito.*verify*(presentadorVentanasMock, *times*(llamaAPresentadorVentanasPresentarConfirmacion)).presentarConfirmacion(*any*(), *any*(), *any*());

Mockito.*verify*(presentadorVentanasMock, *times*(llamaAPresentadorVentanasPresentarExcepcion)).presentarExcepcion(*eq*(excepcion), *any*());

**if**(llamaAPresentadorVentanasPresentarExcepcion == 0){

Mockito.*verify*(presentadorVentanasMock, *times*(llamaAPresentadorVentanasPresentarError)).presentarError(*any*(), *any*(), *any*());

}

Integer cantidadVendedoresDespuesDeEliminar = administrarVendedorController.tablaVendedores.getItems().size();

**if**(llamaAPresentadorVentanasPresentarError != 1 && llamaAPresentadorVentanasPresentarExcepcion != 1 && aceptarVentanaConfirmacion){

*assertNotEquals*(cantidadVendedoresAntesDeEliminar, cantidadVendedoresDespuesDeEliminar);

}

}

};

//Se corre el test en el hilo de JavaFX

corredorTestEnJavaFXThread.apply(test, **null**).evaluate();

}

Código del archivo AltaVendedorController.java

/\*\*

\* Acción que se ejecuta al apretar el botón aceptar.

\*

\* Valida que se hayan insertado datos, los carga al vendedor y deriva la operación a capa lógica.

\* Si la capa lógica retorna errores, éstos se muestran al usuario.

\*/

**public** **void** acceptAction() {

//Se toman los datos ingresados por el usuario

cargarDatos();

//Se validan los datos ingresados por el usuario

StringBuilder error = **new** StringBuilder("");

validarVistaAltaVendedor(error);

//Si hay errores, se los muestra en pantalla, si no, se llama a crear un vendedor en la capa lógica

**if**(!error.toString().isEmpty()){

presentador.presentarError("Revise sus campos", error.toString(), stage);

}

**else**{

crearVendedor();

}

}

/\*\*

\* Método que encapsula el llamado de crear un vendedor a la capa lógica y maneja los errores devueltos

\*/

**private** **void** crearVendedor() {

//Se crea el vendedor a pasarle a la capa lógica

Vendedor vendedor = **new** Vendedor();

vendedor.setNombre(nombre)

.setApellido(apellido)

.setNumeroDocumento(numeroDocumento)

.setTipoDocumento(tipoDoc)

.setSalt(encriptador.generarSal())

.setPassword(encriptador.encriptar(password1.toCharArray(), vendedor.getSalt()));

ResultadoCrearVendedor resultadoCrearVendedor = **null**;

**try**{

//Se llama a crear un vendedor en la capa lógica

resultadoCrearVendedor = coordinador.crearVendedor(vendedor);

StringBuilder error = **new** StringBuilder("");

List<ErrorCrearVendedor> listaErrores = resultadoCrearVendedor.getErrores();

//Se crea un mensaje apropiado a mostrar según el error recibido

parsearErroresLogica(error, listaErrores);

//Si hay errores, se los muestra en pantalla, si no, se presenta una notificación indicando el éxito de la operación

**if**(!error.toString().isEmpty()){

presentador.presentarError("Revise sus campos", error.toString(), stage);

}

**else**{

presentador.presentarToast("Se ha creado el vendedor con éxito", stage);

salir();

}

//Se manejan las excepciones que puede devolver la lógica

} **catch**(PersistenciaException e){

presentador.presentarExcepcion(e, stage);

} **catch**(EntidadExistenteConEstadoBajaException e){

//Si el vendedor ya existía pero fue dado de baja se debe mostrar una ventana preguntando al usuario si desea darlo de alta

manejarVendedorExistenteBaja(vendedor);

} **catch**(Exception e){

presentador.presentarExcepcionInesperada(e, stage); //falta el stage

}

}

/\*\*

\* Método que se llama al querer dar de alta un vendedor que fue dado de baja previamente

\* Si el usuario confirma la operación se llama a la pantalla modificar vendedor

\*

\* **@param** vendedor

\*/

**private** **void** manejarVendedorExistenteBaja(Vendedor vendedor) {

VentanaConfirmacion ventana = presentador.presentarConfirmacion("El vendedor ya existe", "El vendedor está dado de baja. Si continúa podrá darlo de alta nuevamente. ¿Desea continuar?", stage);

**if**(ventana.acepta()){

//Si el usuario acepta se llama a la pantalla modificar vendedor

**try**{

vendedor = coordinador.obtenerVendedor(vendedor);

} **catch**(PersistenciaException e1){

presentador.presentarExcepcion(e1, stage);

}

ModificarVendedorController controlador = (ModificarVendedorController) cambiarmeAScene(ModificarVendedorController.***URLVista***, URLVistaRetorno);

controlador.setVendedor(vendedor);

controlador.setAltaVendedor();

}

}

/\*\*

\* Se convierten los errores devueltos por la capa lógica a errores a mostrar al usuario

\*

\* **@param** error

\* Texto a mostrar al usuario

\* **@param** listaErrores

\* Lista de errores de la capa lógica

\*/

**private** **void** parsearErroresLogica(StringBuilder error, List<ErrorCrearVendedor> listaErrores) {

**if**(listaErrores.contains(ErrorCrearVendedor.***Formato\_Nombre\_Incorrecto***)){

error.append("Nombre Incorrecto").append("\r\n");

}

**if**(listaErrores.contains(ErrorCrearVendedor.***Formato\_Apellido\_Incorrecto***)){

error.append("Apellido Incorrecto").append("\r\n");

}

**if**(listaErrores.contains(ErrorCrearVendedor.***Formato\_Documento\_Incorrecto***)){

error.append("Documento Incorrecto").append("\r\n");

}

**if**(listaErrores.contains(ErrorCrearVendedor.***Ya\_Existe\_Vendedor***)){

error.append("Ya existe un vendedor registrado con ese documento").append("\r\n");

}

}

/\*\*

\* Se valida que los campos ingresados por el usuario sean correctos

\*

\* **@param** error

\* Texto a mostrar en caso de que haya algún error

\*/

**private** **void** validarVistaAltaVendedor(StringBuilder error) {

**if**(nombre.isEmpty()){

error.append("Inserte un nombre").append("\r\n");

}

**if**(apellido.isEmpty()){

error.append("Inserte un apellido").append("\r\n");

}

**if**(tipoDoc == **null**){

error.append("Elija un tipo de documento").append("\r\n");

}

**if**(numeroDocumento.isEmpty()){

error.append("Inserte un numero de documento").append("\r\n");

}

**if**(password1.isEmpty() && password2.isEmpty()){

error.append("Inserte su contraseña").append("\r\n");

}

**if**(!password1.isEmpty() && password2.isEmpty()){

error.append("Inserte su contraseña nuevamente").append("\r\n");

}

**if**(!password1.equals(password2)){

error.append("Sus contraseñas no coinciden, Ingreselas nuevamente").append("\r\n ");

}

}

/\*\*

\* Se cargan los datos de la vista a variables para su posterior utilización

\*/

**private** **void** cargarDatos() {

nombre = textFieldNombre.getText().trim();

apellido = textFieldApellido.getText().trim();

numeroDocumento = textFieldNumeroDocumento.getText().trim();

password1 = passwordFieldContraseña.getText();

password2 = passwordFieldRepiteContraseña.getText();

tipoDoc = comboBoxTipoDocumento.getValue();

}

Prueba de unidad del archivo AltaVendedorControllerTest.java

//Casos de prueba //nombre,apellido,tipoDocumento,numeroDocumento,contraseña,contraseña2,resultadoCrearVendedorEsperado,llamaAPresentadorVentanasPresentarError,llamaAPresentadorVentanasPresentarExcepcion,llamaAPresentadorVentanasPresentarExcepcionInesperada,llamaACrearVendedor,excepcion,aceptarVentanaConfirmacion,llamaACambiarScene

/\* 0 \*/ **new** Object[] { "Juan", "Perez", doc, "12345678", "abc", "abc", ***resultadoCorrecto***, 0, 0, 0, 1, **null**, **true**, 0 }, //prueba correcta

/\* 1 \*/**new** Object[] { "Juan", "Perez", doc, "12345678", "abc", "abc", ***resultadoCrearNombreIncorrecto***, 1, 0, 0, 1, **null**, **true**, 0 }, //prueba nombre incorrecto

/\* 2 \*/**new** Object[] { "Juan", "Perez", doc, "12345678", "abc", "abc", ***resultadoCrearApellidoIncorrecto***, 1, 0, 0, 1, **null**, **true**, 0 }, //prueba apellido incorrecto

/\* 3 \*/**new** Object[] { "Juan", "Perez", doc, "12345678", "abc", "abc", ***resultadoCrearDocumentoIncorrecto***, 1, 0, 0, 1, **null**, **true**, 0 }, //prueba documento incorrecto

/\* 4 \*/**new** Object[] { "Juan", "Perez", doc, "12345678", "abc", "abc", ***resultadoCrearYaExiste***, 1, 0, 0, 1, **null**, **true**, 0 }, //prueba ya existe vendedor

/\* 5 \*/**new** Object[] { "Juan", "Perez", doc, "12345678", "abc", "abc", **new** ResultadoCrearVendedor(ErrorCrearVendedor.***Formato\_Nombre\_Incorrecto***, ErrorCrearVendedor.***Formato\_Apellido\_Incorrecto***), 1, 0, 0, 1, **null**, **true**, 0 }, //prueba nombre y apellido incorrectos

/\* 6 \*/**new** Object[] { "", "Perez", doc, "12345678", "abc", "abc", **null**, 1, 0, 0, 0, **null**, **true**, 0 }, //prueba nombre vacio

/\* 7 \*/**new** Object[] { "Juan", "Perez", doc, "12345678", "abc", "abc", ***resultadoCorrecto***, 0, 0, 0, 1, **new** EntidadExistenteConEstadoBajaException(), **true**, 1 }, //prueba Vendedor Existente y acepta

/\* 8 \*/**new** Object[] { "Juan", "Perez", doc, "12345678", "abc", "abc", ***resultadoCorrecto***, 0, 0, 0, 1, **new** EntidadExistenteConEstadoBajaException(), **false**, 0 }, //prueba Vendedor Existente y cancela

/\* 9 \*/**new** Object[] { "Juan", "Perez", doc, "12345678", "abc", "abc", ***resultadoCorrecto***, 0, 1, 0, 1, **new** SaveUpdateException(**new** Throwable()), **false**, 0 }, //prueba PersistenciaException

**public** **class** AltaVendedorControllerTest {

/\*\*

\* **@param** nombre

\* nombre del vendedor a crear

\* **@param** apellido

\* apellido del vendedor a crear

\* **@param** tipoDocumento

\* tipoDocumento del vendedor a crear

\* **@param** numeroDocumento

\* numeroDocumento del vendedor a crear

\* **@param** contraseña

\* contraseña ingresada

\* **@param** contraseña2

\* segunda contraseña ingresada

\* **@param** resultadoCrearVendedorEsperado

\* apellido del vendedor a crear

\* **@param** llamaAPresentadorVentanasPresentarError

\* indica si se debe presentar una ventana de error

\* **@param** llamaAPresentadorVentanasPresentarExcepcion

\* indica si se debe presentar una ventana de excepción

\* **@param** llamaAPresentadorVentanasPresentarExcepcionInesperada

\* indica si se debe presentar una ventana de excepción inesperada

\* **@param** llamaACrearVendedor

\* indica si se debe llamar al método crear vendedor de la lógica

\* **@param** excepcion

\* excepción devuelta por la lógica

\* **@param** aceptarVentanaConfirmacion

\* indica si el usuario acepta la ventana de confirmación

\* **@param** llamaACambiarScene

\* indica si debe llamar al cambio de pantalla

\* **@throws** Throwable

\*/

@Test

@Parameters

**public** **void** testCrearVendedor(String nombre, String apellido, TipoDocumento tipoDocumento, String numeroDocumento, String contraseña, String contraseña2, ResultadoCrearVendedor resultadoCrearVendedorEsperado, Integer llamaAPresentadorVentanasPresentarError, Integer llamaAPresentadorVentanasPresentarExcepcion, Integer llamaAPresentadorVentanasPresentarExcepcionInesperada, Integer llamaACrearVendedor, Exception excepcion, Boolean aceptarVentanaConfirmacion, Integer llamaACambiarScene) **throws** Throwable {

