



Universidad  
Nacional de  
Córdoba

## Facultad de Ciencias Exactas, Físicas y Naturales

### Trabajo Práctico Final Ing. De Software

Profesor: Miceli, Martín

Integrantes: González, Manuel Simón

Krenz, Matías Ezequiel

Rivero, Franco Fabián

Grupo: Serox

# Introducción

## Inicios

Con el desarrollo del presente informe se llevará a cabo un análisis de todas las fases necesarias para el desarrollo, el modelado y la implementación de un sistema de software. Nuestro objetivo principal, es implementar las metodologías de la Ingeniería de Software para crear un sistema de manera eficiente y organizada, tomando como base un ejemplo de proyecto desarrollado en el Libro “Head First Design Patterns”. Se nos encomendó como tarea principal, implementar mejoras en dicho proyecto y crear un nuevo modelo, añadiendo funcionalidad extra, las cuales demandaran la utilización de nuevos patrones de diseño como Singleton, Testing, Control de Configuraciones, Documentacion, Strategy, entre otros.

## Creación del nuevo modelo

### Elección

Como base inicial, queríamos crear un modelo que se asemejara a un caso particular de la vida real, pero que a su vez fuera funcional y útil. Tomamos como iniciativa, la situación socio-económica que se vive en el presente en país, donde la inseguridad e inestabilidad económica obliga a diversas personas a delinquir para poder subsistir el día a día.

Con esto en mente, se llego a pensar que la mejor elección, sería un sistema de seguridad. Por ello, se decidió crear una simulación de un sistema de ventas en un supermercado. Dicho sistema está destinado a la utilización exclusiva en supermercados o comercios en los cuales se utilizan cajas y sistema de seguridad por medio de sensores que detectan productos que no han sido procesados por el software de la caja con anterioridad.

Los potenciales usuarios del mismo pueden ser:

- Supermercados
- Farmacias
- Tiendas de tecnología
- Tiendas de ropa

Con esta simulación se pretende mejorar y proteger a las personas que viven de su comercio o negocio familiar. Todo relacionados con un fin en particular, evitar el robo de mercadería y proteger sus productos.

### **Explicación**

Como se detallo en el punto anterior la aplicación creada está orientada a proteger a los comerciantes. Está basada en una arquitectura denominada Model View Controller (MVC). Por ello, las partes del sistema estarán subdivididas pero interconectadas entre sí para contribuir a un mismo objetivo. Dicha arquitectura se basa en la creación de una interfaz de usuario (View), a la administración de datos (Model) y el modulo que gestiona y comunica entre las ya mencionadas partes (Controller). Las partes principales de nuestra aplicación son las siguientes:

- SuperModel
- SuperModelInterface
- SuperController
  - SuperView

### **Análisis de modelos previstos**

El modelo principal a partir del cual se creó la aplicación es un sistema en el cual se simulan los latidos del corazón y otro similar en donde se cuenta con la simulación de los golpes de una batería. Ellos están basados también en una arquitectura Model View Controller.

El primero, una vez finalizado el propósito del presente y, al realizar una ejecución, muestra dos ventanas. Una da opciones al usuario (interface)

para manipular los controles de la aplicación. En el caso del HeartModel, nos permite mediante el botón ">>" crear una nueva instancia del objeto luego de haber agregado el patrón de diseño Singleton haciendo que el objeto se cree solo una vez. En la vista (View) muestra una barra que simula los latidos del corazón, en proporción del valor predefinido de BPM. También se incluyo por medio de modificaciones en el código, un JLabel revelando la palabra "Instancias" donde muestra las veces que se intenta crear un nuevo objeto HeartModel (al presionar el botón ">>" de la interface).

Por otro lado, al ejecutar el segundo modelo se despliegan dos ventanas, una con la interface de usuario en donde permite introducir la cantidad de BPM que queremos reproducir mediante un botón SET, pudiendo a partir de este incrementar o disminuir ese valor introducido con los botones (">>") o ("<<") respectivamente. En la vista (View) de este modelo, veremos una ventana en la cual se muestra una barra similar a la del HeartModel que simula los golpes de la batería y también podremos observar una lectura donde se indican las BPM con las que se está trabajando en ese preciso momento.

# **1. Nota de entrega**

## **1.1. Versiones y sus implementaciones**

El trabajo final será entregado en el momento del reléase en un archivo comprimido en formato ".zip" conteniendo tanto el software realizado en su versión final (en formato .jar), la documentación del mismo y esquemas representativos de los pasos intermedios realizados para poder realizar esta versión. A continuación se detalla el avance en cada una de las versiones realizadas:

- v1.0: Se agrego el Singleton para poder crear una sola instancia del Heart model.
- v1.1: Se agregaron los observadores.
- v1.2: Se implementan los cambios en la ventana del Heart (mostrar instancias y aumentarlas con el botón ">>").
- v2.0: Se comienza a desarrollar el nuevo modelo "Supermercado".

- v2.1: Se implementan todas las funcionalidades faltantes al modelo “Supermercado”.
- V3.0: Se intenta implementar la “MultiView” (Selección de modelo en una misma ventana”)

## 1.2. Bugs conocidos

Al momento del release final, no pudimos eliminar un bug: El de la “MultiView”.

Este bug se trabajo por horas para intentar solucionarlo buscando múltiples soluciones, pero llegado al momento de entrega del trabajo, el bug permanecía. Por lo que se decidió presentar el trabajo manteniendo este bug.

## 1.3. Lugar/link del entregable y de las instrucciones de instalación

Los “releases” se entregaran en formato ejecutable .jar y distribuido de forma comprimida en formato .zip.

Para la obtención de los mismos se empleara un link de descarga y además se anexará la documentación correspondiente.

Link de descarga:

<https://github.com/FrancoRivero/IngenieriaEnSoftwareTPfinal>

### Para su instalación:

- ✓ Sólo es necesario poseer la última versión de la maquina virtual de java.
- ✓ Descargue los archivos correspondientes al directorio deseado.
- ✓ Descomprima los archivos embebidos dentro del archivo .zip.
- ✓ Ejecute el archivo java .jar correspondiente.

## 2. Manejo de las configuraciones

### 2.1. Herramientas para control de versiones

Por problemas técnicos y de tiempo, decidimos realizar el trabajo de forma conjunta y almacenando las versiones por medio de carpetas en una PC determinada y luego distribuidas a cada uno de los integrantes. Se trabajo en cada versión hasta finalizarla, de manera que la misma quede completa y sobre ella se puedan realizar nuevas implementaciones de nuestro software.

Si bien se separaron correctamente las versiones, no fueron éstas almacenadas en algún repositorio remoto a medida que se realizaban los releases.

