**ANEXO: CODIGO FUENTE**

**Código fuente de la API:**

**Indice:**

**Implementaciones Páginas**

interface ActionCommand 3

interface Event 3

interface EventListener 3

interface EventManager 4

class EventManagerFactory 4

class exceptionRegisterEvent 4

class ActionHandler 5

class ConcreteEventManager 8

class EventCancel 10

**package** mat7510.eventManagerApi;

**public** **interface** ActionCommand {

**public** **void** execute();

}

**package** mat7510.eventManagerApi;

**public** **interface** Event {

**public** **boolean** equals(Event anotherEvent);

}

**package** mat7510.eventManagerApi;

**public** **interface** EventListener {

**public** **void** eventOccurred(Event e);

}

**package** mat7510.eventManagerApi;

**import** java.util.List;

**public** **interface** EventManager **extends** EventListener {

**public** **void** register(ActionCommand cmd, Event e) **throws** exceptionRegisterEvent ;

**public** **void** register(ActionCommand cmd, List<Event> e) **throws** exceptionRegisterEvent ;

**public** **void** registerWithOrder(ActionCommand cmd, List<Event> e) **throws** exceptionRegisterEvent ;

**public** **void** registerCancellables(Event event1, Event event2) **throws** exceptionRegisterEvent ;

**public** **void** reset();

}

**package** mat7510.eventManagerApi;

**public** **class** EventManagerFactory {

**private** **static** EventManager *eventManager* = **null**;

**private** EventManagerFactory(){};

**public** **static** EventManager createEventManager() {

**if** (*eventManager* == **null**) {

*eventManager* = **new** ConcreteEventManager();

}

**return** *eventManager*;

}

**public** **static** EventManager getInstance(){

**if** (*eventManager* == **null**)

**return** *createEventManager*();

**return** *eventManager*;

}

}

**package** mat7510.eventManagerApi;

**public** **class** exceptionRegisterEvent **extends** Exception{

**private** **static** **final** **long** *serialVersionUID* = 1L;

**private** String text;

**public** exceptionRegisterEvent(String e) {

text = "No se puede registrar los eventos debido a: " + e;

}

@Override

**public** String toString() {

**return** text;

}

}

**package** mat7510.eventManagerApi;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

**public** **class** ActionHandler {

**private** ActionCommand command;

**private** List<Event> events;

**private** ArrayList<Boolean> eventsIndexs;

**int** amountActivated;

**private** **boolean** order;

**static** **public** ActionHandler createActionSingle ( ActionCommand command, Event event){

**return** **new** ActionHandler(command ,event);

}

**static** **public** ActionHandler createActionGroup ( ActionCommand command, List<Event> event){

**return** **new** ActionHandler(command ,event, **false**);

}

**static** **public** ActionHandler createActionGroupOrder ( ActionCommand command, List<Event> event){

**return** **new** ActionHandler(command ,event, **true**);

}

**private** **void** createEvents() {

// Inicializa el array de eventos de control del comando

amountActivated=0;

eventsIndexs = **new** ArrayList<Boolean>();

**for** (**int** index = 0; index < events.size(); index++ ){

eventsIndexs.add(index,**false**);

}

}

**private** ActionHandler ( ActionCommand command, Event event){

**this**.command = command;

events = **new** ArrayList<Event>();

eventsIndexs = **new** ArrayList<Boolean>();

**this**.events.add(event);

setOrder(**false**);

createEvents();

amountActivated=0;

}

**private** ActionHandler ( ActionCommand command, List<Event> events, **boolean** order){

**this**.command = command;

**this**.events = events;

setOrder(order);

createEvents();

amountActivated=0;

}

**private** **void** clearEvents() {

// Inicializa el array de eventos de control del comando

**if**(amountActivated==0)

**return**;

amountActivated=0;

**for** (**int** index = 0; index < events.size(); index++ ){

eventsIndexs.set(index,**false**);

}

}

**public** **void** addEvento( Event event) {

**this**.events.add(event);

}

**public** Iterator<Event> getEventIterator(){

**return** **this**.events.listIterator();

}

**public** **void** setCommand(ActionCommand command) {

**this**.command = command;

}

**public** ActionCommand getCommand() {

**return** command;

}

**public** **boolean** isOrder() {

**return** order;

}

**private** **void** setOrder(**boolean** order) {

**this**.order = order;

}

**public** **boolean** getOrder() {

**return** order;

}

**public** **void** activateEvent (**int** index){

// Cuando se activa se setea en el array como activo(true)

**if**(eventsIndexs.get(index)!=**true**){

**if** ( order == **true** ){

**if**(verifyOrder( index ) == **true**){

eventsIndexs.set(index, **true**);

amountActivated++;

}

}**else**{

eventsIndexs.set(index, **true**);

amountActivated++;

}

}

}

**public** **boolean** isActivedEvent(**int** index){

**return** eventsIndexs.get(index);

}

**private** **void** cancelEventsWithOrder (**int** index){

**for**( **int** i=index ; i < eventsIndexs.size() ; i++){

**if**(eventsIndexs.get(i) == **true**){

eventsIndexs.set(i,**false**);

amountActivated--;

}

}

}

**public** **void** cancelEvent (**int** index){

// Cuando se cancela seteo el estado a desactivado

**if** (eventsIndexs.get(index) == **true**){

**if**(order)

cancelEventsWithOrder (index);

