La Conquista del Mundo

Informe de pruebas de desarrollo

Director de proyecto
Ricardo Ruedas García

Equipo de pruebas
Jorge Colao Adán
Ángel Durán Izquierdo
Antonio Gómez Poblete
Daniel León Romero
Laura Núñez Villa

9 de marzo de 2011

Índice

1. Clase UserManager

1.1. UserManager::registerUser

Nombre del *tester*	Antonio Gómez Poblete
Fecha de asignación	27 de enero de 2011
Fecha de finalización	30 de enero de 2011
Código bajo prueba	`UserManager::registerUser`

A continuación se detallarán las pruebas de desarrollo (Pruebas unitarias con *Junit*).

Lista de los valores de prueba para cada atributo. El criterio elegido para todos los valores de prueba (test data) ha sido: Añadir valores interesantes propensos a error (Conjetura de error).

- `login`

  null

  Cadena vacía

  Usuario existente: `"JorgeCA"`

  Usuario no existente: `"LuisAn"`

- `passwd`

  null

  Cadena vacía

  Cadenas no vacías: `"jorge"`, `"luis"`

- `email`

  null

  Cadena vacía

  Valores incorrectos: `"jorge"`, `"jorge@"`, `"jorge@gmail"`, `"jorge@gmail."`

  Valores correctos: `"jorge.colao@gmail.com"`, `"luis@gmail.com"`

La estrategia para obtener los casos de prueba elegida ha sido *each choice* (Test 1 a 3 y 8 a 12) y añadir algunos casos interesantes (Test 4 a 7). El test 13 se ha elaborado para ver el comportamiento de registerUser cuando el servidor se ha desconectado.

Para elegir tanto los valores de prueba como los casos de pruebas se ha analizado el comportamiento de este caso de uso (caja blanca). Tras este análisis se ha llegado a la conclusión de que hay en dos lugares en los que

se pueden encontrar errores: Si al método registerUser se le da un parámetro erróneo (throw EmptyStringException, MalformedEmailException, NullPointerException), o si el nombre del usuario ya existe en el servidor (throw UserAlreadyExistsException). La primera comprobación es local, por ello si se lanza la excepcion InvalidArgumentException no se hará nada en el servidor y no se analizará si el usuario ya existe o no.

Además, si observamos las comprobaciones que se hacen en el cliente se ve que si login == null, independientemente del valor de los demás atributos se lanzará NullPointerException, por esa razón es independiente el valor de los demas atributos, lo que hace que muchos casos de prueba no sean necesarios de probar. Por otro lado, para probar qué pasa si el email es *null* el resto de los parámetros deberán ser correctos, lo que añade algunos casos interesantes. Además de con *null* lo comentado en este parrafo sucede con el resto de los parámetros.

Teniendo en cuenta las consideraciones anteriormente mencionadas, la tabla completa de los casos de prueba y los resultados esperados son:

	Valores de Prueba	**Objetivo del test**	**Resultado esperado**
T1	(vacío, vacío, vacío)	login	EmptyStringException
T2	(null, null, null)	login	NullPointerException
T3	(JorgeCA, jorge, jorge.colao@gmail.com)	Usuario ya existente	UserAlreadyExistst
T4	(JorgeCA, vacío, jorge.colao@gmail.com)	passwd	EmptyStringException
T5	(JorgeCA, null, jorge.colao@gmail.com)	passwd	NullPointerException
T6	(JorgeCA, jorge, vacío)	email	EmptyStringException
T7	(JorgeCA, jorge, null)	email	NullPointerException
T8	(JorgeCA, jorge, jorge)	email	MalformedEmailException
T9	(JorgeCA, jorge, jorge@)	email	MalformedEmailException
T10	(JorgeCA, jorge, jorge@gmail)	email	MalformedEmailException
T11	(JorgeCA, jorge, jorge@gmail.)	email	MalformedEmailException
T12	(LuisAn, luis, luis@gmail.com)	email	El usuario queda registrado
T13	(Angel&Duran, angel, a@d.com)	Comportamiento registerUser	RemoteException

1.2. UserManager::login

Nombre del *tester*	Angel Durán Izquierdo
Fecha de asignación	27 de enero de 2011
Fecha de finalización	2 de febrero de 2011
Código bajo prueba	`UserManager::createSession`

A continuación se detallarán las pruebas de desarrollo (Pruebas unitarias con *JUnit*).

Lista de los valores de prueba para cada atributo. El criterio elegido para todos los valores de prueba (test data) ha sido: Añadir valores interesantes propensos a error (Conjetura de error).

- `login`

  null

  Cadena vacía

  Usuario existente: `Angel`

  Usuario no existente: `ADuran`

  Nombre de usuario registrado pero introducido con mayusculas: `ADuran`

  Usuario existente con carácteres especiales: `Angel&Duran`

  Usuario existente con numeros: `1111, -1`

- `passwd`

  null

  Cadena vacía

  Cadenas no vacías: `angel`

  Cadenas con carácteres especiales: `22&22`

  Cadenas con numeros: `2222`

La estrategia para obtener los casos de prueba elegida ha sido *each choice* eligiendo los casos mas interesantes.

Para elegir tanto los valores de prueba como los casos de pruebas se ha analizado el comportamiento de este caso de uso (caja blanca). Tras este análisis se ha llegado a la conclusión de que hay en dos lugares en los que se pueden encontrar errores: Si al método createSession se le da un parámetro erróneo (throw InvalidArgumentException), o si el nombre del usuario no se ha registrado en el servidor (throw UserAlreadyExistsException).

Por otro lado se comprueba si al crear una sesion cuando ya se existía una, la anterior se cierra correctamente.

Teniendo en cuenta las consideraciones anteriormente mencionadas, la tabla completa de los casos de prueba y los resultados esperados son:

	Valores de Prueba	**Objetivo del test**	**Resultado esperado**
Test1	(vacío, vacío)	login-pass	InvalidArgument
Test2	(Aduran, vacío)	pass	InvalidArgument
Test3	(vacío, angel)	login	UserAlreadyExiststst
Test4	(Aduran, angel)	login-pass	Sesión creada correctamente
Test5	(null, angel)	login	InvalidArgument
Test6	(Aduran, null)	pass	InvalidArgument
Test7	(null, null)	login-pass	InvalidArgument
Test8	(ADuran, angel)	login	WrongLoginException
Test9	(Aduran, Angel)	pass	WrongLoginException
Test10	(1111, 22&22)	login-pass	Sesión creada correctamente

	Valores de Prueba	Objetivo del test	Resultado esperado
Test11	(-1, 2222)	login-pass	Sesión creada correctamente
Test12	(Angel&Duran, angel)	login	Sesión creada correctamente
Test13	(Aduran y ricki, angel y ricki)	Comportamiento createSession	Sesión creada correctamente
Test14	(Aduran, angel)	Comportamiento createSession	RemoteException
Test15	(Aduran, angel)	Comportamiento createSession	Sesión cerrada correctamente
Test16	(Aduran, angel)	Comportamiento createSession	Sesión creada correctamente

2. Clase GameManager

Nombre del *tester*	Jorge Colao Adán
	Daniel León Romero
Fecha de asignación	18 de febrero de 2011
Fecha de finalización	21 de febrero de 2011
Código bajo prueba	`GameManager`

En este apartado se hablará sobre las pruebas realizadas con la herramienta *JUnit* sobre esta clase.