Luego de finalizar el trabajo, se decidió subir a GitHub (de forma conjunta) todas las versiones realizadas en carpetas separadas y manteniendo un protocolo preestablecido.

Como se menciona anteriormente, el repositorio remoto elegido fue GitHub, utilizando el TortoiseGit para realizar los commit de los archivos al mismo.

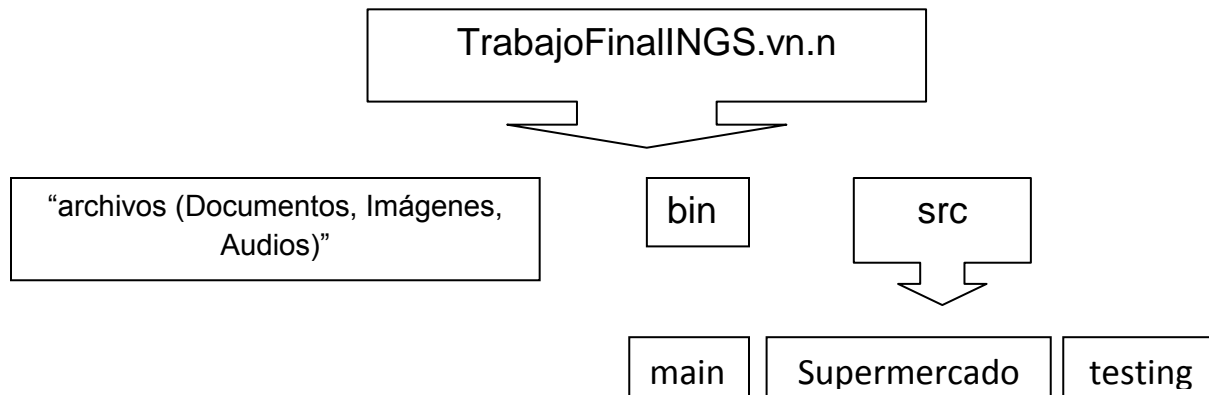
La dirección GitHub es:

<https://github.com/FrancoRivero/IngenieriaEnSoftwareTPfinal>

Cabe mencionar que nuestra decisión también se tomo por razones de seguridad. Como bien se dijo, las versiones fueron cargadas a último momento antes de la entrega, ya que nuestra cuenta GitHub no es paga, por lo que el repositorio cargado puede ser visualizado por cualquier usuario del servicio.

## 2.2. Esquema de directorios

El esquema de directorios que contiene el repositorio es el siguiente (léase de forma descendente teniendo en cuenta los Sub-ítems):



**TrabajoFinalINGS.vn.n**: Carpeta raíz del proyecto. (n.n indica la versión de cada commit).

**bin**: carpeta de archivos compilados.

**src**: carpeta de archivos de código fuente.

**main**: carpeta alojamiento de las vistas, los controladores y los modelos correspondientes a Heart model y a Beat Model.

**Supermercado**: carpeta de alojamiento de la vista, el controlador y el modelo correspondientes al modelo de supermercado.

**testing**: carpeta de alojamiento de los test unitarios.

## **2.3. Normas de etiquetado y nombramiento**

La modalidad seleccionada para el etiquetado de versiones es la siguiente:

Se implemento el mismo nombre para todas las versiones “TrabajoFinalINGS” y a continuación del mismo se concateno la letra “v” (versión) y junto a ella, se comenzó a enumerar la versión correspondiente con dos números separados por un punto.

El primer digito representa cambios importantes en el diseño del código. Por lo que dicho digito lo utilizamos para representar un ítem del enunciado del trabajo (si es que éste se lo tomaba como un cambio importante para el sistema).

El segundo digito representa modificaciones funcionales y hace referencia a pequeños agregados a la versión anterior.

## **2.4. Plan de esquema de ramas y política de fusión de archivos**

Si bien, se pasará a la creación de una nueva versión cuando la actual está funcionando al 100%, es decir, que la misma esté libre de bugs y haya pasado de manera correcta los test a la que se la someta, se tomó como programación estable a una rama principal en donde se almacena y redirige código funcionando y, una rama secundaria en donde se va implementando código nuevo generalmente sin testear.

Una vez que la programación en la rama secundaria se la etiqueta como estable y sin bugs, se realiza un mergeo de este código a la rama principal, donde se le añade el lazo necesario para interconectarlas.



## 2.5. Formato de entrega e instrucciones de instalación

Los “releases” se entregaran en formato ejecutable .jar y distribuido de forma comprimida en formato .zip.

Para la obtención de los mismos se empleara un link de descarga y además se anexará la documentación correspondiente.

Link de descarga:

<https://github.com/FrancoRivero/IngenieriaEnSoftwareTPfinal>

Para su instalación:

- ✓ Sólo es necesario poseer la última versión de la maquina virtual de java.
- ✓ Descargue los archivos correspondientes al directorio deseado.
- ✓ Descomprima los archivos embebidos dentro del archivo .zip.
- ✓ Ejecute el archivo java .jar correspondiente.

## 2.6. Integrantes del equipo y forma de contacto

Integrantes	Contacto	Roll
González, Manuel Simón	Email: mgonzalez1992@outlook.com Facebook Whatsapp	Diseñador grafico Arquitecto
Krenz, Matías Ezequiel	Email: matiaskrenz@hotmail.com Facebook Whatsapp	Programador Tester Arquitecto
Rivero, Franco Fabián	Email: francorivero2012@gmail.com Facebook Whatsapp	Programador Tester Arquitecto

La forma de contacto utilizada por los miembros del grupo fueron redes sociales (facebook) y whatsapp, por medio de estos se acordaban los días y lugares de reunión ya que se avanzaba en el proyecto de manera conjunta. Al comenzar cada reunión, el equipo se dividía las tareas a realizar por cada miembro.

## **2.7. Periodicidad de las reuniones**

Las reuniones están comprendidas a lo largo de una jornada de 4 días de duración. Las reuniones fueron diarias.