//Se crean los mocks necesarios

CoordinadorJavaFX coordinadorMock = *mock*(CoordinadorJavaFX.**class**);

PresentadorVentanas presentadorVentanasMock = *mock*(PresentadorVentanas.**class**);

VentanaError ventanaErrorMock = *mock*(VentanaError.**class**);

VentanaErrorExcepcion ventanaErrorExcepcionMock = *mock*(VentanaErrorExcepcion.**class**);

VentanaErrorExcepcionInesperada ventanaErrorExcepcionInesperadaMock = *mock*(VentanaErrorExcepcionInesperada.**class**);

VentanaConfirmacion ventanaConfirmacionMock = *mock*(VentanaConfirmacion.**class**);

ScenographyChanger scenographyChangerMock = *mock*(ScenographyChanger.**class**);

ModificarVendedorController modificarVendedorControllerMock = *mock*(ModificarVendedorController.**class**);

//Se setea lo que deben devolver los mocks cuando son invocados por la clase a probar

*when*(presentadorVentanasMock.presentarError(*any*(), *any*(), *any*())).thenReturn(ventanaErrorMock);

*when*(presentadorVentanasMock.presentarExcepcion(*any*(), *any*())).thenReturn(ventanaErrorExcepcionMock);

*when*(presentadorVentanasMock.presentarExcepcionInesperada(*any*(), *any*())).thenReturn(ventanaErrorExcepcionInesperadaMock);

*when*(presentadorVentanasMock.presentarConfirmacion(*any*(), *any*(), *any*())).thenReturn(ventanaConfirmacionMock);

*when*(ventanaConfirmacionMock.acepta()).thenReturn(aceptarVentanaConfirmacion);

*when*(scenographyChangerMock.cambiarScenography(*any*(String.**class**), *any*())).thenReturn(modificarVendedorControllerMock);

*doNothing*().when(presentadorVentanasMock).presentarToast(*any*(), *any*());

Vendedor vendedor = **new** Vendedor()

.setNombre(nombre)

.setApellido(apellido)

.setTipoDocumento(tipoDocumento)

.setNumeroDocumento(numeroDocumento)

.setPassword(contraseña);

*when*(coordinadorMock.crearVendedor(vendedor)).thenReturn(resultadoCrearVendedorEsperado);

**if**(excepcion != **null**){

*when*(coordinadorMock.crearVendedor(vendedor)).thenThrow(excepcion);

}

ArrayList<TipoDocumento> tipos = **new** ArrayList<>();

tipos.add(tipoDocumento);

*when*(coordinadorMock.obtenerTiposDeDocumento()).thenReturn(tipos);

//Controlador a probar, se sobreescriben algunos métodos para setear los mocks y setear los datos que ingresaría el usuario en la vista

AltaVendedorController altaVendedorController = **new** AltaVendedorController() {

@Override

**public** **void** inicializar(URL location, ResourceBundle resources) {

**this**.coordinador = coordinadorMock;

**this**.presentador = **new** PresentadorVentanas();

**this**.presentador = presentadorVentanasMock;

setScenographyChanger(scenographyChangerMock);

**super**.inicializar(location, resources);

}

@Override

**public** **void** acceptAction() {

**this**.coordinador = coordinadorMock;

**this**.textFieldNombre.setText(nombre);

**this**.textFieldApellido.setText(apellido);

**this**.comboBoxTipoDocumento.getSelectionModel().select(tipoDocumento);

**this**.textFieldNumeroDocumento.setText(numeroDocumento);

**this**.passwordFieldContraseña.setText(contraseña);

**this**.passwordFieldRepiteContraseña.setText(contraseña);

**super**.acceptAction();

};

@Override

**protected** **void** setTitulo(String titulo) {

}

};

//Los controladores de las vistas deben correrse en un thread de JavaFX

ControladorTest corredorTestEnJavaFXThread = **new** ControladorTest(AltaVendedorController.***URLVista***, altaVendedorController);

Statement test = **new** Statement() {

@Override

**public** **void** evaluate() **throws** Throwable {

//Método a probar

altaVendedorController.acceptAction();

//Se hacen las verificaciones pertinentes para comprobar que el controlador se comporte adecuadamente

Mockito.*verify*(coordinadorMock).obtenerTiposDeDocumento();

Mockito.*verify*(coordinadorMock, *times*(llamaACrearVendedor)).crearVendedor(*any*());

Mockito.*verify*(presentadorVentanasMock, *times*(llamaAPresentadorVentanasPresentarError)).presentarError(*eq*("Revise sus campos"), *any*(), *any*());

Mockito.*verify*(presentadorVentanasMock, *times*(llamaAPresentadorVentanasPresentarExcepcion)).presentarExcepcion(*eq*(excepcion), *any*());

Mockito.*verify*(presentadorVentanasMock, *times*(llamaAPresentadorVentanasPresentarExcepcionInesperada)).presentarExcepcionInesperada(*eq*(excepcion), *any*());

Mockito.*verify*(scenographyChangerMock, *times*(llamaACambiarScene)).cambiarScenography(ModificarVendedorController.***URLVista***, **false**);

}

};

//Se corre el test en el hilo de JavaFX

corredorTestEnJavaFXThread.apply(test, **null**).evaluate();

}

Código del archivo ModificarVendedorController.java

/\*\*

\* Acción que se ejecuta al apretar el botón aceptar.

\*

\* Valida que se hayan insertado datos, los carga al vendedor y deriva la operación a capa lógica.

\* Si la capa lógica retorna errores, éstos se muestran al usuario.

\*/

**public** **void** acceptAction() {

//Se toman los datos de la vista

String nombre = textFieldNombre.getText().trim();

String apellido = textFieldApellido.getText().trim();

String numeroDocumento = textFieldNumeroDocumento.getText().trim();

String passwordAntigua = passwordFieldContraseñaAntigua.getText();

String password1 = passwordFieldContraseñaNueva.getText();

String password2 = passwordFieldRepiteContraseña.getText();

TipoDocumento tipoDoc = comboBoxTipoDocumento.getValue();

StringBuilder error = **new** StringBuilder("");

//Se validan los campos

**if**(nombre.isEmpty()){

error.append("Inserte un nombre").append("\r\n");

}

**if**(apellido.isEmpty()){

error.append("Inserte un apellido").append("\r\n");

}

**if**(tipoDoc == **null**){

error.append("Elija un tipo de documento").append("\r\n");

}

**if**(numeroDocumento.isEmpty()){

error.append("Inserte un numero de documento").append("\r\n");

}

//Si el usuario quiere cambiar la contraseña, se validan también su contraseña antigua y nueva

**if**(checkBoxCambiarContraseña.isSelected()){

**if**(passwordAntigua.isEmpty()){

error.append("Ingrese su antigua contraseña\r\n");

}

**if**(!encriptador.encriptar(passwordAntigua.toCharArray(), vendedor.getSalt()).equals(vendedor.getPassword()) && !passwordAntigua.isEmpty()){

error.append("Su contraseña antigua es incorrecta\r\n");

}

**if**(password1.isEmpty() && password2.isEmpty()){

error.append("Ingrese su nueva contraseña").append("\r\n");

}

**if**(!password1.equals(password2)){

error.append("Sus nuevas contraseñas no coinciden. Ingréselas nuevamente").append("\r\n");

}

}

//Si hay errores se muestra una ventana con un mensaje explicativo de ellos

**if**(!error.toString().isEmpty()){

presentador.presentarError("Revise sus campos", error.toString(), stage);

}

**else**{

//Si no hay errores, se crea un vendedor para luego pasarselo a la capa lógica

vendedor.setNombre(nombre)

.setApellido(apellido)

.setNumeroDocumento(numeroDocumento)

.setTipoDocumento(tipoDoc);

**if**(checkBoxCambiarContraseña.isSelected()){

vendedor.setPassword(encriptador.encriptar(passwordFieldContraseñaNueva.getText().toCharArray(), vendedor.getSalt()));

}

ResultadoModificarVendedor resultadoModificarVendedor = **null**;

**try**{

//Se llama a la capa lógica para que cree el vendedor

resultadoModificarVendedor = coordinador.modificarVendedor(vendedor);

List<ErrorModificarVendedor> listaErrores = resultadoModificarVendedor.getErrores();

//Se convierten los errores devueltos a un mensaje a mostrar al usuario

**if**(listaErrores.contains(ErrorModificarVendedor.***Formato\_Nombre\_Incorrecto***)){

error.append("Nombre Incorrecto").append("\r\n");

}

**if**(listaErrores.contains(ErrorModificarVendedor.***Formato\_Apellido\_Incorrecto***)){

error.append("Apellido Incorrecto").append("\r\n");

}

**if**(listaErrores.contains(ErrorModificarVendedor.***Formato\_Documento\_Incorrecto***)){

error.append("Documento Incorrecto").append("\r\n");

}

**if**(listaErrores.contains(ErrorModificarVendedor.***Otro\_Vendedor\_Posee\_Mismo\_Documento\_Y\_Tipo***)){

error.append("Ya existe otro vendedor registrado con ese documento").append("\r\n");

}

//Si hay errores se muestran al usuario, si no, se presenta una notificación de éxito

**if**(!error.toString().isEmpty()){

presentador.presentarError("Revise sus campos", error.toString(), stage);

}

**else**{

/\*

\* primero sale, y después presenta el Toast

\* para saber en que posición colocarlo según el tamaño de la ventana padre (puede ser Login o administrar vendedor)

\*/

salir();

**if**(esAltaNuevamente){

presentador.presentarToast("Se ha dado de alta el vendedor con éxito", stage);

}

**else**{

presentador.presentarToast("Se ha modificado el vendedor con éxito", stage);

}

}

} **catch**(PersistenciaException e){

presentador.presentarExcepcion(e, stage);

} **catch**(Exception e){

presentador.presentarExcepcionInesperada(e, stage);

}

}

}

Prueba de unidad del archivo ModificarVendedorControllerTest.java

//Casos de prueba

//nombre, apellido, tipoDocumento, numeroDocumento, resultadoEncriptar, contraseñaAntigua, contraseñaNueva, contraseñaNueva2, resultadoModificarVendedorEsperado, llamaAPresentadorVentanasPresentarError, llamaAPresentadorVentanasPresentarExcepcion, llamaAPresentadorVentanasPresentarExcepcionInesperada, llamaAModificarVendedor, excepcion, llamaACambiarScene, checkBoxSeleccionado

/\* 0 \*/ **new** Object[] { "Juan", "Perez", doc, "12345678", "abc", "abc", "abc", "abc", ***resultadoCorrecto***, 0, 0, 0, 1, **null**, 0, **false** }, //prueba correcta

/\* 1 \*/ **new** Object[] { "Juan", "Perez", doc, "12345678", "ac", "abc", "abc", "abc", ***resultadoCorrecto***, 1, 0, 0, 0, **null**, 0, **true** }, //prueba contraseña antigua incorrecta

/\* 2 \*/ **new** Object[] { "Juan", "Perez", doc, "12345678", "abc", "abc", "abc", "abdc", ***resultadoCorrecto***, 1, 0, 0, 0, **null**, 0, **true** }, //prueba contraseñas nuevas no coinciden

/\* 3 \*/ **new** Object[] { "Juan", "Perez", doc, "12345678", "abc", "abc", "abc", "abc", ***resultadoModificarNombreIncorrecto***, 1, 0, 0, 1, **null**, 0, **false** }, //prueba nombre incorrecto

/\* 4 \*/ **new** Object[] { "Juan", "Perez", doc, "12345678", "abc", "abc", "abc", "abc", ***resultadoModificarApellidoIncorrecto***, 1, 0, 0, 1, **null**, 0, **false** }, //prueba apellido incorrecto

/\* 5 \*/ **new** Object[] { "Juan", "Perez", doc, "12345678", "abc", "abc", "abc", "abc", ***resultadoModificarDocumentoIncorrecto***, 1, 0, 0, 1, **null**, 0, **false** }, //prueba documento incorrecto

/\* 6 \*/ **new** Object[] { "Juan", "Perez", doc, "12345678", "abc", "abc", "abc", "abc", ***resultadoCrearYaExiste***, 1, 0, 0, 1, **null**, 0, **false** }, //prueba ya existe vendedor

/\* 7 \*/ **new** Object[] { "Juan", "Perez", doc, "12345678", "abc", "abc", "abc", "abc", **new** ResultadoModificarVendedor(ErrorModificarVendedor.***Formato\_Nombre\_Incorrecto***, ErrorModificarVendedor.***Formato\_Apellido\_Incorrecto***), 1, 0, 0, 1, **null**, 0, **false** }, //prueba nombre y apellido incorrectos

