Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Ingeniería de Software II

Trabajo Práctico 1

Sprint Planning

Integrante	LU	Correo electrónico
Martín Alejandro Miguel	181/09	m2.march@gmail.com
Iván Postolski	216/09	ivan.postolski@gmail.com
Juan Manuel Martinez Caamaño	276/09	jmartinezcaamao@gmail.com
Matías Incem	396/09	matias.incem@gmail.com
Pablo Gauna	334/09	gaunapablo@gmail.com

Índice

1. Introducción

El actual informe presenta el análisis, diseño y desarrollo inicial realizado para el producto **Precio Justo**. En el mismo se detallan el conjunto de *user stories* que abarcan el desarrollo completo de la aplicación, incluyendo tanto las definiciones actuales como los puntos de extensión que se tendrán en cuenta. Se presenta además un primer *diseño orientado a objetos* de la aplicación completa. Este tiene como objetivo ser expansible respecto de los distintos ejes de cambios considerados para esta instancia. Por último, se analiza el alcance, el proceso y el éxito de la primera iteración del proceso de desarrollo.

2. Análisis de la aplicación

2.1. Objetivo

Precio Justo es una aplicación de recolección y procesamiento de datos masivos provistos por redes sociales. La intención es poder recopilar los precios más baratos para ciertos productos, de forma confiable y libre de intereses particulares. La elección del uso de las redes sociales para obtener los datos tiene como objetivo lograr imparcialidad en los mismos.

2.2. Ejes de cambio

Dentro de la resolución del objetivo de la aplicación se establecen, para una primera instancia, las siguientes restricciones, que pasan a conformar *ejes de cambio* de la aplicación:

Obtención de datos

- **Origen de los datos:** En un primer lugar los datos a utilizar por la aplicación se obtendrán exclusivamente de *twitter*. Dentro del flujo de datos del mismo, se espera filtrar aquellos que contengan información relevante para nuestros propósitos.
- **Formato de los tweets:** Considerando la restricción anterior, debe definirse bajo qué criterio un *tweet* es útil. En una primera instancia los mismos deberán tener un formato similar a <Producto> <Precio> <Unidad> <Lugar> #PrecioJusto, donde el *hashtag* es indicador de que el *tweet* está orientado a ser utilizado por nuestra aplicación. Se espera en un futuro poder precindir del *hashtag* y poder interpretar formatos más flexibles.
- **Unidad de los productos:** Otra restricción impuesta para el procesamiento de datos es que los precios estén indicados en su valor por una cierta unidad, siendo restringida las unidades aceptadas para cada producto. Por ejemplo, para tomates se aceptan solo *kilogramos*, y para leche el *litro*.
- **Correctitud de los datos:** Una restricción implícita existente es que los mensajes a leer están bien escritos. Otra arista de cambio para la aplicación es poder soportar un cierto nivel de errores tipográficos en nuestra fuente cruda de datos.

Productos soportados Inicialmente la aplicación tendrá soporte para una cantidad limitada de productos, enfocándose principalmente en aquellos de primera necesidad. La misma debe ser suficientemente flexible para poder cambiar estos productos.

Ubicación La aplicación debe informar el lugar donde puede comprarse el producto al precio establecido. Inicialmente esta información deberá estar presente en el mismo *tweet* en formato de 'calle y altura'. Conciendo que **twitter** permite agregar como información del *tweet* la posición geográfica del mismo, la misma podría utilizarse en reemplazo del texto explicitando la dirección. Se espera que la aplicación pueda usar estos datos.

Por otra parte, es importante tener en cuenta que, de agregarse nuevas fuentes de datos a la aplicación, estas deben ser capaces de indicar dónde se consigue el producto al precio publicitado.

Resumen de la información Dentro de la definición de la aplicación se establece que debe permitírsele al usuario especificar como la información será tratada. En esto se definen las siguientes características.

- **Resumen:** Dentro de la masividad de datos que tendremos para un producto, es necesario definir cómo estos se resumirán para el usuario. En primer lugar, se deben poder priorizar los precios más baratos. Además se definen otras estrategias para tratar los datos, como ser *caminando lo menos posible*. El sistema debe permitir agregar nuevas estrategias fácilmente.
- **Filtrado:** Otra posibilidad de tratamiento de datos que se le ofrece al usuario es el filtrado de los mismos. En este momento se presentan dos filtros: por zona geográfica y por rango de precios. El sistema debe soportar agregar nuevos criterios fácilmente.

- **Productos:** En esta primera instancia de la aplicación se está poniendo como restricción que el usuario pueda elegir un solo producto para buscar. En la versión final, la aplicación debe permitirle hacer búsquedas de varios productos a la vez.

Veracidad de los datos A pesar del supuesto donde los datos provistos por redes sociales son imparciales, esto no nos asegura que los datos sean siempre correctos. Es importante tener en cuenta que los precios suben o que algún precio reportado puede ser una oferta temporal. Para tratar con estas complejidades se decidió que el sistema podrá incorporar en un futuro una forma de especificar la vigencia del precio en caso de ser una oferta. En una primera instancia, la aplicación se compromente a utilizar solo datos que tengan una frescura suficiente de forma que puedan considerarse correctos.

Otra propuesta para esta temática es incrementar la interacción con el usuario y que este pueda votar o reportar las ofertas provistas según si fueron buenas o no. Este tipo de información nos permitiria eliminar datos falsos o desactualizados.

Presentación de los datos La visualización de ofertas que se le dará al usuario no se encuentra completamente definida en esta instancia, es por ello que el diseño del sistema debe ser capaz de adaptarse a distintas propuestas que puedan surgir en el mediano plazo. Una propuesta ya presente es que la aplicación incorpore la información de precios en un mapa.

3. User Stories

En esta sección presentamos las **user stories** definidas para el *backlog* del proyecto. En las mismas se captura la extención completa de la aplicación. Durante el proceso de creación del *product backlog* y la realización del *sprint planning* se estudiaron distintas aristas del sistema a desarrollar y el problema a resolver. Las historias planteadas reflejan distintas resoluciones tomadas respecto al alcance de la aplicación. Así mismo en las *stories* se declararon cuestiones que no podían ser resueltas en esta primera etapa. Este es el caso de los **epics**.

Son sumamente generales y deberán ser desarrolladas en el futuro. Cada uno representa uno de los nortes planteados por los product owners. Alguno de los **epics** se solapan con otras historias planteadas, lo cual significa que el mismo está siendo abordado. En este caso el **epic** toma relevancia para recordar cual es el objetivo global que se busca y generar planteos respecto a correcciones y nuevas funcionalidades que pueda darse al producto final.

US145: *Realizar un primer análisis y diseño de la aplicación*

Descripción: En una primera instancia es importante comprender la naturaleza de la aplicación, entender los requerimientos, sugerir propuestas para resolverlos y bocetar una estructura global que permita la división del trabajo entre los integrantes del grupo.

También es necesario definir un camino de desarrollo, priorizar los requerimientos y establecer líneas de expansión de la aplicación que sean deseables.

Criterios de aceptación:
* El grupo de trabajo acordará en un lenguaje/código común para realizar el diseño orientado a objetos.
* El grupo poseerá un boceto básico del diseño global de la aplicación.
* Los integrantes podrán comenzar a trabajar por separado con un conocimiento básico de las actividades que el grupo debe realizar.

Story Points: 8.0 **Iteración:** Primera **Estado:** Accepted

US144: *Como desarrollador quiero buscar un mecanismo que permita estandarizar los archivos de código Python*

Descripción: Python es sensible a la identación, y en el trabajo en equipo puede ser problemático que los integrantes acostumbren usar distintos criterios de tabulación.

Es necesario encontrar algún mecanismo/programa/utilitario que ayude a manejar este y otros problemas de formato de código.

Criterios de aceptación:
* Los integrantes del grupo trabajarán con código Python estandarizado

Story Points: 2.0 **Iteración:** Primera **Estado:** Accepted

US143: *Como desarrollador quiero tener un mecanismo para hacer tests unitarios*

Descripción: Se quiere disponer de un software que nos permita realizar tests unitarios. El mecanismo debe haber sido testeado en su facilidad de uso y extensibilidad creando tests.

Criterios de aceptación:

* Se dispondrá de un mecanismo de fácil extensión para poder testear funcionalidades unitarias.

Story Points: 3.0 **Iteración:** Primera **Estado:** Accepted

TA192: *Investigar bibliotecas o mecanismos para hacer unit testing en el lenguaje de programación del servidor*

Descripción: El código del servidor está escrito en Python, por lo que se investigará el funcionamiento de PyUnit para realizar tests unitarios.

Horas estimadas: 3.0 **Responsable:** Juan M
Horas faltantes: 0.0 **Estado:** Completed

TA193: *Implementar tests de ejemplo con el mecanismo elegido que sirvan de modelo para futuros tests*

Descripción:

Horas estimadas: 1.0 **Responsable:** Martin M
Horas faltantes: 1.0 **Estado:** Defined

US101: *Como desarrollador quiero investigar las opciones disponibles para obtener información de Twitter*

Descripción: Queremos conocer y contrastar las alternativas sobre la obtención y almacenamiento de la información obtenida desde Twitter.

En principio tenemos las siguientes funcionalidades, en las que tenemos que decidir entre dos implementaciones en cada una:
* Almacenar la información vs. obtenerla en cada búsqueda del usuario.
* Investigar la Search REST API vs. Streaming API de Twitter.

Criterios de aceptación:
* El grupo de trabajo conocerá el método para obtener la información desde Twitter

Story Points: 5.0 **Iteración:** Primera **Estado:** Accepted

TA35: *Investigar cómo funcionan las APIs (REST y Streaming), junto con las SDK de Twitter*

Descripción:

Horas estimadas: 4.0 **Responsable:** Ivan P
Horas faltantes: 0.0 **Estado:** Completed

TA36: *Discutir ventajas y desventajas de cada API y elegir una*

Descripción:

Horas estimadas: 2.0 **Responsable:**
Horas faltantes: 0.0 **Estado:** Completed

TA37: *Investigar y elegir cómo y cuándo vamos a conseguir los datos*

Descripción: Se desea tener un método para conseguir datos para responder una consulta. Podemos obtenerlos siempre desde Twitter y no guardar nada, o bien podemos almacenarlos de antemano y usarlos cuando debemos responder una consulta. Revisar como impacta en el sprint backlog esta decisión.

Horas estimadas: 4.0 **Responsable:**
Horas faltantes: 0.0 **Estado:** Completed

US99: *Como PO espero que la información sea obtenida a través de Twitter*

Descripción: El Product Owner desea que el sistema pueda comunicarse con Twitter para recibir información. Por lo tanto, es necesario asegurarse de que podemos recibir datos usando una conexión con Twitter, y que podemos decodificar y entender los datos recibidos.

Criterios de aceptación:
* El sistema tendrá una conexión con la API de Twitter y podrá descargar los datos que necesita
* Toda la información obtenida podrá ser leída por el sistema, los datos útiles deberán ser identificados, separados y convertidos a un formato utilizable por la aplicación.