La estrategia a seguir para las pruebas de estos apartados será *Each Choice*. La elección de los valores de los parámetos, se realizarán con valores límite para los atributos númericos y valores propensos a error para el resto.

2.1. GameManager::updateGameList

En este primer método como no tiene ningún parámetro lo que tenemos que comprobar es que al hacer el test con *JUnit* no salte ninguna excepción. Además, comprobamos que inicialmente el número de partidas a las que se está unido y las partidas disponibles para unirse es igual a cero, y que después de la ejecución de este caso de uso es distinto de cero.

Debido a que con la herramienta *JUnit* no podemos alcanzar una cobertura alta, lo que haremos será hacer pruebas exploratorias.

2.2. GameManager::createGame

Este método tiene seis parámetros de entrada, por lo que tenemos que utilizar alguna estrategia de generación de casos de prueba. Hemos utilizado *Each Choice* con la herramienta de la página `http://161.67.140.42/CombTestWeb/`. Pero comprobando las combinaciones que hemos obtenido nos hemos dado cuenta de que eran poco exahustivas (Test 1 a 3). Debido a este motivo, hemos añadido otros casos de prueba interesantes (Test 4 a 12), como son; probar para que falle cada parámetro independientemente.

Para elegir tanto los valores de prueba como los casos de pruebas de

los casos interesantes se ha analizado el comportamiento de este método (mediante caja blanca). Después de analizar el método nos hemos dado cuenta de que pude haber los siguientes errores:

- NullPointerException
- EmptyStringException
- NegativeValueException

Esta comprobación es local, por ello si se lanza alguna excepción de las anteriores, no se creará la partida en el servidor.

- `name`

  null

  Cadena vacía

  Cadena válida: `"partida"`

- `description`

  null

  Cadena vacía

  Cadena válida: `"partida guerra mundo"`

- `gameSession`

  null

  Fecha: `"Hora actual"` y `"01/Marzo/2012 a las 14:00"`

- `turnTime`

  Número incorrecto: `"0"`

  Número correcto. `"1"` y `"112"`

- `defTime`

  Número incorrecto: `"0"`

  Número correcto. `"1"` y `"20"`

- `negTime`

  Número incorrecto: `"0"`

  Número correcto. `"1"` y `"33"`

La única comprobación especial que podemos hacer es mirar si salta alguna excepción, si no salta ninguna excepción el método estará correcto, lo que creará una partida en el servidor.

Esta es la tabla de casos de prueba y resultados esperados.

	Valores de Prueba	Objetivo del test	Resultado esperado
Test1	(”partida”, ”partida guerra mundo”, Fecha_Posterior, 112, 1, 1)	crear partida	Funcinamiento Correcto
Test2	(null, null, Fecha_Actual, 1, 20, 33)	name, description	NullPointer
Test3	(vacío, vacío, null, 0, 0, 0)	gameSession	NullPointer
Test4	(vacío, vacío, Fecha_Posterior, 1, 1, 1)	name	EmptyString
Test5	(null, ”partida guerra mundo”, Fecha_Posterior, 1, 1, 1)	name	NullPointer
Test6	(”partida”, null, Fecha_Actual, 1, 1, 33)	description	NullPointer
Test7	(”partida”, ”partida guerra mundo”, null, 1, 1, 33)	gameSession	NullPointer
Test8	(”partida”, ”partida guerra mundo”, Fecha_Actual, 0, 20, 33)	turnTime	NegativeValue
Test9	(”partida”, ”partida guerra mundo”, Fecha_Actual, 112, 0, 33)	defTime	NegativeValue
Test10	(”partida”, ”partida guerra mundo”, Fecha_Actual, 112, 20, 0)	negTime	NegativeValue
Test11	(”partida”, ”partida guerra mundo”, Fecha_Actual, 112, 20, 33)	crear partida	Funcinamiento Correcto
Test12	(”partida”, vacío, Fecha_Posterior, 1, 1, 1)	crear partida	Funcinamiento Correcto

2.3. GameManager::joinGame

Para este caso de uso tan sólo tenemos un parámetro que es el número del juego al que queremos unirnos. Por este motivo tenemos que comprobar los valores límite de la lista de partidas a unirse. Para representar los valores límite, hemos usado de límite inferior los números ”–1” y ”0”, ya que la comparación es que sea menor de ”0”. Y para el límite superior usaremos el tope de partidas disponibles en el servidor y el tope más uno. En nuestro servidor tenemos dos partidas a las que nos podemos unir, por lo tanto, el tope sería “1” y el tope más uno sería “2”.

- `gameSelected`

  Límite inferior: `"-1"` y `"0"`

  Límite superior: `"tope = 1"` y `"tope + 1 = 2"`

También tenemos que comprobar que antes de unirnos a la partida tiene que haber al menos una partida a la que podamos unirnos. Y después de unirnos comprobar que el número de filas en la lista de partidas actuales es mayor que antes y la lista de partidas para unirme es menor. Las posobles excepciones que puede tener este método son:

- ArrayIndexOutOfBoundsException, para números por debajo del rango.
- IndexOutOfBoundsException, pra números por encima del rango.

Esta es la tabla de casos de prueba y resultados esperados.

	Valores de Prueba	Objetivo del test	Resultado esperado
Test1	(-1)	gameIndex (límite inferior)	ArrayIndexOutOfBounds
Test2	(1)	seleccionar segunda partida	Funcionamiento correcto
Test3	(0)	seleccionar primera partida	Funcionamiento correcto
Test4	(2)	gameIndex (límite superior)	IndexOutOfBounds

2.4. GameManager::connectToGame

En este método tenemos dos parámetros, uno es el número de la partida seleccionada para jugar y el otro un evento, que para poder realizar la pruebas tenemos que crearnos una clase privada que implemente *GameEventListener*. De esta forma, nos centramos en el primer parámetro que es el interesante. Como sólo tenemos un parámetro al igual que antes tenemos que sacar los valores límite. En esta ocasión en el servidor tenemos una única partida actual, por lo que el tope es "0" y el tope más uno es "1".

- `gameSelected`

  Límite inferior: `"-1"` y `"0"`

  Límite superior: `"tope = 0"` y `"tope + 1 = 1"`

Además, antes de conectar tenemos que comprobar que almenos tengamos una partida a la cual podamos conectarnos para jugar y que el objeto GameEngine esté a null y que después de ejecutar el conectar no sea null.

Esta es la tabla de casos de prueba y resultados esperados.

	Valores de Prueba	Objetivo del test	Resultado esperado
Test1	(-1, new TestGameEventListener())	gameIndex (límite inferior)	ArrayIndexOutOfBounds
Test2	(0, new TestGameEventListener())	seleccionar partida	Funcionamiento correcto
Test3	(1, new TestGameEventListener())	gameIndex (límite superior)	IndexOutOfBounds
Test4	(0, null)	gameListener	NullPointer

2.5. GameManager::disconnectFromGame

Este método no tiene ningún parámetro de entrada, por lo que para comprobar que funciona correctamente. Tenemos que conectarnos previamante a una partida y luego ejecutar este método, si se lanza la excepción *NullPoin-*

terExcepcition, la ejecución fallará. En caso contrario el método funciona correctamente.