/\* 8 \*/ **new** Object[] { "", "Perez", doc, "12345678", "abc", "abc", "abc", "abc", **null**, 1, 0, 0, 0, **null**, 0, **false** }, //prueba nombre vacio

/\* 9 \*/ **new** Object[] { "Juan", "Perez", doc, "12345678", "abc", "abc", "abc", "abc", ***resultadoCorrecto***, 0, 1, 0, 1, **new** SaveUpdateException(**new** Throwable()), 0, **false** }, //prueba PersistenciaException

/\*\*

\* **@param** nombre

\* nombre del vendedor a modificar

\* **@param** apellido

\* apellido del vendedor a modificar

\* **@param** tipoDocumento

\* tipo de documenteo del vendedor a modificar

\* **@param** numeroDocumento

\* número de documento del vendedor a modificar

\* **@param** resultadoEncriptar

\* resultado devuelto por el mock encriptadorPassword

\* **@param** contraseñaAntigua

\* contraseña antigua del vendedor

\* **@param** contraseñaNueva

\* contraseña nueva del vendedor

\* **@param** contraseñaNueva2

\* campo repetir contraseña del vendedor

\* **@param** resultadoModificarVendedorEsperado

\* resultado devuelto por la lógica

\* **@param** llamaAPresentadorVentanasPresentarError

\* indica si se debe llamar a presentar una ventana de error

\* **@param** llamaAPresentadorVentanasPresentarExcepcion

\* indica si se debe llamar a presentar una ventana de excepción

\* **@param** llamaAPresentadorVentanasPresentarExcepcionInesperada

\* indica si se debe llamar a presentar una ventana de excepción inesperada

\* **@param** llamaAModificarVendedor

\* indica si se debe llamar a modificar un vendedor en la capa lógica

\* **@param** excepcion

\* excepción devuelta por la capa lógica

\* **@param** llamaACambiarScene

\* indica si se debe llamar a cambiar de pantalla

\* **@param** checkBoxSeleccionado

\* checkBox de cambiar contraseña seleccionado o no seleccionado

\* **@throws** Throwable

\*/

@Test

@Parameters

**public** **void** testModificarVendedor(String nombre, String apellido, TipoDocumento tipoDocumento, String numeroDocumento, String resultadoEncriptar, String contraseñaAntigua, String contraseñaNueva, String contraseñaNueva2, ResultadoModificarVendedor resultadoModificarVendedorEsperado, Integer llamaAPresentadorVentanasPresentarError, Integer llamaAPresentadorVentanasPresentarExcepcion, Integer llamaAPresentadorVentanasPresentarExcepcionInesperada, Integer llamaAModificarVendedor, Exception excepcion, Integer llamaACambiarScene, Boolean checkBoxSeleccionado) **throws** Throwable {

//Se crean los mocks necesarios

CoordinadorJavaFX coordinadorMock = *mock*(CoordinadorJavaFX.**class**);

PresentadorVentanas presentadorVentanasMock = *mock*(PresentadorVentanas.**class**);

VentanaError ventanaErrorMock = *mock*(VentanaError.**class**);

VentanaErrorExcepcion ventanaErrorExcepcionMock = *mock*(VentanaErrorExcepcion.**class**);

VentanaErrorExcepcionInesperada ventanaErrorExcepcionInesperadaMock = *mock*(VentanaErrorExcepcionInesperada.**class**);

EncriptadorPassword encriptadorPasswordMock = *mock*(EncriptadorPassword.**class**);

//Se setea lo que deben devolver los mocks cuando son invocados por la clase a probar

*when*(presentadorVentanasMock.presentarError(*any*(), *any*(), *any*())).thenReturn(ventanaErrorMock);

*when*(presentadorVentanasMock.presentarExcepcion(*any*(), *any*())).thenReturn(ventanaErrorExcepcionMock);

*when*(presentadorVentanasMock.presentarExcepcionInesperada(*any*(), *any*())).thenReturn(ventanaErrorExcepcionInesperadaMock);

*doNothing*().when(presentadorVentanasMock).presentarToast(*any*(), *any*());

*when*(encriptadorPasswordMock.encriptar(*eq*(contraseñaAntigua.toCharArray()), *any*())).thenReturn(resultadoEncriptar);

Vendedor vendedor = **new** Vendedor()

.setNombre(nombre)

.setApellido(apellido)

.setTipoDocumento(tipoDocumento)

.setNumeroDocumento(numeroDocumento)

.setPassword(contraseñaAntigua);

*when*(coordinadorMock.modificarVendedor(vendedor)).thenReturn(resultadoModificarVendedorEsperado);

**if**(excepcion != **null**){

*when*(coordinadorMock.modificarVendedor(vendedor)).thenThrow(excepcion);

}

ArrayList<TipoDocumento> tipos = **new** ArrayList<>();

tipos.add(tipoDocumento);

*when*(coordinadorMock.obtenerTiposDeDocumento()).thenReturn(tipos);

//Controlador a probar, se sobreescriben algunos métodos para setear los mocks y setear los datos que ingresaría el usuario en la vista

ModificarVendedorController modificarVendedorController = **new** ModificarVendedorController() {

@Override

**public** **void** inicializar(URL location, ResourceBundle resources) {

**this**.coordinador = coordinadorMock;

**this**.presentador = presentadorVentanasMock;

**this**.encriptador = encriptadorPasswordMock;

**this**.setVendedor(vendedor);

**super**.inicializar(location, resources);

}

@Override

**public** **void** acceptAction() {

**this**.textFieldNombre.setText(nombre);

**this**.textFieldApellido.setText(apellido);

**this**.comboBoxTipoDocumento.getSelectionModel().select(tipoDocumento);

**this**.textFieldNumeroDocumento.setText(numeroDocumento);

**this**.passwordFieldContraseñaAntigua.setText(contraseñaAntigua);

**this**.passwordFieldContraseñaNueva.setText(contraseñaNueva);

**this**.passwordFieldRepiteContraseña.setText(contraseñaNueva2);

**this**.checkBoxCambiarContraseña.setSelected(checkBoxSeleccionado);

**super**.acceptAction();

};

@Override

**protected** **void** setTitulo(String titulo) {

}

};

//Los controladores de las vistas deben correrse en un thread de JavaFX

ControladorTest corredorTestEnJavaFXThread = **new** ControladorTest(ModificarVendedorController.***URLVista***, modificarVendedorController);

Statement test = **new** Statement() {

@Override

**public** **void** evaluate() **throws** Throwable {

//Método a probar

modificarVendedorController.acceptAction();

//Se hacen las verificaciones pertinentes para comprobar que el controlador se comporte adecuadamente

Mockito.*verify*(coordinadorMock).obtenerTiposDeDocumento();

Mockito.*verify*(coordinadorMock, *times*(llamaAModificarVendedor)).modificarVendedor(*any*());

Mockito.*verify*(presentadorVentanasMock, *times*(llamaAPresentadorVentanasPresentarError)).presentarError(*eq*("Revise sus campos"), *any*(), *any*());

Mockito.*verify*(presentadorVentanasMock, *times*(llamaAPresentadorVentanasPresentarExcepcion)).presentarExcepcion(*eq*(excepcion), *any*());

Mockito.*verify*(presentadorVentanasMock, *times*(llamaAPresentadorVentanasPresentarExcepcionInesperada)).presentarExcepcionInesperada(*eq*(excepcion), *any*());

}

};

//Se corre el test en el hilo de JavaFX

corredorTestEnJavaFXThread.apply(test, **null**).evaluate();

}

#### Taskcard 7 Entidad Propietario

Código del archivo Propietario.java

@NamedQueries(value = { @NamedQuery(name = "obtenerPropietarios", query = "SELECT p FROM Propietario p WHERE p.estado.estado = 'ALTA'"),

@NamedQuery(name = "obtenerPropietario", query = "SELECT p FROM Propietario p WHERE p.numeroDocumento = :documento AND p.tipoDocumento.tipo = :tipoDocumento") })

@Entity

@Table(name = "propietario", uniqueConstraints = @UniqueConstraint(name = "propietario\_numerodocumento\_idtipo\_uk", columnNames = { "numerodocumento", "idtipo" }))

/\*\*

\* Entidad que modela a un propietario

\*/

**public** **class** Propietario {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** Integer id; //ID

@Column(name = "nombre", length = 30)

**private** String nombre;

@Column(name = "apellido", length = 30)

**private** String apellido;

@Column(name = "numerodocumento", length = 30)

**private** String numeroDocumento;

@Column(name = "telefono", length = 30)

**private** String telefono;

@Column(name = "email", length = 30)

**private** String email;

//Reclaciones

@ManyToOne(fetch = FetchType.***LAZY***)

@JoinColumn(name = "idtipo", referencedColumnName = "id", foreignKey = @ForeignKey(name = "propietario\_idtipo\_fk"))

**private** TipoDocumento tipoDocumento;

@ManyToOne(fetch = FetchType.***LAZY***, cascade = CascadeType.***ALL***)

@JoinColumn(name = "iddireccion", referencedColumnName = "id", foreignKey = @ForeignKey(name = "propietario\_iddireccion\_fk"))

**private** Direccion direccion;

//Opcionales

@OneToMany(fetch = FetchType.***LAZY***, cascade = CascadeType.***ALL***, orphanRemoval = **true**, mappedBy = "propietario")

**private** Set<Inmueble> inmuebles;

@ManyToOne(fetch = FetchType.***EAGER***)

@JoinColumn(name = "idestado", referencedColumnName = "id", foreignKey = @ForeignKey(name = "propietario\_idestado\_fk"), nullable = **false**)

**private** Estado estado;

#### Taskcard 9 Lógica alta, modificar y baja propietario

Código del archivo GestorPropietario.java

/\*\*

\* Se encarga de validar los datos de un propietario a crear y, en caso de que no haya errores,

\* delegar el guardado del objeto a la capa de acceso a datos.

\*

\* **@param** propietario

\* propietario a crear

\* **@return** un resultado informando errores correspondientes en caso de que los haya

\*

\* **@throws** PersistenciaException

\* se lanza esta excepción al ocurrir un error interactuando con la capa de acceso a datos

\* **@throws** GestionException

\* se lanza una excepción EntidadExistenteConEstadoBaja cuando se encuentra que ya existe un propietario con la misma identificación pero tiene estado BAJA

\*/

**public** ResultadoCrearPropietario crearPropietario(Propietario propietario) **throws** PersistenciaException, GestionException {

ArrayList<ErrorCrearPropietario> errores = **new** ArrayList<>();

validarDatosCrearPropietario(propietario, errores);

Propietario propietarioAuxiliar = persistidorPropietario.obtenerPropietario(**new** FiltroPropietario(propietario.getTipoDocumento().getTipo(), propietario.getNumeroDocumento()));

**if**(**null** != propietarioAuxiliar){

**if**(propietarioAuxiliar.getEstado().getEstado().equals(EstadoStr.***ALTA***)){

errores.add(ErrorCrearPropietario.***Ya\_Existe\_Propietario***);

}

**else**{

**throw** **new** EntidadExistenteConEstadoBajaException();

}

}

**if**(errores.isEmpty()){

ArrayList<Estado> estados = gestorDatos.obtenerEstados();

**for**(Estado e: estados){

**if**(e.getEstado().equals(EstadoStr.***ALTA***)){

propietario.setEstado(e);

**break**;

}

}

persistidorPropietario.guardarPropietario(propietario);

}

**return** **new** ResultadoCrearPropietario(errores.toArray(**new** ErrorCrearPropietario[0]));

}

/\*\*

\* Método auxiliar que encapsula la validación de los formatos de los atributos del propietario a crear

\*

\* **@param** propietario

\* propietario a validar

\* **@param** errores

\* lista de errores encontrados

\*/

**private** **void** validarDatosCrearPropietario(Propietario propietario, ArrayList<ErrorCrearPropietario> errores) {

**if**(!validador.validarNombre(propietario.getNombre())){

errores.add(ErrorCrearPropietario.***Formato\_Nombre\_Incorrecto***);

}

**if**(!validador.validarApellido(propietario.getApellido())){

errores.add(ErrorCrearPropietario.***Formato\_Apellido\_Incorrecto***);

}

**if**(!validador.validarDocumento(propietario.getTipoDocumento(), propietario.getNumeroDocumento())){

errores.add(ErrorCrearPropietario.***Formato\_Documento\_Incorrecto***);

}

**if**(!validador.validarTelefono(propietario.getTelefono())){

errores.add(ErrorCrearPropietario.***Formato\_Telefono\_Incorrecto***);

}

**if**(propietario.getEmail() != **null** && !validador.validarEmail(propietario.getEmail())){

errores.add(ErrorCrearPropietario.***Formato\_Email\_Incorrecto***);

}

**if**(!validador.validarDireccion(propietario.getDireccion())){

errores.add(ErrorCrearPropietario.***Formato\_Direccion\_Incorrecto***);

}

}

/\*\*

\* Se encarga de validar los datos de un propietario a modificar y, en caso de que no haya errores,

\* delegar el guardado del objeto a la capa de acceso a datos.

