

2015

Grupo: Proyecto Omega

Fité Juan Manuel
Rodríguez Felici Joaquín
Turina Tomás
Vendramini Santiago

[TRABAJO FINAL DE INGENIERÍA DE SOFTWARE]

Índice

Nota de entrega	3
Listado de funcionalidad.....	3
Pass/Fail ratio.....	4
Bugs conocidos.....	4
Manejo de configuraciones	5
Control de versiones.....	5
Esquema de directorios.....	5
Normas de etiquetado.....	6
Plan de esquema de ramas.....	6
Política de fusión de archivos.....	7
Forma de entrega de releases.....	7
Roles de los integrantes en la CBC.....	7
Herramientas de seguimiento de bugs.....	8
Requerimientos	9
Casos de uso.....	9
Diagrama de actividades.....	9
Diagrama de secuencias.....	10
Requerimientos funcionales.....	11
Requerimientos no funcionales.....	11
Diagrama de arquitectura preliminar.....	11
Matriz de trazabilidad.....	11
Arquitectura	13
Arquitectura general.....	13
Diagrama de despliegue.....	13
Diagrama de componentes.....	14
Diseño e implementación	15
Diagramas de clases.....	15
Diagramas de paquetes.....	18
Patrones de diseño.....	18
Pruebas unitarias y del sistema	20
Pruebas unitarias.....	20

Casos de prueba del sistema.....	23
Sanity Tests.....	26
Matriz de trazabilidad actualizada.....	26
Datos históricos.....	27
Información adicional.....	28

Nota de entrega

El entregable se encuentra en el repositorio de GitHub, cuyo link se detalla en la sección de manejo de configuraciones. Para su instalación/ejecución deben abrirse los archivos:

- DJTestDrive.jar: Ejecuta el BeatModel.
- HeartTestDrive.jar: Ejecuta el HeartModel.
- BridgeCraneTestDrive.jar: Ejecuta el BridgeCraneModel.
- SimultaneousTestDrive.jar: Ejecuta los tres modelos simultáneamente.
- ExchangeableTestDrive.jar: Ejecuta el BridgeCraneModel con la posibilidad de cambiar de modelo en tiempo de ejecución.

Todos estos archivos se encuentran en la carpeta de ejecutables del repositorio.

Listado de funcionalidad:

A continuación se detallan las modificaciones introducidas en el código provisto:

- Implementación del patrón de diseño Singleton en el HeartModel.
- Implementación de un contador del número de intentos de creación de nuevas instancias HeartModel visible en la vista BeatBar.

En cuanto a las nuevas funcionalidades introducidas, podemos mencionar:

- Implementación de una nueva vista que representa gráficamente y en texto la posición de un puente grúa en un depósito, visto desde arriba.
- Implementación de un nuevo modelo, que se agrega a los dos preexistentes. Éste consiste en una matriz que lleva el control de la posición en la que se encuentra la grúa. Las modificaciones de la misma se explicitan gráficamente en la vista. Para esto se llevo a cabo el desarrollo de un patrón Observer adicional.
- Se modificó la ventana de control agregándole dos botones adicionales que permite el desplazamiento de la grúa. Los dos botones ya implementados permiten el movimiento lateral de la grúa, mientras que los nuevos hacia adelante y atrás. Estos botones provocan una modificación de la matriz mencionada en el apartado anterior.
- Se desarrolló un nuevo controller para implementar correctamente el patrón de arquitectura MVC. En el mismo se crean las vistas y se determina la funcionalidad a los botones.
- Se modificó la BeatBar para el modelo, la cual muestra un mensaje indicando el estado de la grúa. Además, cuando no se encuentra disponible, se habilita la barra indicando que la grúa está trabajando.
- Se implementó un dropdown box en BeatBar que permite cambiar de modelo en tiempo de ejecución. Esta funcionalidad está disponible en el ejecutable ExchangeableTestDrive.
- Se desarrollo un ejecutable denominado SimultaneousTestDrive que permite la ejecución de los tres modelos en forma simultánea.

Estas implementaciones se encuentran en su versión final, correctamente desarrolladas y testeadas.

Pass/Fail Ratio:

Test	Características	Estado
UT1	Comprueba que la función decrease() no realice un decremento cuando bpm es cero.	Pass
UT2	Comprueba que no se seteen bpm negativos.	Pass
UT3	Comprueba el correcto funcionamiento del patrón Singleton en el HeartModel.	Pass
UT4	Comprueba incremento de cuenta al intentar instanciar un nuevo HeartModel.	Pass
UT5	Comprueban la correcta modificación de la matriz del BridgeCraneModel, en caso de ser posible.	Pass
UT6		
UT7		
UT8		
UT9	Comprueban si se realizan cambios de acuerdo a si la contraseña ingresada es correcta o no.	Pass
UT10		
UT11	Comprueban que no se modifique la matriz del BridgeCraneModel, en caso de que sobrepasen los límites.	Pass
UT12		
UT13		
UT14		
UT15	Comprueba la correcta modificación de bpm.	Pass
ST1	Comprueba que la grúa no se mueve en el estado "Cargando".	Pass
ST2	Comprueba que se despliegue solo un HeartModel, y que cuando se intente crear otra instancia se muestre un incremento de un contador en la vista BeatBar.	Pass
ST3	Comprueba el movimiento de la grúa utilizando los botones de control de la vista.	Pass
ST4	Comprueba que se pueda cambiar de modelo en tiempo de ejecución a través del despliegue del nuevo menú de la BeatBar.	Pass
ST5	Comprueba que se ejecuten los tres modelos de manera simultánea.	Pass
ST6	Comprueba el correcto funcionamiento del sistema de seguridad.	Pass
ST7	Comprueba la correcta disposición de las ventanas.	Pass
ST8	Comprueba que la grúa no sobrepase los límites.	Pass
	Promedio	100%

Bugs conocidos:

Como bug conocido podemos mencionar que al presionar el botón set en la vista de control sin haber introducido ningún número entero, el programa lanza una excepción del tipo NumberFormatException. Este bug ya se encontraba presente en el software entregado para modificar.

Para el seguimiento de los bugs se utilizo la herramienta provista por GitHub "issues", la cual se describe mas detalladamente en la sección de manejo de configuraciones.

Manejo de configuraciones

Dirección y formas de acceso a la herramienta de control de versiones:

Para el control de versiones se utilizó el servicio de almacenamiento de repositorios GitHub, el cual ofrece sistemas de revisión de código, manejo de archivos y una interfaz fluida. A su vez, GitHub ofrece planes tanto para repositorios privados como para cuentas gratuitas, las cuales suelen utilizarse para almacenar proyectos de código abierto, como es el caso del nuestro.

Se utilizó la consola para la organización de los archivos, mientras que se hizo la revisión, comparación y resolución de conflictos en el código con la herramienta para Windows TortoiseSVN.

La dirección del repositorio utilizado para este proyecto es la siguiente:

https://github.com/joaquinfelici/isw_tpf/

Esquema de directorios y propósito de cada uno:

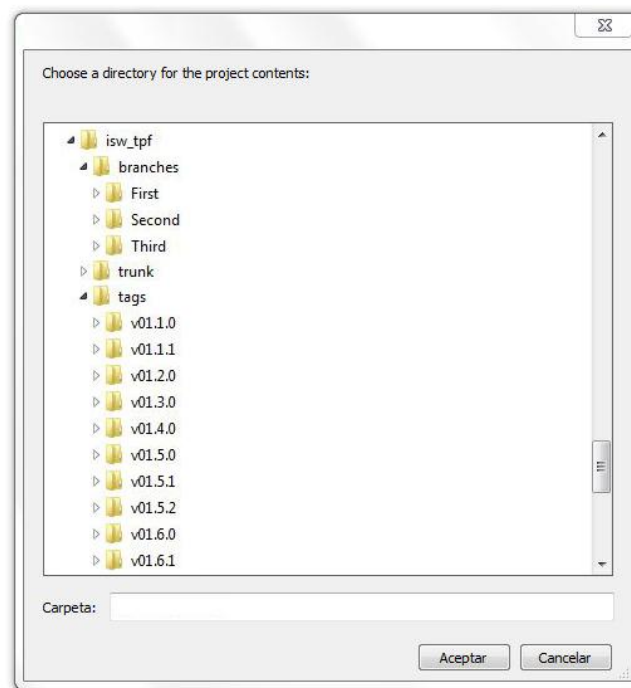


Imagen 1. Esquema de directorios.

A continuación se especifica que función cumple cada carpeta:

- isw_tpf: Es la carpeta principal del proyecto. La misma es la clonación del repositorio de trabajo creado en los servidores de GitHub.