3. Clase GameListModel

3.1. GameListModel::getColumnCount

Nombre del *tester*	Ángel Durán Izquierdo
Fecha de asignación	21 de febrero de 2011
Fecha de finalización	22 de febrero de 2011
Código bajo prueba	`GameListModel::getColumnCount`

A continuación se detallarán las pruebas de desarrollo (Pruebas unitarias con *Junit*).

Este método no tiene ningún parametro por lo que solo se prueba que devuelve el número correcto de columnas, este dato esta fijado en la clase y no es variable. A su vez se comprueba que no se produce ninguna excepción durante la realización de las pruebas.

3.2. GameListModel::getColumnName

Nombre del *tester*	Ángel Durán Izquierdo
Fecha de asignación	21 de febrero de 2011
Fecha de finalización	22 de febrero de 2011
Código bajo prueba	`GameListModel::getColumnName`

A continuación se detallarán las pruebas de desarrollo (Pruebas unitarias con *JUnit*).

Lista de los valores de prueba para cada atributo. El criterio elegido para todos los valores de prueba (test data) ha sido: Añadir valores interesantes propensos a error (Conjetura de error).

Al método se le pasa un entero (col) que indica la columna de la cual queremos recuperar el nombre.

- `col`

  Valor correcto de columna: 0

  Valor negativo: -1

  Valor positivo pero no existe coluna: 6

La estrategia para obtener los casos de prueba elegida ha sido *each choice*.

La tabla completa de los casos de prueba y los resultados esperados son:

	Valores de Prueba	Objetivo del test	Resultado esperado
Test1	(0)	col	Devuelve el dato correcto
Test2	(-1)	col	Excepción
Test3	(6)	col	Excepción

3.3. GameListModel::getGameAt

Nombre del *tester*	Ángel Durán Izquierdo
Fecha de asignación	21 de febrero de 2011
Fecha de finalización	22 de febrero de 2011
Código bajo prueba	`GameListModel::getGameAt`

A continuación se detallarán las pruebas de desarrollo (Pruebas unitarias con *JUnit*).

Lista de los valores de prueba para cada atributo. El criterio elegido para todos los valores de prueba (test data) ha sido: Añadir valores interesantes propensos a error (Conjetura de error).

Al método se le pasa un entero (gameSelected) que indica la posición del juego que se quiere recuperar.

- `col`

  Valor correcto de posición del juego: 0

  Valor positivo pero no existe el juego: 20

La estrategia para obtener los casos de prueba elegida ha sido *each choice*.

La tabla completa de los casos de prueba y los resultados esperados son:

	Valores de Prueba	Objetivo del test	Resultado esperado
Test1	(0)	gameSelected	Devuelve el juego
Test2	(20)	gameSelected	Excepción

3.4. GameListModel::setData

Nombre del *tester*	Ángel Durán Izquierdo
Fecha de asignación	21 de febrero de 2011
Fecha de finalización	22 de febrero de 2011
Código bajo prueba	`GameListModel::setData`

A continuación se detallarán las pruebas de desarrollo (Pruebas unitarias con *JUnit*).

Lista de los valores de prueba para cada atributo. El criterio elegido para todos los valores de prueba (test data) ha sido: Añadir valores interesantes propensos a error (Conjetura de error).

Al método se le pasa una lista de juegos (data).

- `data`

  Lista correcta de juegos: lista de juegos

  Valor nulo: null

La estrategia para obtener los casos de prueba elegida ha sido *each choice*.

La tabla completa de los casos de prueba y los resultados esperados son:

	Valores de Prueba	**Objetivo del test**	**Resultado esperado**
Test1	lista de juegos	data	Devuelve el juego
Test2	null	data	InvalidArgument

3.5. GameListModel::getRowCount

Nombre del *tester*	Ángel Durán Izquierdo
Fecha de asignación	21 de febrero de 2011
Fecha de finalización	22 de febrero de 2011
Código bajo prueba	`GameListModel::getRowCount`

A continuación se detallarán las pruebas de desarrollo (Pruebas unitarias con *JUnit*).

Este método devuelve el número de juegos disponibles, no recibe ningún valor por lo que solo se ha realizado la para comprobar que se devuelve el número correcto de juegos.

3.6. GameListModel::getValueAt

Nombre del *tester*	Ángel Durán Izquierdo
Fecha de asignación	21 de febrero de 2011
Fecha de finalización	22 de febrero de 2011
Código bajo prueba	`GameListModel::getValueAt`

A continuación se detallarán las pruebas de desarrollo (Pruebas unitarias con *JUnit*).

Lista de los valores de prueba para cada atributo. El criterio elegido para todos los valores de prueba (test data) ha sido: Añadir valores interesantes propensos a error (Conjetura de error).

Al método se le pasan dos variables, la primera indica el juego del cual se quiere recuperar la información (rowIndex). El segundo parametro indica que tipo de información debe devolver el método (columnIndex).

- `rowIndex`

  Existe la fila: 0,1

  No existe la fila: 10

- `columnIndex`

  Existe el parametro a devolver: 0,1,2

  No existe el parametro a devolver: 10

La estrategia para obtener los casos de prueba elegida ha sido *pair wise* eligiendo después los casos interesantes. Los test 5 y 6 son redundantes pero se han elegido para conseguir una covertura mayor del código.

La tabla completa de los casos de prueba y los resultados esperados son:

	Valores de Prueba	**Objetivo del test**	**Resultado esperado**
Test1	(0,0)	rowIndex y columnIndex	Devuelve el atributo nombre
Test2	(1,10)	columnIndex	No devuelve nada
Test3	(10,1)	rowIndex	Devuelve el juego
Test4	(10,10)	rowIndex y columnIndex	No devuelve nada
Test5	(1,1)	rowIndex y columnIndex	Devuelve el juego
Test6	(0,2)	rowIndex y columnIndex	Devuelve el número de jugadores

4. Clase PlayerListModel

4.1. PlayerListModel::PlayerListModel

Nombre del *tester*	Ángel Durán Izquierdo
Fecha de asignación	21 de febrero de 2011
Fecha de finalización	22 de febrero de 2011
Código bajo prueba	`PlayerListModel::PlayerListModel`

A continuación se detallarán las pruebas de desarrollo (Pruebas unitarias con *JUnit*).

Lista de los valores de prueba para cada atributo. El criterio elegido para todos los valores de prueba (test data) ha sido: Añadir valores interesantes propensos a error (Conjetura de error).

Al constructor se le pasa dos parametros, el primero es el player propio (selfPlayer) y el segundo es una lista de usuarios (data).

- `selfPlayer`

  Valor correcto: owner

  Valor nulo: null

- `data`

  Valor correcto: lista de jugadores

  Valor nulo: null

La estrategia para obtener los casos de prueba elegida ha sido *pair wise* seleccionando las combinaciones interesantes.