Story Points: 5.0 **Iteración:** Primera **Estado:** Accepted

TA40: *Implementar el conector para Twitter*

Descripción: Necesitamos un conector que pueda realizar una llamada al servidor de Twitter y descargar datos.
* El conector debe poder conectarse a Twitter autenticándose con OAuth2
* El conector debe poder obtener los mensajes de Twitter con un hashtag particular, en nuestro caso, #precioJusto

Horas estimadas: 6.0 **Responsable:**
Horas faltantes: 0.0 **Estado:** Completed

TA41: *Registrar la aplicación en Twitter*

Descripción: Es necesario dar de alta la aplicación en Twitter, para obtener las credenciales para usar la API. Esto es requerido para cualquier aplicación que quiera trabajar con Twitter.

Horas estimadas: 2.0 **Responsable:**
Horas faltantes: 0.0 **Estado:** Completed

TA42: *Traducir los datos obtenidos de Twitter al modelo interno de representación*

Descripción: Debemos tener un traductor que lea los datos que conforman un tweet y los convierta en objetos de nuestro modelo, fundamentalmente en un objeto .ºferta".

Horas estimadas: 4.0 **Responsable:**
Horas faltantes: 0.0 **Estado:** Completed

US113: *Como desarrollador quiero reconocer tweets útiles y convertirlos a ofertas*

Descripción: Cuando el sistema recibe datos de Twitter, necesita poder identificar aquellos que tienen información útil. Esto se determina de acuerdo a su formato. En esta etapa del proyecto, los tweets que son útiles son aquellos que cumplen el formato "< Producto> < Precio> < Unidad> < Lugar> #precioJusto", y además, tengan un producto válido, una unidad soportada por el producto, y un precio y lugar válidos. Es necesario identificar los tweets válidos y extraer la información contenida en ellos.

Criterios de aceptación:
* La aplicación conocerá los productos y unidades aceptados por los administradores y podrá contrastarlos con los datos de los tweets.
* Si un tweet con el hashtag #precioJusto no tiene información correcta o no es relevante, deberá ser ignorado por el sistema.
* Si un tweet con el hashtag es relevante y válido, el sistema deberá componer un objeto 'oferta' que contenga la información dada por dicho tweet.

Story Points: 5.0 **Iteración:** Primera **Estado:** Accepted

TA135: *Implementar un mecanismo para obtener la lista de productos y sus unidades*

Descripción: Para poder validar los tweets, es necesario conocer la lista de productos válidos y de unidades válidas para cada producto. El sistema puede pedir esta lista.

Horas estimadas: 3.0 **Responsable:**
Horas faltantes: 0.0 **Estado:** Completed

TA139: *Implementar el parser que reconozca si el formato de un tweet es correcto*

Descripción: Se deberá reconocer si el tweet tiene la información necesaria de producto, precio, unidad y lugar.
El formato básico a soportar es: < Producto> < Precio> < Unidad> < Lugar> #PrecioJusto

Horas estimadas: 5.0 **Responsable:**
Horas faltantes: 0.0 **Estado:** Completed

TA140: *Con los tweets útiles reconocidos, extraer la información para generar una oferta*

Descripción: Se deben separar los datos validados del tweet para construir un objeto Oferta, que posea la información necesaria y pueda ser leído por la aplicación.

Horas estimadas: 2.0 **Responsable:**
Horas faltantes: 0.0 **Estado:** Completed

US106: *Como desarrollador quiero tener una base para la API REST del backend*

Descripción: Se espera implementar un código inicial del backend que permita agregar fácilmente nuevos métodos a la interfaz REST, los cuales permitirán acceder al funcionamiento interno del mismo.

Criterios de aceptación:
* El backend dispondrá de un método /status que permita verificar si está funcionando correctamente.
* El código de este método será fácilmente replicable para agregar nuevas funcionalidades.

Story Points: 5.0 **Iteración:** Primera **Estado:** Accepted

TA111: *Implementar una operación básica de busqueda de ofertas en nuestro servicio*

Descripción: Esta task cubre la implementacion base del servicio.

Horas estimadas: 4.0 **Responsable:** Martin M
Horas faltantes: 0.0 **Estado:** Completed

TA141: *Investigar bibliotecas para la implementación del servicio*

Descripción: Investigar distintas bibliotecas útiles para la implementación del servicio.
Como primera instancia se presenta la siguiente:

http://www.cherrypy.org/
(http://www.cherrypy.org/)

Horas estimadas: 4.0 **Responsable:** Martin M
Horas faltantes: 0.0 **Estado:** Completed

US124: *Como desarrollador quiero implementar la base del mecanismo de estrategias y filtros*

Descripción: Se desea implementar el código básico que se aprovechará en las futuras implementaciones de filtros y estrategias.
En este código debe tomarse el total de ofertas y devolver un subconjunto, definido por los filtros y las estrategias configuradas.
Un filtro toma un conjunto de ofertas y devuelve el subconjunto de ellas que cumple un criterio determinado, por ejemplo, su precio es menor a 5 pesos, o su dirección está en determinada calle. Otra forma de entender este concepto es como una función Oferta -> Bool que responde si la oferta cumple el criterio o no.
Una estrategia toma un conjunto de ofertas y las ordena de acuerdo a una cualidad, por ejemplo, de menor a mayor precio. Notar que en algunos casos, el resultado final no aprovechará la lista ordenada completa (por ejemplo, podría necesitar solo el primer elemento), por lo que no siempre se necesita hacer todo el procedimiento.

Criterios de aceptación:
* El sistema tendrá código que le permita llamar a un manejador de filtros y estrategias.
* Se podrá invocar la aplicación de un filtro o estrategia "nulo". Esta operación no tendrá ningún efecto en los datos, pero su código será fácilmente extensible para agregar funcionalidades.

Story Points: 5.0 **Iteración:** Primera **Estado:** Accepted

TA143: *Testear la implementación mediante un filtro y una estrategia de prueba*

Descripción:

Horas estimadas: 3.0 **Responsable:** Juan M
Horas faltantes: 0.0 **Estado:** Completed

US147: *Como usuario del backend quiero conocer las estrategias provistas por el sistema*

Descripción: En las capas de frontend que le presenten funcionalidad al usuario deseo poder mostrarle qué estrategias tiene disponible para resumir la información provista por el sistema.

Criterio de aceptación:
* La interfaz rest debe exponer la lista de estrategias disponibles mediante un método /strategies

Story Points: 3.0 **Iteración:** Primera **Estado:** Accepted

US104: *Como desarrollador quiero filtrar las ofertas según un rango de precios*

Descripción: Se debe lograr un código que adquiera los datos de las ofertas y los filtre según un rango de precios determinado. Las ofertas que estén fuera del rango deberán ser descartadas. Esta funcionalidad complementa la desarrollada en la US96, que permite que el sistema obtenga un rango de precios entre los datos enviados por el usuario.

Criterios de aceptación:
* Tenemos ofertas con precios que van desde X hasta Y, y elegimos filtrar desde A hasta Z.
* Luego de aplicado el filtro, las ofertas resultantes serán todas las que cumplan con los otros criterios de la consulta y tengan precio mayor o igual a A y menor o igual a Z.

Story Points: 2.0 **Iteración:** Primera **Estado:** Accepted

TA56: *Implementar el filtrado de las ofertas por rango de precios*

Descripción: El código debe acoplarse al mecanismo de filtros y estrategias implementado en la US124

Horas estimadas: 4.0 **Responsable:** Ivan P
Horas faltantes: 0.0 **Estado:** Completed

US96: *Como usuario espero realizar búsquedas filtrando por un rango de precios*

Descripción: Cuando el usuario realiza una consulta, puede querer definir un rango de precios, de forma que en la respuesta del sistema solo aparezcan las ofertas cuyos precios son mayores a un valor mínimo y menores a un valor máximo.
Nota: Esta story describe las atribuciones del frontend. La funcionalidad que permite que el backend realice un filtro sobre sus datos se encuentra en la US104.

Criterios de aceptación:

* La inferfaz permitirá ingresar, de forma opcional, un rango de precios, formado por un valor MIN y un valor MAX.
* Si los parámetros son ingresados incorrectamente la interfaz debe decirlo. Esto puede ocurrir si el rango es inválido o los números están mal escritos.
* Todas las ofertas sugeridas al finalizar la consulta se encontrarán dentro del rango explicitado.

Story Points: 5.0 **Iteración:** Primera **Estado:** Accepted

TA59: *Implementar en el cliente un medio para establecer el tipo de consulta "por rango" el rango establecido*

Descripción: La interfaz de usuario necesita tener una funcionalidad para elegir si se incluye un rango de precios en la búsqueda. Además, en caso de que se lo elija, también debe tener un campo donde ingresar precio máximo y mínimo.

Horas estimadas: 4.0 **Responsable:** Juan M
Horas faltantes: 0.0 **Estado:** Completed

TA60: *Implementar la comunicación del cliente con el backend, enviando los parámetros*

Descripción: El backend debe tener un método REST para recibir la consulta. El cliente debe utilizar este método de forma correcta enviando los parámetros.

Horas estimadas: 2.0 **Responsable:** Juan M
Horas faltantes: 0.0 **Estado:** Completed

TA61: *Implementar la recepción y exhibición de los datos de respuesta en el frontend*

Descripción: El cliente debe poder recibir la respuesta del backend y exponerla al usuario a través de su interfaz.

Horas estimadas: 3.0 **Responsable:** Juan M
Horas faltantes: 0.0 **Estado:** Completed

US110: *Como desarrollador quiero investigar cómo determinar zonas geográficas*

Descripción: Es necesario establecer un criterio para delimitar las zonas que servirán de filtro en nuestra aplicación.
Algunas opciones son:

* Utilizar la API de google maps si es que tiene en ella información de zonas
* Si existe, utilizar la API del mapa del Gobierno de la Ciudad para usar las zonas delimitadas en la misma
* Definir las zonas manualmente en nuestra aplicación

También se debe determinar si es necesario recortar inicialmente el alcance total de la aplicación a, por ejemplo, Capital Federal.

Criterios de aceptación:
* El grupo de trabajo conocerá el concepto de zona que se utilizará, así como la manera de operar con las zonas.

Story Points: 5.0 **Iteración:** no definida **Estado:** Defined

US88: *Epic: Como usuario quiero obtener la direccion, precio y unidad del producto respetando los filtros ingresados*

Descripción:

Story Points: 30.0 **Iteración:** no definida **Estado:** In-Progress

US82: *Como usuario espero realizar búsquedas filtrando por zona*

Descripción: El sistema debe permitir una funcionalidad en la que el usuario pueda elegir e indicar una zona al momento de hacer su consulta, en caso de que lo desee; si lo hace, las respuestas que reciba deberán estar restringidas a ofertas dentro de dicha zona.

Queda pendiente definir el concepto de zona, y decidir si estarán fijadas por el sistema o pueden ser delimitadas por el usuario durante la consulta.

Esta historia es dependente de la investigación previa realizada en la US110

Criterios de aceptación:
* El usuario podrá elegir buscar ofertas por zona, y en dicho caso, podrá especificar una zona.
* Todas las direcciones de las ofertas devueltas en el resultado estarán dentro de la zona elegida por el usuario.

Story Points: 8.0 **Iteración:** no definida **Estado:** Defined

TA46: *Implementar el filtro por zonas.*

Descripción:

Horas estimadas: 10.0 **Responsable:**
Horas faltantes: 10.0 **Estado:** Defined

US83: *Como usuario quiero realizar búsquedas con la estrategia "gastando lo menos posible"*

Descripción: Cuando se realiza una consulta, se puede elegir la estrategia "gastando lo menos posible", que consiste en seleccionar las ofertas de precio mínimo entre todas las ofertas válidas, y solamente mostrar esas en el resultado.