**else**{

eventsIndexs.set(index,**false**);

amountActivated--;

}

}

}

**private** **boolean** verifyOrder(**int** index) {

// Verifica que no haya ningun evento con indice menor al que se esta revisando dentro de la lista

**int** i=0;

**if**(amountActivated==index-1){

**while**(i<index){

**if**(eventsIndexs.get(i)==**false**)

**return** **false**;

i++;

}

}

**return** **true**;

}

**public** **boolean** isActive() {

**if**(amountActivated==eventsIndexs.size())

**return** **true**;

**else** **return** **false**;

}

**public** **void** cleanState(){

// Vuelve los eventos al momento inicial

clearEvents();

}

}

**package** mat7510.eventManagerApi;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

**public** **class** ConcreteEventManager **implements** EventManager {

**private** List<ActionHandler> actions;

**private** List<EventCancel> eventsCancel;

**public** ConcreteEventManager (){

actions = **new** ArrayList<ActionHandler>();

eventsCancel = **new** ArrayList<EventCancel>();

}

**public** **void** eventOccurred(Event e) {

**if**(e==**null**)

**throw** **new** NullPointerException("Evento nulo");

Iterator<EventCancel> it = eventsCancel.iterator();

EventCancel eventCancel;

**boolean** isFound=**false**;

**while** (it.hasNext() && !isFound){

eventCancel = it.next();

**if** ( e.equals(eventCancel.getEventSource())){

notifyChange ( eventCancel.getEventToBeCancel(),**false**);

isFound=**true**;

}

}

notifyChange (e,**true**);

}

**private** **void** notifyChange(Event event, **boolean** marcar) {

Iterator<ActionHandler> itActions = actions.iterator();

Iterator<Event> itEvents;

ActionHandler action;

Event actionEvent;

**int** index;

**while** (itActions.hasNext()){

action = itActions.next();

itEvents = action.getEventIterator();

index = 0;

**while** (itEvents.hasNext()){

actionEvent = itEvents.next();

**if** (event.equals(actionEvent) && marcar == **true**){

**if**(!action.isActivedEvent(index)){

// marca el evento

action.activateEvent(index);

**if** (action.isActive()){

action.getCommand().execute();

action.cleanState();

}

//Se recorre toda la lista para verificar si registran dos veces el mismo evento

**if**(action.getOrder()==**true**)

**break**;

}

}

**if** (event.equals(actionEvent) && marcar == **false**){

// Se desactivan todos los eventos que cumplan

action.cancelEvent(index);

}

index ++;

}

}

}

@Override

**public** **void** register(ActionCommand cmd, Event e) **throws** exceptionRegisterEvent {

**if**(e==**null**)

**throw** **new** exceptionRegisterEvent("Evento nulo");

**if**(cmd==**null**)

**throw** **new** exceptionRegisterEvent("Comando nulo");

ActionHandler action = ActionHandler.*createActionSingle* (cmd, e);

**this**.actions.add(action);

}

@Override

**public** **void** register(ActionCommand cmd, List<Event> e) **throws** exceptionRegisterEvent {

**if**(e== **null** || e.isEmpty())

**throw** **new** exceptionRegisterEvent("La lista de eventos esta vacia");

**if**(cmd==**null**)

**throw** **new** exceptionRegisterEvent("Comando nulo");

ActionHandler action = ActionHandler.*createActionGroup* (cmd, e);

**this**.actions.add(action);

}

@Override

**public** **void** registerWithOrder(ActionCommand cmd, List<Event> e) **throws** exceptionRegisterEvent {

**if**(e== **null** || e.isEmpty())

**throw** **new** exceptionRegisterEvent("La lista de eventos esta vacia");

**if**(cmd==**null**)

**throw** **new** exceptionRegisterEvent("Comando nulo");

ActionHandler action = ActionHandler.*createActionGroupOrder* (cmd, e);

**this**.actions.add(action);

}

@Override

**public** **void** registerCancellables(Event event1, Event event2) **throws** exceptionRegisterEvent {

**if**(event1==**null** || event2==**null**)

**throw** **new** exceptionRegisterEvent("Evento null");

**if**(event1.equals(event2))

**throw** **new** exceptionRegisterEvent("Los eventos que se desean registrar son iguales");

EventCancel eventCancel = **new** EventCancel (event1, event2);

**this**.eventsCancel.add(eventCancel);

}

@Override

**public** **void** reset() {

actions.clear();

eventsCancel.clear();

}

}

**package** mat7510.eventManagerApi;

**public** **class** EventCancel {

**private** Event eventSource;

**private** Event eventToBeCancel;

**public** EventCancel ( Event eventSource , Event eventToBeCancel){

**this**.eventSource = eventSource;

**this**.eventToBeCancel = eventToBeCancel;

}

**public** **void** setEventToBeCancel(Event eventToBeCancel) {

**this**.eventToBeCancel = eventToBeCancel;

}

**public** Event getEventToBeCancel() {

**return** eventToBeCancel;

}

**public** **void** setEventSource(Event eventSource) {

**this**.eventSource = eventSource;

}

**public** Event getEventSource() {

**return** eventSource;

}

}

**Código fuente clases de Ejemplo de Dominio:**

**Indice:**

**Implementaciones Páginas**

class BasicActionCommand 13

class BasicActionReceiver 13

class BasicEvent 14

class BasicEventSource 14

**public** **class** BasicActionCommand **implements** ActionCommand {

BasicActionReceiver receiver;