- branches: Esta carpeta contiene todas las ramas de nuestro repositorio. Cada carpeta (rama) dentro de la misma representa distintas etapas de nuestro proyecto que nos sirven para organizarnos y llevar un control del mismo cuando se ingresan cambios significativos al código.
- trunk: Es la rama principal de nuestro repositorio, la misma contiene el proyecto finalizado con todas las funcionalidades ya ingresadas.
- tags: Esta carpeta contiene todas las versiones que se fueron desarrollando del trabajo. Las mismas fueron creadas como releases prematuros del proyecto final, donde en cada uno se fueron ingresando tanto cambios en funcionalidades o implementación de nuevas funciones, así como correcciones de bugs encontrados.

Normas de etiquetado:

Para el etiquetado o numeración de versiones se hará uso de la convención más utilizada, y la recomendada por el servicio de release de GitHub en <http://semver.org/>. Esta consiste en incluir:

- La letra v como acrónimo de "versión".
- El numero que indica la versión del proyecto, que solo cambia cuando hay una refactorización importante o cambios significativos en la arquitectura.
- Un número intermedio que se incrementa cada vez que se agregan funcionalidades al sistema.
- Un tercer número que aumenta con respecto a la versión anterior si se han encontrado y solucionado uno o más bugs.

Por ejemplo, sin tener en cuenta el código provisto por la cátedra, nuestra versión inicial (incluyendo las primeras funcionalidades requeridas) fue etiquetada como v01.1.0.

La política de nombramiento de archivos puede apreciarse en la Imagen 1.

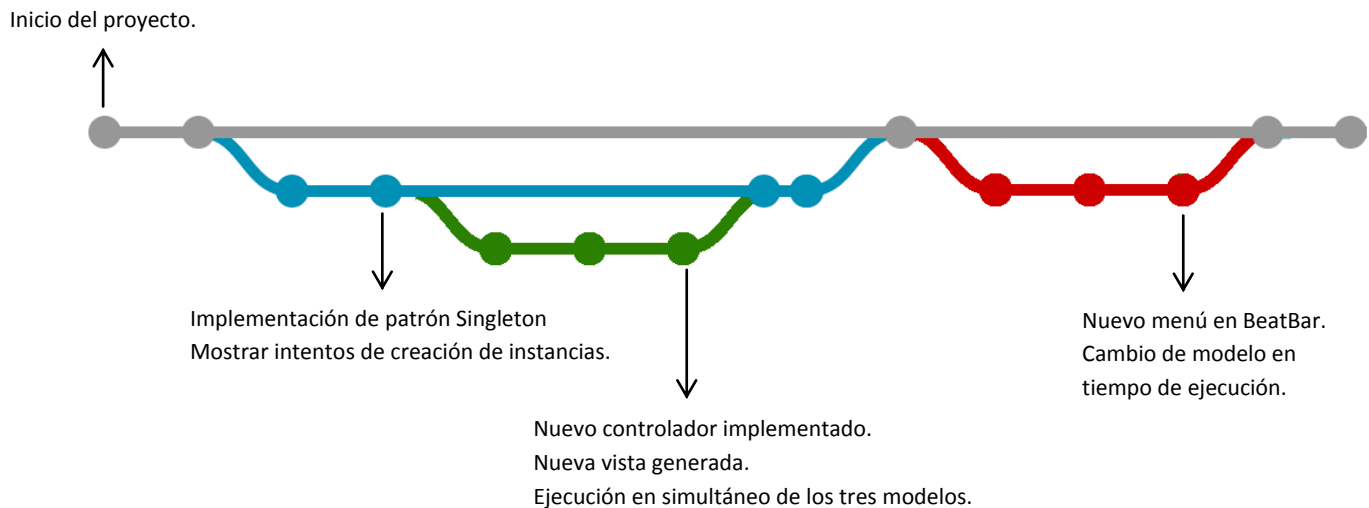
Plan del esquema de ramas a usar:

Al iniciar el proyecto, se realizó un plan con respecto a las branches a crear para una implementación simple y organizada. El plan contiene las siguientes ramas:

- First: contiene la primer parte del trabajo con las primeras modificaciones en sus funcionalidades.
- Second: contiene la vista nueva implementada con las nuevas funcionalidades que la acompañan.
- Third: contiene las nuevas funciones que permiten ejecutar los diferentes modelos y cambiarlos en tiempo de ejecución.

Políticas de fusión de archivos:

Como se mencionó anteriormente, se ha hecho un plan para tener tres branches, aparte de *master*: la azul representa First, la verde Second, y la roja Third. El plan para sus divisiones y merges se muestra en el siguiente diagrama:



Forma de entrega de los releases, instrucciones de instalación:

Se entrega solo un release al incorporar todas las funcionalidades. Antes, se realizan pre-releases incorporando solo algunas (pero con el programa operativo).

En cuanto a las instrucciones de instalación, el usuario debe contar con la última versión de java instalada en su ordenador, y debe descomprimir el archivo provisto en el repositorio. Asegurarse de que sea el último release lanzado. Para ejecutar el programa, debe correrse el archivo .jar deseado incluido dentro del .zip del proyecto.

Roles de los integrantes en la CCB y formas de contacto:

El equipo de trabajo está conformado por cuatro integrantes. Como el grupo es pequeño y no hay impedimentos geográficos para la comunicación directa, se realizan reuniones diarias. Al inicio de cada una de ellas se discute sobre las tareas realizadas, las que quedan por realizarse y la forma de proceder. Luego se asignan las tareas a grupos de dos integrantes, y el equipo se divide para avanzar en la actividad que le compete.

Los roles de cada integrante se muestran en la siguiente tabla:

<i>Integrantes</i>	<i>Roles en la CBC</i>
Fité, Juan Manuel	Arquitecto Desarrollador Master
Rodríguez Felici, Joaquín	Gestor de proyectos Desarrollador Master
Turina, Tomás Bernardo	Analista Desarrollador Master
Vendramini, Santiago David	Arquitecto Desarrollador Master

Herramienta de seguimiento de bugs:

Los bugs del código fueron encontrados de forma manual, al ser corridos y comprobados por alguno de los cuatro desarrolladores pertenecientes al proyecto. El seguimiento de los mismos fue facilitado por la herramienta provista por GitHub llamada "issues". En la misma, el programador que encontró el bug puede informar al resto de su existencia, y dejarlo registrado para recordar solucionarlo él mismo, o que lo resuelva otro miembro del equipo.

Requerimientos

En esta sección se desarrollan los requerimientos del sistema. Para un mejor entendimiento se utilizaron diferentes diagramas implementados con el software StarUML.

Casos de uso

En este tipo de diagrama se puede identificar a simple vista y sin entrar en detalle la interacción entre los actores y los casos de uso. Se observa claramente que, tanto el Usuario (actor externo) como la Vista (actor interno), pueden utilizar el sistema para dos propósitos cada uno. Se observa debajo, en la imagen 2.

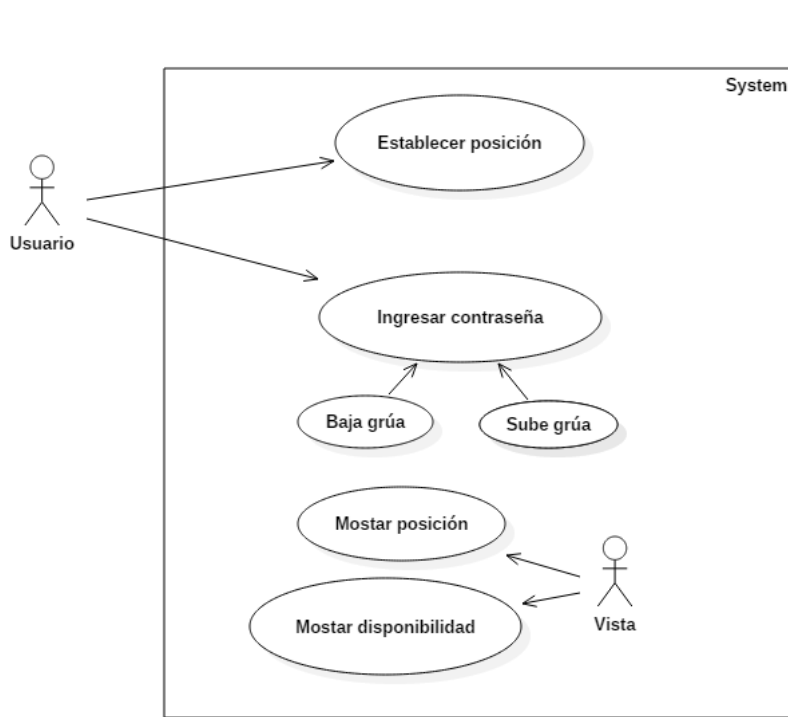


Imagen 2. Casos de uso.