Criterios de aceptación:
* Cuando el usuario haga una consulta, podrá elegir que se aplique la estrategia "gastando lo menos posible"
* Si está seleccionada, el sistema solamente retornará como respuesta las ofertas con el precio más económico entre las que encontró para un determinado producto.

Story Points: 3.0 **Iteración:** no definida **Estado:** Defined

US85: *Como usuario quiero elegir un producto entre los soportados por el sistema para obtener precios de dicho producto*

Descripción: Se desea tener una funcionalidad básica del sistema, que es la siguiente: el usuario ingresa el nombre de un producto válido, y el sistema devuelve un conjunto de ofertas relacionadas con dicho producto, las cuales se consiguen a partir de información de tweets.

Criterios de aceptación:
* El usuario podrá elegir un producto para realizar su consulta, y sabrá si el producto elegido es válido.
* El sistema devolverá una lista de ofertas para el producto, que incluyen información de precio por unidad y lugar.

Story Points: 3.0 **Iteración:** Primera **Estado:** Accepted

TA108: *Implementar una forma de exportar los productos soportados en el backend*

Descripción: La interfaz REST debe exportar, mediante un método /products, la lista de productos soportados junto con su unidad.

Horas estimadas: 4.0 **Responsable:** Juan M
Horas faltantes: 0.0 **Estado:** Completed

TA109: *Implementar la obtención de los productos en el frontend*

Descripción: Se necesita que el cliente de la aplicación consulte satisfactoriamente al backend y pueda obtener la lista de los productos soportados y sus unidades para operar con ellos o presentárselos al usuario.

Horas estimadas: 4.0 **Responsable:** Juan M
Horas faltantes: 0.0 **Estado:** Completed

TA110: *Mostrar al usuario la lista de productos soportados y permitirle elegir un producto*

Descripción: La interfaz con la que se comunica el usuario le debe dar a elegir al mismo uno o más productos de la lista de productos válidos, y esta selección debe estar visible durante el resto de la consulta.

Horas estimadas: 4.0 **Responsable:** Juan M
Horas faltantes: 0.0 **Estado:** Completed

US105: *Como desarrollador quiero tener una implementación básica del frontend y backend*

Descripción: Se necesita tener un código del backend que provea una API para comunicarse con el frontend. También necesitamos definir un frontend que sepa comunicarse con esa API y además provea un interfaz para que el usuario pueda realizar consultas.

Criterios de aceptación:
* El backend deberá ponerse en funcionamiento y proveer una API REST.
* El frontend tendrá una interfaz visible para el usuario. Además podrá comunicarse con la API del backend.
* El código podrá ser fácilmente extensible para agregar funcionalidad.

Story Points: 13.0 **Iteración:** Primera **Estado:** Accepted

TA145: *Instalar todos los componentes necesarios para el desarrollo*

Descripción: Necesitamos que todos los integrantes del grupo instalemos las aplicaciones que se necesitan para desarrollar el software: Python, Tweepy, Git.

Horas estimadas: 5.0 **Responsable:** Juan M
Horas faltantes: 0.0 **Estado:** Completed

TA146: *Crear el proyecto inicial*

Descripción: Se debe crear un proyecto que incluya espacio para el código fuente, la documentación y los diagramas de diseño. Cada uno de ellos deberá ser completado a medida que avanza el proyecto.

Horas estimadas: 4.0 **Responsable:** Juan M
Horas faltantes: 0.0 **Estado:** Completed

TA147: *Agregar un primer código que le dé cierta funcionalidad al cliente frontend*

Descripción: Se necesita un código básico de un frontend que pueda comunicarse con el backend de nuestro sistema a través de una API, y con el usuario por medio de una interfaz.

Horas estimadas: 4.0 **Responsable:** Juan M
Horas faltantes: 0.0 **Estado:** Completed

TA148: *Generar el código de la interfaz que siente las bases para la fácil implementación de agregados*

Descripción: Se debe hacer un primer esbozo de una interfaz de usuario.

Horas estimadas: 5.0 **Responsable:** Juan M
Horas faltantes: 0.0 **Estado:** Completed

TA149: *Implementar la conexión con el servicio REST*

Descripción: El frontend debe ser capaz de comunicarse con el backend, por medio de la API REST de este último.

Horas estimadas: 4.0 **Responsable:** Juan M
Horas faltantes: 0.0 **Estado:** Completed

TA150: *Validar los funcionamientos generados*

Descripción: Es necesario testear la funcionalidad del frontend y backend en cuanto a comunicación.

Horas estimadas: 4.0 **Responsable:** Juan M
Horas faltantes: 0.0 **Estado:** Completed

US102: *Como usuario quiero realizar búsquedas filtrando por la estrategia 'caminando lo menos posible'*

Descripción: En una consulta del usuario se puede elegir la estrategia 'caminando lo menos posible', que elige, entre las ofertas válidas para un producto, aquella que se encuentra más cerca de la posición actual del usuario.
Considerar: la funcionalidad será distinta si se permite la búsqueda simultánea de varios productos (Ver US126).

Criterios de aceptación:
* En la interfaz deberá figurar la opción para elegir la estrategia 'caminando lo menos posible'
* El resultado de aplicar esta estrategia deberá mostrar la oferta que cumpla con los demás filtros (rango, zona, etc.), y además sea la mas cercana a la posición del usuario.
* En caso de que el sistema no logre ubicar al usuario, un error será mostrado al usuario, indicando que no es posible ejecutar la consulta porque el sistema no conoce su ubicación.

Story Points: 3.0 **Iteración:** no definida **Estado:** Defined

US100: *Como PO quiero enfocarme en productos de primera necesidad.*

Descripción: Desde los requerimientos iniciales se estableció que el sistema se enfocará en buscar ofertas para productos de primera necesidad. Cuales son estos productos deberá ser definido por el product owner.

Criterios de aceptación:
* El sistema admite como productos a buscar aquellos definidos por el product owner.

Story Points: 1.0 **Iteración:** no definida **Estado:** Defined

TA54: *Consultar al PO qué productos con qué unidades son de interés*

Descripción:

Horas estimadas: 1.0 **Responsable:**
Horas faltantes: 1.0 **Estado:** Defined

TA55: *Modificar el sistema para que tenga en cuenta las especificaciones del PO*

Descripción:

Horas estimadas: 1.0 **Responsable:**
Horas faltantes: 1.0 **Estado:** Defined

US89: *Como PO espero poder agregar y eliminar productos de la lista de productos soportados junto con su respectiva unidad*

Descripción: El sistema está pensado para tratar con productos "de primera necesidad", y el objetivo es tener una lista cerrada de productos soportados. En principio esta lista es fija, pero se quiere que en un futuro pueda ser modificada, agregando o quitando productos según las decisiones del Product Owner, que puede querer que un producto sea o no soportado por la aplicación.

Criterios de aceptación:
* Un administrador del sistema podrá incluir un nuevo producto para ser soportado por la aplicación, junto con una o más unidades de medida asociadas. Los usuarios podrán consultar por dicho producto luego de que haya sido dado de alta, y el funcionamiento deberá ser correcto al igual que con los productos originales.
* Un administrador podrá eliminar un producto de la lista. Los usuarios no podrán volver a consultar sobre dicho producto en la aplicación.
* Un administrador podrá agregar una nueva unidad asociada a un producto existente. Las respuestas a las consultas de dicho producto incluirán ofertas con dicho producto y unidad. También podrá ser eliminada una unidad existente, y las ofertas que usen esa unidad no aparecerán.

Story Points: 8.0 **Iteración:** no definida **Estado:** Defined

US87: *Como usuario quiero que la información consultada se muestre en un mapa, indicando los lugares de compra y los precios de los productos*

Descripción: El mapa es un método visualmente eficiente para mostrar las ofertas que son respuesta a una consulta del usuario. La dirección puede ser señalada con un punto en el mapa, y el resto de la información puede verse en el epígrafe que acompaña al punto.

Criterios de aceptación:
* Cuando el usuario realiza una consulta, podrá elegir que los resultados se muestren en un mapa.
* Las direcciones de todas las ofertas resultantes aparecerán como puntos en el mapa, y los datos restantes de las ofertas estarán escritos al costado de cada punto.

Story Points: 13.0 **Iteración:** no definida **Estado:** Defined

TA105: *Investigar la API de google maps*

Descripción: En la api de google maps es necesario investigar cómo:

* Ingresarle los datos de los lugares a mostrar
* Ingresar junto con los puntos de interés datos de los mismos
* Mostrar el mapa en el cliente

Horas estimadas: **Responsable:**
Horas faltantes: **Estado:** Defined

TA106: *Implementar en el cliente la comunicación con Google Maps*

Descripción: El código implementado debe tomar los resultados devueltos por el backend y traducirlos y enviárselos a la api de google maps para que cree el mapa con los datos de interés.

Horas estimadas: **Responsable:**
Horas faltantes: **Estado:** Defined

TA107: *Implementar en el cliente el display del mapa con los resultados*

Descripción: Utilizando los resultados de la api de maps, el código debe permitir visualizar en el cliente el mapa generado.

Horas estimadas: **Responsable:**
Horas faltantes: **Estado:** Defined

US118: *Como desarrollador quiero filtrar las ofertas por la estrategia çaminando lo menos posible"*

Descripción: **Requerimiento:**
El usuario debe poder comunicar la información sobre su ubicación actual. En un principio podría transmitirla directamente por texto, indicando < Calle> < Altura> ; más adelante podría hacerlo utilizando algún método de geolocalización. (Ver US127)

Criterio de aceptación:

* El usuario ingresa una búsqueda de un producto y selecciona la estrategia çaminando lo menos posible".
* El resultado de la búsqueda será, entre las ofertas que cumplen las demás restricciones seleccionadas, la mas cercana a la posición informada por el usuario.
* En caso de no haber ofertas disponibles que cumplan las restricciones, se informará de esto al usuario mediante un mensaje.

Story Points: 5.0 **Iteración:** no definida **Estado:** Defined

US86: *Como usuario quiero ingresar el nombre de un producto y que el sistema me ofrezca los productos válidos más similares lexicográficamente*

Descripción: Se desea que el usuario pueda ingresar el nombre del producto que quiere buscar en formato "string", dentro de un campo de texto provisto por la interfaz. El sistema deberá mostrar los productos válidos que se parecen lexicográficamente al ingresado, y el usuario tendrá que elegir uno.

Criterios de aceptación:
* El usuario puede ingresar cualquier palabra o frase en el campo de texto llamado "producto". Por ejemplo, "japón".
* El sistema deberá mostrar una lista con productos que se parecen lexicográficamente al texto ingresado. Por ejemplo, "jamón; jabón"(suponiendo que en el momento de la consulta ambos son productos válidos del sistema).
* De no haber ningún producto que se considere parecido, se deberá indicar que no hay conicidencias. El usuario deberá poder borrar y reescribir el producto.

