

Ingeniería de Software II  
Sistema “El precio justo”

Grupo 6  
1º Cuatrimestre 2013

LU	Nombre	email
667/06	Daniel Foguelman	dj.foguelman@gmail.com
767/03	Hernán Modrow	hmodrow@gmail.com
511/00	Leonardo Tilli	leotilli@gmail.com
836/02	Paula Verghelet	pverghelet@gmail.com

I  
REENTREGAR

## **1 Parte II**

### **2 Introducción**

En esta segunda parte no estamos realizando la reentrega de la Parte I, planificación del sprint, dado que no pudimos realizar las correcciones indicadas oportunamente por falta de tiempo. Si bien las tendremos en cuenta en futuros sprints. Por tanto en esta segunda parte solo incluiremos aquellas cosas que no estaban en dicha entrega, es decir el informe respecto al seguimiento, product increment, retrospectiva y diseño orientado a objetos.

### 3 Seguimiento

#### 3.1 Beneficio del framework Scrum

El seguimiento del proyecto, utilizando la asignación de tareas por intermedio de RallyDev no nos resultó beneficiosa. Así como en otros escenarios (laborales por ejemplo) donde el seguimiento de las horas es informado a un stakeholder con interés en el avance, no nos dio resultados para mantener la velocidad esperada. Esto se refleja en el burndownchart donde se observa una súbita caída en las horas faltantes para finalizar el sprint.

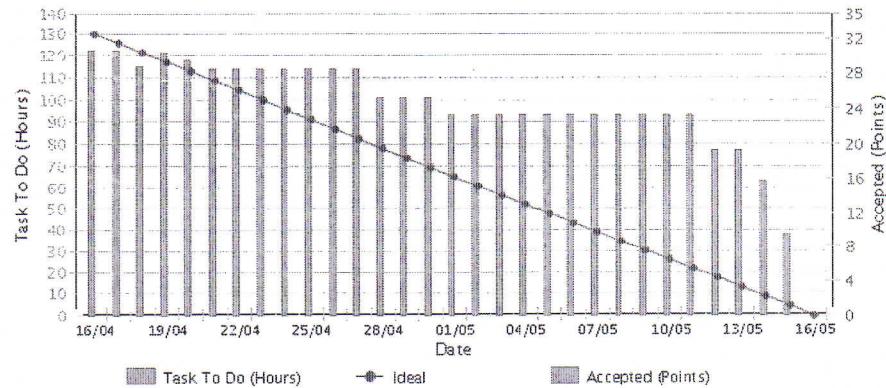
Los motivos para esto son diversos:

- Conocimiento grupal de las tareas
- Y de lo faltante (mucha comunicación por mail)
- Falta de comunicación del equipo con el stakeholder (falta de presión acaso?)

Cabe mencionar que al no poder realizar actualización de horas retroactivas, algo bastante común de realizar en otras herramientas de seguimiento, el burnout chart no es fiel reflejo de nuestro avance.

La utilización de Scrum para el manejo del proyecto nos brindó muchos beneficios en la primer etapa de análisis siendo una herramienta de mucha ayuda al descomponer los requerimientos en user stories y luego en tareas.

#### 3.2 Burndown chart



## 4 Product Increment

### 4.1 Arquitectura

Durante el análisis generamos diversas User Stories de investigación, estas nos permitieron hacer un análisis de la arquitectura antes de comenzar a implementarla. Con pruebas de concepto que nos permitieron entender la factibilidad de dicha implementación.

Los resultados de dicho análisis se encuentran al final de esta sección, en los siguientes párrafos haremos un análisis de la tecnología elegida.

### 4.2 Lenguaje

La decisión de utilizar un lenguaje de tipado dinámico fue tácita, quisimos evitar las restricciones semánticas que nos impone un lenguaje de tipado estático. También buscamos un lenguaje de Objetos Puro, dejando Python afuera de esta decisión. Finalmente optamos por Ruby, dinámicamente tipado, de objetos puro, con closures.