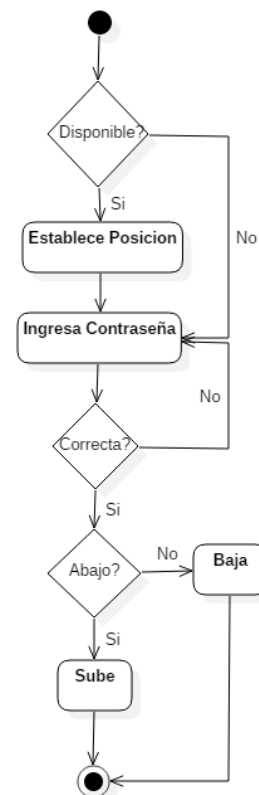


Imagen 3. Diagrama de actividades

Diagramas de actividades

En este caso, el diagrama ofrece una vista sencilla del flujo de trabajo que el usuario del sistema experimentará. Representa la manipulación de la grúa. Se observa arriba, en la imagen 3.

Diagramas de secuencias

Para representar las interacciones de las partes del sistema se utilizaron dos diagramas de secuencias. El primero (Imagen 4) es para el caso en que se quiere establecer una nueva posición. Se utilizó, a modo de ejemplo, solo cuando se presiona la tecla “^”, ya que para el resto el diagrama es análogo. En el segundo diagrama (Imagen 5) se observa el caso de uso en el cual se ingresa la contraseña.

Imagen 4

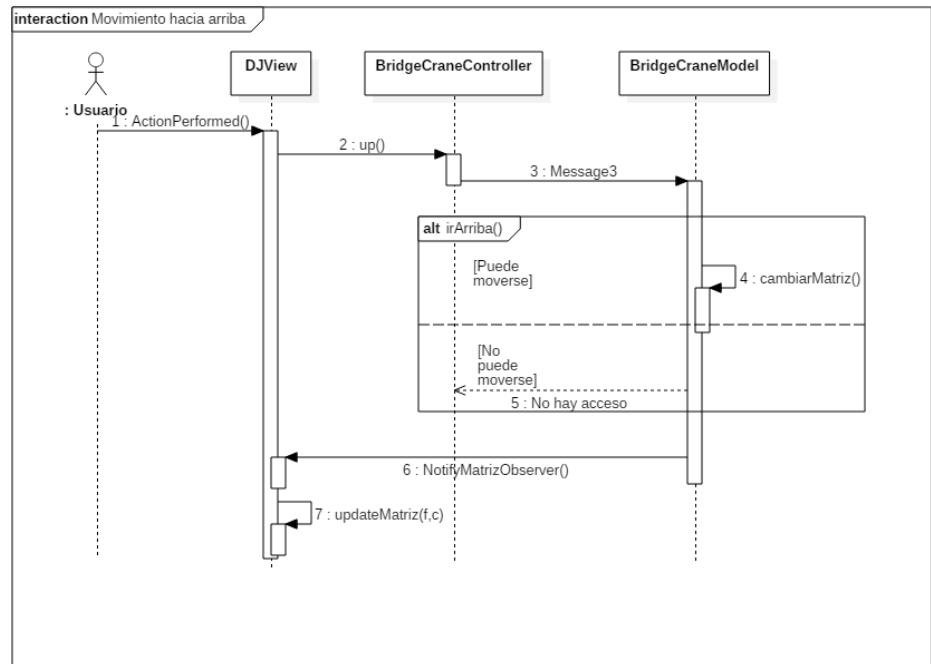
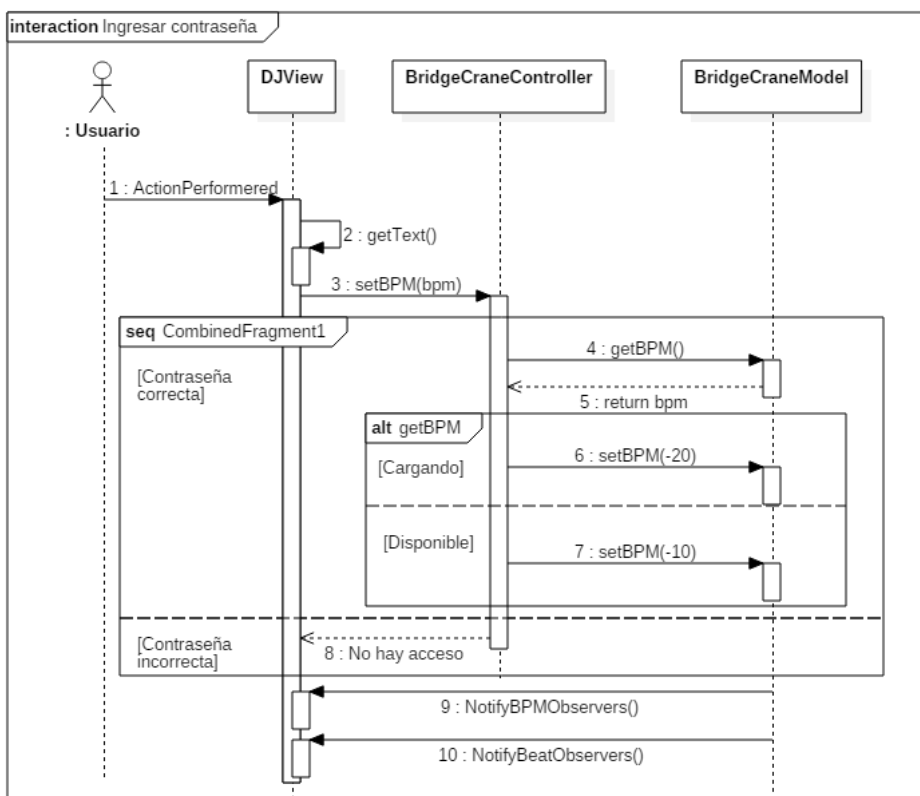


Imagen 5



Requerimientos funcionales

- 1.El sistema debe impedir el movimiento de la grúa cuando está en proceso de carga.
- 2.El sistema debe crear solo una instancia del modelo HeartModel. En caso de que se intente crear otra, solo se muestra el número de intentos.
- 3.El sistema debe permitir el movimiento de la grúa utilizando los botones de control de la vista.
- 4.El sistema debe mostrar de forma gráfica y en texto la posición de la grúa.
- 5.El sistema debe avisar en forma gráfica y en texto que la grúa está en proceso de carga.
- 6.El sistema debe permitir, a través de un menú, cambiar de modelo en tiempo de ejecución.
- 7.El sistema debe permitir ejecutar los tres modelos de manera simultánea.

Requerimientos no funcionales

- 1.El sistema debe ejecutarse a través de los TestDrives exportados a formato .jar ejecutable y contenidos dentro de la carpeta del programa.
- 2.El sistema debe tener un control de seguridad (mediante solicitud de contraseña) para bajar o subir la grúa.
- 3.El sistema debe impedirle a la grúa ir más allá de sus límites.
- 4.El proceso de carga debe requerir la orden del usuario tanto para bajar como para subir, para garantizar el tiempo que se necesita para una carga determinada.
- 5.El sistema debe evitar, cuando se abren múltiples ventanas, que queden solapadas.

Diagrama de arquitectura preliminar

De manera general, en la siguiente imagen se aprecia la arquitectura preliminar, la cual consiste en un modelo MVC. Además se puede observar la interacción de una de las partes del modelo con el usuario.

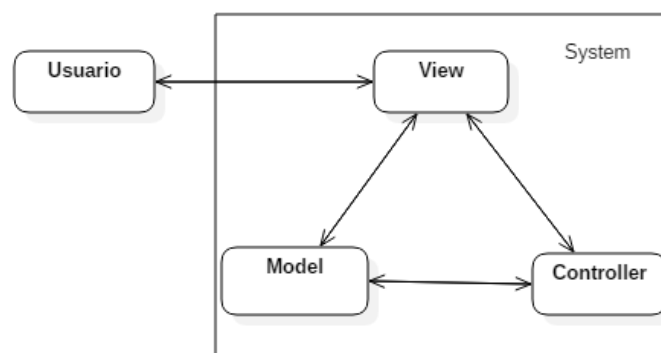


Imagen 6. Diagrama de arquitectura preliminar

Matriz de trazabilidad

En la siguiente tabla se representa la matriz de trazabilidad, que relaciona los requerimientos funcionales y no funcionales con los casos de uso.