Story Points: 5.0 **Iteración:** no definida **Estado:** Defined

US127: *Como desarrollador quiero poder obtener la informaciòn de localización del usuario*

Descripción: En la actualidad, muchos dispositivos móviles tienen la capacidad de obtener información acerca de la ubicación actual del usuario, y comunicar este dato a las aplicaciones que están corriendo. La aplicación de Precio Justo la necesita para algunas de sus funcionalidades, por ejemplo la estrategia çaminando lo menos posible". Por lo tanto, es útil poder recibir la localización del usuario y operar con ella.
En caso de no estar disponible la información en forma automática, un método alternativo para obtenerla es el de pedirle al usuario que la ingrese manualmente.

Criterios de aceptación:
* El sistema conocerá la ubicación actual del usuario en cualquier momento que la necesite para operar.

Story Points: 8.0 **Iteración:** no definida **Estado:** Defined

US91: *Como usuario de la comunidad de Twitter quiero poder especificar una fecha de expiración para mi tweet de precioJusto*

Descripción: Cuando el usuario de Twitter publica un tweet de precioJusto, es posible que sepa que la oferta que está informando sólo es válida por tiempo limitado. En ese caso querría informar la fecha de expiración de la oferta cuando publica el tweet.
Por otra parte, queremos investigar otras maneras de conseguir el dato de una fecha de expiración. Por ejemplo, el usuario de Twitter que la publicó puede querer avisar que ya no es válida. También un usuario de nuestro sistema, que recibió la oferta en su búsqueda pero descubrió que ya no era válida, puede querer informarlo.

Además, queremos designar una fecha de expiración por defecto en las ofertas que no tienen una, para que los usuarios no reciban información obsoleta. Se desea investigar un valor razonable para este default.

Criterios de aceptación:
* El sistema tiene una fecha de expiración asignada a cada oferta.
* Si un usuario de Twitter publica un tweet que incluye una fecha límite, esta fecha deberá asignarse a la oferta.
* Ninguna oferta deberá aparecer como parte de un resultado de búsqueda si su fecha de expiración ya pasó.

Story Points: **Iteración:** no definida **Estado:** Defined

US112: *Epic: Como usuario deseo tener una interfaz para acceder a la aplicación desde distintos dispositivos*

Descripción: Se quiere desarrollar distintas interfaces para que el usuario consulte a la aplicación. Por el momento el Product Owner no requiere la implementación de ninguna interfaz en particular, pero a futuro puede hacerlo. Se deberá desarrollar la interfaz pedida, y asegurar que puede comunicarse correctamente con nuestro sistema.

Criterios de aceptación:
* El usuario podrá interactuar con el sistema correctamente a través de la interfaz de su dispositivo.

Story Points: **Iteración:** no definida **Estado:** Defined

US98: *Epic: Como PO espero que se pueda obtener información de tweets sin el hashtag precioJusto, y de otros medios de comunicación*

Descripción: A futuro se desea incorporar nuevas fuentes de información para conocer ofertas. Por el momento se requiere que las ofertas se obtengan por medio de tweets terminados con el hashtag #precioJusto. Sabemos que en próximas etapas del desarrollo se puede requerir que el sistema obtenga información de cualquier tweet, y también es posible que se quiera buscar en otras redes sociales como Facebook y Google+. Cuando esto se pida, deberemos asegurar que podemos extraer información correcta y completa de estas fuentes.

Criterios de aceptación:
* Cuando se designen nuevas fuentes de información, el sistema deberá poder leer, interpretar y usar los datos obtenidos de dichas fuentes.

Story Points: **Iteración:** no definida **Estado:** Defined

US92: *Como usuario espero que las ofertas obtenidas como respuesta sean las más recientes*

Descripción: Se debe proveer al sistema mecanismos que le permitan reconocer ofertas obsoletas entre los tweets que consigue. Consideramos que una oferta es obsoleta si existe un tweet más reciente que informa sobre el mismo producto en el mismo lugar, con un precio actualizado. Se desea que el sistema reconozca cuál es el tweet menos reciente y lo descarte por ser información obsoleta.
En caso de que el precio en ambos tweets sea el mismo, se lo considera información repetida, por lo que también se puede descartar al más antiguo sin inconvenientes.

Criterios de aceptación:
* El usuario realiza una búsqueda por un producto, para el cual el sistema conoce dos ofertas del mismo lugar, con distinto precio y distinta fecha.
* La oferta de fecha menos reciente no deberá ser incluida en la respuesta en ningún caso. La más reciente será incluida en el caso de que cumpla con todos los criterios impuestos por los filtros y estrategias de la búsqueda.

Story Points: **Iteración:** no definida **Estado:** Defined

US95: *Como usuario quiero poder interactuar con la información mostrada por el sistema (actualizar, denunciar, votar)*

Descripción: Se desea que los usuarios que recibieron una respuesta del sistema, en forma de una o más ofertas, puedan interactuar con ellas. Un usuario que descubre que la información ofrecida está desactualizada puede querer modificar esa oferta directamente desde el sistema (sin necesidad de abrir Twitter y escribir un tweet). Si encuentra que la información es incorrecta o engañosa puede querer denunciarla para que sea removida; también puede querer puntuar la calidad de la oferta de acuerdo a si su experiencia fue positiva o negativa.

Criterios de aceptación:
* Un usuario recibe una oferta desde el sistema, y encuentra que, en la realidad, el precio o la dirección han cambiado. El usuario enviará una corrección de estos datos directamente desde el sistema. En consultas subsiguientes, los usuarios recibirán la información actualizada.
* Un usuario encuentra que en la realidad la oferta ya no existe, o bien es fraudulenta por algún motivo. El usuario denunciará la información desde el sistema. Cuando el sistema recibe suficiente evidencia de que la información es inválida, esta no deberá aparecer como resultado de ninguna consulta.
* Un usuario quiere valorar la experiencia de compra de un producto que consiguió a través de una oferta del sistema. El usuario ingresará su valoración, y en consultas subsiguientes, los usuarios la verán cuando reciban la oferta.

Story Points: **Iteración:** no definida **Estado:** Defined

US120: *Como usuario de la comunidad de Twitter quiero que mis tweets se prioricen a la hora de mostrar las ofertas según su cantidad de favs o retweets*

Descripción: Twitter provee funcionalidad para que los tweets puedan ser marcados como favoritos ("favs"), o compartidos (retweets"), y se suele tomar estos datos como indicadores de la popularidad de un tweet. Queremos que nuestro sistema obtenga información de favs y retweets de los tweets que usamos para conocer ofertas, y que considere a los tweets más populares como "más confiablesz "de mejor calidad".

Criterios de aceptación:
* Un usuario de Twitter escribe un tweet con una oferta que nuestro sistema puede leer. Este tweet recibe gran cantidad de favs y retweets.
* En una consulta cuyo resultado incluya a este tweet, el sistema deberá darle prioridad por sobre los otros tweets que tengan menor cantidad de favs y retweets.

Story Points: **Iteración:** no definida **Estado:** Defined

US126: *Epic: Como usuario quiero poder buscar más de un producto en una misma consulta*

Descripción: Se quiere poder extender la funcionalidad actual de consultas del usuario para que pueda realizar una consulta con múltiples productos. Se debe investigar cómo afecta esta modificación a las funcionalidades ya existentes. Por ejemplo, en la estrategia çaminando lo menos posible", además de considerar la distancia entre el usuario y el producto, se debe contemplar la distancia entre cada par de productos entre los seleccionados.

Criterios de aceptación:
* Un usuario quiere buscar ofertas para dos o más productos en una misma consulta. El sistema deberá devolver resultados correctos para todos ellos.
* Los filtros y estrategias se aplicarán como corresponde considerando el hecho de que hay varios productos involucrados en el cálculo.

Story Points: **Iteración:** no definida **Estado:** Defined

4. Diseño

Comenzaremos explicando el diseño del sistema desde afuera hacia adentro. Es decir desde las capas mas externas del mismo hacia el corazón de nuestro modelo. Para cada diagrama haremos incapié en sus puntos destacados y referencia a aquellos otros que puedan ayudar o que complementen la comprensión del mismo.

4.1. Frontend-Backend

La aplicación desarrollada se descompone en dos componentes, un *Frontend*, a traves del cuál el usuario accede a los servicios del sistema y se concentra en la presentación de la información; y un *Backend*, al cual se le delegan las tareas de construir un resultado y comunicarselo al *Frontend* a partir del input del usuario.

A continuación detallaremos el diseño del Frontend. Dado que nos encontramos en la primer iteración del proceso de desarrollo, el foco se concentró en proveer una interfaz clara para comunicar el Frontend con el Backend, y no una implementación completa de la capa de presentación de los datos.

La clase **WebFrontend** es la encargada de la presentación de los datos. Esta clase posee un único método utilizado por la librería *CherryPy* que lo exporta en el formato de un método rest[1]. La capa de presentación de los datos se encuentra suficientemente desacoplada de la capa de comunicación con el Backend, de forma que puede cambiarse por completo la primera, sin afectar el resto de funcionamiento.

La clase **OfferBackend** representa al Backend de la aplicación, con el cuál el Frontend se comunica. Se decidió que esta clase debe ser *abstracta* debido a que podrían haber varios métodos de comunicación con el Backend (Por ejemplo, mediante un servicio REST, de forma local, etc...). Son las subclases de OfferBackend las encargadas de definir esta cuestión. Un caso particular de esto es **OfferRestBackend**, que utiliza REST para comunicarse con el Backend.

La clase *RestCommunicator* provee una interfaz con la cual comunicarnos con un servicio REST como el implementado.

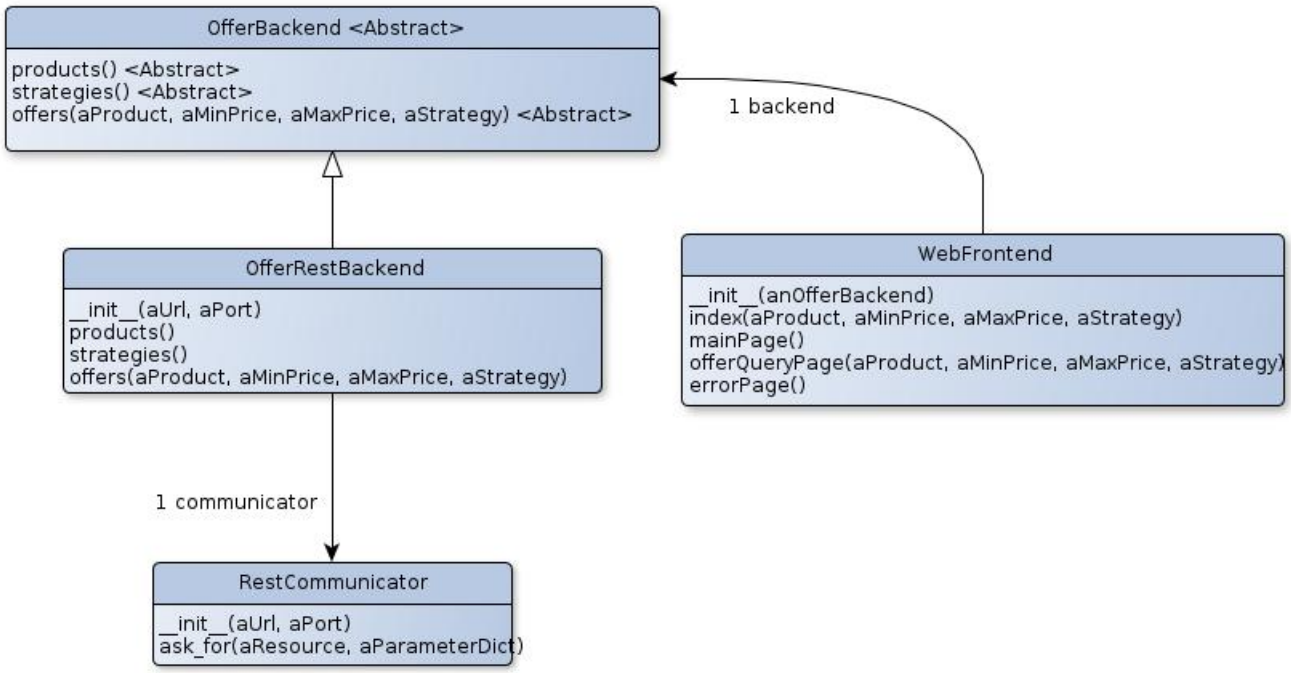

Figura 1: Frontend

[1] https://en.wikipedia.org/wiki/Representational_state_transfer

4.2. REST Api

Luego de que el *Backend* haga su pedido a través de *RestCommunicator* el mismo llegara a nuestra *REST Api*. Para lograr esto utilizamos de nuevo a *CherryPy* que nos provee un mensaje *quickstart* para elegir una clase que hará el manejo de dichos pedidos. El siguiente diagrama ilustra dicha clase. El resto de las colaboraciones con estas clases serán explicadas en detalle durante secciones posteriores.