No es casualidad que la elección del lenguaje nos afectara también en la elección de las tecnologías de soporte de aplicación:

- Servidor SQL, para el soporte de datos de backend
- RESTful server, para la interfaz de aplicación
- Engine de templating para las vistas

### 4.3 Servidor SQL

Utilizamos SQLite 3 por su portabilidad y facilidad de instalación.

### 4.4 Server

Para brindar los servicios al usuario, utilizamos HTML con una API HTTPRest. Nuestro servidor es Sinatra. De fácil utilización solo es necesario conectar los servicios HTTP a la capa del controlador. ¿Por qué decimos que es REST? Básicamente no depende del estado, es orientado a cliente-servidor y está orientado en capas.

Esto nos permite desacoplar la aplicación de la interfaz de usuario, en la versión actual entregamos una interfaz modesta en HAML, un lenguaje de templating cuya transformación a HTML es nativa en Ruby + Sinatra.

### 4.5 Modelo

Para el modelo de aplicación aplicamos una versión reducida de MVC.

Por un lado tenemos el servidor de aplicación (sinatra como comentábamos en la sección anterior) y por otro lado tenemos un controller cuyas responsabilidades se ven acotadas a la validación del input de usuario y la delegación del procesamiento a ciertas acciones definidas en la capa de servicios.

Los servicios que actualmente soporta el controlador son los de búsqueda.

## 4.6 Extensibilidad

Nuestro diseño como veremos (y justificaremos) en las siguientes secciones, es extensible en:

- Tipos de filtros: nuevos tipos de búsquedas podrán agregarse sin modificar ni la validación de los parámetros ni la interfaz de usuario. Esto lo conseguimos mediante una sencilla técnica de metaprogramación.
- Tipos de consumo de datos de Twitter: actualmente soportamos la api on-line y la api off-line.
- Estrategias de parsing de tweets: en esta versión acotada utilizamos una extracción de datos posicional por medio de regular expressions, esto podrá ser modificado para utilizar un interprete de lenguaje natural que extraiga estos datos cuya ambigüedad es tan variada.
- Interfaces de usuario: actualmente brindamos una interfaz HTML pero podría interactuar con una UI distinta, braile, de escritorio, distribuida, etc.

## 5 Retrospectiva

Principalmente disfrutamos de aprender Ruby, siempre es bueno aprender un lenguaje nuevo.

Creemos que las dificultades no estuvieron en el aprendizaje de nuevas herramientas, si no más bien en tratar de mantener el ritmo de trabajo.

Mención aparte merece el tema del trabajo en equipo. Acá nos encontramos con algunas dificultades, muchas por desconocimiento de cómo trabaja el otro. Ya que salvo Daniel y Hernan el resto nunca había trabajado en un TP juntos. Algunos miembros se adaptaron al dinámica de grupo que se estableció y otros no.

Esto creemos que afectó en la entrega dado que en la recta final algunos miembros del grupo se sobrecargaron de tareas.

También hicimos mucho foco en las herramientas y la implementación y no tanto en el diseño formal. Nos dejamos llevar por el entusiasmo por lo nuevo, si bien estamos convencidos de que el diseño que pensamos es bueno.

Este foco por la implementación y el ritmo tardío de trabajo hizo que nos comunicáramos poco con el product owner para validar nuestro avance. Tal vez acá nos podría haber llamado un poco más la atención, como entendemos que hubiera sucedido en un escenario real (si bien sabemos que ya estamos grandecitos).

✓. ¿Qué nos se puede mejorar para el 2º Sprint?  
✓. ¿Cómo resultó la retrostímula?

## 6 Diseño Orientado a Objetos

### 6.1 Ciclo de vida de una consulta

El siguiente diagrama muestra la interacción en los distintos componentes de la aplicación. Los distintos componentes serán explicados en las siguientes secciones.

¿ ¿Llamación en lenguaje NATURAL?