**public** BasicActionCommand(BasicActionReceiver receiver) {

**this**.receiver = receiver;

}

**public** **void** execute() {

receiver.setState(**true**);

}

}

**package** mat7510.eventManagerApi.domainExamples.basicDomain;

**public** **class** BasicActionReceiver {

**private** **boolean** state = **false**;

**public** **void** setState(**boolean** state) {

**this**.state = state;

}

**public** **boolean** getState() {

**return** state;

}

}

**public** **class** BasicEvent **implements** Event {

**private** String eventName;

**public** BasicEvent(String eventName) {

**this**.eventName = eventName;

}

**public** String getEventName() {

**return** **this**.eventName;

}

**public** **boolean** equals(Event anotherEvent) {

**if** (!(anotherEvent **instanceof** BasicEvent)) {

**return** **false**;

}

**return** ((BasicEvent)anotherEvent).eventName.equals(**this**.eventName);

}

@Override

**public** String toString() {

**return** **this**.eventName;

}

}

**public** **class** BasicEventSource {

**private** String eventToTrigger;

**public** BasicEventSource(String eventToTrigger) {

**this**.eventToTrigger = eventToTrigger;

}

**private** EventListener listener;

**public** **void** addListener(EventListener listener) {

**this**.listener = listener;

}

**public** **void** triggerEvent() **throws** exceptionRegisterEvent {

listener.eventOccurred(**new** BasicEvent(eventToTrigger));

}

**public** **void** triggerEventNull() **throws** exceptionRegisterEvent {

BasicEvent event = **null**;

listener.eventOccurred(event);

}

}

**Código fuente de los Test:**

**Indice:**

**Implementaciones Páginas**

class BasicTest 16

class MultipleEventsTest 17

class MultipleEventsWithCancelablesTest 19

class MultipleEventsWithOrderAndCancelablesTest 21

class MultipleEventsWithOrderTest 23

class NullTriggerEvent 25

class BombaDeAguaTest 26

**public** **class** BasicTest {

**private** EventManager mngr;

**private** BasicEventSource eventSource;

**private** BasicActionReceiver actionReceiver;

**private** **static** **final** String *EVENTO* = "evento";

@Before

**public** **void** setUp() **throws** Exception {

mngr = EventManagerFactory.*getInstance*();

eventSource = **new** BasicEventSource(*EVENTO*);

eventSource.addListener(mngr);

actionReceiver = **new** BasicActionReceiver();

}

@After

**public** **void** tearDown() {

// nada por ahora...

}

@Test

**public** **void** testBasicContext() **throws** exceptionRegisterEvent {

// Registramos en el Manager la accion - evento

**try**{

mngr.register(**new** BasicActionCommand(actionReceiver), **new** BasicEvent(*EVENTO*));

}**catch**(exceptionRegisterEvent e){

System.*out*.println(e.toString());

}

// El Source dispara el Evento...

eventSource.triggerEvent();

// Y si todo funciona bien, el Receiver deberia haber sufrido

// el cambio de estado, por la accion ejecutada...

*assertTrue*(actionReceiver.getState());

}

}

**public** **class** MultipleEventsTest {

**private** **static** **final** String *EVENTO1* = "evento 1";

**private** **static** **final** String *EVENTO2* = "evento 2";

**private** **static** **final** String *EVENTO3* = "evento 3";

**private** EventManager mngr;

**private** BasicEventSource eventSource1;

**private** BasicEventSource eventSource2;

**private** BasicEventSource eventSource3;

**private** BasicActionReceiver actionReceiver;

**private** BasicActionReceiver actionReceiveRepeated;

@Before

**public** **void** setUp() **throws** Exception {

mngr = EventManagerFactory.*getInstance*();

eventSource1 = **new** BasicEventSource(*EVENTO1*);

eventSource2 = **new** BasicEventSource(*EVENTO2*);

eventSource3 = **new** BasicEventSource(*EVENTO3*);

eventSource1.addListener(mngr);

eventSource2.addListener(mngr);

eventSource3.addListener(mngr);

actionReceiver = **new** BasicActionReceiver();

actionReceiveRepeated = **new** BasicActionReceiver();

}

@After

**public** **void** tearDown() {

// nada por ahora...

}

@Test

**public** **void** testBasicContext() **throws** exceptionRegisterEvent {

// Registramos en el Manager la accion - evento

List<Event>events = **new** ArrayList<Event>();

events.add(**new** BasicEvent(*EVENTO1*));

events.add(**new** BasicEvent(*EVENTO2*));

events.add(**new** BasicEvent(*EVENTO3*));

**try**{

mngr.register(**new** BasicActionCommand(actionReceiver), events);

}**catch**(exceptionRegisterEvent e){

System.*out*.println(e.toString());

}

// El Source dispara el Evento...

eventSource1.triggerEvent();

eventSource2.triggerEvent();

eventSource3.triggerEvent();

// Y si todo funciona bien, el Receiver deberia haber sufrido

// el cambio de estado, por la accion ejecutada...