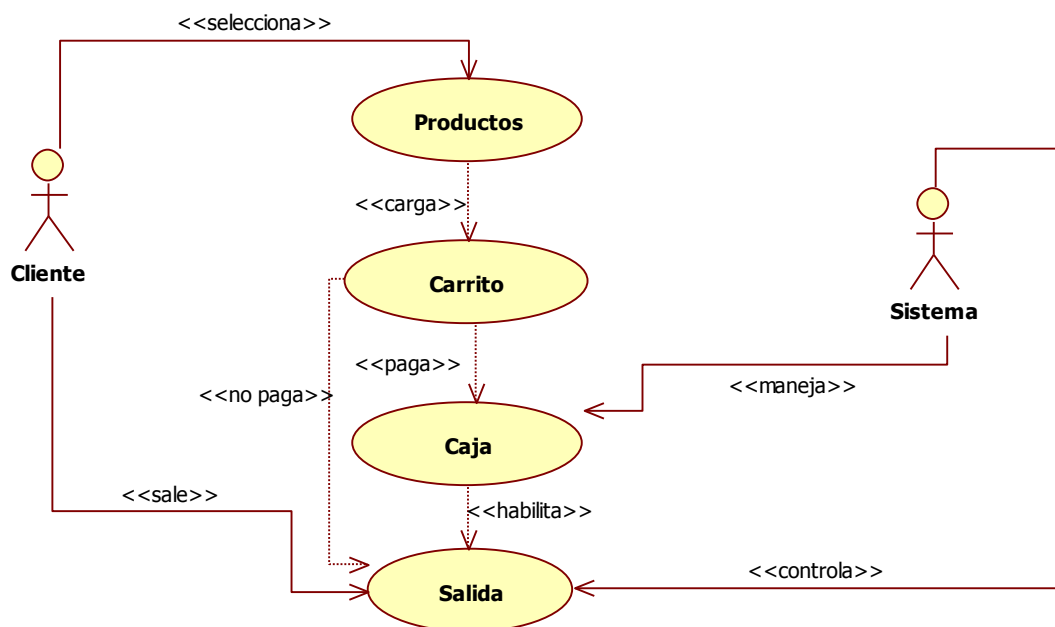
## **2.8. Seguimiento de Bugs. Método seleccionado**

El método seleccionado por el equipo fue simple, ya que a medida que se avanza en el código, se fueron realizando pruebas de implementación manual o utilizando la herramienta propia de java. En caso de encontrar un error, este se solucionaba y se procedía con normalidad en las modificaciones e incorporaciones de código.

## 3.Requerimientos

### 3.1. Diagrama caso de uso

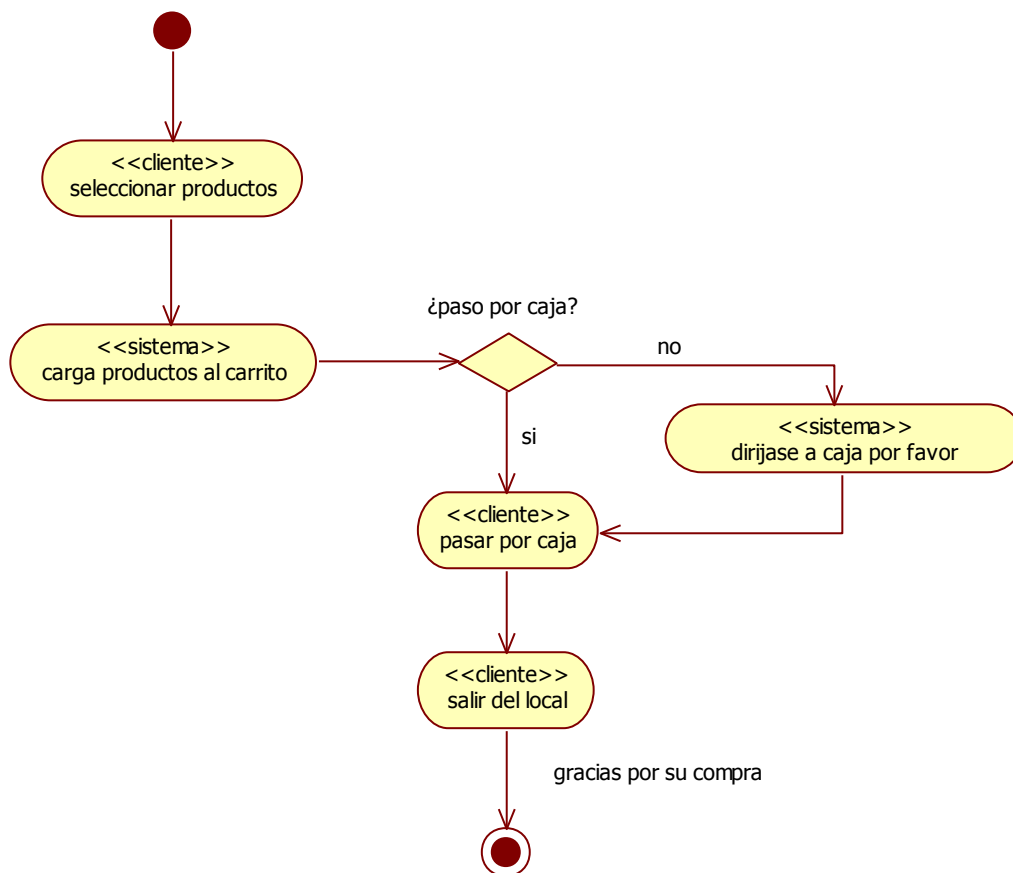
Se adjunta diagrama de casos de uso. Con él, es posible distinguir los diferentes sujetos que interactúan con el sistema y como se relacionan entre sí. El cliente, por su parte, es el encargado de llevar a cabo el uso de las funcionalidades que el sistema ofrece. Por su parte, el sistema, debe comportarse acorde a lo que el cliente realice o requiera en cierta interacción.