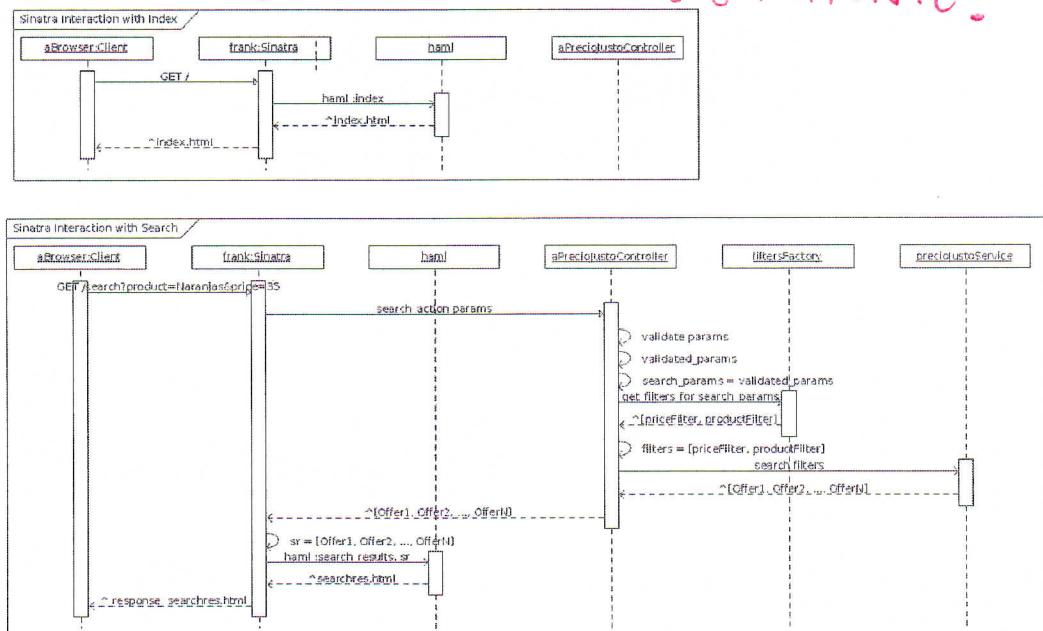


Figure 1: Diagrama de secuencia de sinatra

### 6.2 Factory de servicios

Dado que twitter permite dos tipos de obtención de tweets, vía streaming o por búsquedas, es que implementamos dos formas de trabajar con los tweets, pero que desde el lado de clientes de la búsqueda de ofertas de El Precio de Justo mantienen la misma interfaz.

Para esto último decidimos utilizar el patrón de abstract factory con dos implementaciones, una online y otra offline. Cada una de estas implementaciones nos permiten crear la clase de servicio apropiada (la clase que encapsula la lógica de la aplicación), los filtros permitidos y la implementación específica de estos.

Esto puede verse en la figura 2.

FACTA EXPLICAR.

### 6.3 Filtros

Dado que el filtrado de los tweets de las ofertas buscadas se realizan de distinta manera, los filtros si bien deberán proveer las misma funcionalidad son