Requerimientos		Establecer Posición	Ingresar Contraseña	Bajar Grúa	Subir Grúa	Mostrar Posición	Mostrar Disponibilidad
Funcionales	1	1					
	2						
	3	1					
	4					1	
	5					1	1
	6						
	7						
No Funcionales	1						
	2		1	1	1		
	3	1					
	4			1	1		
	5	1				1	1

Arquitectura

Arquitectura general

Para la realización del software se utilizó la arquitectura MVC (Model View Controller), ya que el software sobre el cual se trabajó ya tenía este tipo de arquitectura. Además que dicho patrón separa acciones sobre datos del sistema, lo que beneficia la presentación, interacción y manejo del mismo. Esto favorece la implementación ya que permite desarrollar nuevos modelos y controladores para ellos sin tener q realizar grandes modificaciones a las vistas. De esta manera quedan totalmente compatibilizados e independizados los modelos preexistentes con el nuevo desarrollo.

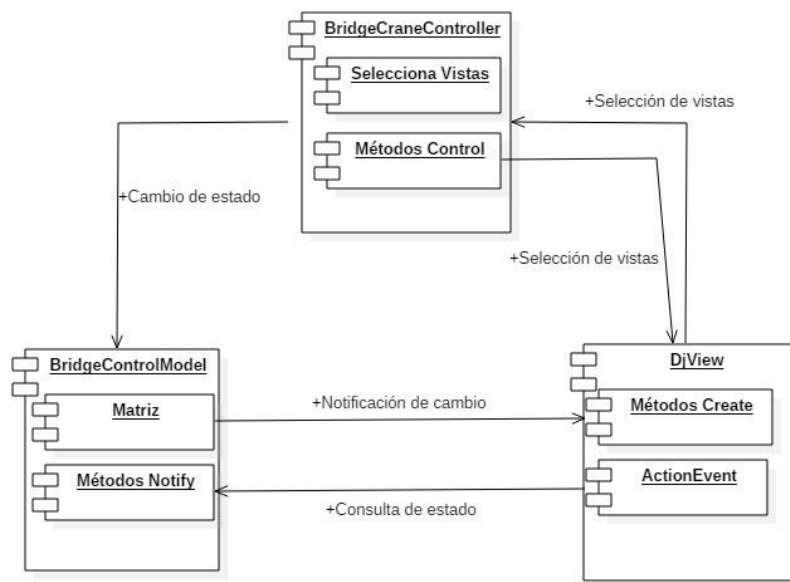


Imagen 7. Arquitectura general

Diagrama de despliegue

Se incorpora el siguiente diagrama para otorgar un mayor entendimiento de la disposición física de los componentes del sistema y su interacción.

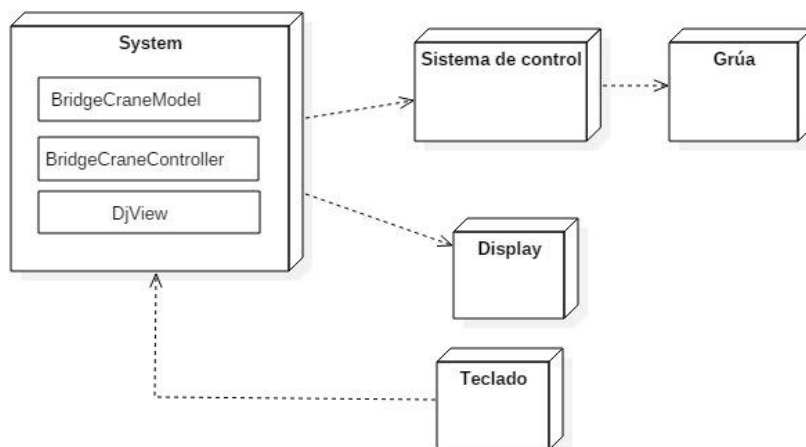


Imagen 8. Diagrama de despliegue

Diagrama de componentes

De la misma manera, se presenta el siguiente diagrama de componentes que brinda una vista de los subsistemas y de la comunicación entre ellos.

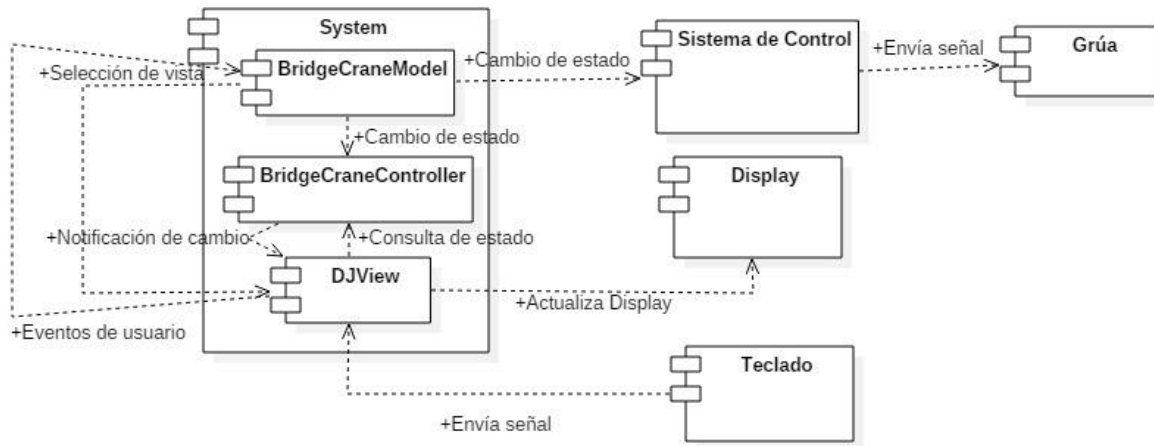


Imagen 9. Diagrama de componentes

Diseño e implementación

Diagramas de clases

Se presenta a continuación un diagrama de clase por cada modelo del sistema (los dos preexistentes y el desarrollado para cumplir con los requisitos):