\*

\* **@param** propietario

\* propietario a modificar

\* **@return** un resultado informando errores correspondientes en caso de que los haya

\*

\* **@throws** PersistenciaException

\* se lanza esta excepción al ocurrir un error interactuando con la capa de acceso a datos

\*/

**public** ResultadoModificarPropietario modificarPropietario(Propietario propietario) **throws** PersistenciaException {

ArrayList<ErrorModificarPropietario> errores = **new** ArrayList<>();

validarDatosModificarPropietario(propietario, errores);

Propietario propietarioAuxiliar;

propietarioAuxiliar = persistidorPropietario.obtenerPropietario(**new** FiltroPropietario(propietario.getTipoDocumento().getTipo(), propietario.getNumeroDocumento()));

**if**(**null** != propietarioAuxiliar && !propietario.equals(propietarioAuxiliar)){

errores.add(ErrorModificarPropietario.***Ya\_Existe\_Propietario***);

}

**if**(errores.isEmpty()){

ArrayList<Estado> estados = gestorDatos.obtenerEstados();

**for**(Estado e: estados){

**if**(e.getEstado().equals(EstadoStr.***ALTA***)){

propietario.setEstado(e);

**break**;

}

}

persistidorPropietario.modificarPropietario(propietario);

}

**return** **new** ResultadoModificarPropietario(errores.toArray(**new** ErrorModificarPropietario[0]));

}

/\*\*

\* Método auxiliar que encapsula la validación de los formatos de los atributos del propietario a modificar

\*

\* **@param** propietario

\* propietario a validar

\* **@param** errores

\* lista de errores encontrados

\*/

**private** **void** validarDatosModificarPropietario(Propietario propietario, ArrayList<ErrorModificarPropietario> errores) {

**if**(!validador.validarNombre(propietario.getNombre())){

errores.add(ErrorModificarPropietario.***Formato\_Nombre\_Incorrecto***);

}

**if**(!validador.validarApellido(propietario.getApellido())){

errores.add(ErrorModificarPropietario.***Formato\_Apellido\_Incorrecto***);

}

**if**(!validador.validarDocumento(propietario.getTipoDocumento(), propietario.getNumeroDocumento())){

errores.add(ErrorModificarPropietario.***Formato\_Documento\_Incorrecto***);

}

**if**(!validador.validarTelefono(propietario.getTelefono())){

errores.add(ErrorModificarPropietario.***Formato\_Telefono\_Incorrecto***);

}

**if**(!validador.validarEmail(propietario.getEmail())){

errores.add(ErrorModificarPropietario.***Formato\_Email\_Incorrecto***);

}

**if**(!validador.validarDireccion(propietario.getDireccion())){

errores.add(ErrorModificarPropietario.***Formato\_Direccion\_Incorrecto***);

}

}

/\*\*

\* Se encarga de validar que exista el propietario a eliminar, se setea el estado en BAJA y,

\* en caso de que no haya errores, delegar el guardado del objeto a la capa de acceso a datos.

\*

\* **@param** propietario

\* propietario a eliminar

\* **@return** un resultado informando errores correspondientes en caso de que los haya

\*

\* **@throws** PersistenciaException

\* se lanza esta excepción al ocurrir un error interactuando con la capa de acceso a datos

\*/

**public** ResultadoEliminarPropietario eliminarPropietario(Propietario propietario) **throws** PersistenciaException {

ArrayList<Estado> estados = gestorDatos.obtenerEstados();

**for**(Estado e: estados){

**if**(e.getEstado().equals(EstadoStr.***BAJA***)){

propietario.setEstado(e);

**break**;

}

}

persistidorPropietario.modificarPropietario(propietario);

**return** **new** ResultadoEliminarPropietario();

}

/\*\*

\* Obtiene el listado de propietarios solicitándolo a la capa de acceso a datos

\*

\* **@return** el listado de propietarios solicitado

\*

\* **@throws** PersistenciaException

\* se lanza esta excepción al ocurrir un error interactuando con la capa de acceso a datos

\*/

**public** ArrayList<Propietario> obtenerPropietarios() **throws** PersistenciaException {

**return** persistidorPropietario.listarPropietarios();

}

/\*\*

\* Obtiene un propietario en base a su tipo y número de documento solicitándolo a la capa de acceso a datos

\*

\* **@param** filtro

\* filtro que contiene el tipo y número de documento del propietario buscado

\*

\* **@return** el propietario solicitado

\*

\* **@throws** PersistenciaException

\* se lanza esta excepción al ocurrir un error interactuando con la capa de acceso a datos

\*/

**public** Propietario obtenerPropietario(FiltroPropietario filtro) **throws** PersistenciaException {

Propietario propietarioAuxiliar;

propietarioAuxiliar = persistidorPropietario.obtenerPropietario(**new** FiltroPropietario(filtro.getTipoDocumento(), filtro.getDocumento()));

**return** propietarioAuxiliar;

}

Prueba de unidad del archivo GestorPropietario.java

//Casos de prueba

//resValNombre, resValApellido, resValDocumento, resValTelefono, resValEmail, resValDireccion, resObtenerPropietario, guardar, resultadoCrearPropietarioEsperado, excepcion, excepcionEsperada

/\* 0 \*/ **new** Object[] { **true**, **true**, **true**, **true**, **true**, **true**, **null**, 1, ***resultadoCrearCorrecto***, **null**, **null** }, //Test correcto

/\* 1 \*/ **new** Object[] { **false**, **true**, **true**, **true**, **true**, **true**, **null**, 0, ***resultadoCrearNombreIncorrecto***, **null**, **null** }, //Nombre Incorrecto

/\* 2 \*/ **new** Object[] { **true**, **false**, **true**, **true**, **true**, **true**, **null**, 0, ***resultadoCrearApellidoIncorrecto***, **null**, **null** }, //Apellido Incorrecto

/\* 3 \*/ **new** Object[] { **true**, **true**, **false**, **true**, **true**, **true**, **null**, 0, ***resultadoCrearDocumentoIncorrecto***, **null**, **null** }, //Documento Incorrecto

/\* 4 \*/ **new** Object[] { **true**, **true**, **true**, **false**, **true**, **true**, **null**, 0, ***resultadoCrearTelefonoIncorrecto***, **null**, **null** }, //Teléfono Incorrecto

/\* 5 \*/ **new** Object[] { **true**, **true**, **true**, **true**, **false**, **true**, **null**, 0, ***resultadoCrearEmailIncorrecto***, **null**, **null** }, //Email Incorrecto

/\* 6 \*/ **new** Object[] { **true**, **true**, **true**, **true**, **true**, **false**, **null**, 0, ***resultadoCrearDireccionIncorrecto***, **null**, **null** }, //Dirección incorrecta

/\* 7 \*/ **new** Object[] { **false**, **false**, **true**, **true**, **true**, **true**, **null**, 0, **new** ResultadoCrearPropietario(ErrorCrearPropietario.***Formato\_Nombre\_Incorrecto***, ErrorCrearPropietario.***Formato\_Apellido\_Incorrecto***), **null**, **null** }, //Nombre y apellido incorrectos

/\* 8 \*/ **new** Object[] { **true**, **true**, **true**, **true**, **true**, **true**, *propietario*, 0, ***resultadoCrearYaExiste***, **null**, **null** }, //Ya existe un propietario con el mismo documento

/\* 9 \*/ **new** Object[] { **true**, **true**, **true**, **true**, **true**, **true**, **null**, 1, **null**, **new** SaveUpdateException(**new** Exception()), **new** SaveUpdateException(**new** Exception()) }, //La base de datos tira una excepción

/\* 10 \*/ **new** Object[] { **true**, **true**, **true**, **true**, **true**, **true**, propietario2, 0, **null**, **null**, **new** EntidadExistenteConEstadoBajaException() } //El propietario ya existe pero con estado BAJA

/\*\*

\* Test para el método crear propietario

\*

\* **@param** resValNombre

\* resultado devuelto por el mock validador de formato al validar nombre

\* **@param** resValApellido

\* resultado devuelto por el mock validador de formato al validar apellido

\* **@param** resValDocumento

\* resultado devuelto por el mock validador de formato al validar documento

\* **@param** resValTelefono

\* resultado devuelto por el mock validador de formato al validar teléfono

\* **@param** resValEmail

\* resultado devuelto por el mock validador de formato al validar email

\* **@param** resValDireccion

\* resultado devuelto por el mock validador de formato al validar dirección

\* **@param** resObtenerPropietario

\* resultado devuelto por la base de datos al obtener un propietario

\* **@param** guardar

\* indica si se debe mandar a guardar el propietario

\* **@param** resultadoCrearPropietarioEsperado

\* resultado esperado del test

\* **@param** excepcion

\* excepción devuelta por la base de datos

\* **@param** excepcionEsperada

\* excepción esperada que debería lanzar el gestor

\* **@throws** Exception

\*/

@Test

@Parameters

**public** **void** testCrearPropietario(Boolean resValNombre, Boolean resValApellido, Boolean resValDocumento, Boolean resValTelefono, Boolean resValEmail, Boolean resValDireccion, Propietario resObtenerPropietario, Integer guardar, ResultadoCrearPropietario resultadoCrearPropietarioEsperado, Throwable excepcion, Throwable excepcionEsperada) **throws** Exception {

//Inicialización de los mocks

PropietarioService propietarioServiceMock = *mock*(PropietarioService.**class**);

ValidadorFormato validadorFormatoMock = *mock*(ValidadorFormato.**class**);

GestorDatos gestorDatosMock = *mock*(GestorDatos.**class**);

//Clase anónima necesaria para inyectar dependencias

GestorPropietario gestorPropietario = **new** GestorPropietario() {

{

**this**.persistidorPropietario = propietarioServiceMock;

**this**.validador = validadorFormatoMock;

**this**.gestorDatos = gestorDatosMock;

}

};

ArrayList<Estado> estados = **new** ArrayList<>();

estados.add(**new** Estado(EstadoStr.***BAJA***));

estados.add(**new** Estado(EstadoStr.***ALTA***));

//Setear valores esperados a los mocks

*when*(validadorFormatoMock.validarNombre(*propietario*.getNombre())).thenReturn(resValNombre);

*when*(validadorFormatoMock.validarApellido(*propietario*.getApellido())).thenReturn(resValApellido);

*when*(validadorFormatoMock.validarDocumento(*propietario*.getTipoDocumento(), *propietario*.getNumeroDocumento())).thenReturn(resValDocumento);

*when*(validadorFormatoMock.validarTelefono(*propietario*.getTelefono())).thenReturn(resValTelefono);

*when*(validadorFormatoMock.validarEmail(*propietario*.getEmail())).thenReturn(resValEmail);

*when*(validadorFormatoMock.validarDireccion(*propietario*.getDireccion())).thenReturn(resValDireccion);

*when*(propietarioServiceMock.obtenerPropietario(*any*())).thenReturn(resObtenerPropietario);

*when*(gestorDatosMock.obtenerEstados()).thenReturn(estados);

//Setear la excepcion devuelta por la base de datos, si corresponde

**if**(excepcion != **null**){

*doThrow*(excepcion).when(propietarioServiceMock).guardarPropietario(*propietario*);

}

**else**{

*doNothing*().when(propietarioServiceMock).guardarPropietario(*propietario*);

}

//Llamar al método a testear y comprobar resultados obtenidos, que se llaman a los métodos deseados y con los parámetros correctos

**if**(excepcionEsperada == **null**){

*assertEquals*(resultadoCrearPropietarioEsperado.getErrores(), gestorPropietario.crearPropietario(*propietario*).getErrores());

}

**else**{

**try**{

gestorPropietario.crearPropietario(*propietario*);

Assert.*fail*("Debería haber fallado!");

} **catch**(Exception e){

*assertEquals*(excepcionEsperada.getClass(), e.getClass());