La tabla completa de los casos de prueba y los resultados esperados son:

	Valores de Prueba	**Objetivo del test**	**Resultado esperado**
Test1	(owner, lista de jugadores)	selfPlayer y data	El objeto se crea correctamente
Test2	(null, null)	selfPlayer y data	NullPointerException

4.2. PlayerListModel::getColumnName

Nombre del *tester*	Ángel Durán Izquierdo
Fecha de asignación	21 de febrero de 2011
Fecha de finalización	22 de febrero de 2011
Código bajo prueba	`PlayerListModel::getColumnName`

A continuación se detallarán las pruebas de desarrollo (Pruebas unitarias con *JUnit*).

Lista de los valores de prueba para cada atributo. El criterio elegido para todos los valores de prueba (test data) ha sido: Añadir valores interesantes propensos a error (Conjetura de error).

Al método se le pasa un entero (col) que indica la columna de la cual queremos recuperar el nombre.

- `col`

  Valor correcto de columna: 1

  Valor de coluna invalido: 5

La estrategia para obtener los casos de prueba elegida ha sido *each choice*.

La tabla completa de los casos de prueba y los resultados esperados son:

	Valores de Prueba	**Objetivo del test**	**Resultado esperado**
Test1	(1)	col	Devuelve el dato correcto
Test2	(5)	col	Excepción

4.3. PlayerListModel::getPlayerAt

Nombre del *tester*	Ángel Durán Izquierdo
Fecha de asignación	21 de febrero de 2011
Fecha de finalización	22 de febrero de 2011
Código bajo prueba	`PlayerListModel::getPlayerAt`

A continuación se detallarán las pruebas de desarrollo (Pruebas unitarias con *JUnit*).

Lista de los valores de prueba para cada atributo. El criterio elegido para todos los valores de prueba (test data) ha sido: Añadir valores interesantes propensos a error (Conjetura de error).

Al método se le pasa un entero (index) que indica la posición del jugador que se quiere recuperar.

- `index`

  Valor correcto: 0

  Valor de posición que no existe: 5

La estrategia para obtener los casos de prueba elegida ha sido *each choice*.

La tabla completa de los casos de prueba y los resultados esperados son:

	Valores de Prueba	**Objetivo del test**	**Resultado esperado**
Test1	(0)	index	Devuelve el jugador correcto
Test2	(5)	index	Excepción

4.4. PlayerListModel::setData

Nombre del *tester*	Ángel Durán Izquierdo
Fecha de asignación	21 de febrero de 2011
Fecha de finalización	22 de febrero de 2011
Código bajo prueba	`PlayerListModel::setData`

A continuación se detallarán las pruebas de desarrollo (Pruebas unitarias con *JUnit*).

Lista de los valores de prueba para cada atributo. El criterio elegido para todos los valores de prueba (test data) ha sido: Añadir valores interesantes propensos a error (Conjetura de error).

Al método se le pasa una lista de jugadores (data).

- `data`

  Lista correcta de jugadores: lista de jugadores

Valor nulo: null

La estrategia para obtener los casos de prueba elegida ha sido *each choice*.

La tabla completa de los casos de prueba y los resultados esperados son:

	Valores de Prueba	**Objetivo del test**	**Resultado esperado**
Test1	lista de jugadores	data	setdata correcto
Test2	null	data	Excepción

4.5. PlayerListModel::getRowCount

Nombre del *tester*	Ángel Durán Izquierdo
Fecha de asignación	21 de febrero de 2011
Fecha de finalización	22 de febrero de 2011
Código bajo prueba	`PlayerListModel::getRowCount`

A continuación se detallarán las pruebas de desarrollo (Pruebas unitarias con *JUnit*).

Este método devuelve el número de jugadores disponibles, no recibe ningún valor por lo que solo se ha realizado la para comprobar que se devuelve el número correcto de jugadores.

4.6. PlayerListModel::getColumnCount

Nombre del *tester*	Ángel Durán Izquierdo
Fecha de asignación	21 de febrero de 2011
Fecha de finalización	22 de febrero de 2011
Código bajo prueba	`PlayerListModel::getColumnCount`

A continuación se detallarán las pruebas de desarrollo (Pruebas unitarias con *Junit*).

Este método no tiene ningún parametro por lo que solo se prueba que devuelve el número correcto de columnas, este dato esta fijado en la clase y no es variable. A su vez se comprueba que no se produce ninguna excepción durante la realización de las pruebas.

4.7. PlayerListModel::getValueAt

Nombre del *tester*	Ángel Durán Izquierdo
Fecha de asignación	21 de febrero de 2011
Fecha de finalización	22 de febrero de 2011
Código bajo prueba	`PlayerListModel::getValueAt`

A continuación se detallarán las pruebas de desarrollo (Pruebas unitarias con *JUnit*).

Lista de los valores de prueba para cada atributo. El criterio elegido para todos los valores de prueba (test data) ha sido: Añadir valores interesantes propensos a error (Conjetura de error).

Al método se le pasan dos variables, la primera indica el jugador del cual se quiere recuperar la información (rowIndex). El segundo parametro indica que tipo de información debe devolver el método (columnIndex).

- `rowIndex`

  Existe la fila: 0

  No existe la fila: 3

- `columnIndex`

  Existe el parametro a devolver: 0,1,2

  No existe el parametro a devolver: 10

La estrategia para obtener los casos de prueba elegida ha sido *pair wise* eligiendo después los casos interesantes. Los test 2 y 3 son redundantes pero se han elegido para conseguir una covertura mayor del código.

La tabla completa de los casos de prueba y los resultados esperados son:

	Valores de Prueba	**Objetivo del test**	**Resultado esperado**
Test1	(0,0)	rowIndex y columnIndex	Devuelve el nombre del jugador
Test2	(0,1)	columnIndex	Devuelve turno del jugador
Test3	(0,2)	rowIndex	Devuelve si el jugador esta conectado
Test4	(3,0)	rowIndex	IndexOutOfBoundsException
Test5	(0,10)	columnIndex	IndexOutOfBoundsException

4.8. PlayerListModel::getActivePlayer

Nombre del *tester*	Ángel Durán Izquierdo
Fecha de asignación	21 de febrero de 2011
Fecha de finalización	22 de febrero de 2011
Código bajo prueba	`PlayerListModel::getActivePlayer`

A continuación se detallarán las pruebas de desarrollo (Pruebas unitarias con *Junit*).

Este método devuelve el jugador activo en un momento determinando, dicha función no requiere de parametros por lo que unicamente se prueba que al haber un jugador activo el método devuelve este jugador.

4.9. PlayerListModel::getSelfPlayer

Nombre del *tester*	Ángel Durán Izquierdo
Fecha de asignación	21 de febrero de 2011
Fecha de finalización	22 de febrero de 2011
Código bajo prueba	`PlayerListModel::getSelfPlayer`

A continuación se detallarán las pruebas de desarrollo (Pruebas unitarias con *Junit*).

Este método devuelve el jugador propio, no recibe ningún parametro y por lo tanto se ha probado simplemente que al llamar a la función devuelve el jugador adecuado.