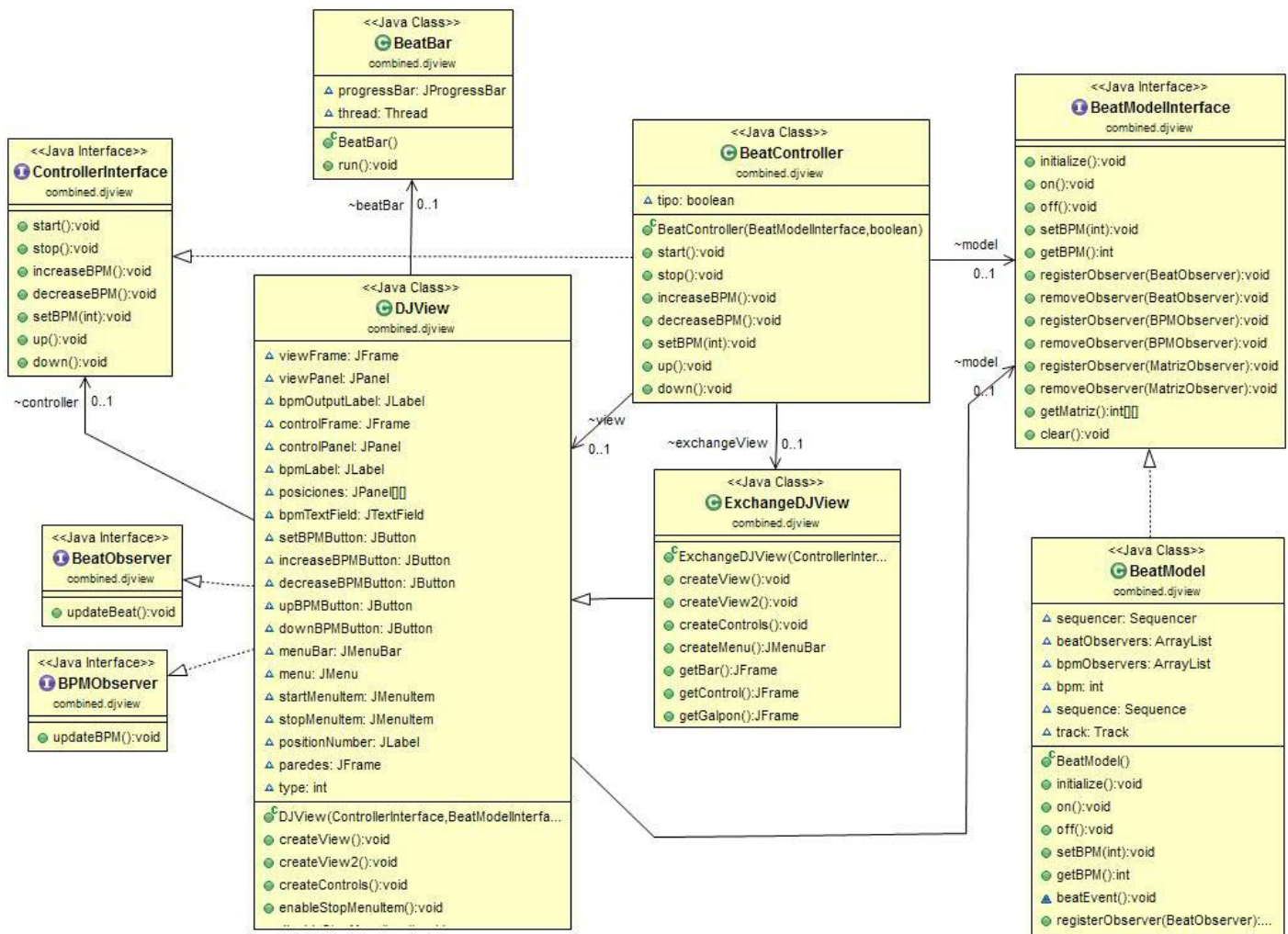


Imagen 10. Diagrama de clases de BeatModel

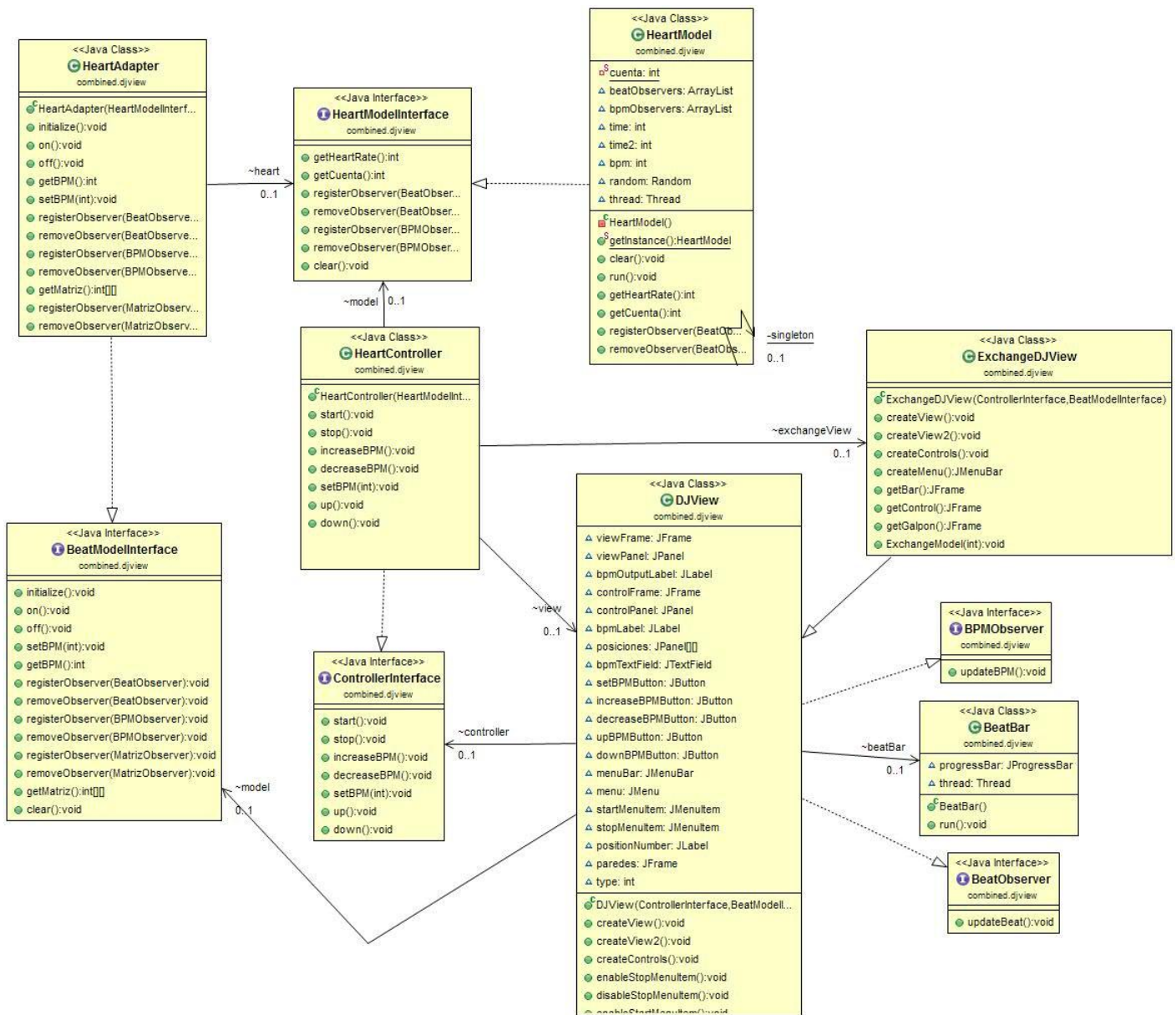


Imagen 11. Diagrama de clases de HeartModel

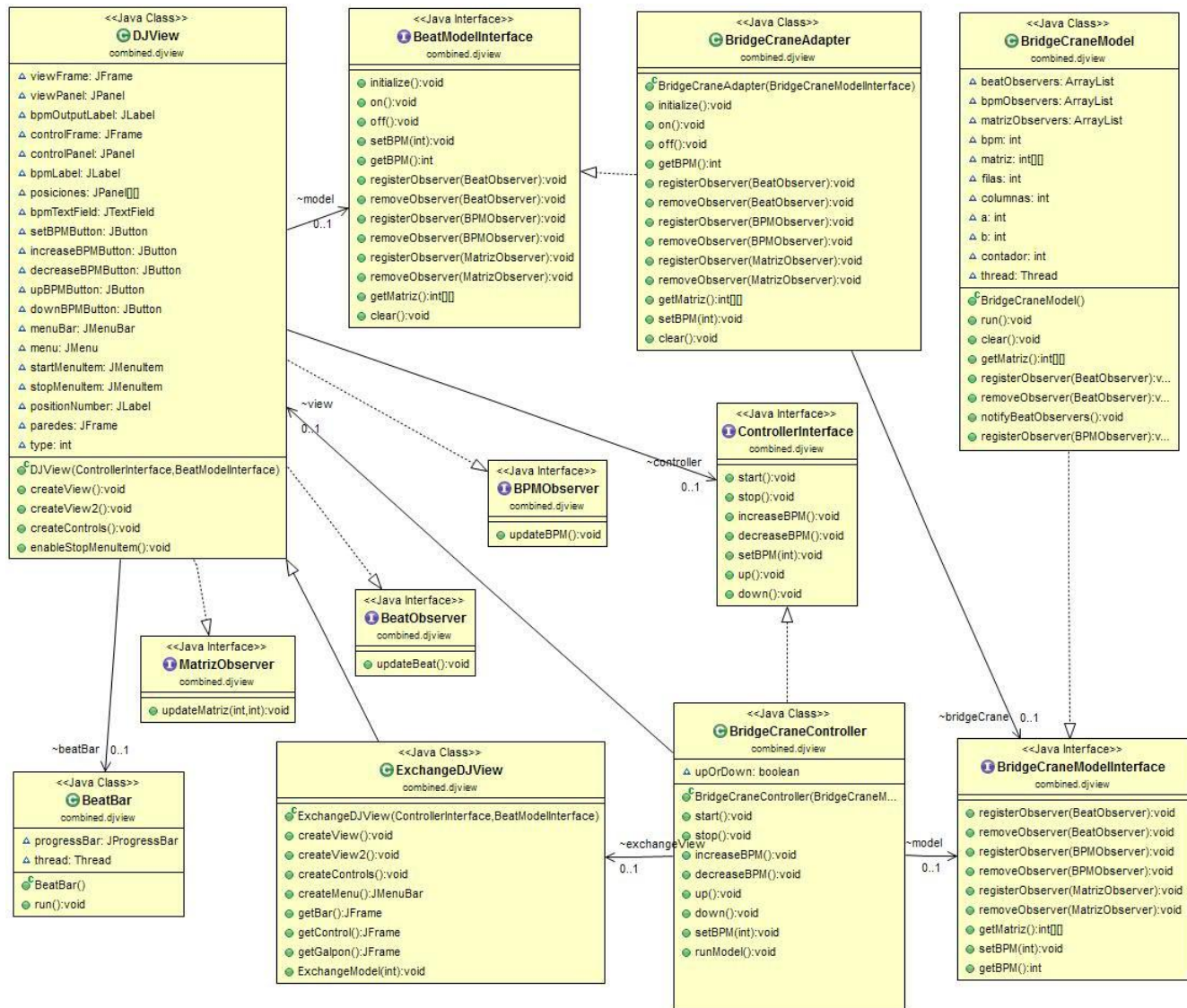


Imagen 12. Diagrama de clases de BridgeCraneModel

Diagramas de paquetes

El diagrama de paquetes de agrupamiento de clases es el siguiente:

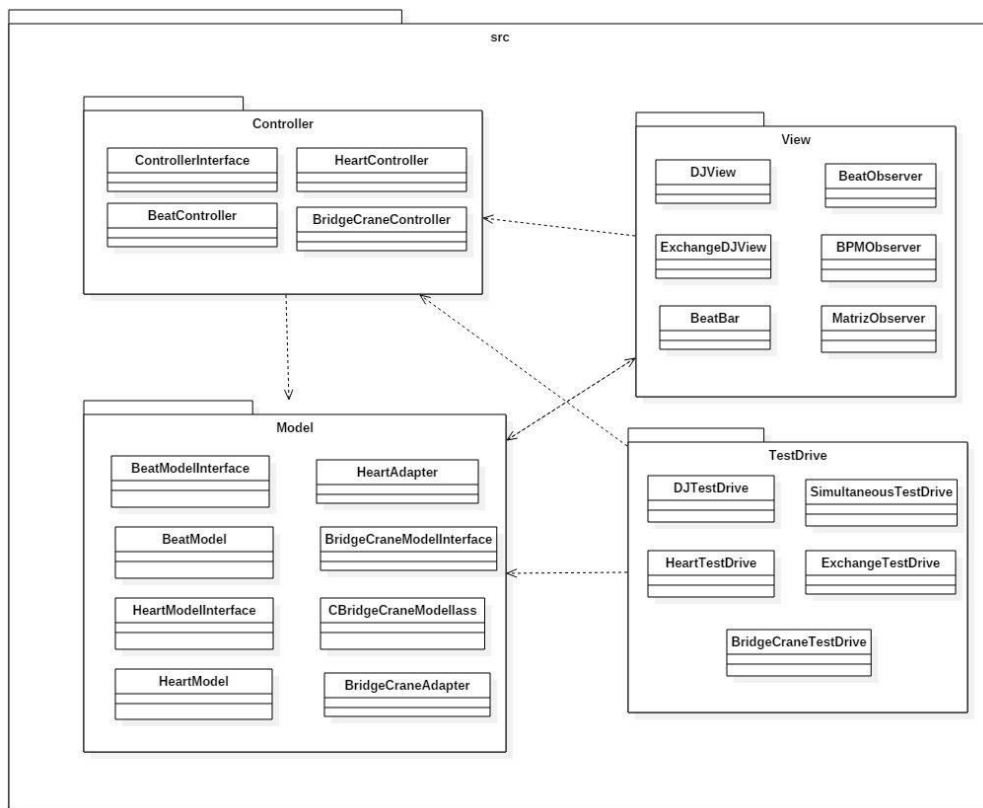


Imagen 13. Diagrama de componentes

Patrones de diseño

En la implementación de nuestro modelo, hay dos patrones de diseño que resaltan. Por un lado, el patrón Adapter, el cual permite establecer un puente o conexión entre una clase (o interfaz) padre, y una clase incompatible con el resto. En otras palabras, permite la cooperación entre clases para extender sus funcionalidades a clases de otros tipos.

Es decir, nuestro nuevo modelo (representado por la clase `BridgeCraneModel`) necesita hacer uso de la clase `BeatModelInterface`, pero sus métodos y funcionalidades son incompatibles con ésta. Por esta razón, se crea la clase `BridgeCraneAdapter`, que resuelve la incompatibilidad entre las clases, implementando los métodos de la interfaz para incorporar las funcionalidades de `BridgeCraneModel`.

Puede observarse a continuación un extracto del diagrama de clases, que representa las relaciones necesarias para la implementación del patrón Adapter.

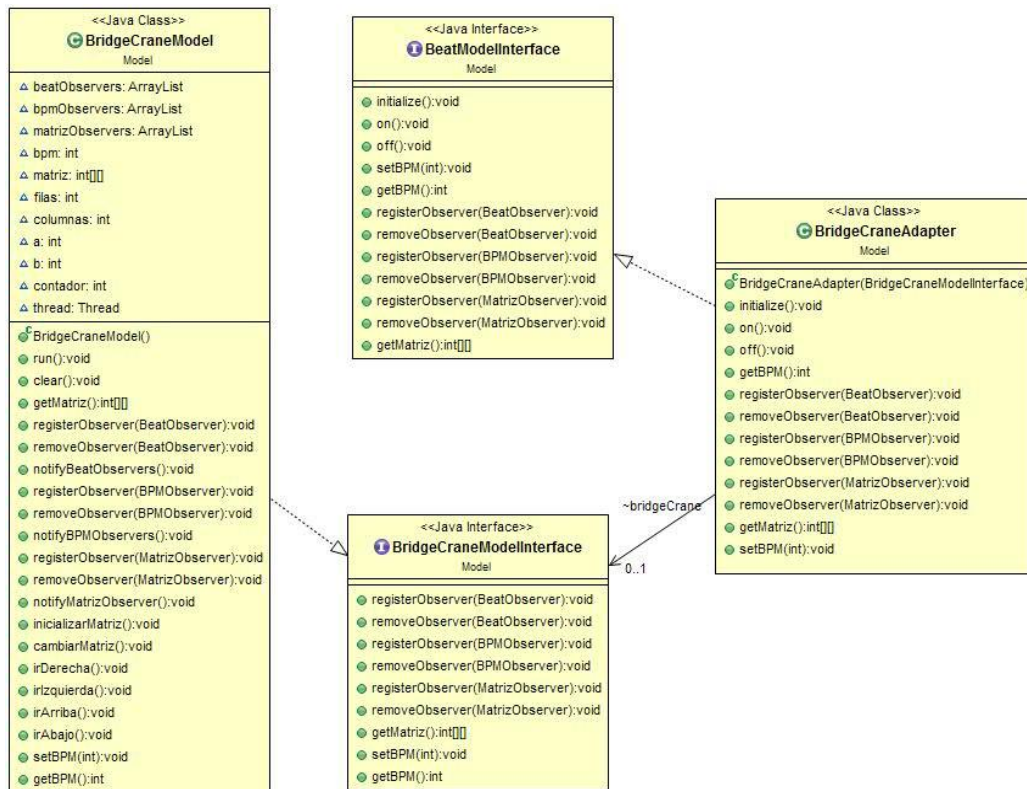


Imagen 14. Diagrama del patrón de diseño Adapter

Por otro lado, se hizo uso del patrón Observer. Este se utiliza para relacionar distintos objetos entre sí, en torno a uno principal, de modo que, cuando el principal cambie de estado, los demás sean informados de manera automática.

El patrón se compone de un objeto observado (en nuestro caso un objeto de la clase BridgeCraneModel), y de sujetos observadores (instancia de la clase DJView).

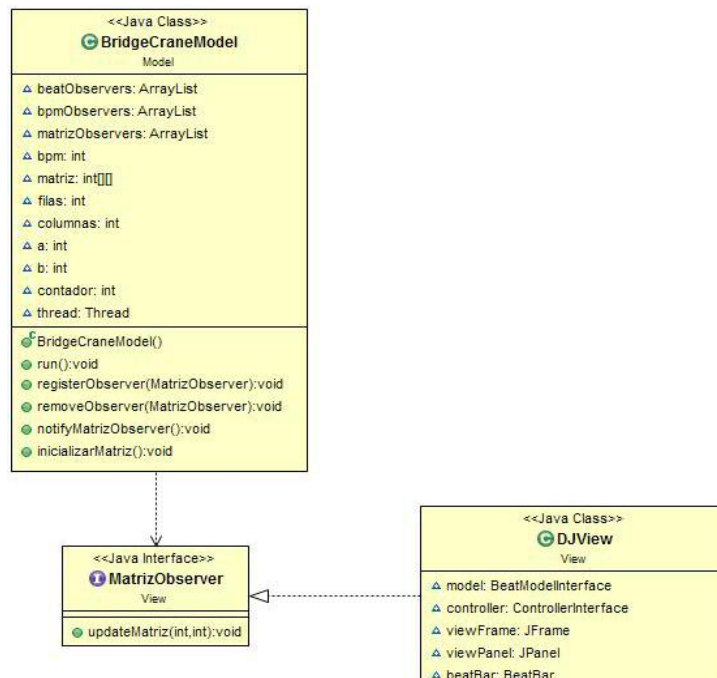


Imagen 15. Diagrama del patrón de diseño Observer

La clase a ser observada debe proveer métodos necesarios, como lo son `registerObserver()`, `removeObserver()` y `notifyObservers()`. Para el caso del observador de la matriz de nuestro modelo, y sus constantes cambios, se creó la interfaz `MatrizObserver`. Cuando una instancia de `DJView` quiera observar los cambios de la matriz contenida en `BridgeCraneModel`, deberá incorporarse como observador con el método `registerMatrizObserver(MatrizObserver o)`. Luego, el modelo informa de cambios al observador `DJView` con el método `notifyMatrizObservers()`.