}

}

*verify*(validadorFormatoMock).validarNombre(*propietario*.getNombre());

*verify*(validadorFormatoMock).validarApellido(*propietario*.getApellido());

*verify*(validadorFormatoMock).validarDocumento(*propietario*.getTipoDocumento(), *propietario*.getNumeroDocumento());

*verify*(validadorFormatoMock).validarTelefono(*propietario*.getTelefono());

*verify*(validadorFormatoMock).validarEmail(*propietario*.getEmail());

*verify*(validadorFormatoMock).validarDireccion(*propietario*.getDireccion());

*verify*(gestorDatosMock, *times*(guardar)).obtenerEstados();

*verify*(propietarioServiceMock, *times*(guardar)).guardarPropietario(*propietario*);

}

//Casos de prueba

//resValNombre, resValApellido, resValDocumento, resValTelefono, resValEmail, resValDireccion, resObtenerPropietario, modificar, resultadoModificarPropietarioEsperado, excepcion, excepcionEsperada

/\* 0 \*/ **new** Object[] { **true**, **true**, **true**, **true**, **true**, **true**, **null**, 1, ***resultadoModificarCorrecto***, **null**, **null** }, //Test correcto (se modifica el documento)

/\* 1 \*/ **new** Object[] { **false**, **true**, **true**, **true**, **true**, **true**, **null**, 0, ***resultadoModificarNombreIncorrecto***, **null**, **null** }, //Nombre Incorrecto

/\* 2 \*/ **new** Object[] { **true**, **false**, **true**, **true**, **true**, **true**, **null**, 0, ***resultadoModificarApellidoIncorrecto***, **null**, **null** }, //Apellido Incorrecto

/\* 3 \*/ **new** Object[] { **true**, **true**, **false**, **true**, **true**, **true**, **null**, 0, ***resultadoModificarDocumentoIncorrecto***, **null**, **null** }, //Documento Incorrecto

/\* 4 \*/ **new** Object[] { **true**, **true**, **true**, **false**, **true**, **true**, **null**, 0, ***resultadoModificarTelefonoIncorrecto***, **null**, **null** }, //Teléfono Incorrecto

/\* 5 \*/ **new** Object[] { **true**, **true**, **true**, **true**, **false**, **true**, **null**, 0, ***resultadoModificarEmailIncorrecto***, **null**, **null** }, //Email Incorrecto

/\* 6 \*/ **new** Object[] { **true**, **true**, **true**, **true**, **true**, **false**, **null**, 0, ***resultadoModificarDireccionIncorrecto***, **null**, **null** }, //Dirección incorrecta

/\* 7 \*/ **new** Object[] { **false**, **false**, **true**, **true**, **true**, **true**, **null**, 0, **new** ResultadoModificarPropietario(ErrorModificarPropietario.***Formato\_Nombre\_Incorrecto***, ErrorModificarPropietario.***Formato\_Apellido\_Incorrecto***), **null**, **null** }, //Nombre y apellido incorrectos

/\* 8 \*/ **new** Object[] { **true**, **true**, **true**, **true**, **true**, **true**, propietario2, 0, ***resultadoModificarYaSePoseeMismoDocumento***, **null**, **null** }, //Ya existe un propietario con el mismo documento

/\* 9 \*/ **new** Object[] { **true**, **true**, **true**, **true**, **true**, **true**, *propietario*, 1, ***resultadoModificarCorrecto***, **null**, **null** }, //Test correcto, no se cambió el documento

/\* 10 \*/ **new** Object[] { **true**, **true**, **true**, **true**, **true**, **true**, **null**, 1, **null**, **new** SaveUpdateException(**new** Exception()), **new** SaveUpdateException(**new** Exception()) } //La base de datos tira una excepción

/\*\*

\* Test para el método modificar propietario

\*

\* **@param** resValNombre

\* resultado devuelto por el mock validador de formato al validar nombre

\* **@param** resValApellido

\* resultado devuelto por el mock validador de formato al validar apellido

\* **@param** resValDocumento

\* resultado devuelto por el mock validador de formato al validar documento

\* **@param** resValTelefono

\* resultado devuelto por el mock validador de formato al validar teléfono

\* **@param** resValEmail

\* resultado devuelto por el mock validador de formato al validar email

\* **@param** resValDireccion

\* resultado devuelto por el mock validador de formato al validar dirección

\* **@param** resObtenerPropietario

\* resultado devuelto por la base de datos al obtener un propietario

\* **@param** modificar

\* indica si se debe mandar a guardar el propietario modificado

\* **@param** resultadoModificarPropietarioEsperado

\* resultado esperado del test

\* **@param** excepcion

\* excepción devuelta por la base de datos

\* **@param** excepcionEsperada

\* excepción esperada que debería lanzar el gestor

\* **@throws** Exception

\*/

@Test

@Parameters

**public** **void** testModificarPropietario(Boolean resValNombre, Boolean resValApellido, Boolean resValDocumento, Boolean resValTelefono, Boolean resValEmail, Boolean resValDireccion, Propietario resObtenerPropietario, Integer modificar, ResultadoModificarPropietario resultadoModificarPropietarioEsperado, Throwable excepcion, Throwable excepcionEsperada) **throws** Exception {

//Inicialización de los mocks

PropietarioService propietarioServiceMock = *mock*(PropietarioService.**class**);

ValidadorFormato validadorFormatoMock = *mock*(ValidadorFormato.**class**);

GestorDatos gestorDatosMock = *mock*(GestorDatos.**class**);

//Clase anónima necesaria para inyectar dependencias

GestorPropietario gestorPropietario = **new** GestorPropietario() {

{

**this**.persistidorPropietario = propietarioServiceMock;

**this**.validador = validadorFormatoMock;

**this**.gestorDatos = gestorDatosMock;

}

};

ArrayList<Estado> estados = **new** ArrayList<>();

estados.add(**new** Estado(EstadoStr.***BAJA***));

estados.add(**new** Estado(EstadoStr.***ALTA***));

//Setear valores esperados a los mocks

*when*(validadorFormatoMock.validarNombre(*propietario*.getNombre())).thenReturn(resValNombre);

*when*(validadorFormatoMock.validarApellido(*propietario*.getApellido())).thenReturn(resValApellido);

*when*(validadorFormatoMock.validarDocumento(*propietario*.getTipoDocumento(), *propietario*.getNumeroDocumento())).thenReturn(resValDocumento);

*when*(validadorFormatoMock.validarTelefono(*propietario*.getTelefono())).thenReturn(resValTelefono);

*when*(validadorFormatoMock.validarEmail(*propietario*.getEmail())).thenReturn(resValEmail);

*when*(validadorFormatoMock.validarDireccion(*propietario*.getDireccion())).thenReturn(resValDireccion);

*when*(propietarioServiceMock.obtenerPropietario(*any*())).thenReturn(resObtenerPropietario);

*when*(gestorDatosMock.obtenerEstados()).thenReturn(estados);

//Setear la excepcion devuelta por la base de datos, si corresponde

**if**(excepcion != **null**){

*doThrow*(excepcion).when(propietarioServiceMock).modificarPropietario(*propietario*);

}

**else**{

*doNothing*().when(propietarioServiceMock).modificarPropietario(*propietario*);

}

//Llamar al método a testear y comprobar resultados obtenidos, que se llaman a los métodos deseados y con los parámetros correctos

**if**(excepcionEsperada == **null**){

*assertEquals*(resultadoModificarPropietarioEsperado.getErrores(), gestorPropietario.modificarPropietario(*propietario*).getErrores());

}

**else**{

**try**{

gestorPropietario.modificarPropietario(*propietario*);

Assert.*fail*("Debería haber fallado!");

} **catch**(Exception e){

*assertEquals*(excepcionEsperada.getClass(), e.getClass());

}

}

*verify*(validadorFormatoMock).validarNombre(*propietario*.getNombre());

*verify*(validadorFormatoMock).validarApellido(*propietario*.getApellido());

*verify*(validadorFormatoMock).validarDocumento(*propietario*.getTipoDocumento(), *propietario*.getNumeroDocumento());

*verify*(validadorFormatoMock).validarTelefono(*propietario*.getTelefono());

*verify*(validadorFormatoMock).validarEmail(*propietario*.getEmail());

*verify*(validadorFormatoMock).validarDireccion(*propietario*.getDireccion());

*verify*(propietarioServiceMock).obtenerPropietario(*any*());

*verify*(gestorDatosMock, *times*(modificar)).obtenerEstados();

*verify*(propietarioServiceMock, *times*(modificar)).modificarPropietario(*propietario*);

}

//Casos de prueba

//resObtenerPropietario, eliminar, resultadoEliminarPropietarioEsperado, excepcion, excepcionEsperada

/\* 0 \*/ **new** Object[] { *propietario*, 1, ***resultadoEliminarCorrecto***, **null**, **null** }, //Test correcto

/\* 1 \*/ **new** Object[] { *propietario*, 1, **null**, **new** SaveUpdateException(**new** Exception()), **new** SaveUpdateException(**new** Exception()) } //La base de datos tira una excepción

/\* @param resObtenerPropietario

\* resultado devuelto por la base de datos al obtener un propietario

\* @param eliminar

\* indica si se debe eliminar el propietario

\* @param resultadoEliminarPropietarioEsperado

\* resultado esperado del test

\* @param excepcion

\* excepción devuelta por la base de datos

\* @param excepcionEsperada

\* excepción esperada que debería lanzar el gestor

\* @throws Exception

\*/

@Test

@Parameters

**public** **void** testEliminarPropietario(Propietario resObtenerPropietario, Integer eliminar, ResultadoEliminarPropietario resultadoEliminarPropietarioEsperado, Throwable excepcion, Throwable excepcionEsperada) **throws** Exception {

//Inicialización de los mocks

PropietarioService propietarioServiceMock = *mock*(PropietarioService.**class**);

GestorDatos gestorDatosMock = *mock*(GestorDatos.**class**);

//Clase anónima necesaria para inyectar dependencias

GestorPropietario gestorPropietario = **new** GestorPropietario() {

{

**this**.persistidorPropietario = propietarioServiceMock;

**this**.gestorDatos = gestorDatosMock;

}

};

ArrayList<Estado> estados = **new** ArrayList<>();

estados.add(**new** Estado(EstadoStr.***ALTA***));

estados.add(**new** Estado(EstadoStr.***BAJA***));

//Setear valores esperados a los mocks

*when*(gestorDatosMock.obtenerEstados()).thenReturn(estados);

//Setear la excepcion devuelta por la base de datos, si corresponde

**if**(excepcion != **null**){

*doThrow*(excepcion).when(propietarioServiceMock).modificarPropietario(*propietario*);

}

**else**{

*doNothing*().when(propietarioServiceMock).modificarPropietario(*propietario*);

}

//Llamar al método a testear y comprobar resultados obtenidos, que se llaman a los métodos deseados y con los parámetros correctos

**if**(excepcionEsperada == **null**){

*assertEquals*(resultadoEliminarPropietarioEsperado.getErrores(), gestorPropietario.eliminarPropietario(*propietario*).getErrores());

}

**else**{

**try**{

gestorPropietario.eliminarPropietario(*propietario*);

Assert.*fail*("Debería haber fallado!");

} **catch**(Exception e){

*assertEquals*(excepcionEsperada.getClass(), e.getClass());

}

}

**if**(eliminar != 0){

*assertEquals*(EstadoStr.***BAJA***, *propietario*.getEstado().getEstado());

}

*verify*(gestorDatosMock, *times*(eliminar)).obtenerEstados();

*verify*(propietarioServiceMock, *times*(eliminar)).modificarPropietario(*propietario*);

}

#### Taskcard 10 Persistidor Propietario

Código del archivo PropietarioServiceImpl.java

/\*

\* Método para guardar en la base de datos un propietario

\*/

@Override

@Transactional(rollbackFor = PersistenciaException.**class**) //si falla hace rollback de la transacción

**public** **void** guardarPropietario(Propietario propietario) **throws** PersistenciaException {

Session session = getSessionFactory().getCurrentSession();

**try**{

session.save(propietario);

} **catch**(Exception e){

**throw** **new** SaveUpdateException(e);

}

}

/\*

\* Método para modificar en la base de datos un propietario

\*/

@Override

@Transactional(rollbackFor = PersistenciaException.**class**) //si falla hace rollback de la transacción

**public** **void** modificarPropietario(Propietario propietario) **throws** PersistenciaException {

Session session = getSessionFactory().getCurrentSession();

**try**{

session.update(propietario);

} **catch**(Exception e){

**throw** **new** SaveUpdateException(e);

}

}

/\*

\* Método para obtener de la base de datos un propietario

\*/

@Override

@Transactional(readOnly = **true**, rollbackFor = PersistenciaException.**class**) //si falla hace rollback de la transacción

**public** Propietario obtenerPropietario(FiltroPropietario filtro) **throws** PersistenciaException {

Propietario propietario = **null**;

Session session = getSessionFactory().getCurrentSession();

**try**{

//named query ubicada en entidad propietario

propietario = (Propietario) session.getNamedQuery("obtenerPropietario").setParameter("tipoDocumento", filtro.getTipoDocumento()).setParameter("documento", filtro.getDocumento()).uniqueResult();

} **catch**(NoResultException e){

**return** **null**;

} **catch**(NonUniqueResultException e){

**return** **null**;

} **catch**(Exception e){

**throw** **new** ConsultaException(e);

}

**return** propietario;

}

/\*

\* Método para obtener de la base de datos todos los propietarios con estado ALTA

\*/

@Override

@Transactional(readOnly = **true**, rollbackFor = PersistenciaException.**class**) //si falla hace rollback de la transacción

**public** ArrayList<Propietario> listarPropietarios() **throws** PersistenciaException {

ArrayList<Propietario> propietarios = **new** ArrayList<>();

Session session = getSessionFactory().getCurrentSession();

**try**{

//named query ubicada en entidad propietario

**for**(Object o: session.getNamedQuery("obtenerPropietarios").list()){

**if**(o **instanceof** Propietario){

propietarios.add((Propietario) o);

}

}

} **catch**(Exception e){

**throw** **new** ConsultaException(e);

}

**return** propietarios;

}

#### Taskcard 12 Clases de datos

Código del archivo Direccion.java

/\*

\* Entidad que modela una direccion.