### 3.2. Diagrama de actividades

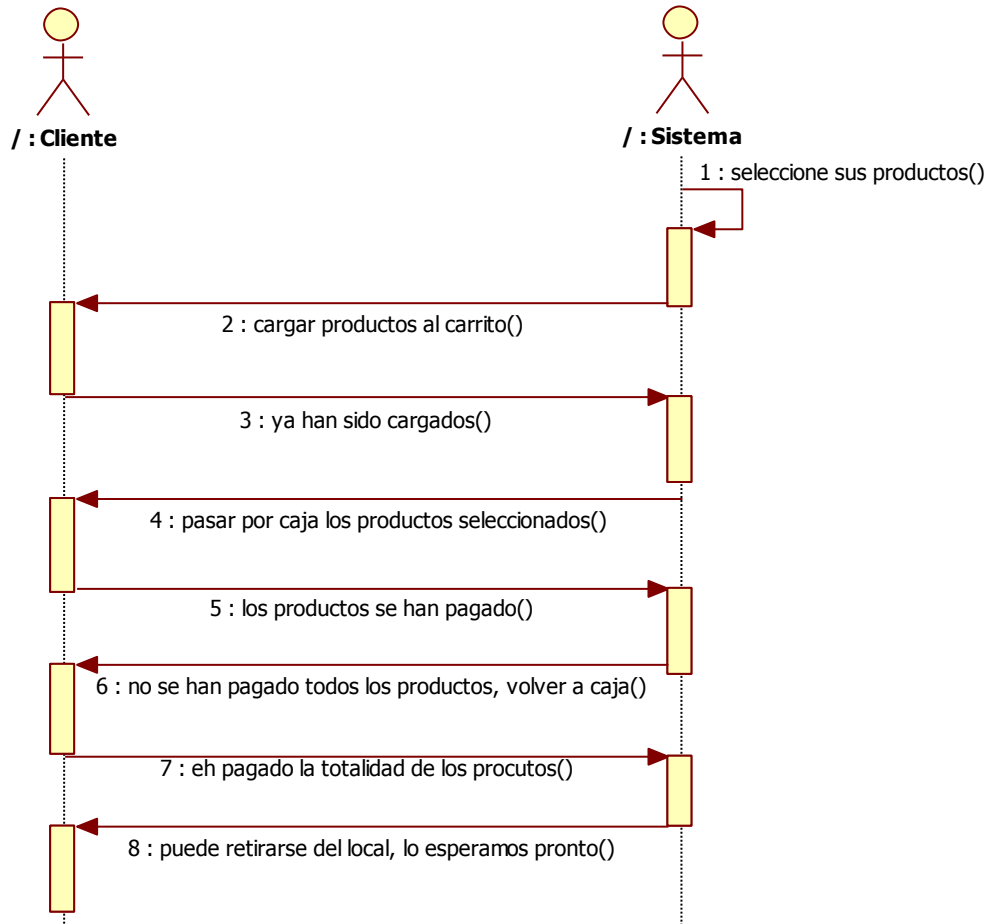
Como bien se dijo, el cliente es el encargado de manipular y controlar el sistema acorde a sus necesidades.

Observemos el siguiente diagrama. El cliente debe seleccionar los productos disponibles en la góndola para agregarlos al carrito (el sistema los cargará automáticamente al sistema). Luego, el cliente decide si desea salir del local sin pagar los productos o si desea salir pasando previamente por caja. El sistema reaccionará de una u otra forma dependiendo de la acción del cliente.



### 3.3. Diagrama de secuencia

Aquí se logra observar la interacción entre cliente-sistema y, los pasos necesarios para lograr una ejecución completa y satisfactoria del proyecto.



### **3.4. Requerimientos**

#### **Requerimientos funcionales**

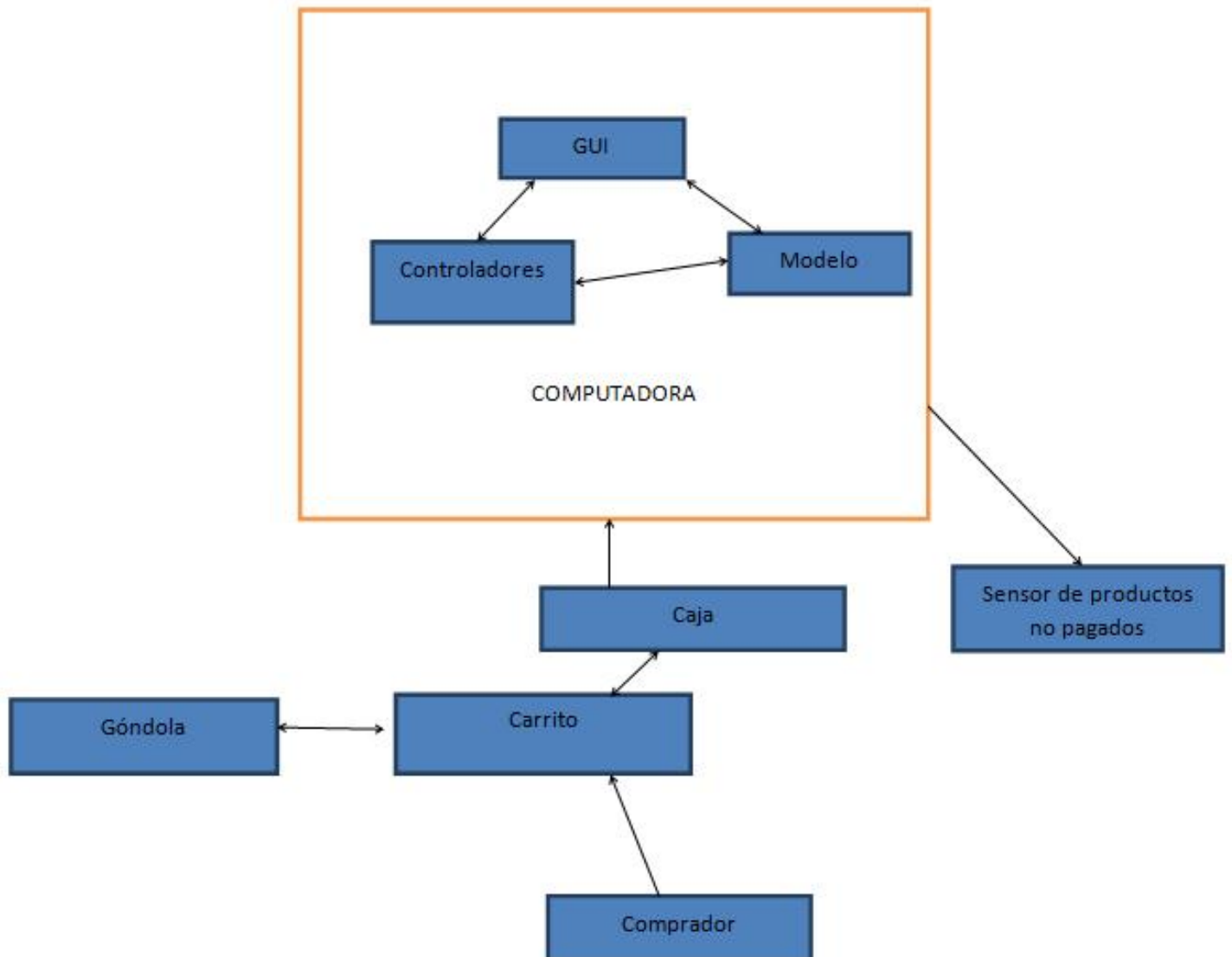
- El sistema debe crear una única instancia (usando el patrón Singleton).
- El sistema debe generar nuevas instancias cada vez que se clickea en el botón ">>" de la ventana HeartBeat.
- El sistema debe tener una caja para pagar los productos.
- El sistema debe tener un regulador del tamaño del carrito.
- El sistema debería mostrar un mensaje de aviso si el sensor de salida detecta mercadería no pagada.
- El sistema debe mostrar en pantalla los productos agregados al carrito.
- El sistema debe mostrar en pantalla un mensaje de agradecimiento si el sensor no detecta mercadería no pagada.
- El sistema debería mostrar la cantidad de productos pagados.