*assertTrue*(actionReceiver.getState());

//Eventos Repetidos

List<Event>eventsRepeated = **new** ArrayList<Event>();

eventsRepeated.add(**new** BasicEvent(*EVENTO1*));

eventsRepeated.add(**new** BasicEvent(*EVENTO2*));

eventsRepeated.add(**new** BasicEvent(*EVENTO1*));

**try**{

mngr.register(**new** BasicActionCommand(actionReceiveRepeated), eventsRepeated);

}**catch**(exceptionRegisterEvent e){

System.*out*.println(e.toString());

}

eventSource1.triggerEvent();

eventSource2.triggerEvent();

*assertTrue*(actionReceiveRepeated.getState());

}

}

**public** **class** MultipleEventsWithCancelablesTest {

**private** **static** **final** String *EVENTO1* = "evento 1";

**private** **static** **final** String *EVENTO2* = "evento 2";

**private** **static** **final** String *EVENTO3* = "evento 3";

**private** **static** **final** String *EVENTO4* = "evento 4";

**private** EventManager mngr;

**private** BasicEventSource eventSource1;

**private** BasicEventSource eventSource2;

**private** BasicEventSource eventSource3;

**private** BasicEventSource eventSource4;

**private** BasicActionReceiver actionReceiver;

@Before

**public** **void** setUp() **throws** Exception {

mngr = EventManagerFactory.*getInstance*();

eventSource1 = **new** BasicEventSource(*EVENTO1*);

eventSource2 = **new** BasicEventSource(*EVENTO2*);

eventSource3 = **new** BasicEventSource(*EVENTO3*);

eventSource4 = **new** BasicEventSource(*EVENTO4*);

eventSource1.addListener(mngr);

eventSource2.addListener(mngr);

eventSource3.addListener(mngr);

eventSource4.addListener(mngr);

actionReceiver = **new** BasicActionReceiver();

}

@After

**public** **void** tearDown() {

// nada por ahora...

}

@Test

**public** **void** testBasicContext() **throws** exceptionRegisterEvent {

// Registramos en el Manager la accion - evento

List<Event>events = **new** ArrayList<Event>();

events.add(**new** BasicEvent(*EVENTO1*));

events.add(**new** BasicEvent(*EVENTO2*));

events.add(**new** BasicEvent(*EVENTO3*));

**try**{

mngr.register(**new** BasicActionCommand(actionReceiver), events);

mngr.registerCancellables(**new** BasicEvent(*EVENTO4*),**new** BasicEvent(*EVENTO1*));

}**catch**(exceptionRegisterEvent e){

System.*out*.println(e.toString());

}

// El Source dispara el Evento...

eventSource1.triggerEvent();

eventSource3.triggerEvent();

eventSource4.triggerEvent();

eventSource2.triggerEvent();

// Y si todo funciona bien, el Receiver no deberia haber sufrido

// el cambio de estado, por la ejecucion del evento cancelable que deberia desactivar event1

*assertFalse*(actionReceiver.getState());

}

}

**public** **class** MultipleEventsWithOrderAndCancelablesTest {

**private** **static** **final** String *EVENTO1* = "evento 1";

**private** **static** **final** String *EVENTO2* = "evento 2";

**private** **static** **final** String *EVENTO3* = "evento 3";

**private** **static** **final** String *EVENTO4* = "evento 4";

**private** EventManager mngr;

**private** BasicEventSource eventSource1;

**private** BasicEventSource eventSource2;

**private** BasicEventSource eventSource3;

**private** BasicEventSource eventSource4;

**private** BasicActionReceiver actionReceiver;

@Before

**public** **void** setUp() **throws** Exception {

mngr = EventManagerFactory.*getInstance*();

eventSource1 = **new** BasicEventSource(*EVENTO1*);

eventSource2 = **new** BasicEventSource(*EVENTO2*);

eventSource3 = **new** BasicEventSource(*EVENTO3*);

eventSource4 = **new** BasicEventSource(*EVENTO4*);

eventSource1.addListener(mngr);

eventSource2.addListener(mngr);

eventSource3.addListener(mngr);

eventSource4.addListener(mngr);

actionReceiver = **new** BasicActionReceiver();

}

@After

**public** **void** tearDown() {

// nada por ahora...

}

@Test

**public** **void** testBasicContext() **throws** exceptionRegisterEvent {

// Registramos en el Manager la accion - evento

List<Event>events = **new** ArrayList<Event>();

events.add(**new** BasicEvent(*EVENTO1*));

events.add(**new** BasicEvent(*EVENTO2*));

events.add(**new** BasicEvent(*EVENTO3*));

**try**{

mngr.registerWithOrder(**new** BasicActionCommand(actionReceiver), events);

mngr.registerCancellables(**new** BasicEvent(*EVENTO4*),**new** BasicEvent(*EVENTO1*));

}**catch**(exceptionRegisterEvent e){

System.*out*.println(e.toString());

}

// El Source dispara el Evento...

eventSource1.triggerEvent();

eventSource2.triggerEvent();

//Deberia borrar el estado de event1 y event2

eventSource4.triggerEvent();

eventSource1.triggerEvent();

eventSource3.triggerEvent();

// Y si todo funciona bien, el Receiver no deberia haber sufrido

// el cambio de estado, deberia estar solo activado event1

*assertFalse*(actionReceiver.getState());

eventSource2.triggerEvent();

eventSource3.triggerEvent();

// Y si todo funciona bien, el Receiver deberia haber sufrido

// el cambio de estado

*assertTrue*(actionReceiver.getState());