\* Pertenece a la taskcard 12 de la iteración 1 y a la historia de usuario 3

\*/

**public** **class** Direccion {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** Integer id; //ID

@Column(name = "numero", length = 30, nullable = **false**)

**private** String numero;

@Column(name = "piso", length = 30)

**private** String piso;

@Column(name = "departamento", length = 30)

**private** String departamento;

@Column(name = "otros", length = 100)

**private** String otros;

//Relaciones

@ManyToOne(fetch = FetchType.***LAZY***, cascade = CascadeType.***ALL***)

@JoinColumn(name = "idcalle", referencedColumnName = "id", foreignKey = @ForeignKey(name = "direccion\_idcalle\_fk"))

**private** Calle calle;

@ManyToOne(fetch = FetchType.***LAZY***, cascade = CascadeType.***ALL***)

@JoinColumn(name = "idbarrio", referencedColumnName = "id", foreignKey = @ForeignKey(name = "direccion\_idbarrio\_fk"))

**private** Barrio barrio;

@ManyToOne(fetch = FetchType.***LAZY***, cascade = CascadeType.***ALL***)

@JoinColumn(name = "idlocalidad", referencedColumnName = "id", foreignKey = @ForeignKey(name = "direccion\_idlocalidad\_fk"))

**private** Localidad localidad;

Código del archivo Barrio.java

/\*\*

\* Entidad que modela un barrio

\* Pertenece a la taskcard 12 de la iteración 1 y a la historia de usuario 3

\*/

**public** **class** Barrio {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** Integer id; //ID

@Column(name = "nombre", length = 50, nullable = **false**)

**private** String nombre;

//Relaciones

@ManyToOne(fetch = FetchType.***LAZY***, cascade = CascadeType.***ALL***)

@JoinColumn(name = "idlocalidad", referencedColumnName = "id", foreignKey = @ForeignKey(name = "barrio\_idlocalidad\_fk"), nullable = **false**)

**private** Localidad localidad;

Código del archivo Calle.java

/\*

\* Entidad que modela una calle.

\* Pertenece a la taskcard 12 de la iteración 1 y a la historia de usuario 3

\*/

**public** **class** Calle {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** Integer id; //ID

@Column(name = "nombre", length = 50, nullable = **false**)

**private** String nombre;

//Relaciones

@ManyToOne(fetch = FetchType.***LAZY***, cascade = CascadeType.***ALL***)

@JoinColumn(name = "idlocalidad", referencedColumnName = "id", foreignKey = @ForeignKey(name = "calle\_idlocalidad\_fk"), nullable = **false**)

**private** Localidad localidad;

Código del archivo País.java

/\*

\* Entidad que modela un país.

\* Pertenece a la taskcard 12 de la iteración 1 y a la historia de usuario 3

\*/

**public** **class** Pais {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** Integer id; //ID

@Column(name = "nombre", length = 50, nullable = **false**)

**private** String nombre;

Código del archivo Provincia.java

/\*

\* Entidad que modela una provincia.

\* Pertenece a la taskcard 12 de la iteración 1 y a la historia de usuario 3

\*/

**public** **class** Provincia {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** Integer id; //ID

@Column(name = "nombre", length = 50, nullable = **false**)

**private** String nombre;

//Relaciones

@ManyToOne(fetch = FetchType.***LAZY***, cascade = CascadeType.***ALL***)

@JoinColumn(name = "idpais", referencedColumnName = "id", foreignKey = @ForeignKey(name = "provincia\_idpais\_fk"), nullable = **false**)

**private** Pais pais;

Código del archivo Localidad.java

/\*

\* Entidad que modela una localidad.

\* Pertenece a la taskcard 12 de la iteración 1 y a la historia de usuario 3

\*/

**public** **class** Localidad {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** Integer id; //ID

@Column(name = "nombre", length = 50, nullable = **false**)

**private** String nombre;

//Relaciones

@ManyToOne(fetch = FetchType.***LAZY***, cascade = CascadeType.***ALL***)

@JoinColumn(name = "idprovincia", referencedColumnName = "id", foreignKey = @ForeignKey(name = "localidad\_idprovincia\_fk"), nullable = **false**)

**private** Provincia provincia;

Código del archivo TipoDocumento.java

/\*

\* Entidad que modela el tipo de un documento.

\* Pertenece a la taskcard 12 de la iteración 1 y a la historia de usuario 3

\*/

**public** **class** TipoDocumento {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** Integer id; //ID

@Column(name = "tipo\_documento", nullable = **false**)

@Enumerated(EnumType.***STRING***)

**private** TipoDocumentoStr tipo;

Código del archivo TipoInmueble.java

/\*

\* Entidad que modela el tipo de un inmueble.

\* Pertenece a la taskcard 12 de la iteración 1 y a la historia de usuario 3

\*/

**public** **class** TipoInmueble {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** Integer id; //ID

@Enumerated(EnumType.***STRING***)

@Column(name = "tipo", length = 30)

**private** TipoInmuebleStr tipo;

Código del archivo Orientacion.java

/\*

\* Entidad que modela la orientacion física que tiene un inmueble.

\* Pertenece a la taskcard 12 de la iteración 1 y a la historia de usuario 3

\*/

**public** **class** Orientacion {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** Integer id; //ID

@Column(name = "orientacion\_enum", length = 10)

@Enumerated(EnumType.***STRING***)

**private** OrientacionStr orientacion;

#### Taskcard 14 Lógica alta, modificación y baja inmueble

Código del archivo PropietarioServiceImpl.java

/\*\*

\* Método para crear un inmueble. Primero se validan las reglas de negocia y luego se persiste.

\* Pertenece a la taskcard 14 de la iteración 1 y a la historia 3

\*

\* **@param** inmueble

\* a guardar

\* **@return** resultado de la operación

\* **@throws** PersistenciaException

\* si falló al persistir

\*/

**public** ResultadoCrearInmueble crearInmueble(Inmueble inmueble) **throws** PersistenciaException {

Set<ErrorCrearInmueble> errores = **new** HashSet<>();

validarInmuebleAlta(inmueble, errores);

**if**(errores.isEmpty()){

ArrayList<Estado> estados = gestorDatos.obtenerEstados();

**for**(Estado e: estados){

**if**(e.getEstado().equals(EstadoStr.***ALTA***)){

inmueble.setEstado(e);

**break**;

}

}

persistidorInmueble.guardarInmueble(inmueble);

}

**return** **new** ResultadoCrearInmueble(errores.toArray(**new** ErrorCrearInmueble[0]));

}

/\*\*

\* Método auxiliar que encapsula la validación de los atributos del inmueble a crear

\*

\* **@param** inmueble

\* inmueble a validar

\* **@param** errores

\* lista de errores encontrados

\*/

**private** **void** validarInmuebleAlta(Inmueble inmueble, Set<ErrorCrearInmueble> errores) **throws** PersistenciaException {

**if**(inmueble.getFechaCarga() == **null**){

errores.add(ErrorCrearInmueble.***Fecha\_Vacia***);

}

**if**(inmueble.getPropietario() != **null**){

Propietario propietario = gestorPropietario.obtenerPropietario(**new** FiltroPropietario(inmueble.getPropietario().getTipoDocumento().getTipo(), inmueble.getPropietario().getNumeroDocumento()));

**if**(propietario == **null**){

errores.add(ErrorCrearInmueble.***Propietario\_Inexistente***);

}

}

**else**{

errores.add(ErrorCrearInmueble.***Propietario\_Vacio***);

}

**if**(inmueble.getPrecio() == **null**){

errores.add(ErrorCrearInmueble.***Precio\_Vacio***);

}

**else** **if**(!validador.validarDoublePositivo(inmueble.getPrecio())){

errores.add(ErrorCrearInmueble.***Precio\_Incorrecto***);

}

**if**(inmueble.getFondo() != **null** && !validador.validarDoublePositivo(inmueble.getFondo())){

errores.add(ErrorCrearInmueble.***Fondo\_Incorrecto***);

}

**if**(inmueble.getFrente() != **null** && !validador.validarDoublePositivo(inmueble.getFrente())){

errores.add(ErrorCrearInmueble.***Frente\_Incorrecto***);

}

**if**(inmueble.getSuperficie() != **null** && !validador.validarDoublePositivo(inmueble.getSuperficie())){

errores.add(ErrorCrearInmueble.***Superficie\_Incorrecta***);

}

**if**(inmueble.getTipo() == **null**){

errores.add(ErrorCrearInmueble.***Tipo\_Vacio***);

}

**if**(!validador.validarDireccion(inmueble.getDireccion())){

errores.add(ErrorCrearInmueble.***Formato\_Direccion\_Incorrecto***);

}

**if**(inmueble.getDireccion() == **null** || inmueble.getDireccion().getBarrio() == **null** || inmueble.getDireccion().getBarrio().getLocalidad() == **null** || inmueble.getDireccion().getLocalidad() == **null** || inmueble.getDireccion().getLocalidad().getProvincia() == **null** || inmueble.getDireccion().getLocalidad().getProvincia().getPais() == **null** || inmueble.getDireccion().getCalle() == **null** || inmueble.getDireccion().getCalle().getLocalidad() == **null**){

errores.add(ErrorCrearInmueble.***Formato\_Direccion\_Incorrecto***);

}

**if**(!validarDatosEdificio(inmueble.getDatosEdificio())){

errores.add(ErrorCrearInmueble.***Datos\_Edificio\_Incorrectos***);

}

}

/\*\*

\* Método que se encarga de la modificación de los datos de un inmueble.

\* Se validan todos los datos correspondientes al inmueble, corroborando los que son obligatorios.

\* Se valida que exista el inmueble en la base de datos y en caso de que no haya errores, delega el guardado del objeto a la capa de acceso a datos.

\*

\* **@param** inmueble

\* inmueble a modificar.

\*

\* **@return** un resultado informando errores correspondientes en caso de que los haya.

\*

\* **@throws** PersistenciaException

\* Se lanza esta excepción al ocurrir un error interactuando con la capa de acceso a datos.