#### **Requerimientos no funcionales**

- El tiempo de respuesta debe ser inmediato si el sensor detecta mercadería no pagada.
- El sistema debe ser de fácil utilización y entendimiento.
- El sistema no debe dejar agregar más productos al carrito si el mismo ya se encuentra lleno.
- El sistema no debería fallar en el sistema de sensores, de lo contrario podría culparse a un cliente que no tuvo un acto de maldad
- El sistema debe dejar agregar productos al carrito de forma instantánea
- El sistema debe dejar pagar los productos a una velocidad mínima de un producto por segundo

### 3.5. Diagrama de arquitectura preliminar

Sobre arquitectura se hablara en el siguiente punto. Aquí presentamos un diagrama “de prueba” de la arquitectura.



## 4.Arquitectura

Intentando guiarnos de los requerimientos del sistema, algunas funciones hechas en nuestro programa son ajenas a la parte del software por lo que no las tendremos en cuenta para la arquitectura.

Como el software es mayor parte una interfaz gráfica (haciendo alusión al requerimiento de usabilidad), decidimos usar Model View Controller, que se implementa a partir del patrón Observer y el patrón Strategy.

Para la implementación de nuestro sistema, fue necesario crear, testear e implementar múltiples vistas, modelos y controladores, por lo que el patrón MVC nos fue de gran ayuda.

Adentrándose más en su definición, podemos decir que es un patrón de arquitectura de software que separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones.

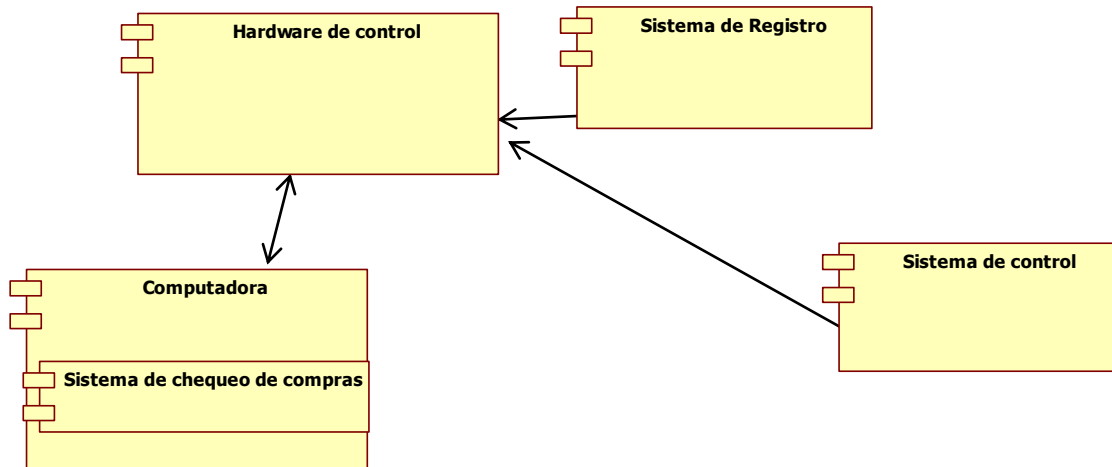
El mismo se basa de los 3 componentes anteriormente mencionados:

Modelo: Es la representación de la información con la cual el sistema opera, por lo tanto gestiona todos los accesos a dicha información, tanto consultas como actualizaciones, implementando también los privilegios de acceso que se hayan descrito en las especificaciones de la aplicación (lógica de negocio). Envía a la 'vista' aquella parte de la información que en cada momento se le solicita para que sea mostrada (típicamente a un usuario). Las peticiones de acceso o manipulación de información llegan al 'modelo' a través del 'controlador'.

Controlador: responde a acciones del usuario e invoca peticiones al “modelo” cuando se hace alguna solicitud sobre la información; puede también enviar comandos a su vista asociada si se solicita un cambio en la forma en que se presenta en modelo. En definitiva el controlador hace de intermediario entre la vista y el modelo.

Vista: presenta al modelo en un formato adecuado para interactuar (interfaz de usuario) requiere del modelo la información que debe representar a la salida.



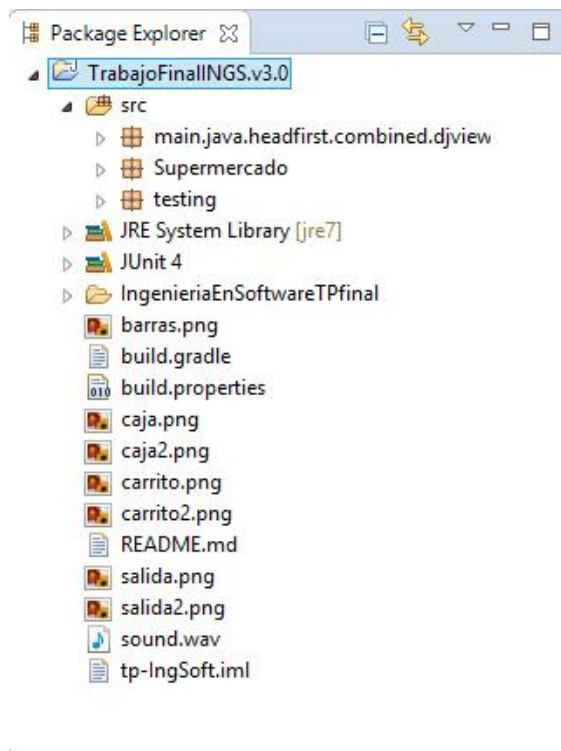


Vemos que la estructura no solo depende de la parte de software sino que también tenemos una gran parte de hardware necesario. Tal es así que el sistema de control está hecho a base de sensores y el sistema de registro a base de memorias que son las que registran la cantidad de compras y objetos pagados. De este modo si esta todo como debe ser el sistema no se activa en caso contrario debería saltar una alarma que alerte al personal de seguridad que los objetos que lleva un determinado sujeto no fueron pagados previamente en caja.

## 5. Diseño e implementación

### 5.1. Diagrama de paquetes





































Inicialmente presentaremos el diagrama de paquetes de agrupamientos de clases para poder entender luego, como distribuimos, agrupamos y exponemos los diagramas de clases y de objetos.



Como observamos, el trabajo se dividió en 3 paquetes diferentes.