}

}

**public** **class** MultipleEventsWithOrderTest {

**private** **static** **final** String *EVENTO1* = "evento 1";

**private** **static** **final** String *EVENTO2* = "evento 2";

**private** **static** **final** String *EVENTO3* = "evento 3";

**private** EventManager mngr;

**private** BasicEventSource eventSource1;

**private** BasicEventSource eventSource2;

**private** BasicEventSource eventSource3;

**private** BasicActionReceiver actionReceiver;

**private** BasicActionReceiver actionReceiveRepeated;

@Before

**public** **void** setUp() **throws** Exception {

mngr = EventManagerFactory.*getInstance*();

eventSource1 = **new** BasicEventSource(*EVENTO1*);

eventSource2 = **new** BasicEventSource(*EVENTO2*);

eventSource3 = **new** BasicEventSource(*EVENTO3*);

eventSource1.addListener(mngr);

eventSource2.addListener(mngr);

eventSource3.addListener(mngr);

actionReceiver = **new** BasicActionReceiver();

actionReceiveRepeated = **new** BasicActionReceiver();

}

@After

**public** **void** tearDown() {

// nada por ahora...

}

@Test

**public** **void** testBasicContext() **throws** exceptionRegisterEvent {

// Registramos en el Manager la accion - evento

List<Event>events = **new** ArrayList<Event>();

events.add(**new** BasicEvent(*EVENTO1*));

events.add(**new** BasicEvent(*EVENTO2*));

events.add(**new** BasicEvent(*EVENTO3*));

**try**{

mngr.registerWithOrder(**new** BasicActionCommand(actionReceiver), events);

}**catch**(exceptionRegisterEvent e){

System.*out*.println(e.toString());

}

// El Source dispara el Evento...

eventSource1.triggerEvent();

eventSource3.triggerEvent();

eventSource2.triggerEvent();

eventSource3.triggerEvent();

// Y si todo funciona bien, el Receiver deberia haber sufrido

// el cambio de estado, por la accion ejecutada...

*assertTrue*(actionReceiver.getState());

// Inicializo el estado nuevamente

actionReceiver.setState(**false**);

//Corroboro que los estados de los eventos no hayan quedado en true

eventSource1.triggerEvent();

*assertFalse*(actionReceiver.getState());

eventSource2.triggerEvent();

eventSource3.triggerEvent();

*assertTrue*(actionReceiver.getState());

// Inicializo el estado nuevamente

actionReceiver.setState(**false**);

eventSource2.triggerEvent();

eventSource3.triggerEvent();

eventSource1.triggerEvent();

//Corroboro que si los eventos no se dan en ese orden no se ejecuta el comando

*assertFalse*(actionReceiver.getState());

//Eventos Repetidos

// Registramos en el Manager la accion - evento

List<Event>eventsRepeated = **new** ArrayList<Event>();

eventsRepeated.add(**new** BasicEvent(*EVENTO1*));

eventsRepeated.add(**new** BasicEvent(*EVENTO2*));

eventsRepeated.add(**new** BasicEvent(*EVENTO1*));

**try**{

mngr.registerWithOrder(**new** BasicActionCommand(actionReceiveRepeated), eventsRepeated);

}**catch**(exceptionRegisterEvent e){

System.*out*.println(e.toString());

}

eventSource1.triggerEvent();

eventSource2.triggerEvent();

*assertFalse*(actionReceiveRepeated.getState());

eventSource1.triggerEvent();

*assertTrue*(actionReceiveRepeated.getState());

}

}

**public** **class** NullTriggerEvent {

**private** **static** **final** String *EVENTO1* = "evento 1";

**private** EventManager mngr;

**private** BasicEventSource eventSource1;

@Before

**public** **void** setUp() **throws** Exception {

mngr = EventManagerFactory.*getInstance*();

eventSource1 = **new** BasicEventSource(*EVENTO1*);

eventSource1.addListener(mngr);

}

@After

**public** **void** tearDown() {

// nada por ahora...

}

@Test

**public** **void** testBasicContext() **throws** exceptionRegisterEvent {

// Registramos en el Manager la accion - evento

List<Event>events = **null**;

Event eventTest = **null**;

**try**{

mngr.registerWithOrder(**null**, events);

*fail*("Excepted exeptionRegisterEvent");

}**catch**(exceptionRegisterEvent e){

System.*out*.println(e.toString());

}

//Se verifica la validación del registro de eventos con orden

**try**{

mngr.registerWithOrder(**null**, **null**);

*fail*("Excepted exeptionRegisterEvent");

}**catch**(exceptionRegisterEvent e){

System.*out*.println(e.toString());

}

//Se verifica la validación del registro de eventos sin orden

**try**{

mngr.register(**null**, events);

*fail*("Excepted exeptionRegisterEvent");

}**catch**(exceptionRegisterEvent e){

System.*out*.println(e.toString());

}

//Se verifica la validación del registro de evento simple

**try**{

mngr.register(**null**, eventTest);

*fail*("Excepted exeptionRegisterEvent");

}**catch**(exceptionRegisterEvent e){

System.*out*.println(e.toString());

}

//Se verifica la validación de disparo de evento nulo

**try**{

eventSource1.triggerEventNull();

*fail*("Excepted exeptionRegisterEvent");

}

**catch**(Exception e){

System.*out*.println(e.toString());