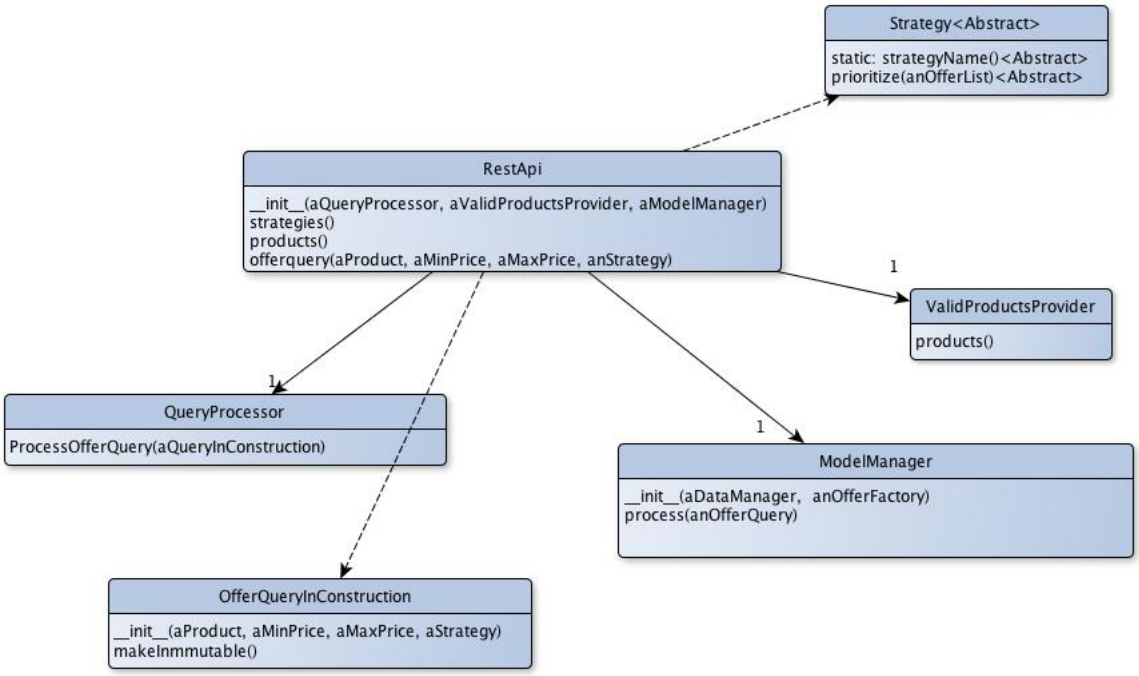

Figura 2: RestApi class

El comportamiento más interesante se encuentra en el mensaje *offerquery*. El diagrama de secuencias debajo presenta el comportamiento global del **Backend**, iniciado por el mensaje *offerquery* recibido por la **RestApi**, clase que se encarga de la comunicación con el exterior. Esta clase sabe traducir desde y hacia el lenguaje externo, en este caso el protocolo Rest y el lenguaje Json. La funcionalidad luego de la misma es recibir el mensaje en formato *rest*, convertirlo en una *query* de nuestro modelo llamando a un QueryProcessor, para así obtener una OfferQuery. Esta OfferQuery se le pasa al ModelManager que nos devolverá el resultado deseado y que RestApi se encargará de traducir al lenguaje utilizado en el exterior de la aplicación.

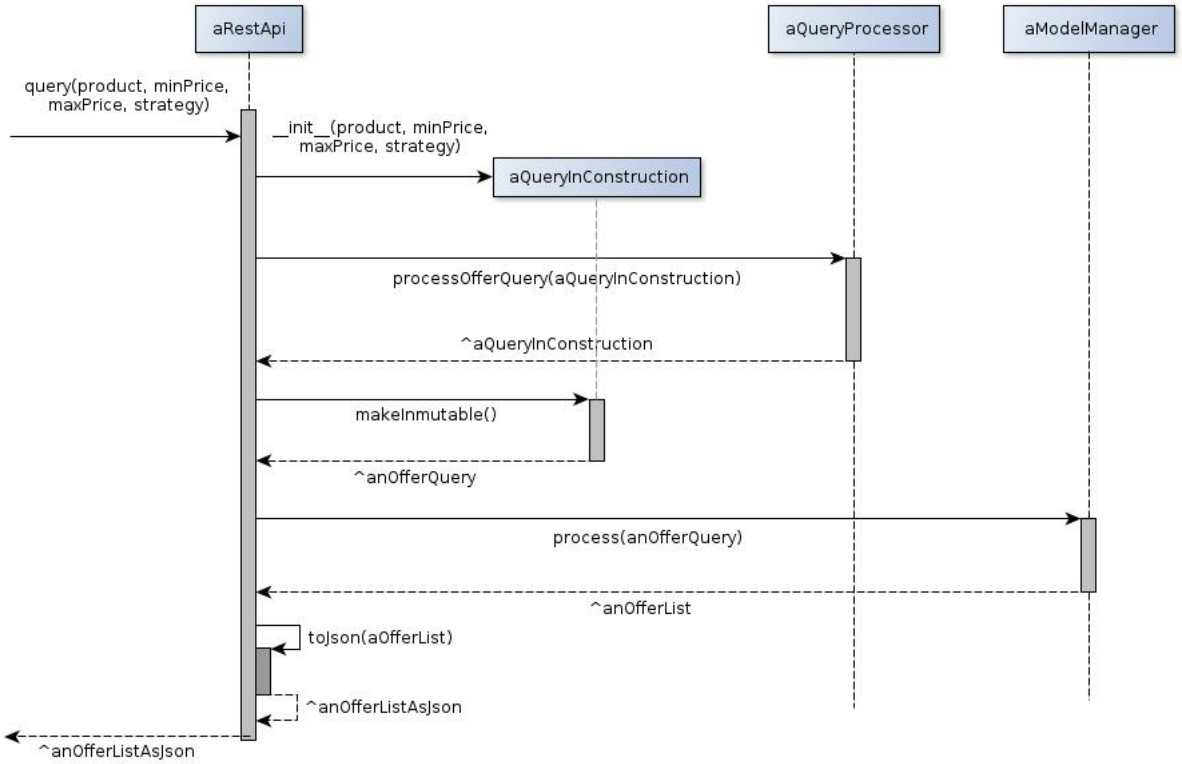

Figura 3: RestApi sequence

4.3. QueryConstruction

En esta sección explicaremos con mas detalle el proceso y el diseño involucrados en la construccion de *querys* que ocurre en los primeros mensajes del diagrama previo (*RestApi sequence*).

En este diagrama se destacan las clases de objetos que se encargan de, dada una **OfferQueryInConstruction** (Objeto que representa una consulta por ofertas sin procesar), construir el conjunto de restricciones a aplicar sobre las ofertas. Son los objetos de tipo **QueryProcessor** los encargados de construir los **Filter** y **Strategy** que determinan cuales serán los resultados finales de la consulta. En caso de que alguno de los párametros introducidos por el usuario no sean válidos, el **QueryProcessor** correspondiente informará de esta situación mediante una excepción. La clase **QueryProcessor** y sus subclases, siguen el patron de diseño *Composite*.

Los objetos de tipo **ValidProductsProvider** llevan cuenta de cuales son los productos válidos. Un objeto de este tipo es utilizado por un **ProductQueryProcessor** para verificar que el producto ingresado por el usuario se encuentre en la lista de los productos soportados por el sistema.

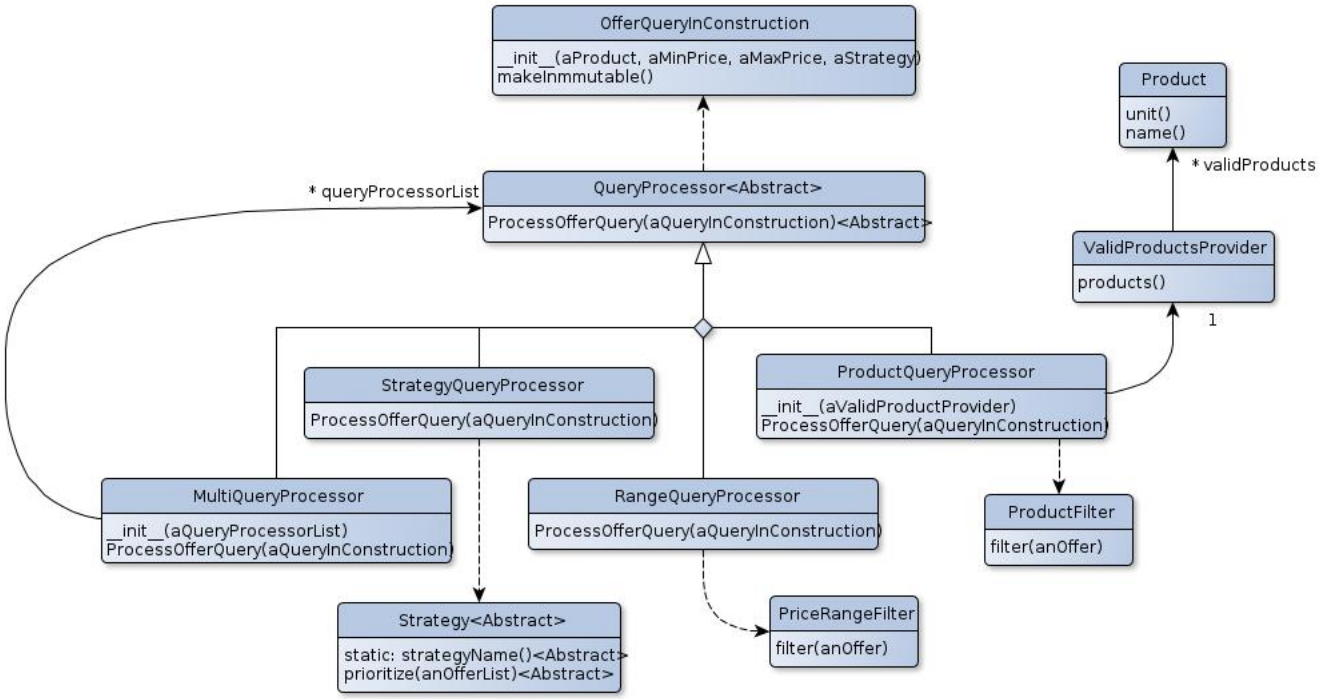

Figura 4: Query Processor

Como fue mencionado anteriormente, un objeto de tipo **OfferQueryInConstruction** representa una query del usuario, la cual todavía no se terminó de procesar. Una vez que se terminaron de construir los filtros y estrategias correspondientes a la consulta del usuario, es necesario hacerla inmutable, obteniendo así un objeto con tipo **OfferQuery**. Los objetos de esta clase poseen únicamente un **Filter** y una **Strategy**.