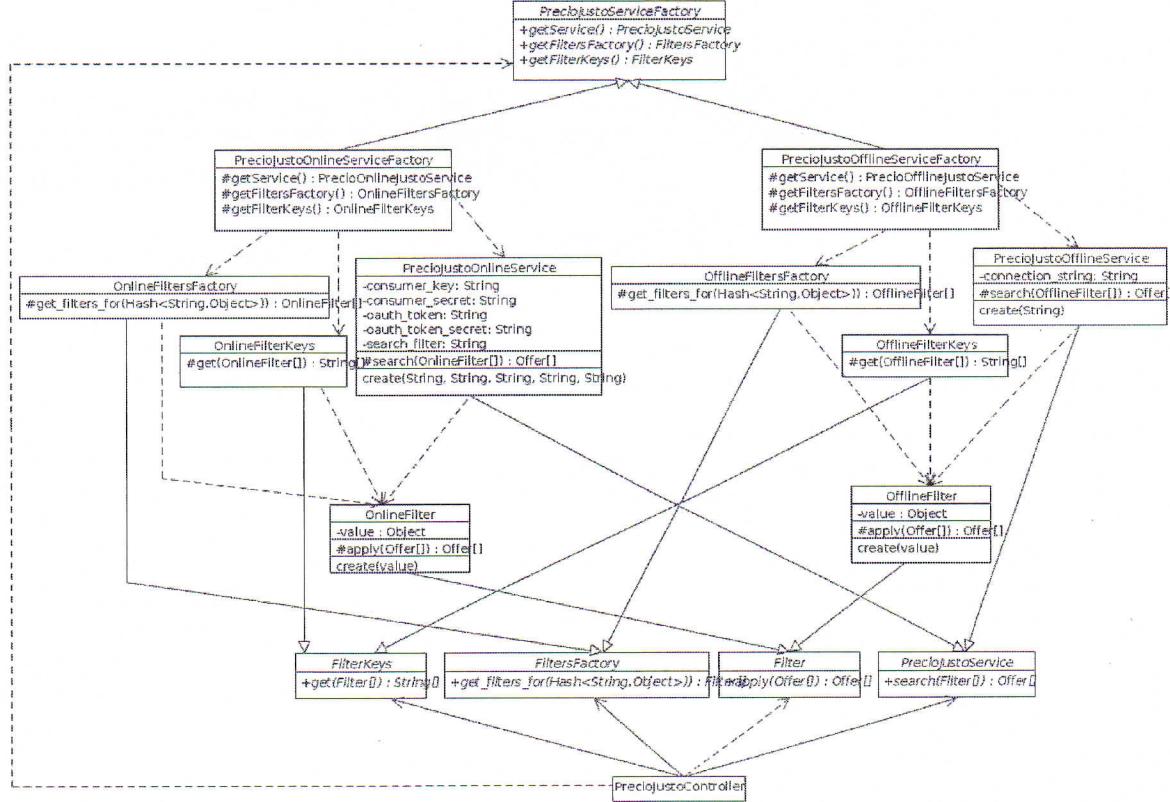


Figure 2: Diagrama de clases de creación de servicios online y offline

implementativamente distintos.

En un caso filtrarán las ofertas que se vayan obteniendo de twitter en vivo y en el otro caso se realizará un búsqueda sobre el motor de base de datos.

Los filtros modelados son por producto, precio y ubicación, si bien es posible extenderlos.

Esto puede verse en la figura 3.

*No Se tarda, falta ejecución*

#### 6.4 Extracción de datos de un tweet

La extracción de las información de las ofertas que se encuentra en los tweets se realizar mediante objetos de la clase OfferFromTweetExtractor (clase abstracta).

Para la demostración realizamos una implementación que extrae la información mediante expresiones regulares para cada pedazo de información dentro del texto, salvo para la geolocalización del tweet.

Si bien no está implementado para la demo, es deseable delegar la creación de las instancias de Offer. Para ello pensamos realizar mediante un OfferBuilder, la tarea del mismo sería realizar las acciones necesarias para crear una nueva instantancia.