4.10. PlayerListModel::getPlayerByName

Nombre del *tester*	Ángel Durán Izquierdo
Fecha de asignación	21 de febrero de 2011
Fecha de finalización	22 de febrero de 2011
Código bajo prueba	`PlayerListModel::getPlayerByName`

A continuación se detallarán las pruebas de desarrollo (Pruebas unitarias con *JUnit*).

Lista de los valores de prueba para cada atributo. El criterio elegido para todos los valores de prueba (test data) ha sido: Añadir valores interesantes propensos a error (Conjetura de error).

Al método se le pasa un String (name) que indica el nombre del jugador que se quiere recuperar.

- `name`

  Valor correcto : .ºponente"

  Valor jugador inexistente: "no existe"

  Valor nulo: null

La estrategia para obtener los casos de prueba elegida ha sido *each choice*.

La tabla completa de los casos de prueba y los resultados esperados son:

	Valores de Prueba	**Objetivo del test**	**Resultado esperado**
Test1	(.ºponente")	name	Devuelve el juego
Test2	("no existe")	name	No devuelve nada
Test2	(null)	name	Excepción

5. Clase MapModel

Nombre del *tester*	Antonio Gómez Poblete
Fecha de asignación	21 de febrero de 2011
Fecha de finalización	25 de febrero de 2011
Código bajo prueba	`MapModel`

A diferencia de otras clases como los gestores, la clase `MapModel` únicamente contiene métodos para implementar el patrón MVC, no implementa casi ninguna otra funcionalidad. Estos métodos son en su mayoría para acceder a los datos que el modelo proporciona y no filtran los atributos de entrada (en ocasiones no tienen) como pueden hacerlo otras clases de dominio. Por esta razón para comprobar esta clase se va a hacer usa de técnicas de caja blanca , intentando cubrir una máxima cobertura del código.

A continuación se van a ir mencionando los distintos test que se han realizado a esta clase para obtener un cobertura de un 99 %, teniendo en cuenta que como precondición debe darse que exista un mapModel (recién creado) y dos jugadores (`Antonio` y `Ambrosio`) sin ningún territorio.

5.1. MapModelTest::testMapModel

En este test se comprueba que el constructor de la clase `MapModel` funciona adecuadamente. Para ello se pregunta por cada uno de los territorios si son distintos a `null` y si tienen el identificador que les corresponde.

5.2. MapModelTest::testSetData

En este test se comprueba que si los territorios son asignados nuevamente, la información es correcta.

Este test además de probar la función `setData`, prueba también el método `updateTerritory` ya que este es llamado desde `setData`.

5.3. MapModelTest::testUpdateTerritory

En este test se actualiza un territorio existente, para ello se le asigna un jugador (`Antonio`) y se comprueba que ningún país tienen un jugador salvo el país de `Antonio`.

5.4. MapModelTest::testgetColumnName

En este test se comprueba que `testgetColumnName` devuelve el nombre correcto para todos los casos.

5.5. MapModel::getValueAt

A diferencia de los métodos anteriormente mencionados, para probar `getValueAt` se ha aplicado también un enfoque de caja negra.

El criterio elegido para todos los valores de prueba (test data) ha sido: Añadir valores límite y valores interesantes propensos a error (Conjetura de error)

- `rowIndex`

  Valores fuera de rango: `map.getRowCount()`, `-1`.

  `0`

- `columnIndex`

  Valores fuera de rango: `map.getColumnCount()`, `-1`.

  `0`

La estrategia para obtener los casos de prueba elegida ha sido *each choice* y añadir algunos casos interesantes .

Teniendo en cuenta las consideraciones anteriormente mencionadas, la tabla completa de los casos de prueba y los resultados esperados son:

	Valores de Prueba	**Resultado esperado**
Test1	(-1, 0)	ArrayIndexOutOfBoundsException
Test2	(`map.getRowCount()`, 0)	IndexOutOfBoundsException
Test3	(0, -1)	IndexOutOfBoundsException
Test4	(0, `map.getColumnCount()`)	IndexOutOfBoundsException

5.5.1. MapModelTest::testgetValueAt5

Este test ha sido creado para completar los casos de pruebas antes mencionados y así obtener una mayor cobertura. Además los anteriores test esperaban el mismo resultado (todos producían error).

En este test los dos jugadores (`Antonio` y `Ambrosio`) son asignados a dos territorios. Luego se pregunta por la información de dos territorios (14, 6) y se comprueba que al no tener espías solo se mostrará `¿?` y el nombre del país.

Para finalizar se añade un espía al jugador `Antonio` que es el `selfPlayer` y se comprueba que ahora si es posible saber la información del país (14) también se comprueba que todos los atributos de este son correctos.

6. Clase TerritoryDecorator

Nombre del *tester*	Antonio Gómez Poblete
Fecha de asignación	21 de febrero de 2011
Fecha de finalización	25 de febrero de 2011
Código bajo prueba	`TerritoryDecorator`

Gran parte de la lógica de `TerritoryDecorator` ha sido ya probada al realizar los casos de pruebas descritos en la sección anterior, 90 % de cobertura. No obstante quedan algunas funcionalidades sin probar que se describirán a continuación con los siguientes test (se llegará a un 99 %).

Para realizar todos los test partimos como precondición (método `setUp()`)que se ha creado un `MapModel`, que existe el jugador `selfPlayer` y que a dos territorios se les ha asignado jugador.

6.1. TerritoryDecoratorTest::testgetclone1

En este test se clonan tres territorios (0, 6, 14) y se comprueba que el identificador del original y la copia sean el mismo. También se comprueba que los territorios copiados sean los mismo que los originales llamando directamente a la función `equal`, de este modo en este test se comprobará el `clone` y el `equal` de la clase `TerritoryDecorator`.

6.2. TerritoryDecoratorTest::testgetEqual1

En este test se comparan territorios diferentes (6, 0, 14) con la función `equal`. Así se comprobarán que ambas posibilidades (true, false) del `equal` funcionan correctamente.

6.3. TerritoryDecoratorTest::testgetName1

El objetivo aquí es comprobar que funciona bien el `getName` de territorio. Para ello se comprueba que el nombre de dos países es diferente (14, 6).

6.4. TerritoryDecoratorTest::testgetAdjacentTerritories1

En este test se comprueba que la función `getAdjacentTerritories` se comporta correctamente, para ello realizamos algunos test intentando probar las dos posibilidades (ser o no adyacente a un país).

- Se comprueba que `getAdjacentTerritories()` no devuelva `null`.

- Se comprueba que el territorio 0 es adyacente al 1 y el 1 al 0.
- Se comprueba que el territorio 0 no es adyacente al 30.
- Se comprueba que el territorio 6 es adyacente al 0 y el 0 al 6.
- Se comprueba que el territorio 6 no es adyacente al 17.

7. Clase GameEngine

Nombre del *tester*	Jorge Colao Adán
	Daniel León Romero
Fecha de asignación	1 de marzo de 2011
Fecha de finalización	8 de marzo de 2011
Código bajo prueba	`GameEngine`

En este apartado se hablará sobre las pruebas realizadas con la herramienta *JUnit* sobre esta clase.