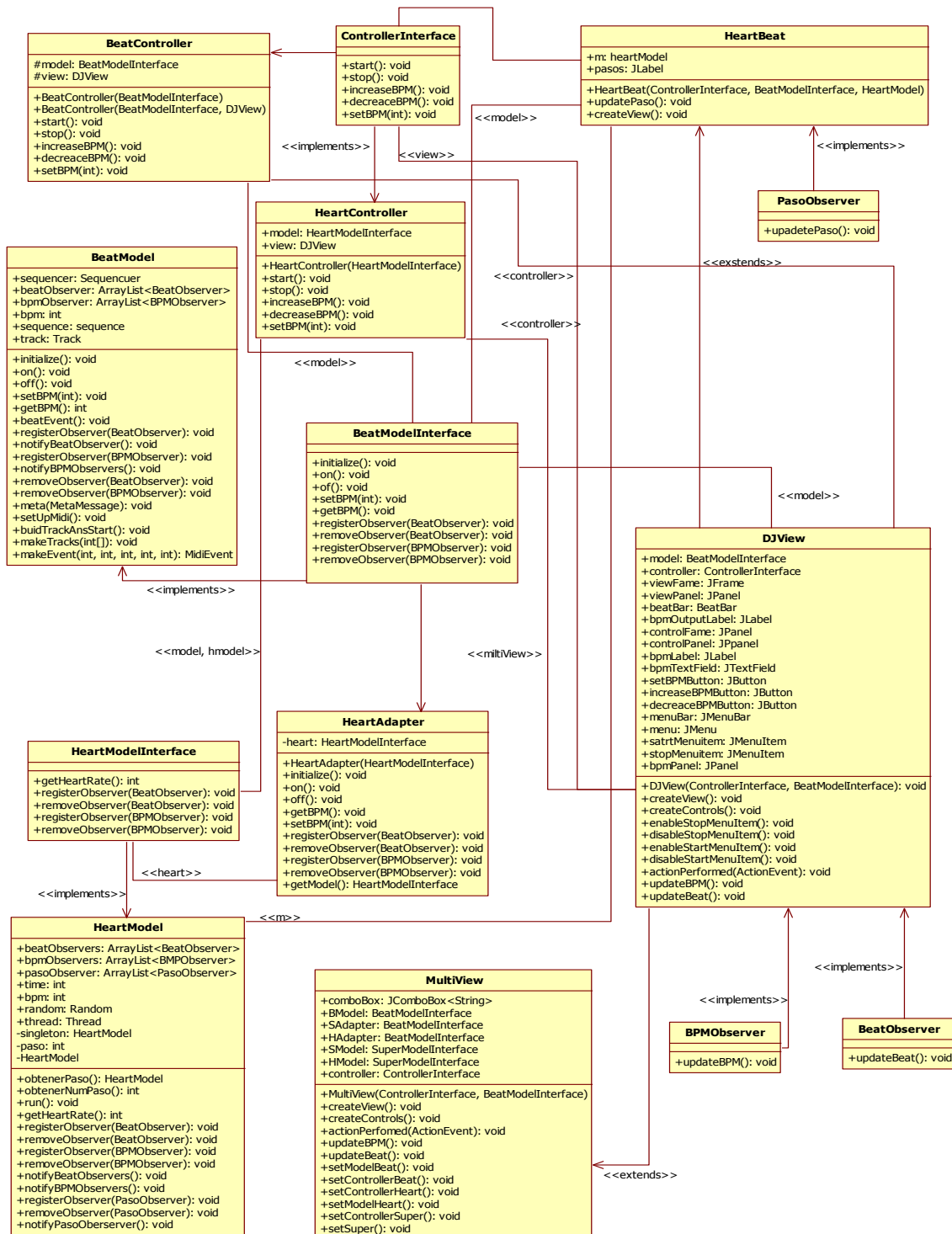
El primer paquete corresponde al proyecto inicial en el cual nos teníamos que basar (ejemplo de proyecto del libro “Head First Design Patterns”. El segundo paquete corresponde al proyecto que debimos implementar basándonos en la experiencia obtenida con la asignatura cursante. Y, por último, vemos un tercer paquete correspondiente a los test unitarios realizados.

Viéndolos con más detalle:

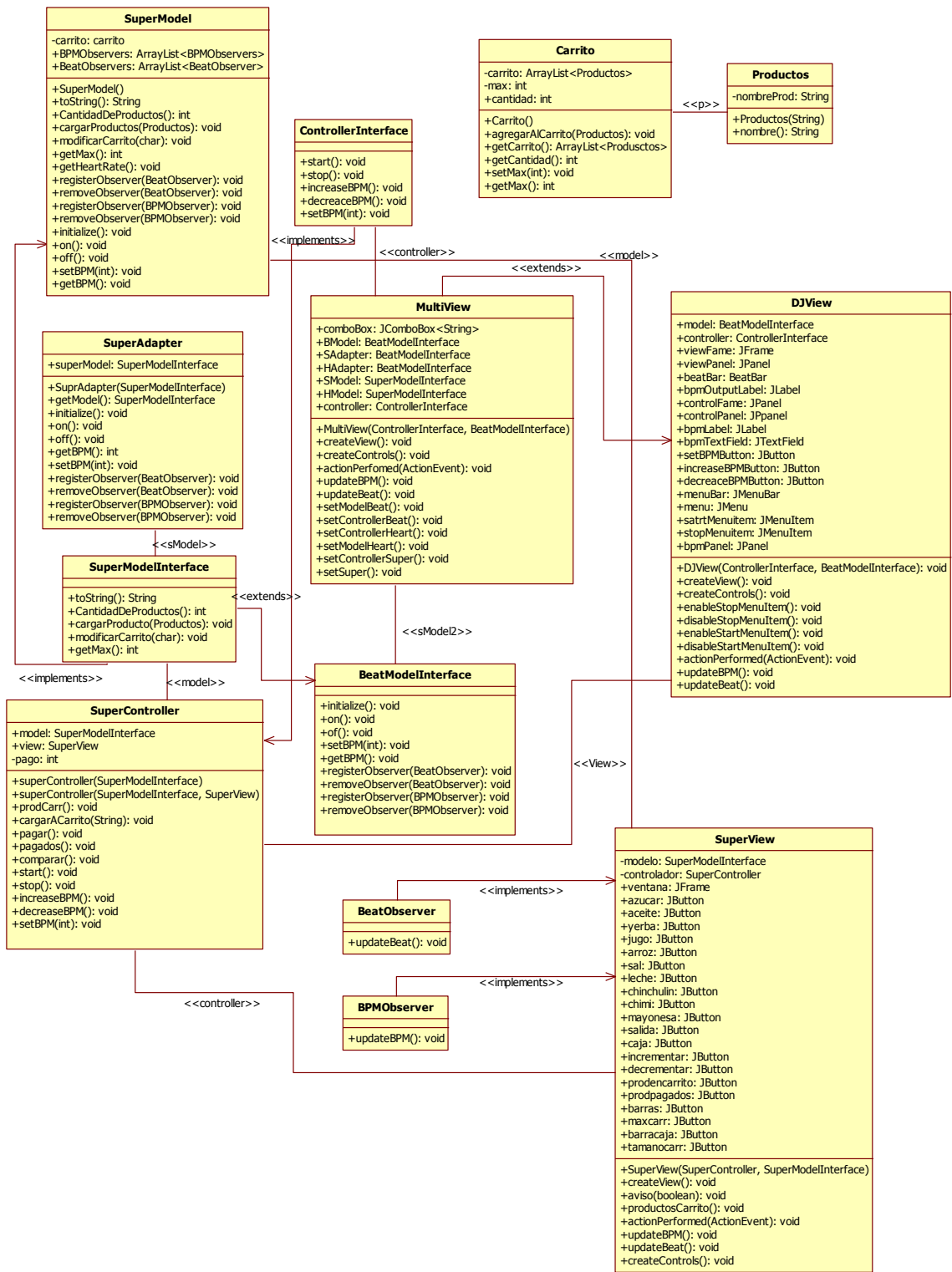
- ▲  main.java.headfirst.combined.djview
  - ▷  BeatBar.java
  - ▷  BeatController.java
  - ▷  BeatModel.java
  - ▷  BeatModelInterface.java
  - ▷  BeatObserver.java
  - ▷  BPMObserver.java
  - ▷  ControllerInterface.java
  - ▷  DJTestDrive.java
  - ▷  DJView.java
  - ▷  HeartAdapter.java
  - ▷  HeartBeat.java
  - ▷  HeartController.java
  - ▷  HeartModel.java
  - ▷  HeartModelInterface.java
  - ▷  HeartTestDrive.java
  - ▷  PasoObserver.java
- ▲  Supermercado
  - ▷  Carrito.java
  - ▷  MultiView.java
  - ▷  MySuperTestDrive.java
  - ▷  Productos.java
  - ▷  SuperAdapter.java
  - ▷  SuperController.java
  - ▷  SuperModel.java
  - ▷  SuperModelInterface.java
  - ▷  SuperView.java
  - ▷  TestDriveMultiView.java
  - ▷  TresModelosTestDrive.java
- ▲  testing
  - ▷  Test1.java
  - ▷  Test2.java
  - ▷  Test3.java
  - ▷  Test4.java
  - ▷  Test5.java
  - ▷  Test6.java