Las **Strategy** representan distintas políticas para priorizar ofertas, por ejemplo, *Mas baratas primero*. Los **Filter** representan condiciones booleanas que deben cumplir las ofertas para poder encontrarse en el conjunto de resultado. La clase **Filter** cumple con el patrón de diseño *Composite*. La clase **MultiFilter** sigue el patrón *TemplateMethod*, esta clase define un esqueleto para la función *filter*, sin embargo, son las clases derivadas las que implementan los métodos *reduceOp* y *baseCase*.

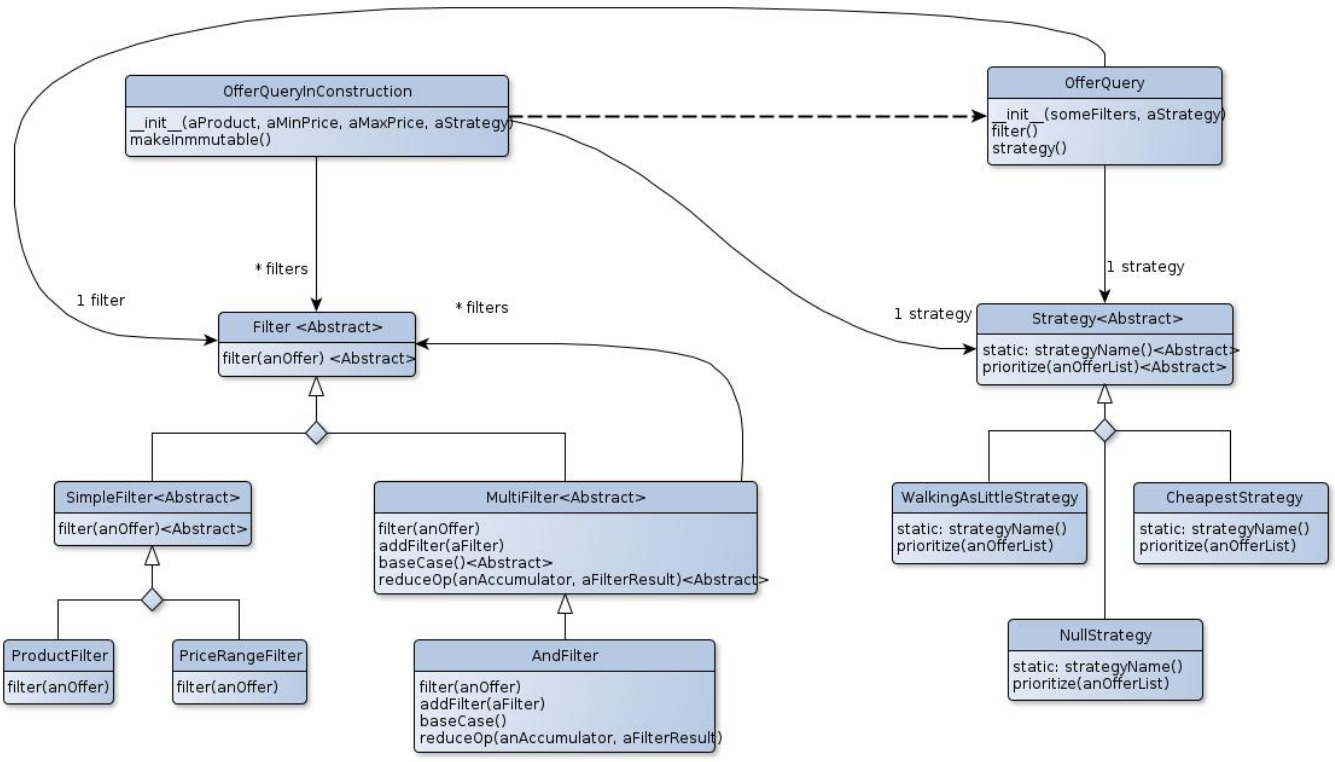

Figura 5: Filters and Strategies

4.4. Filters

En el siguiente diagrama se presenta el diagrama de secuencia para un **AndFilter**, que posee como colaborador a un **ProductFilter** y un **PriceRangeFilter**. Un **AndFilter** es un filtro que da su aprobación cuando todos los filtros contiene internamente también dan su aprobación. La interacción que se captura en este diagrama es la del mensaje *filter* con *anOffer* pasada como parámetro, a un **AndFilter**. El método asociado a este mensaje está implementado en **MultiFilter**.

Este método envia los mensajes *baseCase* y *reduceOp* a sí mismo. Los métodos asociados a estos mensajes están implementados en **AndFilter**. *baseCase* es utilizado para inicializar un acumulador interno, mientras que *reduceOp* se aplica a este acumulador y al resultado de aplicar cada filtro.

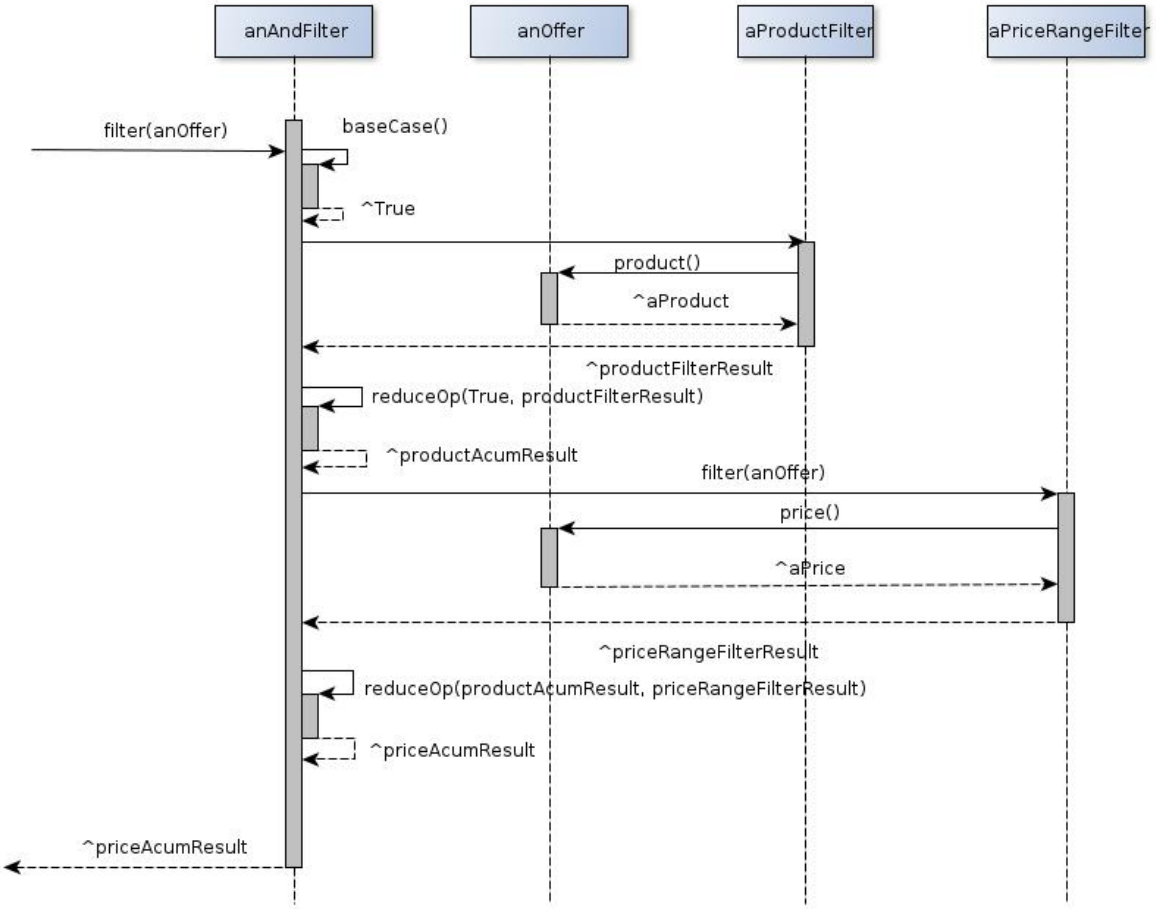

Figura 6: andFilter

4.5. OfferQueryProcessing

El siguiente diagrama gira entorno a la clase **ModelManager**, cuya responsabilidad es procesar una *query* contra las ofertas que el sistema encuentra en *Twitter*. Su nombre nace de su interacción con sus colaboradores internos (**DataManager**, **OfferFactory**) y parámetros (**OfferQuery**). Para realizar el procesamiento de una *query* el mismo debe ir interactuando uno a uno con los mismos. Dicha interacción resulta a nivel gráfico en una forma de estrella.

El primer eje de cambio que tuvimos en consideración y se ve reflejado en el diagrama es la posibilidad de tener más de una fuente de datos además de la REST api de *Twitter*. Este eje se logra facilmente agregando otro **DataSource** concreto a la instancia de DataManager con el ménsaje *addConnector*.

El siguiente eje de cambio está en el uso de la interfaz *iterable* por **OfferFactory** la misma genera un desacoplamiento de fácil reemplazo por cualquier tipo de almacenamiento de datos que provea una manera de iterarlos mediante el mensaje *next()*. Por ejemplo, en siguientes versiones del sistema podría utilizarse algun mecanismo de persistencia que ayude a eliminar el costo de hacer el request a nuestros conectores cada vez y la clase **OfferFactory** no debería modificarse.