}

}

}

**public** **class** BombaDeAguaTest {

**private** **static** EventManager *mngr*;

**private** TanqueDeAgua tanque;

**private** MedidorDePresionDeAgua medidor;

**private** BombaDeAgua bomba;

@BeforeClass

**public** **static** **void** setUpBeforeClass() **throws** Exception {

*mngr* = EventManagerFactory.*getInstance*();

}

@AfterClass

**public** **static** **void** tearDownAfterClass() **throws** Exception {

}

@Before

**public** **void** setUp() **throws** Exception {

// Siempre reseteamos el Manager:

*mngr*.reset();

// Tenemos una bomba de agua:

bomba = **new** BombaDeAgua();

// (inicia apagada)

// Tenemos un tanque de agua

// con capacidad de 10.000 litros

tanque = **new** TanqueDeAgua(**new** BigDecimal(10000));

// Va a estar lleno cuando llegue a su 95% de capacidad:

tanque.setLimiteSuperior(**new** BigDecimal(9500));

// Y va a estar vacio cuando tenga menos de 100 litros:

tanque.setLimiteInferior(**new** BigDecimal(100));

// (el tanque inicia vacio)

// Tenemos un Medidor de Presion de Agua

// conectado a la Red de Abastecimiento

medidor = **new** MedidorDePresionDeAgua(RedDeAbastecimientoDeAgua.*instance*());

// Y vamos conectando los objetos al Manejador de Sucesos...

// El Tanque y el Medidor son Disparadores de Sucesos

// El Manejador escuchara los Sucesos

tanque.addEventListener(*mngr*);

medidor.addEventListener(*mngr*);

// La Red de Agua TAMBIEN es disparador de Sucesos:

RedDeAbastecimientoDeAgua.*instance*().addListener(*mngr*);

// Y entonces "enchufamos" las acciones con sus eventos...

registrarAccionesYEventos();

// LISTO PARA LA ACCION!

}

**private** **void** registrarAccionesYEventos() **throws** exceptionRegisterEvent {

// Cuando cambia la presion de la Red (evento),

// el medidor debe testear la presion (action)

*mngr*.register(**new** ControlarPresionCmd(medidor),

**new** CambioDePresionDeAguaEvent());

// Cuando el Medidor detecta que hay presion y ademas

// el tanque indica que esta vacio (2 eventos)

// entonces la Bomba debe encenderse:

List<Event> eventosParaEncenderLaBomba = **new** ArrayList<Event>(2);

eventosParaEncenderLaBomba.add(**new** HayPresionEvent(medidor));

eventosParaEncenderLaBomba.add(**new** TanqueVacioEvent(tanque));

*mngr*.register(**new** EncenderBombaCmd(bomba), eventosParaEncenderLaBomba);

// Ahora, si no hay presion, o bien si el tanque se llena

// entonces la bomba debe apagarse:

*mngr*.register(**new** ApagarBombaCmd(bomba), **new** NoHayPresionEvent(medidor));

*mngr*.register(**new** ApagarBombaCmd(bomba), **new** TanqueLlenoEvent(tanque));

// Pot ultimo, indicamos los Eventos Cancelables entre si

// Si el Tanque esta lleno, entonces no esta vacio

*mngr*.registerCancellables(**new** TanqueLlenoEvent(tanque), **new** TanqueVacioEvent(tanque));

*mngr*.registerCancellables(**new** TanqueVacioEvent(tanque), **new** TanqueLlenoEvent(tanque));

// Y si el Medidor detecta que Hay presion, entonces Deja de "no haber presion"

*mngr*.registerCancellables(**new** HayPresionEvent(medidor), **new** NoHayPresionEvent(medidor));

*mngr*.registerCancellables(**new** NoHayPresionEvent(medidor), **new** HayPresionEvent(medidor));

}

@After

**public** **void** tearDown() **throws** Exception {

}

**private** **void** printEstado(String str1, String str2){

System.*out*.println("");

**if**(str1.length() >0)

System.*out*.println("Acción: "+str1);

**else** System.*out*.println("Estado Inicial:");

System.*out*.println("Limite inferior del tanque: "+tanque.getLimiteInferior());

System.*out*.println("Limite superior del tanque: "+tanque.getLimiteSuperior());

System.*out*.println("Litros en el tanque: "+tanque.getContenido());

System.*out*.println("Presión Mínima: "+medidor.getPresionMinima());

System.*out*.println("Presión: "+RedDeAbastecimientoDeAgua.*instance*().getPresionActual());

**if**(str2.length() >0)

System.*out*.println("Resultado: "+str2);

}

@Test

**public** **void** testBombaDeAguaTanqueLlenoYPresion() {

// La presion minima para detectar presion sera 50 PSI

medidor.setPresionMinima(**new** BigDecimal(50));

printEstado("","");

*assertTrue*(! bomba.isEncendida());

*assertTrue*(tanque.isTanqueVacio());

// llenamos el tanque un poquito...

tanque.llenar(**new** BigDecimal(80));

// El tanque no se lleno, y la bomba sigue apagada

*assertTrue*(! tanque.isTanqueLleno());

*assertTrue*(! bomba.isEncendida());

*assertTrue*(tanque.isTanqueVacio());

printEstado("Lleno un ponco el tanque","El tanque sigue con bajo caudal pero no hay presión para encender la bomba.");

// Damos un poquito de presion a la Red:

RedDeAbastecimientoDeAgua.*instance*().setPresionActual(**new** BigDecimal(10));

// El medidor aun no detecto presion y la bomba sigue apagada:

*assertTrue*(! medidor.hayPresion());

*assertTrue*(! bomba.isEncendida());

printEstado("Subo un poco la presión","El tanque esta vacio y la presión sigue siendo insuficiente para encender la bomba.");

// Ahora le damos suficiente presion a la Red,

// cosa que arranque la bomba:

RedDeAbastecimientoDeAgua.*instance*().setPresionActual(**new** BigDecimal(110));

// El medidor debe haber detectado la presion

// y la bomba DEBE HABER ENCENDIDO:

printEstado("Subo la presión","Enciendo la bomba");

*assertTrue*(tanque.isTanqueVacio());

*assertTrue*(medidor.hayPresion());

*assertTrue*(bomba.isEncendida());

}

}

**Código fuente Dominio de Ejemplo Bomba de Agua:**

**Indice:**

**Implementaciones Páginas**

class ApagarBombaCmd 30

abstract class BombaCmd 30

class BombaDeAgua 30

class CambioDePresionDeAguaEvent 30

class ControlarPresionCmd 31

class EncenderBombaCmd 31

class HayPresionEvent 31

class MedidorDePresionDeAgua 32

class NoHayPresionEvent 33

class RedDeAbastecimientoDeAgua 34

class TanqueDeAgua 35

class TanqueLlenoEvent 37

class TanqueVacioEvent 37

**public** **class** ApagarBombaCmd **extends** BombaCmd {

**public** ApagarBombaCmd(BombaDeAgua b) {

**super**(b);

}

**public** **void** execute() {

bomba.apagar();

}

}

**public** **abstract** **class** BombaCmd **implements** ActionCommand {

**protected** BombaDeAgua bomba;

**public** BombaCmd(BombaDeAgua b) {

**this**.bomba = b;

}

}

**public** **class** BombaDeAgua {

**private** Boolean estado = **false**;

**public** **void** encender() {

estado = **true**;

}

**public** **void** apagar() {

estado = **false**;

}

**public** Boolean isEncendida() {

**return** estado;

}

}

**public** **class** CambioDePresionDeAguaEvent **implements** Event {

**public** **boolean** equals(Event anotherEvent) {

**if** (!(anotherEvent **instanceof** CambioDePresionDeAguaEvent)) {

**return** **false**;

}

// Un cambio de presion es siempre lo mismo...

**return** **true**;

}

}

**public** **class** ControlarPresionCmd **implements** ActionCommand {

**private** MedidorDePresionDeAgua medidor;

**public** ControlarPresionCmd(MedidorDePresionDeAgua medidor) {

**this**.medidor = medidor;

}

**public** **void** execute() {

**this**.medidor.controlarPresion();

}

}

**public** **class** EncenderBombaCmd **extends** BombaCmd {

**public** EncenderBombaCmd(BombaDeAgua b) {

**super**(b);

}

**public** **void** execute() {

bomba.encender();

}

}

**public** **class** HayPresionEvent **implements** Event {

**private** MedidorDePresionDeAgua medidor;

**public** HayPresionEvent(MedidorDePresionDeAgua medidor) {

**this**.medidor = medidor;

}

@Override

**public** **boolean** equals(Event anotherEvent) {

**if** (!(anotherEvent **instanceof** HayPresionEvent)) {

**return** **false**;

}

**if** (**this**.medidor.equals(((HayPresionEvent)anotherEvent).medidor)) {

**return** **true**;

}

**return** **false**;

}

}

**public** **class** MedidorDePresionDeAgua {

**private** List<EventListener> listeners = **new** ArrayList<EventListener>();

/\*\*

\* indicador de cual es la presion minima en PSI

\* Si la presion esta por debajo de este indice,

\* El medidor indicara que NO HAY PRESION

\*

\* Valor default 100 PSI

\*/

**private** BigDecimal presionMinima = **new** BigDecimal(100);

/\*\*

\* La red de abastecimiento a medir

\*/

**private** RedDeAbastecimientoDeAgua red;

**public** MedidorDePresionDeAgua(RedDeAbastecimientoDeAgua red) {

**this**.red = red;

}

**public** **void** addEventListener(EventListener eventListener) {

**this**.listeners.add(eventListener);

}

**public** **void** setPresionMinima(BigDecimal presionMinima) {

**this**.presionMinima = presionMinima;

}

**public** BigDecimal getPresionMinima() {

**return** presionMinima;

}

**public** Boolean hayPresion() {

**return** (red.getPresionActual().compareTo(presionMinima) >= 0);

}

/\*\*

\* Cuando nos piden controlar la presion

\* Nos fijamos cual es la presion de la red

\* E informamos si hay presion o no, en base al indicador

\* de presionMinima

\*/

**public** **void** controlarPresion() {

Event e;

**if** (hayPresion()) {

e = **new** HayPresionEvent(**this**);

} **else** {

e = **new** NoHayPresionEvent(**this**);

}

**for** (EventListener listener : listeners) {

listener.eventOccurred(e);