## 5.2. Diagrama de clases

Teniendo en mente lo anteriormente expuesto, se adjuntan a continuación, los diagramas de clases correspondientes al primer paquete (“main.java.headfirst.combines.djview”).



A continuación, se adjuntan los diagramas de clases correspondientes al segundo paquete (“Supermercado”).



## 6.Pruebas unitarias y del sistema

### 6.1. UnitTest

Para realizar Unit test y corroborar que el desarrollo del programa de extendía de forma satisfactoria, se utilizó Junit propio de java.

Para el sistema en general, se realizaron en total 6 test unitarios:

#### Pruebas unitarias de HeartModel y BeatModel

<b>Test 1</b>	Verifica que solo se pueda crear una instancia (usando el patrón Singleton) de la clase HeartModel.
Ejecución	<ul style="list-style-type: none"><li>• Crea un HeartModel</li><li>• Verifica que el objeto sea único</li></ul>
Estado	PASS

<b>Test 2</b>	Verifica que salte un aviso cuando se intenta crear más de una instancia y que cuente cuantas veces se intento crearlo
Ejecución	<ul style="list-style-type: none"><li>• Crea un HeartModel</li><li>• Verifica que el objeto no se pueda crear más de una vez</li></ul>
Estado	PASS

## Pruebas unitarias de Modelo de Supermercado

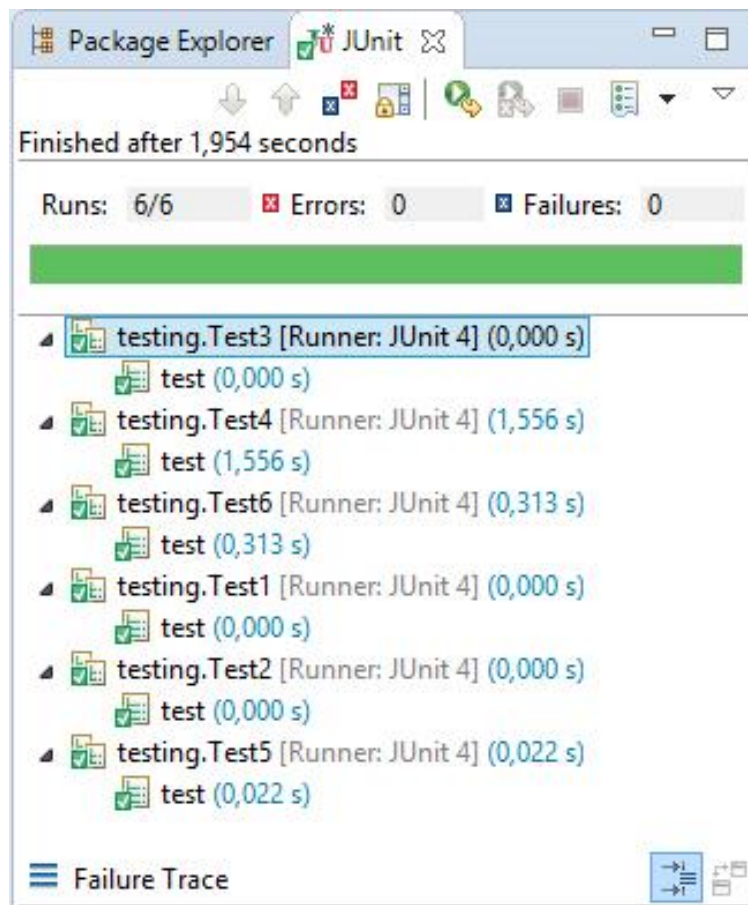
<b>Test 3</b>	Verifica que en el carrito no se puedan agregar más productos que los de su capacidad
Ejecución	<ul style="list-style-type: none"><li>• Crea un carrito con un valor por defecto de capacidad</li><li>• Crea productos</li><li>• Añade productos al carrito</li></ul>
Estado	PASS

<b>Test 4</b>	Verifica que no se puedan pagar más productos que los que hay en el carrito
Ejecución	<ul style="list-style-type: none"><li>• Añade productos al carrito</li><li>• Paga más veces que productos en el carrito</li></ul>
Estado	PASS

<b>Test 5</b>	Verifica que se muestre una ventana con una advertencia si se agrega un producto en el carrito y no se pago
Ejecución	<ul style="list-style-type: none"><li>• Carga un producto al carrito</li><li>• Salir</li></ul>
Estado	PASS