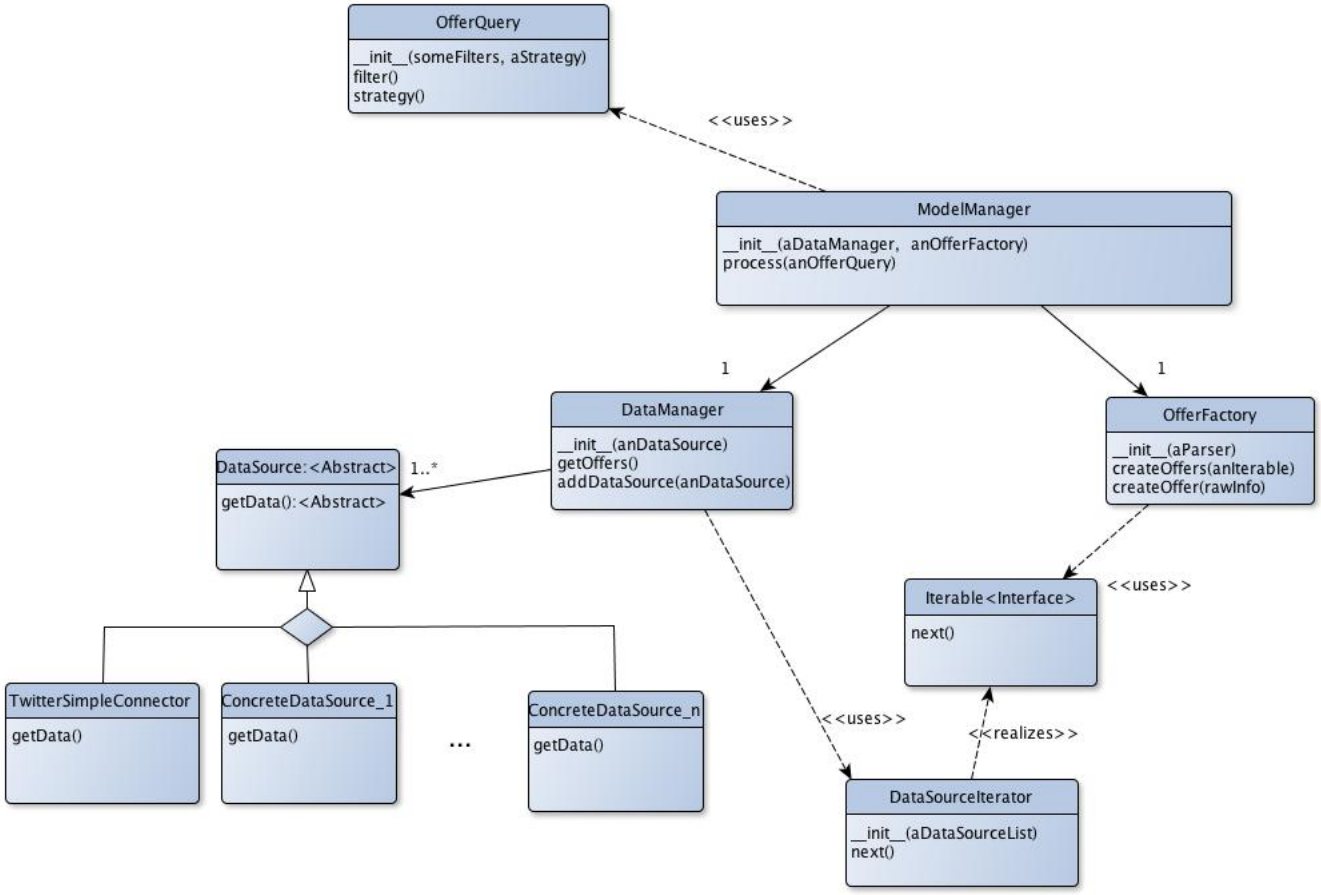

Figura 7: Model Manager Class

La interacción primordial de **ModelManager** esta en el mensaje *process(anOfferQuery)* que es llamado por **RestApi** luego de el armado de la *anOfferQuery* visto en la sección previa. Vemos entonces como esta compuesta esta interacción en el siguiente diagrama.

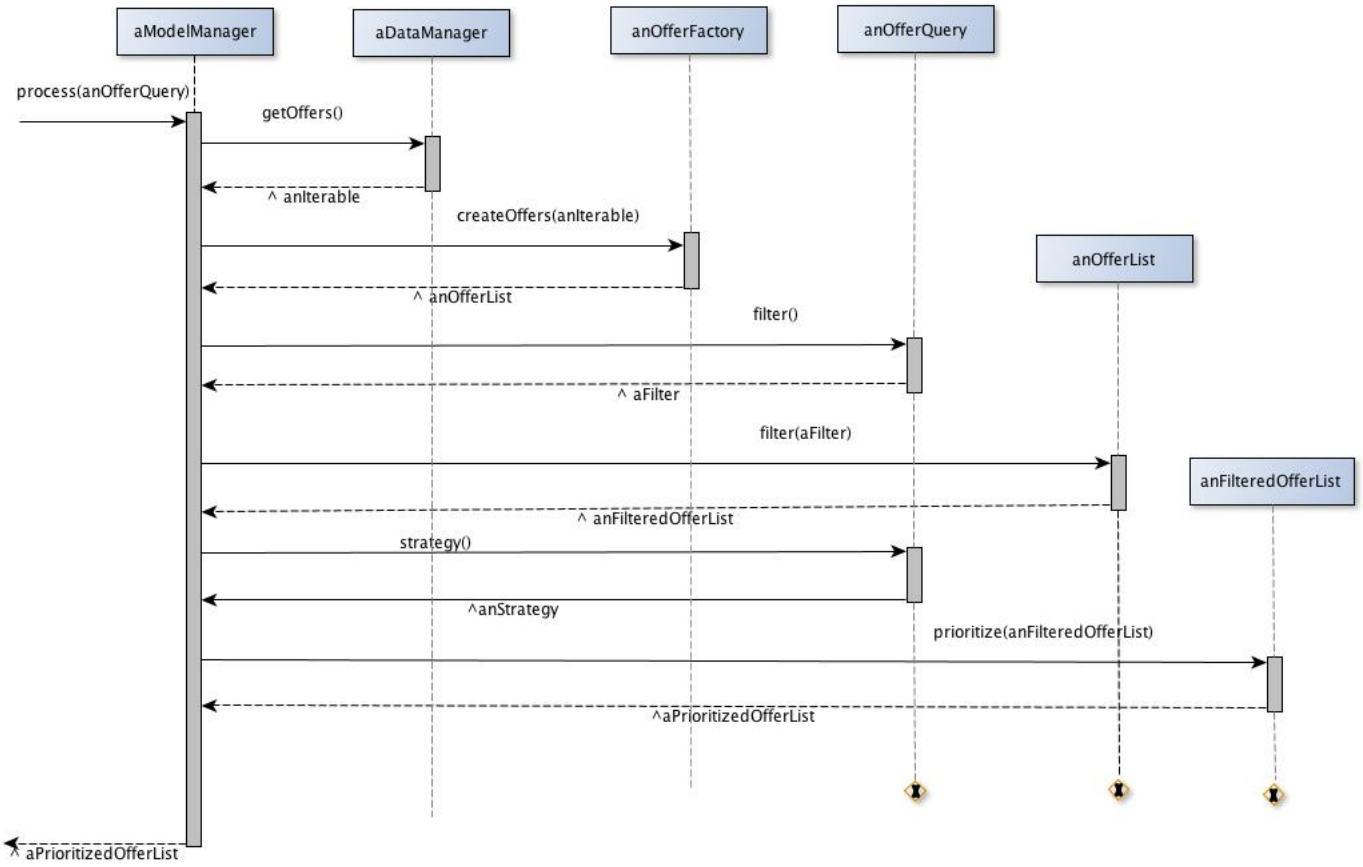

Figura 8: Model Manager Sequence

En esta secuencia se vislumbra lo que viene en la próxima sección vinculada al armado de *ofertas* desde los datos obtenidos desde *Twitter* con el mensaje *getOffers()*.Luego de obtener un iterador de los datos se delega en **OfferFactory** la responsabilidad de traducción, parseo y validación de los datos iterables hasta devolver una lista de ofertas.

4.6. Offer Building with OfferFactory

Para pasar de los Tweets (u otro medio, ya que en este punto es transparente el origen de la información) a ofertas entendibles en nuestro sistema. Implementamos un **OfferFactory** el cual es capaz de transformar una lista de tweets (validos o no) en una lista de ofertas válidas.

Para esto el OfferFactory es inicializado con un **ParserChain** que se encarga de intentar parsear la información de los twists devolviendo como resultado un **OfferBuilder** o la excepción **ParserError** en caso de no ser posible para el parser extraer la información para rellenar el OfferBuilder.

El ParserChain es Inicializado con una lista de **SpecificParser**, los cuales son llamados en el orden en el que vienen en la lista, tratando de extraer la información que necesitan y volcándola dentro del OfferBuilder que se está rellenado.

Este diseño nos da una gran flexibilidad para los cambios futuros, tanto en agregar nuevas formas de extraer datos, como también alterar la forma en la que se extraen los dados ahora sin modificar otras.

Como por ejemplo sería muy fácil agregar el validador ortográfico para poder detectar productos mal escritos sin necesidad de alterar ninguna otra parte (en el diseño se muestra una clase tentativa que le agrega a **ProductParser** la posibilidad de chequear esto).

Lo mismo se aplica con la posibilidad de agregar validaciones a la **Location** la cual por el momento solo consiste en un string que no tiene ningún tipo de chequeo sobre su validez.

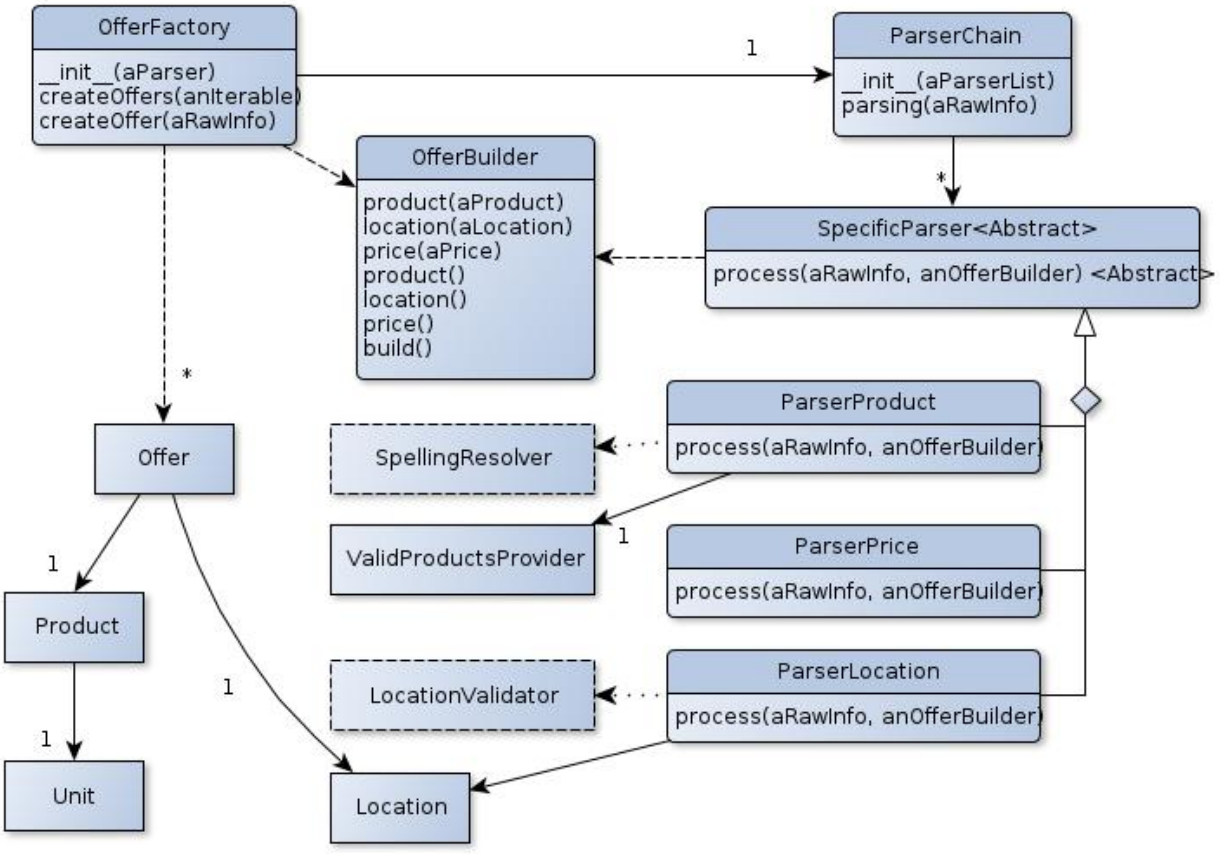

Figura 9: Offer Factory

En este diagrama se muestra el envío del mensaje *createOffers* a un **OfferFactory** pasándole como parámetro un objeto Iterable, este almacena un conjuntos de objetos del tipo **RawInfo** los cuales pueden ser pedidos uno tras otro utilizando el mensaje *next*.