La estrategia a seguir para las pruebas de estos apartados será *Each Choice*. La elección de los valores de los parámetos, se realizarán con valores límite para los atributos númericos y valores propensos a error para el resto.

A la hora de realizar las pruebas en la clase GameEngine tenemos unos atributos privados a los que no podemos acceder. Para acceder a ellos se necesita la clase *PrivateAccesor.java*, disponible en la página `http://onjava.com/pub/a/onjava/2003/11/12/reflection.html?page=2`. Por ejemplo, para acceder a la variable *mCurrentAttack* de la clase GameEngine, se tendría declarar un objeto de esta manera:

```
Object o = PrivateAccessor.getPrivateField(gameEngine,
"mCurrentAttack");
```

Para poder probar los ataques primero tenemos que conectarnos a la partida. La conexión se realiza con el método connectToGame que tiene como argumentos el número de la partida a la que hay que conectarse y un objeto de la clase *GameEventListener*. Para crear este objeto tenemos que crear una clase privada que implemente *GameEventListener*.

7.1. GameEngine::attackTerritory

Este método tiene seis parámetros de entrada, por lo que tenemos que utilizar alguna estrategia de generación de casos de prueba. Hemos utilizado *Each Choice*, combinando las entradas para que falle una y sólo una, en cada caso de prueba y poder saber en donde esta el error.

Para elegir tanto los valores de prueba como los casos de pruebas de los casos interesantes se ha analizado el comportamiento de este método (mediante caja blanca). Después de analizar el método nos hemos dado cuenta de que pude haber los siguientes errores:

- PendingAttackException
- ArrayIndexOutOfBoundsException
- IndexOutOfBoundsException
- UnocupiedTerritoryException
- NegativeValueException
- NotEnoughUnitsException
- InvalidTerritoryException

Esta comprobación es local, por ello si se lanza alguna excepción de las anteriores, no se realizará el ataque.

Estos son los valores de los atributos para los casos de prueba:

- `src`

  Número incorrecto: `"-1"`, `"41"` y `"42"`

  Número correcto: `"0"`, territorio en el que se encuentra *JorgeCA*

- `dst`

  Número incorrecto: `"-1"`, `"0"`, `"41"` y `"42"`

  Número correcto: `"2"`, territorio en el que se encuentra *Aduran*

- `soldiers`

  Número incorrecto: `"-1"` y `"21"`

  Número correcto: `"0"` y `"20"`

- `cannons`

  Número incorrecto: `"-1"` y `"7"`

  Número correcto: `"0"` y `"6"`

- `missiles`

  Número incorrecto: `"-1"` y `"2"`

  Número correcto: `"0"` y `"1"`

- `icbm`

  Número incorrecto: `"-1"` y `"7"`

  Número correcto: `"0"` y `"6"`

La tabla siguiente recoge los resultados esperados.

	Valores de Prueba	**Objetivo del test**	**Resultado esperado**
Test1	(0, 2, 0, 0, 1, 6)	realizar ataque	Funcionamiento Correcto
Test2	(0, 2, 0, 0, 1, 6)	realizar 2 ataques	PendingAttack
Test3	(-1, 2, 0, 0, 1, 6)	src	ArrayIndexOutOfBounds
Test4	(42, 2, 0, 0, 1, 6)	src	IndexOutOfBounds
Test5	(41, 2, 0, 0, 1, 6)	src	UnocupiedTerritory
Test6	(0, -1, 0, 0, 1, 6)	dst	ArrayIndexOutOfBounds
Test7	(0, 42, 0, 0, 1, 6)	dst	IndexOutOfBounds
Test8	(0, 0, 0, 0, 1, 6)	dst	InvalidTerritory
Test9	(0, 2, -1, 0, 1, 6)	soldiers	NegativeValue
Test10	(0, 2, 21, 0, 1, 6)	negTime	NotEnoughUnits
Test11	(0, 2, 20, -1, 1, 6)	cannons	NegativeValue
Test12	(0, 2, 20, 7, 1, 6)	cannons	NotEnoughUnits
Test13	(0, 2, 20, 6, -1, 6)	missiles	NegativeValue
Test14	(0, 2, 20, 6, 2, 6)	missiles	NotEnoughUnits
Test15	(0, 2, 20, 6, 1, -1)	icbm	NegativeValue
Test16	(0, 2, 20, 6, 1, 7)	icbm	NotEnoughUnits
Test17	(0, 2, 0, 0, 0, 0)	realizar ataque	Funcionamiento Correcto
Test18	(0, 2, 20, 6, 0, 0)	realizar ataque	Funcionamiento Correcto

7.2. GameEngine::territoryUnderAttack

Este método tiene tres parámetros de entrada, por lo que se utilizará *Each Choice*. Combinando las entradas para que falle una y sólo una, en cada caso de prueba y poder saber en donde está el error.

Para elegir tanto los valores de prueba como los casos de pruebas de los casos interesantes se ha analizado el comportamiento de este método (mediante caja blanca). Después de analizar el método nos hemos dado cuenta de que sólo pude darse la excepción de *NullPointerException*.

Esta comprobación es local, por ello si se lanza la excepción anterior, no se informará de que te quieren atacar.

Estos son los valores de los atributos para los casos de prueba:

- `src`

  Territorio 2 de Aduran

  null

- `dst`

Territorio 0 de JorgeCA

null

- `arsenal`

  Arsenal(5, 4, 1, 0)

  null

La tabla siguiente recoge los resultados esperados, para los casos de prueba.

	Valores de Prueba	**Objetivo del test**	**Resultado esperado**
Test1	(T2_Aduran, T0_JorgeCA, Arsenal)	informar del ataque	Funcionamiento Correcto
Test2	(null, T0_JorgeCA, Arsenal)	src	NullPointer
Test3	(T2_Aduran, null, Arsenal)	dst	NullPointer
Test4	(T2_Aduran, T0_JorgeCA, null)	arsenal	NullPointer

7.3. GameEngine::acceptAttack

Este método es una respuesta al método anterior. No tiene ningún parámetro de entrada, por lo que para poder probarlo sólo podemos comprobar que antes de realizar este método la variable *mCurrentAttack* no sea null y después de ejecutarlo que sea null.

Analizando el comportamiento del método se puede observar que solamente puede dar un error, *OutOfTurnException*.

Esta comprobación es local, por ello si se lanza la excepción OutOfTurnException, no se aceptará el ataque.

Se han realizado dos casos de prueba:

- Ejecutando primero el método *territoryUnderAttack*. El cuál da un valor la variable *mCurrentAttack* y el método aceptar ataque no da error.
- Sin ejecutar primero el método *territoryUnderAttack*. En este caso no se le da valor a la varible *mCurrentAttack* y como es null falla la ejecución de aceptar ataque.