Pruebas unitarias y del sistema

Pruebas unitarias

El resultado de todas las pruebas unitarias realizadas en el software se muestra a continuación:

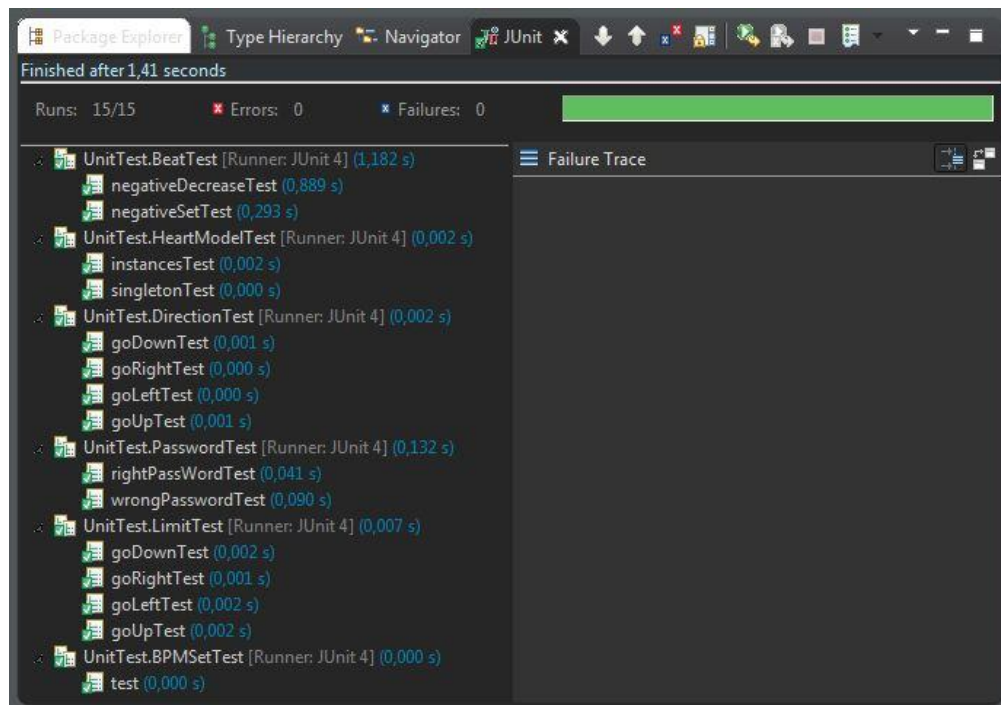


Imagen 16. Resultado de pruebas unitarias

Los tests realizados fueron los siguientes:

(UT1) BeatTest: negativeDecreaseTest

Comprueba que si la frecuencia del `BeatModel` está seteada en 0, una acción `decrease()` no disminuya el valor, sino que se mantenga en 0.

(UT2) BeatTest: negativeSetTest

Verifica que al intentar setear una frecuencia menor a 0 el valor no cambie, sino que se mantenga en el último seteado.

(UT3) HeartModelTest: singletonTest

Este test ha de verificar que al instanciar nuevamente la clase HeartModel, el objeto retornado sea siempre el mismo (variable static).

(UT4) HeartModelTest: instancesTest

Comprueba que una nueva instanciación de la clase HeartModel (intento de creación de un nuevo objeto) incrementa la cuenta.

(UT5) DirectionTest: goRightTest

Verifica la correcta actualización en la variable columna (y por ende en la matriz) al ejecutar al método irDerecha() y el movimiento està permitido según los límites.

(UT6) DirectionTest: goLeftTest

Verifica la correcta actualización en la variable columna (y por ende en la matriz) al ejecutar al método irIzquierda() y el movimiento està permitido según los límites.

(UT7) DirectionTest: goUpTest

Verifica la correcta actualización en la variable fila (y por ende en la matriz) al ejecutar al método irArriba() y el movimiento està permitido según los límites.

(UT8) DirectionTest: goDownTest

Verifica la correcta actualización en la variable fila (y por ende en la matriz) al ejecutar al método irAbajo() y el movimiento està permitido según los límites.

(UT9) PasswordTest: rightPasswordTest

Comprueba que el ingreso correcto de contraseña setee el valor esperado de frecuencia de modelo.

(UT10) PasswordTest: wrongPasswordTest

Comprueba que el ingreso incorrecto de contraseña mantenga el valor anterior de frecuencia, sin efectivizar cambios.

(UT11) LimitTest: goRightTest

Comprueba que, cuando la posición actual se encuentra en la ultima columna, una llamada al método irDerecha() no provoque ningún cambio.

(UT12) LimitTest: goLeftTest

Comprueba que, cuando la posición actual se encuentra en la primer columna, una llamada al método irIzquierda() no provoque ningún cambio.

(UT13) LimitTest: goUpTest

Comprueba que, cuando la posición actual se encuentra en la primer fila, una llamada al método irArriba() no provoque ningún cambio.

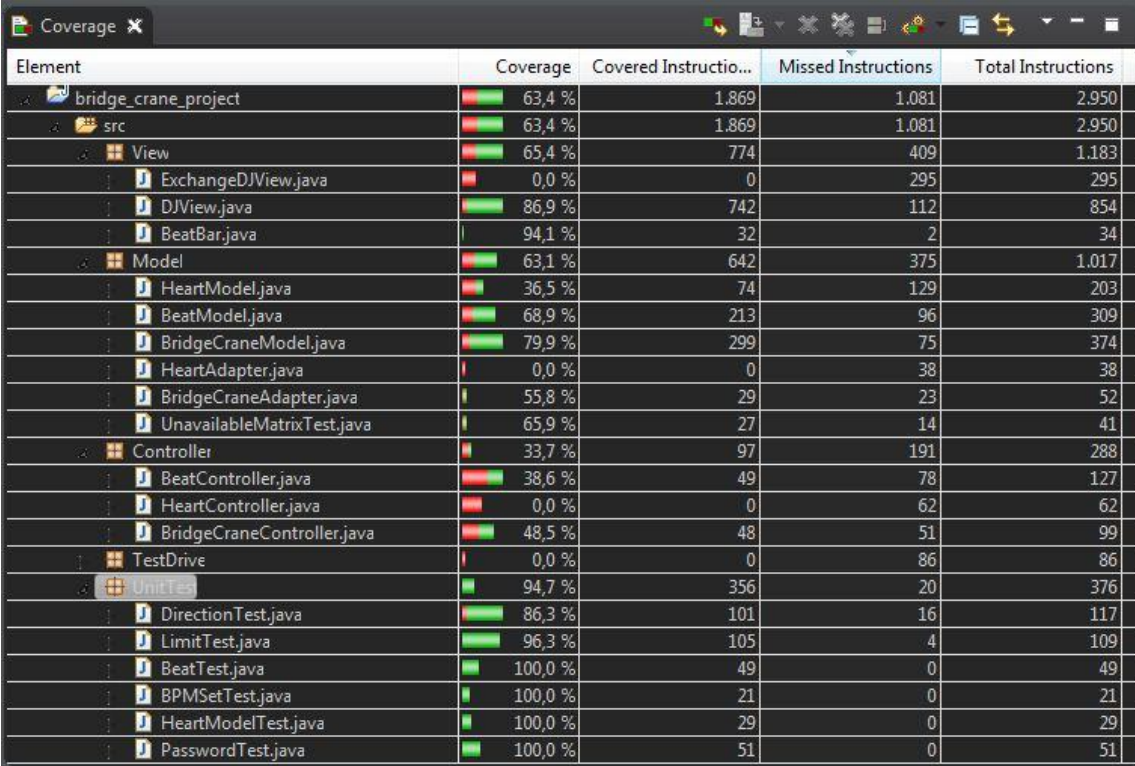
(UT14) LimitTest: goDownTest

Comprueba que, cuando la posición actual se encuentra en la última fila, una llamada al método irAbajo() no provoque ningún cambio.