\*/

**public** ResultadoModificarInmueble modificarInmueble(Inmueble inmueble) **throws** PersistenciaException {

ArrayList<ErrorModificarInmueble> errores = **new** ArrayList<>();

validarInmuebleModificar(inmueble, errores);

Inmueble inmuebleAuxiliar = persistidorInmueble.obtenerInmueble(inmueble.getId());

**if**(inmuebleAuxiliar == **null**){

errores.add(ErrorModificarInmueble.***Inmueble\_Inexistente***);

}

**if**(errores.isEmpty()){

HistorialInmueble historialInmuebleAuxiliar = **new** HistorialInmueble();

DatosEdificio datosEdificioAuxiliar = inmuebleAuxiliar.getDatosEdificio();

HistorialDatosEdificio historialDatosEdificioAuxiliar = **new** HistorialDatosEdificio()

.setAguaCaliente(datosEdificioAuxiliar.getAguaCaliente())

.setAguaCorriente(datosEdificioAuxiliar.getAguaCaliente())

.setAntiguedad(datosEdificioAuxiliar.getAntiguedad())

.setBaños(datosEdificioAuxiliar.getBaños())

.setCloacas(datosEdificioAuxiliar.getCloacas())

.setDormitorios(datosEdificioAuxiliar.getDormitorios())

.setGaraje(datosEdificioAuxiliar.getGaraje())

.setGasNatural(datosEdificioAuxiliar.getGasNatural())

.setLavadero(datosEdificioAuxiliar.getLavadero())

.setPatio(datosEdificioAuxiliar.getPatio())

.setPavimento(datosEdificioAuxiliar.getPavimento())

.setPiscina(datosEdificioAuxiliar.getPiscina())

.setPropiedadHorizontal(datosEdificioAuxiliar.getPropiedadHorizontal())

.setSuperficie(datosEdificioAuxiliar.getSuperficie())

.setTelefono(datosEdificioAuxiliar.getTelefono())

.setDatosEdificio(datosEdificioAuxiliar)

.setHistorialInmueble(historialInmuebleAuxiliar);

historialInmuebleAuxiliar

.setDireccion(inmuebleAuxiliar.getDireccion())

.setEstado(inmuebleAuxiliar.getEstado())

.setFechaCarga(inmuebleAuxiliar.getFechaCarga())

.setFechaYHoraCambio(**new** Date())

.setFondo(inmuebleAuxiliar.getFondo())

.setFrente(inmuebleAuxiliar.getFrente())

.setObservaciones(inmuebleAuxiliar.getObservaciones())

.setOrientacion(inmuebleAuxiliar.getOrientacion())

.setPrecio(inmuebleAuxiliar.getPrecio())

.setPropietario(inmuebleAuxiliar.getPropietario())

.setSuperficie(inmuebleAuxiliar.getSuperficie())

.setTipo(inmuebleAuxiliar.getTipo())

.setInmueble(inmuebleAuxiliar)

.setHistorialDatosEdificio(historialDatosEdificioAuxiliar)

.getFotos().addAll(inmuebleAuxiliar.getFotos());

persistidorHistorial.guardarHistorialInmueble(historialInmuebleAuxiliar);

persistidorInmueble.modificarInmueble(inmueble);

}

**return** **new** ResultadoModificarInmueble(errores.toArray(**new** ErrorModificarInmueble[0]));

}

/\*\*

\* Método auxiliar que encapsula la validación de los atributos del inmueble a modificar

\*

\* **@param** inmueble

\* inmueble a validar

\* **@param** errores

\* lista de errores encontrados

\*/

**private** **void** validarInmuebleModificar(Inmueble inmueble, ArrayList<ErrorModificarInmueble> errores)

**throws** PersistenciaException {

**if**(inmueble.getFechaCarga() == **null**){

errores.add(ErrorModificarInmueble.***Fecha\_Vacia***);

}

**if**(inmueble.getPropietario() != **null**){

Propietario propietario = gestorPropietario.obtenerPropietario(**new** FiltroPropietario(inmueble.getPropietario().getTipoDocumento().getTipo(), inmueble.getPropietario().getNumeroDocumento()));

**if**(propietario == **null**){

errores.add(ErrorModificarInmueble.***Propietario\_Inexistente***);

}

}

**else**{

errores.add(ErrorModificarInmueble.***Propietario\_Vacio***);

}

**if**(inmueble.getPrecio() == **null**){

errores.add(ErrorModificarInmueble.***Precio\_Vacio***);

}

**if**(!validador.validarDoublePositivo(inmueble.getPrecio())){

errores.add(ErrorModificarInmueble.***Precio\_Incorrecto***);

}

**if**(inmueble.getFondo() != **null** && !validador.validarDoublePositivo(inmueble.getFondo())){

errores.add(ErrorModificarInmueble.***Fondo\_Incorrecto***);

}

**if**(inmueble.getFrente() != **null** && !validador.validarDoublePositivo(inmueble.getFrente())){

errores.add(ErrorModificarInmueble.***Frente\_Incorrecto***);

}

**if**(inmueble.getSuperficie() != **null** && !validador.validarDoublePositivo(inmueble.getSuperficie())){

errores.add(ErrorModificarInmueble.***Superficie\_Incorrecta***);

}

**if**(inmueble.getTipo() == **null**){

errores.add(ErrorModificarInmueble.***Tipo\_Vacio***);

}

**if**(!validador.validarDireccion(inmueble.getDireccion())){

errores.add(ErrorModificarInmueble.***Formato\_Direccion\_Incorrecto***);

}

**if**(!validarDatosEdificio(inmueble.getDatosEdificio())){

errores.add(ErrorModificarInmueble.***Datos\_Edificio\_Incorrectos***);

}

}

/\*\*

\* Se encarga de validar que exista el inmueble a eliminar, se setea el estado en BAJA y,

\* en caso de que no haya errores, delegar el guardado del objeto a la capa de acceso a datos.

\*

\* **@param** inmueble

\* inmueble a eliminar

\* **@return** un resultado informando errores correspondientes en caso de que los haya

\*

\* **@throws** PersistenciaException

\* se lanza esta excepción al ocurrir un error interactuando con la capa de acceso a datos

\*/

**public** ResultadoEliminarInmueble eliminarInmueble(Inmueble inmueble) **throws** PersistenciaException {

ArrayList<Estado> estados = gestorDatos.obtenerEstados();

**for**(Estado e: estados){

**if**(e.getEstado().equals(EstadoStr.***BAJA***)){

inmueble.setEstado(e);

}

}

persistidorInmueble.modificarInmueble(inmueble);

**return** **new** ResultadoEliminarInmueble();

}

/\*\*

\* Obtiene el listado de inmuebles solicitándola a la capa de acceso a datos

\*

\* **@return** el listado de inmuebles solicitados

\*

\* **@throws** PersistenciaException

\* se lanza esta excepción al ocurrir un error interactuando con la capa de acceso a datos

\*/

**public** ArrayList<Inmueble> obtenerInmuebles() **throws** PersistenciaException {

**return** persistidorInmueble.listarInmuebles();

}

/\*\*

\* Método auxiliar que encapsula la validación de los atributos del edificio

\*

\* **@param** datosEdificio

\* datos del edificio a validar

\*

\* **@return** True si los datos son correctos

\* False si no son correctos

\*/

**protected** Boolean validarDatosEdificio(DatosEdificio datosEdificio) {

**if**(datosEdificio == **null**){

**return** **false**;

}

**if**(datosEdificio.getAguaCaliente() == **null**){

**return** **false**;

}

**if**(datosEdificio.getAguaCorriente() == **null**){

**return** **false**;

}

**if**(datosEdificio.getCloacas() == **null**){

**return** **false**;

}

**if**(datosEdificio.getGaraje() == **null**){

**return** **false**;

}

**if**(datosEdificio.getGasNatural() == **null**){

**return** **false**;

}

**if**(datosEdificio.getLavadero() == **null**){

**return** **false**;

}

**if**(datosEdificio.getPatio() == **null**){

**return** **false**;

}

**if**(datosEdificio.getPavimento() == **null**){

**return** **false**;

}

**if**(datosEdificio.getPiscina() == **null**){

**return** **false**;

}

**if**(datosEdificio.getPropiedadHorizontal() == **null**){

**return** **false**;

}

**if**(datosEdificio.getTelefono() == **null**){

**return** **false**;

}

**if**(datosEdificio.getAntiguedad() != **null** && !validador.validarEnteroPositivo(datosEdificio.getAntiguedad())){

**return** **false**;

}

**if**(datosEdificio.getBaños() != **null** && !validador.validarEnteroPositivo(datosEdificio.getBaños())){

**return** **false**;

}

**if**(datosEdificio.getDormitorios() != **null** && !validador.validarEnteroPositivo(datosEdificio.getDormitorios())){

**return** **false**;

}

**if**(datosEdificio.getSuperficie() != **null** && !validador.validarDoublePositivo(datosEdificio.getSuperficie())){

**return** **false**;

}

**return** **true**;

}

Prueba de unidad del archivo GetorInmueble.java

//Casos de prueba

//inmueble, resultado, validadorMock, propietario, excepcion

/\* 0 \*/ **new** Object[] { inmuebleCorrecto, **new** ResultadoCrearInmueble(), validadorCorrecto, propietario, **null** }, //inmueble correcto

/\* 1 \*/ **new** Object[] { inmuebleSinFecha, **new** ResultadoCrearInmueble(ErrorCrearInmueble.***Fecha\_Vacia***), validadorCorrecto, propietario, **null** }, //inmueble sin fecha de carga

/\* 2 \*/ **new** Object[] { inmuebleSinPropietario, **new** ResultadoCrearInmueble(ErrorCrearInmueble.***Propietario\_Vacio***), validadorCorrecto, propietario, **null** }, //inmueble sin propietario

/\* 3 \*/ **new** Object[] { inmuebleSinTipo, **new** ResultadoCrearInmueble(ErrorCrearInmueble.***Tipo\_Vacio***), validadorCorrecto, propietario, **null** }, //inmueble sin TipoInmueble

/\* 4 \*/ **new** Object[] { inmuebleSinPrecio, **new** ResultadoCrearInmueble(ErrorCrearInmueble.***Precio\_Vacio***), validadorCorrecto, propietario, **null** }, //inmueble sin precio

/\* 5 \*/ **new** Object[] { inmuebleCorrecto, **new** ResultadoCrearInmueble(ErrorCrearInmueble.***Formato\_Direccion\_Incorrecto***), validadorFormatoDireccionIncorrecto, propietario, **null** }, //inmueble con formato de direccion incorrecta

/\* 6 \*/ **new** Object[] { inmueblePrecioIncorrecto, **new** ResultadoCrearInmueble(ErrorCrearInmueble.***Precio\_Incorrecto***), validadorDoubleIncorrecto, propietario, **null** }, //inmueble con formato de precio incorrecto

/\* 7 \*/ **new** Object[] { inmuebleFrenteIncorrecto, **new** ResultadoCrearInmueble(ErrorCrearInmueble.***Frente\_Incorrecto***), validadorDoubleIncorrecto, propietario, **null** }, //inmueble con formato de frente incorrecto

/\* 8 \*/ **new** Object[] { inmuebleFondoIncorrecto, **new** ResultadoCrearInmueble(ErrorCrearInmueble.***Fondo\_Incorrecto***), validadorDoubleIncorrecto, propietario, **null** }, //inmueble con formato de fondo incorrecto

/\* 9 \*/ **new** Object[] { inmuebleSuperficieIncorrecta, **new** ResultadoCrearInmueble(ErrorCrearInmueble.***Superficie\_Incorrecta***), validadorDoubleIncorrecto, propietario, **null** }, //inmueble con formato de superficie incorrecto

/\* 10 \*/ **new** Object[] { inmuebleDatosEdificioIncorrectos, **new** ResultadoCrearInmueble(ErrorCrearInmueble.***Datos\_Edificio\_Incorrectos***), validadorCorrecto, propietario, **null** }, //inmueble con datosEdificio Incorrectos

/\* 11 \*/ **new** Object[] { inmuebleCorrecto, **new** ResultadoCrearInmueble(ErrorCrearInmueble.***Propietario\_Inexistente***), validadorCorrecto, **null**, **null** }, //propietario del inmueble no está persistido

/\* 12 \*/ **new** Object[] { inmuebleCorrecto, **null**, validadorCorrecto, **null**, **new** ObjNotFoundException("", **new** Exception()) }, //el persistidor tira una PersistenciaException

/\* 13 \*/ **new** Object[] { inmuebleCorrecto, **null**, validadorCorrecto, **null**, **new** Exception() } //el persistidor tira una excepción inesperada

/\*\*

\* Prueba el método crearInmueble(), el cual corresponde con la taskcard 14 de la iteración 1 y a la historia 3

\*

\* **@param** inmueble

\* inmueble a crear

\* **@param** resultado

\* resultado que se espera que devuelva el gestor

\* **@param** validadorMock

\* clase vacía que utiliza el gestor para validar

\* **@param** propietario

\* propietario "en la base de datos" a comparar con el propietario del inmueble a guardar para verificar que exista en la base de datos