7.4. GameEngine::requestNegotiation

Al igual que el método anterior, también es la respuesta al método *territoryUnderAttack*. Como valores de entrada tiene dos parámetos:

- `money`

Número incorrecto: `"-1"` y `"201"`

Número correcto: `"0"` y `"200"`

- `sodiers`

  Número incorrecto: `"-1"` y `"21"`

  Número correcto: `"0"` y `"20"`

Los posibles excepciones que pueden lanzarse al ejecutar este método son:

- OutOfTurnException
- NegativeValueException
- NotEnoughUnitsException
- NotEnoughMoneyException
- NullPointerException

La tabla siguiente recoge los resultados esperados.

	Valores de Prueba	**Objetivo del test**	**Resultado esperado**
Test1	(0, 20)	pedir negociación	Funcionamiento Correcto
Test2	(0, 20)	(*)	NullPointer
Test3	(-1, 0)	money	NegativeValue
Test4	(201, 0)	money	NotEnoughMoney
Test5	(200, -1)	soldiers	NegativeValue
Test6	(200, 21)	soldiers	NotEnoughUnits

(*) El caso de prueba; Test2 es igual que el primero, pero no se ha realizado el *territoryUnderAttack*, por lo que la variable *mCurrentAttack* es null.

7.5. GameEngine::resolveAttack

Este método es la respuesta al método *acceptAttack*. No tiene ningún parámetro de entrada, por lo que para poder probarlo sólo podemos comprobar que antes de realizar este método la variable *mCurrentAttack* no sea null y después de ejecutarlo que sea null.

Se han realizado dos casos de prueba:

- Ejecutando primero el método *territoryUnderAttack*. Para poder ejecutar el método resolver ataque se necesita que primero la variable

mCurrentAttack tenga un valor no nulo, por este motivo ejecutamos antes el método *territoryUnderAttack* y luego el método resolver ataque no da error.

- Sin ejecutar primero el método *territoryUnderAttack*. En este caso no se le da un valor a la varible *mCurrentAttack* y como es null falla la ejecución de resolver ataque.

7.6. GameEngine::resolveNegotiation

Este método es la respuesta al método *requestNegotiation*. La variable *mCurrentAttack* se comporta igual que el caso anterior.

Se han realizado dos casos de prueba:

- Ejecutando primero el método *territoryUnderAttack*. Para poder ejecutar el método resolver negociación se necesita que primero la variable *mCurrentAttack* tenga un valor no nulo, por este motivo ejecutamos antes el método territoryUnderAttack y luego el método resolver negociación no da error.
- Sin ejecutar primero el método *territoryUnderAttack*. En este caso no se le da un valor a la varible *mCurrentAttack* y como es null falla la ejecución de resolver negociación.

7.7. GameEngine::buyUnits

El método **buyUnits(int index, int soldiers, int cannons, int missiles, int icbm, int antimissiles)** tiene seis parámetros de entrada, y para comprobar su funcionamiento nos hemos inclinado por un enfoque de caja blanca. Hemos optado por un criterio de cobertura *modificada de condición decisión* usando valores interesantes. Las excepciones que este método lanza son las siguientes:

- **PendingAttackException** Cuando se intenta comprar y hay un ataque en curso.
- **UnocupiedTerritoryException** Si se intentan comprar unidades en un territorio no asignado a ningún jugador.
- **InvalidTerritoryException** Si el dueño del territorio es otro jugador.
- **NegativeValueException** Si se intenta comprar un número negativo de algún tipo de unidad.

- **NotEnoughMoneyException** Si el jugador no tiene dinero suficiente para comprar las unidades.
- **ArrayIndexOutOfBoundsException** Si el índice del país es negativo.
- **IndexOutOfBoundsException** Si el índice del país es mayor de 41.

A continuación se muestra una tabla que muestra los diferentes valores para los casos de prueba y el resultado esperado para cada uno de ellos.

	Valores de Prueba	**Objetivo del test**	**Resultado esperado**
Test1	(0, 1, 0, 0, 0, 0)	Comprar unidades	Funcionamiento Correcto
Test2	(2, 1, 0, 0, 0, 0)	index	InvalidTerritoryException
Test3	(0, 3, 0, 0, 0, 0)		NotEnoughMoneyException
Test4	(0, -1, 0, 0, 0, 0)	soldiers	NegativeValueException
Test5	(0, 0, -1, 0, 0, 0)	cannons	NegativeValueException
Test6	(0, 0, 0, -1, 0, 0)	missiles	NegativeValueException
Test7	(0, 0, 0, 0, -1, 0)	icbm	NegativeValueException
Test8	(0, 0, 0, 0, 0, -1)	antimissiles	NegativeValueException
Test9	(-1, 0, 0, 0, 0, 0)	index	ArrayIndexOutOfBoundsException
Test10	(42, 0, 0, 0, 0, 0)	index	IndexOutOfBoundsException
Test11	(12, 1, 0, 0, 0, 0)	index	UnocupiedTerritoryException
Test12	(12, 1, 0, 0, 0, 0)	Comprar con ataque en curso	PendingAttackException

7.7.1. Observaciones

- **Test 2** El territorio no pertenece al usuario.
- **Test 3** Antes de comprar, establecemos el dinero del jugador a 0 gallifantes.
- **Test 11** El territorio no está asignado a ningún jugador.
- **Test 12** Antes de ejecutar buyUnits, lanzamos un ataque.

7.8. GameEngine::moveUnits

El método **moveUnits(int src, int dst, int soldiers, int[] cannons, int missiles, int icbm, int antimissiles)** tiene siete parámetros de entrada (uno de ellos compuesto por tres enteros), y para comprobar su funcionamiento nos hemos inclinado por un enfoque de caja blanca. Hemos optado por un criterio de cobertura *modificada de condición decisión* usando valores interesantes. Las excepciones que este método lanza son las siguientes:

- **PendingAttackException** Cuando se intenta mover tropas y hay un ataque en curso.
- **UnocupiedTerritoryException** Si se intenta mover desde o hacia un país no asignado a ningún jugador.

- **InvalidTerritoryException** Si el dueño del territorio origen o destino es otro jugador; si los territorios destino y origen no son adyacentes.
- **NegativeValueException** Si se intenta mover un número negativo de algún tipo de unidad.
- **ArrayIndexOutOfBoundsException** Si el origen o el destino son negativos.
- **IndexOutOfBoundsException** Si el origen o el destino son mayores de 41.

A continuación se muestra una tabla que muestra los diferentes valores para los casos de prueba y el resultado esperado para cada uno de ellos.