En este caso particular el objeto Iterable va a tener dos elementos en su interior, de los cuales el primero es un dato válido ya que el objeto RawInfo contiene un text parseable por el sistema(ej:"Leche a 5 p el lt en varela al 20 #precioJusto"), mientas que el segundo contiene información no parseable(ej:"los mejores productos #precioJusto").

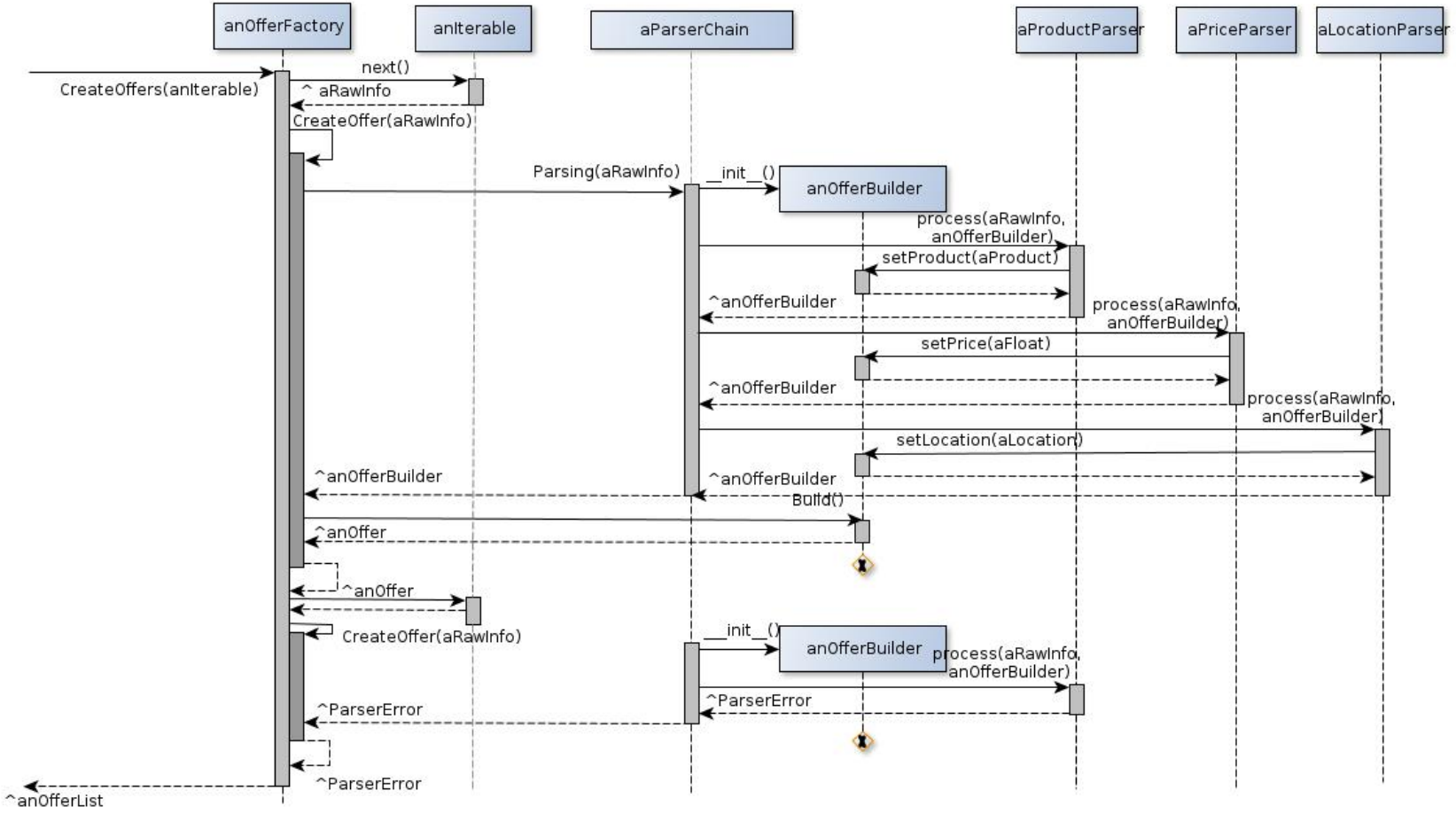

Figura 10: Offer Factory

5. Primera Iteración

En un esquema de desarrollo iterativo incremental, el producto final se construye en iteraciones. En cada iteración se establecen ciertos objetivos, que apuntan a agregar una nueva funcionaliad al producto. Para el momento de este informe, se desarrolló solo la primera iteración. En la misma, el objetivo definido fue tener una primera versión de la aplicación con una funcionalidad limitada, pero completa frente al espíritu del proyecto. El resultado de una iteración se denomina *product increment* y deberá permitir buscar las ofertas de un producto en *twitter*, filtrarlos con algún criterio y priorizalos mediante alguna estrategia.

A continuación explicamos cómo resulto el desarrollo de la primera iteración.

5.1. Desarrollo

En el ámbito del trabajo que se está realizando, es importante remarcar el hecho que el desarrollo iterativo incremental de la aplicación fue en el marco de un trabajo práctico cuyos objetivos exceden (e incluso son conflictivos con) los objetivos normales de una iteración. Es así que el trabajo se dividió en dos aristas: la de *diseño orientado a objetos* y la de *desarrollo del increment*.

Lo primero que se hizo fue definir los requerimientos y el alcance de la aplicación al generar el *Product Backlog*, cuyas historias detallarían estas decisiones. Luego surgió la necesidad de enfocarnos completamente en el diseño de la aplicación, lo que conformaba un punto muy importante del trabajo práctico. Esta sección de trabajo entra en conflicto con la metodología de desarrollo ya que en la misma la visibilidad completa del diseño no solo no es importante sino que debe postergarse para extenderse y corregirse contínuamente con las iteraciones. A razón de este trabajo inicial de diseño el desarrollo quedó en segundo plano hasta la segunda mitad del *sprint*, como puede verse en el **burndown chart**.

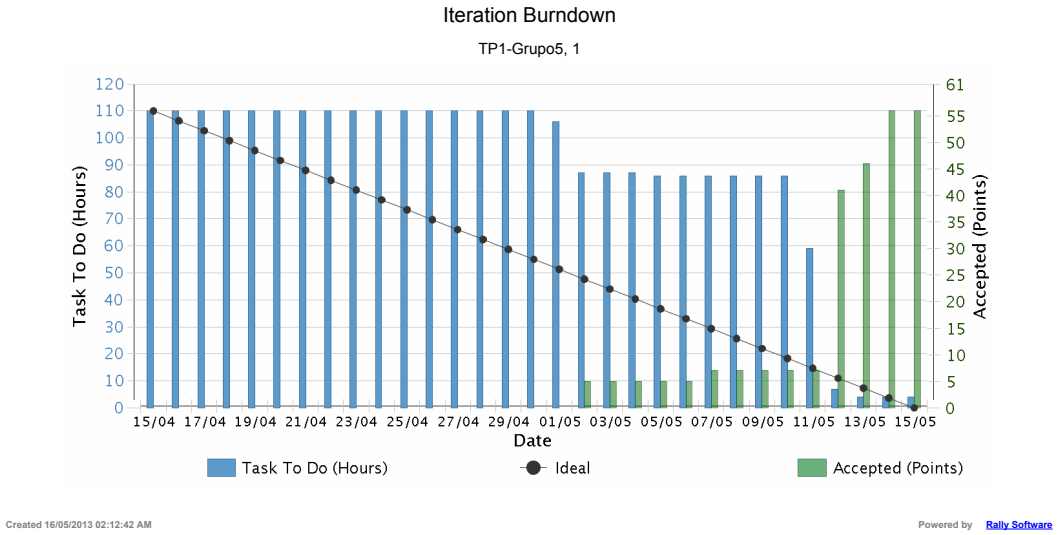

Figura 11: **Burndown chart:** progreso de terminación de tareas en el *sprint*.

Una vez contentos con el porcentaje de diseño realizado para los fines del trabajo práctico, el desarrollo de código adquirió prioridad. Considerando que los integrantes del equipo, teniendo otras responsabilidades, no podían dedicarse diariamente al proyecto, puede observarse en el gráfico que los días de trabajo están espaciados. Las actividades de

finalización de tarea fueron bastante postergadas, por lo que la cantidad de historias aceptadas creció abrúptamente en los últimos días.

5.2. Logros

El primer *product increment* se enfocó en lograr una primera prueba de un sistema funcionando completamente. Su desarrollo se orientó a que hubiera algo para presentar al *Product Owner* y al usuario final, así como para que fuera fácilmente expansible a los ejes de cambio definidos. Entre las funcionalidades básicas logradas se encuentran:

- Un frontend web que permite simple usabilidad para el usuario.
- Consulta a *twitter* de tweets con hashtag **#PrecioJusto** para obtener precios de productos.
- Interpretación de los *tweets* para obtener la información.
- Priorización de los menores precios encontrados para un producto.
- Filtrado de los precios acorde a un mínimo y máximo.

5.3. Revisión

Al finalizar el *sprint* realizamos una revisón sobre cómo trabajó el equipo y qué mejoras podrían hacerse. Llegamos a las siguientes conclusiones.

- **Poca comunicación con el PO:** a lo largo del *sprint*, la falta de comunicación con el **Product Owner** originó muchas sorpresas en cuanto a requerimientos y expectativas tanto del desarrollo del *increment* como del trabajo práctico. En futuras iteraciones es recomendable reunirse con el **Product Owner** con mayor frecuencia.
- **Seguimiento espaciado del proyecto:** a causa de otras responsabilidades y distracciones, las horas dedicadas al desarrollo estuvieron espaciadas y la mayor parte del trabajo se concentró en unos pocos días. En el futuro es recomendable una mayor frecuencia de trabajo de forma que los desvíos y las dudas puedan resolverse con mayor velocidad.
- **Poco asado:** evidentemente el trabajo del equipo se vio afectado por la falta de asado en las reuniones, que bajó la moral y denigró el espíritu de trabajo. En las próximas iteraciones la bondiola y vacío serán protagonistas.
- **Disparidad en la formación de los integrantes:** a causa de las distintas experiencias y líneas de trabajo de cada uno de los integrantes, nos encontramos con diferencias en la experiencia con las herramientas y variedad de forma de pensar las problemáticas. En el futuro será importante tener en cuenta que es necesario dedicarle más atención a cómo comunicamos las ideas dentro del equipo, así como equiparar mejor el conocimiento de las tecnologías.