\* **@param** excepcion

\* es la excepcion que debe lanzar el mock del persistidor, si la prueba involucra procesar una excepcion de dicho persistidor, debe ser nulo propietario para que se use

\* **@throws** Exception

\*/

**public** **void** testCrearInmueble(Inmueble inmueble, ResultadoCrearInmueble resultado, ValidadorFormato validadorMock, Propietario propietario, Throwable excepcion) **throws** Exception {

GestorDatos gestorDatosMock = **new** GestorDatos() {

@Override

**public** ArrayList<Estado> obtenerEstados() **throws** PersistenciaException {

**return** **new** ArrayList<>();

}

@Override

**public** ArrayList<EstadoInmueble> obtenerEstadosInmueble() **throws** PersistenciaException {

**return** **new** ArrayList<>();

}

};

GestorPropietario gestorPropietarioMock = **new** GestorPropietario() {

@Override

**public** Propietario obtenerPropietario(FiltroPropietario filtro) **throws** PersistenciaException {

**if**(propietario != **null**){

**return** propietario;

}

**if**(excepcion == **null**){

**return** **null**;

}

**if**(excepcion **instanceof** PersistenciaException){

**throw** (PersistenciaException) excepcion;

}

**new** Integer("asd");

**return** **null**;

}

};

InmuebleService persistidorInmuebleMock = **new** InmuebleService() {

@Override

**public** Inmueble obtenerInmueble(Integer id) **throws** PersistenciaException {

**return** **null**;

}

@Override

**public** **void** modificarInmueble(Inmueble inmueble) **throws** PersistenciaException {

}

@Override

**public** ArrayList<Inmueble> listarInmuebles() **throws** PersistenciaException {

**return** **null**;

}

@Override

**public** **void** guardarInmueble(Inmueble inmueble) **throws** PersistenciaException {

}

@Override

**public** ArrayList<Inmueble> listarInmuebles(FiltroInmueble filtro) **throws** PersistenciaException {

**return** **null**;

}

};

GestorInmueble gestorInmueble = **new** GestorInmueble() {

{

**this**.gestorPropietario = gestorPropietarioMock;

**this**.persistidorInmueble = persistidorInmuebleMock;

**this**.gestorDatos = gestorDatosMock;

**this**.validador = validadorMock;

}

};

//creamos la prueba

Statement test = **new** Statement() {

@Override

**public** **void** evaluate() **throws** Throwable {

**if**(resultado != **null**){

*assertEquals*(resultado, gestorInmueble.crearInmueble(inmueble));

}

**else**{

**try**{

gestorInmueble.crearInmueble(inmueble);

Assert.*fail*("Debería haber fallado!");

} **catch**(PersistenciaException e){

Assert.*assertEquals*((excepcion), e);

} **catch**(Exception e){

**if**(excepcion **instanceof** PersistenciaException){

Assert.*fail*("Debería haber tirado una PersistenciaException y tiro otra Exception!");

}

}

}

}

};

//Ejecutamos la prueba

**try**{

test.evaluate();

} **catch**(Throwable e){

**throw** **new** Exception(e);

}

}

//Casos de prueba

// inmueble, resultadoValidarFondo, resultadoValidarFrente, resultadoValidarSuperficie, resultadoValidarDireccion, resultadoValidarDatosEdificio, resultadoValidarPrecio, retornaInmueble, retornaPropietario, resultado, excepcion

/\* 0 \*/ **new** Object[] { inmuebleCorrecto, **true**, **true**, **true**, **true**, **true**, **true**, **true**, **true**, ***resultadoModificarCorrecto***, **null** }, //inmueble correcto

/\* 1 \*/ **new** Object[] { inmuebleSinFecha, **true**, **true**, **true**, **true**, **true**, **true**, **true**, **true**, ***resultadoModificarFecha\_Vacia***, **null** }, //inmueble sin fecha de carga

/\* 2 \*/ **new** Object[] { inmuebleSinPropietario, **true**, **true**, **true**, **true**, **true**, **true**, **true**, **true**, ***resultadoModificarPropietario\_Vacio***, **null** }, //inmueble sin propietario

/\* 3 \*/ **new** Object[] { inmuebleSinTipo, **true**, **true**, **true**, **true**, **true**, **true**, **true**, **true**, ***resultadoModificarTipo\_Vacio***, **null** }, //inmueble sin TipoInmueble

/\* 4 \*/ **new** Object[] { inmuebleSinPrecio, **true**, **true**, **true**, **true**, **true**, **true**, **true**, **true**, ***resultadoModificarPrecio\_Vacio***, **null** }, //inmueble sin precio

/\* 5 \*/ **new** Object[] { inmuebleFondoIncorrecto, **false**, **true**, **true**, **true**, **true**, **true**, **true**, **true**, ***resultadoModificarFondo\_Incorrecto***, **null** }, //inmueble con formato de fondo incorrecto

/\* 6 \*/ **new** Object[] { inmuebleFrenteIncorrecto, **true**, **false**, **true**, **true**, **true**, **true**, **true**, **true**, ***resultadoModificarFrente\_Incorrecto***, **null** }, //inmueble con formato de frente incorrecto

/\* 7 \*/ **new** Object[] { inmuebleSuperficieIncorrecta, **true**, **true**, **false**, **true**, **true**, **true**, **true**, **true**, ***resultadoModificarSuperficie\_Incorrecta***, **null** }, //inmueble con formato de superficie incorrecto

/\* 8 \*/ **new** Object[] { inmuebleCorrecto, **true**, **true**, **true**, **false**, **true**, **true**, **true**, **true**, ***resultadoModificarFormato\_Direccion\_Incorrecto***, **null** }, //inmueble con formato de direccion incorrecta

/\* 9 \*/ **new** Object[] { inmuebleDatosEdificioIncorrectos, **true**, **true**, **true**, **true**, **false**, **true**, **true**, **true**, ***resultadoModificarDatos\_Edificio\_Incorrectos***, **null** }, //inmueble con datosEdificio Incorrectos

/\* 10 \*/ **new** Object[] { inmuebleCorrecto, **true**, **true**, **true**, **true**, **true**, **false**, **true**, **true**, ***resultadoModificarPrecio\_Incorrecto***, **null** }, //inmueble con formato de precio incorrecto

/\* 11 \*/ **new** Object[] { inmuebleCorrecto, **true**, **true**, **true**, **true**, **true**, **true**, **true**, **false**, ***resultadoModificarPropietario\_Inexistente***, **null** }, //propietario del inmueble no está persistido

/\* 12 \*/ **new** Object[] { inmuebleCorrecto, **true**, **true**, **true**, **true**, **true**, **true**, **false**, **true**, ***resultadoModificarInmueble\_Inexistente***, **null** }, //Inmueble no está persistido

/\* 13 \*/ **new** Object[] { inmuebleCorrecto, **true**, **true**, **true**, **true**, **true**, **true**, **true**, **true**, **null**, **new** ObjNotFoundException("", **new** Exception()) }, //el persistidor tira una PersistenciaException

/\* 14 \*/ **new** Object[] { inmuebleCorrecto, **true**, **true**, **true**, **true**, **true**, **true**, **true**, **true**, **null**, **new** SaveUpdateException(**new** Exception()) }, //el persistidor tira una SaveUpdateException

/\*\*

\* Prueba el método modificarInmueble(), el cual corresponde con la taskcard 14 de la iteración 1 y a la historia 3

\*

\* **@param** inmueble

\* inmueble a modificar

\* **@param** resultadoValidarFondo

\* resultado devuelto por el mock validador al validar el fondo del inmueble

\* **@param** resultadoValidarFrente

\* resultado devuelto por el mock validador al validar el frente del inmueble

\* **@param** resultadoValidarSuperficie

\* resultado devuelto por el mock validador al validar la superficie del inmueble

\* **@param** resultadoValidarDireccion

\* resultado devuelto por el mock validador al validar la dirección del inmueble

\* **@param** resultadoValidarDatosEdificioEsperado

\* resultado devuelto por el mock validador al validar el fondo del inmueble

\* **@param** resultadoValidarPrecio

\* resultado devuelto por el mock validador al validar el precio del inmueble

\* **@param** retornaInmueble

\* indica si el persistidor devuelve un inmueble

\* **@param** retornaPropietario

\* indica si el persistidor devuelve un propietario

\* **@param** resultadoEsperado

\* resultado que se espera que devuelva el gestor

\* **@param** propietario

\* propietario "en la base de datos" a comparar con el propietario del inmueble a guardar para verificar que exista en la base de datos

\* **@param** excepcion

\* es la excepcion que debe lanzar el mock del persistidor, si la prueba involucra procesar una excepcion de dicho persistidor, debe ser nulo propietario para que se use

\* **@throws** Exception

\*/

@Test

@Parameters

**public** **void** testModificarInmueble(Inmueble inmueble, Boolean resultadoValidarFondo, Boolean resultadoValidarFrente, Boolean resultadoValidarSuperficie, Boolean resultadoValidarDireccion, Boolean resultadoValidarDatosEdificioEsperado, Boolean resultadoValidarPrecio, Boolean retornaInmueble, Boolean retornaPropietario, ResultadoModificarInmueble resultadoEsperado, Throwable excepcion) **throws** Exception {

GestorPropietario gestorPropietarioMock = Mockito.*mock*(GestorPropietario.**class**);

InmuebleService persistidorInmuebleMock = Mockito.*mock*(InmuebleService.**class**);

HistorialService persistidorHistorialMock = Mockito.*mock*(HistorialService.**class**);

ValidadorFormato validadorFormatoMock = Mockito.*mock*(ValidadorFormato.**class**);

Mockito.*when*(validadorFormatoMock.validarDoublePositivo(inmueble.getFondo())).thenReturn(resultadoValidarFondo);

Mockito.*when*(validadorFormatoMock.validarDoublePositivo(inmueble.getFrente())).thenReturn(resultadoValidarFrente);

Mockito.*when*(validadorFormatoMock.validarDoublePositivo(inmueble.getSuperficie())).thenReturn(resultadoValidarSuperficie);

Mockito.*when*(validadorFormatoMock.validarDireccion(inmueble.getDireccion())).thenReturn(resultadoValidarDireccion);

Mockito.*when*(validadorFormatoMock.validarDoublePositivo(inmueble.getPrecio())).thenReturn(resultadoValidarPrecio);

**if**(retornaPropietario){

Mockito.*when*(gestorPropietarioMock.obtenerPropietario(*any*())).thenReturn(inmueble.getPropietario());

}

**else**{

Mockito.*when*(gestorPropietarioMock.obtenerPropietario(*any*())).thenReturn(**null**);

}

**if**(retornaInmueble){

Mockito.*when*(persistidorInmuebleMock.obtenerInmueble(inmueble.getId())).thenReturn(inmueble);

}

**else**{

Mockito.*when*(persistidorInmuebleMock.obtenerInmueble(inmueble.getId())).thenReturn(**null**);

}

**if**(excepcion != **null**){

Mockito.*doThrow*(excepcion).when(persistidorInmuebleMock).modificarInmueble(inmueble);

}

GestorInmueble gestorInmueble = **new** GestorInmueble() {

{

**this**.validador = validadorFormatoMock;

**this**.gestorPropietario = gestorPropietarioMock;

**this**.persistidorInmueble = persistidorInmuebleMock;

**this**.persistidorHistorial = persistidorHistorialMock;

}

};

//creamos la prueba

Statement test = **new** Statement() {

@Override

**public** **void** evaluate() **throws** Throwable {

**if**(excepcion == **null**){

*assertEquals*(resultadoEsperado, gestorInmueble.modificarInmueble(inmueble));

*assertEquals*(resultadoValidarDatosEdificioEsperado, gestorInmueble.validarDatosEdificio(inmueble.getDatosEdificio()));

**if**(!resultadoEsperado.hayErrores()){

Mockito.*verify*(persistidorInmuebleMock).modificarInmueble(*any*());

Mockito.*verify*(persistidorHistorialMock).guardarHistorialInmueble(*any*());

}

}

**else**{

**try**{

gestorInmueble.modificarInmueble(inmueble);

Assert.*fail*("Debería haber fallado!");

} **catch**(PersistenciaException e){

Assert.*assertEquals*((excepcion), e);

} **catch**(Exception e){

**if**(excepcion **instanceof** PersistenciaException){

Assert.*fail*("Debería haber tirado una PersistenciaException y tiro otra Exception!");

}

}

}

}

};

//Ejecutamos la prueba

**try**{

test.evaluate();

} **catch**(Throwable e){

**throw** **new** Exception(e);

}

}