	Valores de Prueba	**Objetivo del test**	**Resultado esperado**
Test1	(0, 1, 0, [0,0,0] , 0, 0, 0)	Mover unidades	Funcionamiento Correcto
Test2	(0, 1, 1, [0,0,0], 1, 1, 1)	Mover unidades	Funcionamiento Correcto
Test3	(0, 1, 1, [1,1,1], 1, 1, 1)	Mover unidades	Funcionamiento Correcto
Test4	(0, 1, 100, [0,0,0], 0, 0, 0)	soldiers	NotEnoughUnitsException
Test5	(0, 1, 0, [100,0,0], 0, 0, 0)	cannons[0]	NotEnoughUnitsException
Test6	(0, 1, 0, [0,100,0], 0, 0, 0)	cannons[1]	NotEnoughUnitsException
Test7	(0, 1, 0, [0,0,100], 0, 0, 0)	cannons[2]	NotEnoughUnitsException
Test8	0, 1, 0, [0,0,0], 110, 0, 0)	missiles	NotEnoughUnitsException
Test9	(0, 1, 0, [0,0,0], 0, 110, 0)	icbm	NotEnoughUnitsException
Test10	(0, 1, 0, [0,0,0], 0, 0, 110)	antimissiles	NotEnoughUnitsException
Test11	(0, 7, 0, [0,0,0], 0, 0, 0)	Mover a territorios no adyacentes	InvalidTerritoryException
Test12	(-1, 0, 0, [0,0,0], 0, 0, 0)	src	ArrayIndexOutOfBoundsException
Test13	(0, -1, 0, [0,0,0], 0, 0, 0)	dst	ArrayIndexOutOfBoundsException
Test14	(0, 2, 0, [0,0,0], 0, 0, 0)	dst	InvalidTerritoryException
Test15	(0, 1, -1, [0,0,0], 0, 0, 0)	soldiers	NegativeValueException
Test16	(0, 1, 0, [-1,0,0], 0, 0, 0)	cannons [0]	NegativeValueException
Test17	(0, 1, 0, [0,-1,0], 0, 0, 0)	cannons [1]	NegativeValueException
Test18	(0, 1, 0, [0, 0,-1], 0, 0, 0)	cannons [2]	NegativeValueException
Test19	(0, 1, 0, [0,0,0], -1, 0, 0)	missiles	NegativeValueException
Test20	(0, 1, 0, [0,0,0], 0, -1, 0)	icbm	NegativeValueException
Test21	(0, 1, 0, [0,0,0], 0, 0, -1)	antimissiles	NegativeValueException
Test22	(42, 1, 0, [0,0,0], 0, 0, 0)	src	IndexOutOfBoundsException
Test23	(0, 42, 0, [0,0,0], 0, 0, 0)	dst	IndexOutOfBoundsException
Test24	(0, 42, 0, [0,0,0], 0, 0, 0)	Mover con un ataque en curso	PendingAttackException
Test25	(12, 2, 0, [0,0,0], 0, 0, 0)	src	UnocupiedTerritoryException
Test26	(0, 12, 0, [0,0,0], 0, 0, 0)	dst	UnocupiedTerritoryException
Test27	(2, 0, 0, [0,0,0], 0, 0, 0)	src	InvalidTerritoryException

7.8.1. Observaciones

- Los territorios pertenecientes al jugador son el 0, 1, y 7.
- El territorio 12 no está asignado a ningún jugador.
- Los territorios 0, 1 y 2 son adyacentes.

- El territorio 7 no es adyacente al 0.
- **Test 24** Antes de ejecutar moveUnits, lanzamos un ataque.

7.9. GameEngine::deploySpy

El método **deploySpy(int index)** tiene solamente un parámetro de entrada, y para comprobar su funcionamiento nos hemos inclinado por un enfoque de caja blanca. Hemos optado por un criterio de cobertura *modificada de condición decisión* usando valores interesantes. Las excepciones que este método lanza son las siguientes:

- **PendingAttackException** Cuando se intenta enviar un espía y hay un ataque en curso.
- **NotEnoughMoneyException** Cuando el jugador no tiene dinero para comprar el espía.
- **InvalidTerritoryException** Cuando el territorio pertenece al propio jugador.
- **ArrayIndexOutOfBoundsException** Si el índice del país es negativo.
- **IndexOutOfBoundsException** Si el índice del país es mayor de 41.

A continuación se muestra una tabla que muestra los diferentes valores para los casos de prueba y el resultado esperado para cada uno de ellos.

	Valores de Prueba	**Objetivo del test**	**Resultado esperado**
Test1	(0)	index	InvalidTerritoryException
Test2	(2)	index	Funcionamiento Correcto
Test3	(-1)	index	ArrayIndexOutOfBoundsException
Test4	(42)	index	IndexOutOfBoundsException
Test5	(2)	index	NotEnoughMoneyException
Test6	(2)	Enviar espia con ataque en curso	PendingAttackException

7.9.1. Observaciones

- **Test 2** Antes de ejecutar deploySpy, nos aseguramos de que el jugador tenga dinero suficiente.
- **Test 5** Antes de ejecutar deploySpy, establecemos la cantidad de dinero del jugador a 0.
- **Test 6** Antes de ejecutar deploySpy, lanzamos un ataque.

7.10. GameEngine::buyTerritory

El método **buyTerritory(int index)** tiene solamente un parámetro de entrada, y para comprobar su funcionamiento nos hemos inclinado por un enfoque de caja blanca. Hemos optado por un criterio de cobertura *modificada de condición decisión* usando valores interesantes. Las excepciones que este método lanza son las siguientes:

- **PendingAttackException** Cuando se intenta comprar un territorio y hay un ataque en curso.
- **NotEnoughMoneyException** Cuando el jugador no tiene dinero para comprar el territorio.
- **InvalidTerritoryException** Cuando el territorio pertenece a algún jugador o no es adyacente a ninguno de los pertenecientes al jugador que solicita la compra.
- **ArrayIndexOutOfBoundsException** Si el índice del país es negativo.
- **IndexOutOfBoundsException** Si el índice del país es mayor de 41.

A continuación se muestra una tabla que muestra los diferentes valores para los casos de prueba y el resultado esperado para cada uno de ellos.

	Valores de Prueba	**Objetivo del test**	**Resultado esperado**
Test1	(6)	Comprar territorio	Funcionamiento correcto
Test2	(22)	index	InvalidTerritoryException
Test3	(2)	index	OcupiedTerritoryException
Test4	(6)	Comprar sin el dinero suficiente	NotEnoughMoneyException
Test5	(6)	Comprar con ataque en curso	PendingAttackException
Test6	(-1)	index	ArrayIndexOutOfBoundsException
Test7	(42)	index	IndexOutOfBoundsException

7.10.1. Observaciones

- El jugador es propietario de los territorios 0,1 y 7. El territorio 2 también tiene propietario.
- El territorio 22 no es adyacente a ninguno perteneciente al jugador.
- **Test 4** Antes de ejecutar buyTerritory, establecemos el dinero del jugador a 0.
- **Test 5** Antes de ejecutar buyTerritory, lanzamos un ataque.

8. Clase UnitInfo

8.1. UnitInfo

Nombre del *tester*	Ángel Durán Izquierdo
Fecha de asignación	29 de febrero de 2011
Fecha de finalización	29 de febrero de 2011
Código bajo prueba	`UnitInfo`

Se han realizado las pruebas pertinentes sobre la clase UnitInfo, esta clase contiene la información necesaria de las diferentes unidades que componen un arsenal, debido a la baja complejidad de la lógica de esta clase lo único que se ha realizado son pruebas para corroborar que los métodos de dicha clase devuelven la información de forma correcta. Se han probado las siguientes funciones:

Método	**Resultado esperado**
getSoldierCost	Resultado correcto
getCannonCost	Resultado correcto
getMissileCost	Resultado correcto
getICBMCost	Resultado correcto
getAntiMissileCost	Resultado correcto
getSpyCost	Resultado correcto