(UT15) BPMTest: bpmSetTest

Verifica que al llamar a la función setBPM(), se modifique correctamente el campo BPM de la clase BeatCraneModel.

Se proveen a continuación los porcentajes de cobertura de código, generados a partir de la extensión Eclemma para Eclipse.



Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
bridge_crane_project	63,4 %	1.869	1.081	2.950
src	63,4 %	1.869	1.081	2.950
View	65,4 %	774	409	1.183
ExchangeDJView.java	0,0 %	0	295	295
DJView.java	86,9 %	742	112	854
BeatBar.java	94,1 %	32	2	34
Model	63,1 %	642	375	1.017
HeartModel.java	36,5 %	74	129	203
BeatModel.java	68,9 %	213	96	309
BridgeCraneModel.java	79,9 %	299	75	374
HeartAdapter.java	0,0 %	0	38	38
BridgeCraneAdapter.java	55,8 %	29	23	52
UnavailableMatrixTest.java	65,9 %	27	14	41
Controller	33,7 %	97	191	288
BeatController.java	38,6 %	49	78	127
HeartController.java	0,0 %	0	62	62
BridgeCraneController.java	48,5 %	48	51	99
TestDrive	0,0 %	0	86	86
UnitTest	94,7 %	356	20	376
DirectionTest.java	86,3 %	101	16	117
LimitTest.java	96,3 %	105	4	109
BeatTest.java	100,0 %	49	0	49
BPMSetTest.java	100,0 %	21	0	21
HeartModelTest.java	100,0 %	29	0	29
PasswordTest.java	100,0 %	51	0	51

Imagen 17. Porcentajes de cobertura

Casos de prueba del sistema

Se muestran a continuación los casos de prueba del sistema realizados, con instrucciones para su comprobación y los resultados obtenidos por los miembros del equipo.

ST1	
Descripción	El sistema deberá impedir el movimiento de la grúa cuando está en proceso de carga.
Instrucciones de ejecución	<ol style="list-style-type: none">1. Ejecutar el archivo BridgeCraneTestDrive.jar2. Mover la grúa a la posición deseada.3. Ingresar la contraseña en el cuadro de texto.4. Cuando esté en proceso de carga, intentar mover la grúa a otra posición.
Resultado esperado	La grúa debería mantenerse en la misma posición.
Pass/Fail	Pass

ST2	
Descripción	El sistema debe crear solo una instancia estática de la clase HeartModel. En el caso de que se intente crear otra, se muestra el número de intentos en la ventana BeatBar.
Instrucciones de ejecución	<ol style="list-style-type: none">1. Ejecutar el archivo HeartDriveTest.jar2. Presionar el botón >>.3. Comprobar que el número mostrado en la ventana BeatBar incrementa cada vez que se presiona el botón.
Resultado esperado	El valor mostrado en la BeatBar debería incrementarse cada vez que se presiona el botón >>.
Pass/Fail	Pass

ST3	
Descripción	El sistema debe permitir el movimiento de la grúa utilizando los botones de control de la vista.
Instrucciones de ejecución	<ol style="list-style-type: none"> 1. Ejecutar el archivo BridgeCraneTestDrive.jar 2. Presionar alguno de los botones de dirección ubicados en la ventana de control. 3. Comprobar que la grúa se desplace en la dirección seleccionada.
Resultado esperado	La grúa debería desplazarse en la dirección seleccionada.
Pass/Fail	Pass

ST4	
Descripción	El sistema debe permitir, a través de un menú, cambiar de modelo en tiempo de ejecución.
Instrucciones de ejecución	<ol style="list-style-type: none"> 1. Ejecutar el archivo ExchangeableTestDrive.jar 2. Desplegar el menú "Model" de la ventana BeatBar. 3. Seleccionar el modelo deseado.
Resultado esperado	Se debería ejecutar el modelo seleccionado.
Pass/Fail	Pass

ST5	
Descripción	El sistema debe permitir ejecutar los tres modelos de manera simultánea.
Instrucciones de ejecución	<ol style="list-style-type: none"> 1. Ejecutar el archivo SimultaneousTestDrive.jar
Resultado esperado	Los tres modelos deberían aparecer en pantalla.
Pass/Fail	Pass

ST6	
Descripción	El sistema debe tener un control de seguridad (mediante solicitud de contraseña) para bajar o subir la grúa.
Instrucciones de ejecución	<ol style="list-style-type: none"> 1. Ejecutar el archivo BridgeCraneTestDrive.jar 2. Ingresar una contraseña en el cuadro de texto. 3. Presionar botón set.
Resultado esperado	Cuando la contraseña es correcta, el sistema cambia de estado. Cuando es incorrecta, no realiza ningún cambio.
Pass/Fail	Pass

ST7	
Descripción	El sistema debe evitar, cuando se abren múltiples ventanas, que queden solapadas.
Instrucciones de ejecución	<ol style="list-style-type: none"> 1. Abrir cualquiera de los archivos ejecutables .jar provistos en el último release.
Resultado esperado	Al desplegarse las ventanas, deberían estar distribuidas de tal manera que puedan visualizarse sin solaparse.
Pass/Fail	Pass

ST8	
Descripción	El sistema debe impedirle a la grúa ir más allá de sus límites.
Instrucciones de ejecución	<ol style="list-style-type: none"> 1. Ejecutar el archivo BridgeCraneTestDrive.jar 2. Intentar mover la grúa fuera de los límites del recuadro con los botones de la ventana de control.
Resultado esperado	Cuando se está al borde del recuadro y se intenta desplazarse más allá del mismo, no se produce ningún cambio.
Pass/Fail	Pass

Smoke o Sanity Tests

El conjunto de casos de prueba que componen este tipo de tests, demostrando que se cumplen los requerimientos principales del sistema es aquel conformado por ST2, ST3, ST4 y ST5.

Matriz de trazabilidad actualizada

		Establecer Posición	Ingresar Contraseña	Bajar Grúa	Subir Grúa	Mostrar Posición	Mostrar Disponibilidad
Requerimientos Funcionales	1	1					
	2						
	3	1					
	4					1	
	5					1	1
	6						
	7						
Requerimientos no funcionales	1						
	2		1	1	1		
	3	1					
	4			1	1		
	5	1				1	1
Unit Tests	1						
	2						
	3						
	4						
	5	1					
	6	1					
	7	1					
	8	1					
	9		1	1	1		
	10		1				
	11	1					
	12	1					
	13	1					
	14	1					
	15			1	1		
System Tests	1	1		1	1	1	
	2						
	3	1				1	
	4						
	5						
	6		1	1	1		1
	7					1	1
	8	1				1	

Datos históricos

Integrantes	Lun 22	Mar 23	Mie 24	Jue 25	Vie 26	Sab 27	Dom 28	Total(Hs)
Juan	3	8	6,5	11	9	5	3	45,5
Joaquín	3	3,5	7	11	9	5	3	41,5
Tomás	3	7	7	11	8,5	5	3	44,5
Santiago	3	8	7,5	11	9	5	3	46,5

Información adicional

En la realización del práctico pudimos familiarizarnos con las distintas etapas de un proceso de desarrollo de software, entre las cuales podemos mencionar:

- La herramienta de control de versiones GitHub junto con su funcionalidad que permite un correcto seguimiento y notificación de bugs.
- El desarrollo de diagramas UML con la herramienta StarUML que nos permite ver el programa y sus funcionalidades desde distintos puntos de vista.
- La implementación de distintos tipos de patrones de diseño para la elaboración del proyecto, permitiéndonos simplificar y organizar en gran medida nuestro trabajo.
- Utilización de la herramienta JUnit y EcEmma provista por el software Eclipse, las cuales nos permitieron desarrollar los UnitTests y observar el porcentaje de cobertura del código.
- Comprensión del patrón de arquitectura MVC.
- El desarrollo de una interfaz gráfica.

Por otra parte, al concluir con el proyecto vimos que lo que más problemas nos generó fue el hecho de comenzar a solo una semana de su entrega, lo que nos llevo a tener que dedicarle una fuerte carga horaria de trabajo diaria. Este problema provoco que realicemos refactorizaciones en el código que no hubieran sido necesarias de haber comprendido más tempranamente el funcionamiento del programa, y como consecuencia faltaron realizar algunas. Además, no se pudo seguir correctamente el plan de manejo de configuraciones, reflejado principalmente en el uso de la herramienta de reporte de bugs de GitHub.