<b>Test 6</b>	Verifica que se muestre una ventana con un texto si se agrega un producto en el carrito y el mismo se paga
Ejecución	<ul style="list-style-type: none"><li>• Carga un producto al carrito</li><li>• Pagar producto</li><li>• Salir</li></ul>
Estado	PASS

## Pruebas de test unitarios en Eclipse





## 6.2. Casos de prueba del sistema

### Pruebas para requerimientos funcionales

<b>Prueba 1</b>	Prueba de ajuste y regulador del tamaño del carrito
Ejecución	<ul style="list-style-type: none"><li>• Iniciar programa</li><li>• Presionar los botones "&lt;&lt;" y "&gt;&gt;"</li></ul>
Estado	PASS

<b>Prueba 2</b>	Prueba de mensaje de aviso si el sensor detecta mercadería no pagada
Ejecución	<ul style="list-style-type: none"><li>• Iniciar programa</li><li>• Agregar un producto al carrito</li><li>• Presionar el botón "salir"</li></ul>
Estado	PASS

<b>Prueba 3</b>	Prueba mostrar en pantalla los productos agregados al carrito
Ejecución	<ul style="list-style-type: none"><li>• Iniciar programa</li><li>• Agregar 3 producto al carrito</li><li>• Presionar el botón "carrito"</li></ul>
Estado	PASS

<b>Prueba 4</b>	Prueba mostrar en pantalla mensaje de agradecimiento si sensor no detecta mercadería no pagada
Ejecución	<ul style="list-style-type: none"><li>• Iniciar programa</li><li>• Agregar un producto al carrito</li><li>• Presionar el botón "caja"</li><li>• Salir</li></ul>
Estado	PASS

<b>Prueba 5</b>	Prueba mostrar en pantalla mensaje de agradecimiento si sensor no detecta mercadería no pagada
Ejecución	<ul style="list-style-type: none"><li>• Iniciar programa</li><li>• Agregar un producto al carrito</li><li>• Presionar el botón "caja"</li><li>• Salir</li></ul>
Estado	PASS

<b>Prueba 6</b>	Prueba mostrar cantidad de productos pagados
Ejecución	<ul style="list-style-type: none"> <li>• Iniciar programa</li> <li>• Agregar 6 productos al carrito</li> <li>• Pagar productos</li> </ul>
Estado	PASS

## Pruebas para requerimientos no funcionales

<b>Prueba 1</b>	El sistema no debe dejar agregar más productos al carrito si el mismo ya se encuentra lleno
Ejecución	<ul style="list-style-type: none"> <li>• Iniciar programa</li> <li>• Setear capacidad de carrito en 7</li> <li>• Intentar agregar 8 productos</li> </ul>
Estado	PASS

<b>Prueba 2</b>	El sistema debe dejar agregar productos al carrito de forma instantánea
Ejecución	<ul style="list-style-type: none"> <li>• Iniciar programa</li> <li>• Setear capacidad de carrito en 20</li> <li>• Agregar productos al carrito de forma rápida</li> </ul>
Estado	PASS

<b>Prueba 3</b>	El sistema debe dejar pagar los productos a una velocidad mínima de un producto por segundo
Ejecución	<ul style="list-style-type: none"> <li>• Iniciar programa</li> <li>• Setear capacidad de carrito en 10</li> <li>• Agregar productos al carrito</li> <li>• Pagar productos a una velocidad de 1 producto por segundo</li> </ul>
Estado	PASS

## Casos de prueba alternativos

Prueba 1	El sistema debe dejar pagar los productos a una velocidad instantánea
Ejecución	<ul style="list-style-type: none"><li>• Iniciar programa</li><li>• Setear capacidad de carrito en 10</li><li>• Agregar productos al carrito</li><li>• Pagar productos a una velocidad de 1 producto por segundo</li></ul>
Estado	FAIL

### 6.3. PASS/FAIL ratio de Pruebas

Como se dijo anteriormente, el método seleccionado por el equipo fue que a medida que se avanzo en el código, se fueron realizando pruebas de implementación manual o utilizando la herramienta propia de java. En caso de encontrar un error, este se solucionaba y se procedía con normalidad en las modificaciones e incorporaciones de código.

En nuestro caso, para todo tipo de pruebas realizadas el porcentaje obtenido fue de un 100% de PASS Ratio y un 0% de FAIL Ratio.

### 6.4. Numero de Bugs encontrados

**Hemos encontrado 2 bugs:**

- Bug conocido: Agregar productos al carrito hasta que esté lleno. Luego decrementar el tamaño del carrito y volver a agregar productos. Por ejemplo, setear el tamaño del carrito en 7. Agregar 7 productos. Decrementar el tamaño del carrito a 2. Seguir agregando productos. Este bug permite agregar una cantidad infinita de productos al carrito si se realiza lo anteriormente descrito. El bug no pudo ser corregido.

- Bug conocido: Intentar mostrar la vista del modelo de Supermercado a través de la ventana de MiltiView.  
El bug no pudo ser corregido.

## 7.Datos históricos

Integrantes	Jueves	Viernes	Sábado	Domingo	Total
González	6	9	4	7	26
Krenz	6	11	10	8	35
Rivero	6	8	9	5	28

Los detalles de cómo se dividieron las tareas cada integrante del grupo se presentó en el punto 3.6.

El cuadro anterior, refleja el esfuerzo de trabajo en Personas/Horas distribuidas a lo largo de la duración del proyecto (4 días).

## 8. Información adicional

Luego de finalizar el presente trabajo practico, podemos afirmar que mas allá de aprender nuevas técnicas de Ingeniería de Software, pudimos ponerlas a prueba e interactuar con ellas, fijando más los conceptos claves y ponerlos en práctica desarrollando un proyecto de software real.

Si bien el modelo desarrollado se basa en un caso de la vida real, el mismo sólo se creó con fines educativos y de práctica. Es decir, sabiendo que es un programa de uso comercial y que el mismo no tiene funcionalidad real alguna, es totalmente imposible implementarse en la vida real, ya que seríamos irresponsables por el uso inadecuado de la propiedad intelectual.

Dejando de lado dichos comentarios, podemos agregar que aprendimos a utilizar sistemas basados en un patrón de diseño denominado Model View Controller, aprendimos a usar de manera correcta repositorios remotos y control de versiones de software. Además, reafirmamos conceptos relacionados a UML, aprendiendo a manejar herramientas necesarias para crear los diagramas. Y por último, aprendimos y entendimos la importancia de los test, tanto los unitarios como los del sistema, para llevar a cabo el desarrollo exitoso de un software.