}

}

}

**public** **class** NoHayPresionEvent **implements** Event {

**private** MedidorDePresionDeAgua medidor;

**public** NoHayPresionEvent(MedidorDePresionDeAgua medidor) {

**this**.medidor = medidor;

}

@Override

**public** **boolean** equals(Event anotherEvent) {

**if** (!(anotherEvent **instanceof** NoHayPresionEvent)) {

**return** **false**;

}

**if** (**this**.medidor.equals(((NoHayPresionEvent)anotherEvent).medidor)) {

**return** **true**;

}

**return** **false**;

}

}

**public** **class** RedDeAbastecimientoDeAgua {

/\*\*

\* La instancia SINGLETON

\*/

**private** **static** RedDeAbastecimientoDeAgua *instance* = **new** RedDeAbastecimientoDeAgua ();

/\*\*

\* Los listeners (clientes de la red)

\*/

**private** List<EventListener> listeners = **new** ArrayList<EventListener>();

/\*\*

\* La presion actual de la Red

\* Se setea por afuera

\*/

**private** BigDecimal presionActual = BigDecimal.*ZERO*;

**private** RedDeAbastecimientoDeAgua() {

}

**public** **static** RedDeAbastecimientoDeAgua instance() {

**return** *instance*;

}

**public** **void** addListener(EventListener listener) {

**this**.listeners.add(listener);

}

**public** **void** setPresionActual(BigDecimal presionActual) {

**this**.presionActual = presionActual;

**for** (Iterator<EventListener> iterator = listeners.iterator(); iterator.hasNext();) {

EventListener listener = iterator.next();

listener.eventOccurred(**new** CambioDePresionDeAguaEvent());

}

}

**public** BigDecimal getPresionActual() {

**return** presionActual;

}

}

**public** **class** TanqueDeAgua {

**private** List<EventListener> listeners = **new** ArrayList<EventListener>();

**private** BigDecimal capacidadMaxima;

**private** BigDecimal contenido;

**private** BigDecimal limiteInferior;

**private** BigDecimal limiteSuperior;

**public** TanqueDeAgua(BigDecimal capacidadMaxima) {

**this**(capacidadMaxima, BigDecimal.*ZERO*, capacidadMaxima);

}

**public** TanqueDeAgua(BigDecimal capacidadMaxima,

BigDecimal limiteInferior,

BigDecimal limiteSuperior) {

// No se realizan validaciones en pos de la simplicidad (es un ejemplo)

**this**.capacidadMaxima = capacidadMaxima;

**this**.limiteInferior = limiteInferior;

**this**.limiteSuperior = limiteSuperior;

**this**.contenido = BigDecimal.*ZERO*;

}

**public** **void** addEventListener(EventListener eventListener) {

**this**.listeners.add(eventListener);

}

**public** **void** llenar(BigDecimal litros) {

contenido =

contenido.add(litros).compareTo(capacidadMaxima) > 0 ?

capacidadMaxima : contenido.add(litros);

**if**(isTanqueLleno()) {

**for** (EventListener listener : listeners) {

listener.eventOccurred(**new** TanqueLlenoEvent(**this**));

}

}**else** **if**(isTanqueVacio()){

**for** (EventListener listener : listeners) {

listener.eventOccurred(**new** TanqueVacioEvent(**this**));

}

}

}

**public** **void** vaciar(BigDecimal litros) {

contenido =

contenido.subtract(litros).compareTo(BigDecimal.*ZERO*) < 0 ?

BigDecimal.*ZERO* : contenido.subtract(litros);

**if**(isTanqueVacio()) {

**for** (EventListener listener : listeners) {

listener.eventOccurred(**new** TanqueVacioEvent(**this**));

}

}**else** **if**(isTanqueLleno()) {

**for** (EventListener listener : listeners) {

listener.eventOccurred(**new** TanqueLlenoEvent(**this**));

}

}

}

**public** **boolean** isTanqueLleno() {

**return** contenido.compareTo(limiteSuperior) >= 0;

}

**public** **boolean** isTanqueVacio() {

**return** contenido.compareTo(limiteInferior) <= 0;

}

**public** BigDecimal getCapacidadMaxima() {

**return** capacidadMaxima;

}

**public** **void** setLimiteInferior(BigDecimal limiteInferior) {

**this**.limiteInferior = limiteInferior;

}

**public** BigDecimal getLimiteInferior() {

**return** limiteInferior;

}

**public** **void** setLimiteSuperior(BigDecimal limiteSuperior) {

**this**.limiteSuperior = limiteSuperior;

}

**public** BigDecimal getLimiteSuperior() {

**return** limiteSuperior;

}

**public** BigDecimal getContenido() {

**return** contenido;

}

}

**public** **class** TanqueLlenoEvent **implements** Event {

**private** TanqueDeAgua tanque;

**public** TanqueLlenoEvent(TanqueDeAgua tanque) {

**this**.tanque = tanque;

}

@Override

**public** **boolean** equals(Event anotherEvent) {

**if** (!(anotherEvent **instanceof** TanqueLlenoEvent)) {

**return** **false**;

}

**if** (**this**.tanque.equals(((TanqueLlenoEvent)anotherEvent).tanque)) {

**return** **true**;

}

**return** **false**;

}

}

**public** **class** TanqueVacioEvent **implements** Event {

**private** TanqueDeAgua tanque;

**public** TanqueVacioEvent(TanqueDeAgua tanque) {

**this**.tanque = tanque;

}

@Override

**public** **boolean** equals(Event anotherEvent) {

**if** (!(anotherEvent **instanceof** TanqueVacioEvent)) {

**return** **false**;

}

**if** (**this**.tanque.equals(((TanqueVacioEvent)anotherEvent).tanque)) {

**return** **true**;

}

**return** **false**;

}

}