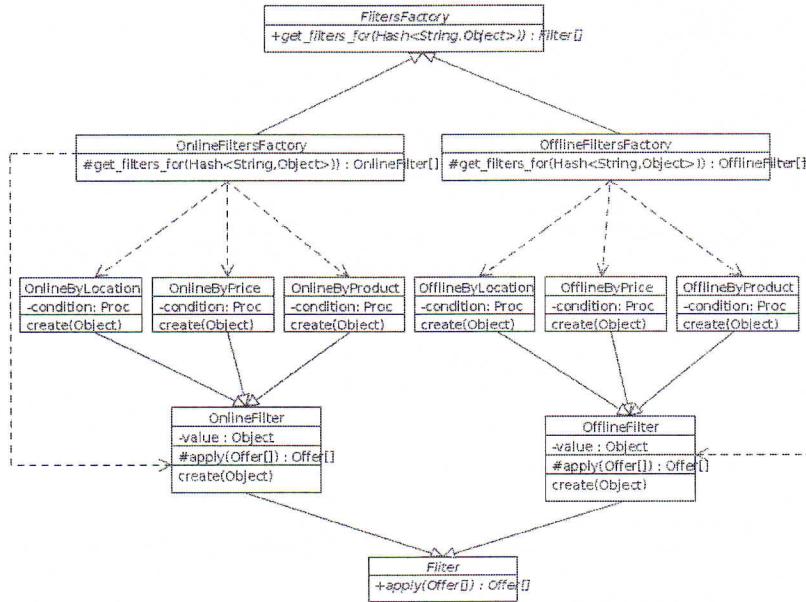


Figure 3: Diagrama de clases de creación de filtros

La idea es que cada pedazo de información generé una instancia de un objeto polimórfico de la información que modela (ej: el precio, producto, unidad). Cuando nos referimos a objetos polimórficos, queremos decir que la información podría no poder generar un objeto válido y queremos que resulte transparente para los objetos con los que interactúan con dicho objeto (la idea es poder implementar Null Object Pattern).

## 6.5 Servicio online

En este caso la clase PrecioJustoOnlineService encapsula la lógica de la aplicación utilizando como fuente de información la búsquedas directas en Twitter mediante la API de Search.

Para aplicar los filtros utiliza las clases derivadas de OnlineFilter y utiliza OfferFromTweetPositionalExtractor para la extracción de las ofertas.

## 6.6 Servicio offline

### 6.6.1 PrecioJustoOfflineService

En este caso la clase PrecioJustoOfflineService encapsula la lógica de la aplicación utilizando como fuente de información una base de datos SQLite3. La cual es alimentada mediante un servicio independiente.

Para aplicar los filtros utiliza las clases derivadas de OfflineFilter. Las ofertas ya fueron parseadas antes de ser insertadas en la BD.

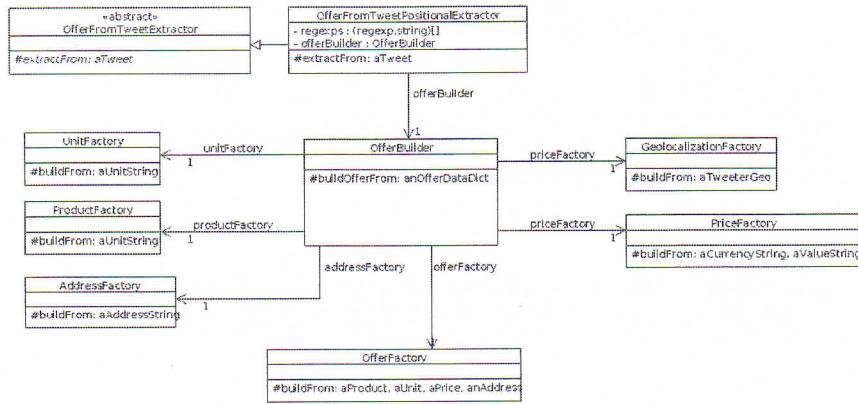


Figure 4: Diagrama de clases de extraccion datos de tweet

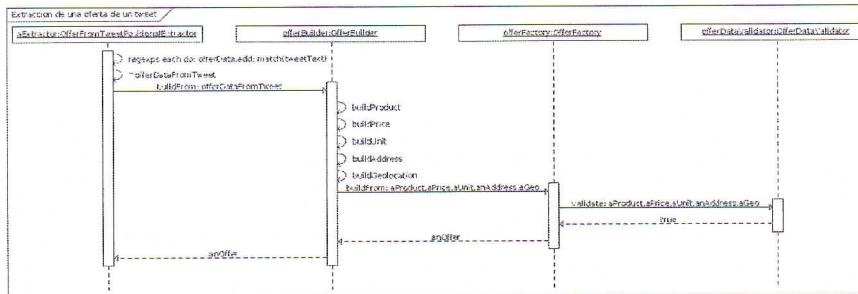


Figure 5: Diagrama de secuencia de extraccion de datos de tweet

### 6.6.2 Servicio de Streaming

Este proceso independiente utiliza la API de Streaming de Twitter para obtener las ofertas y persistirlas en la base de datos.

Para la extracción de las ofertas de los tweets utiliza el extractor OfferFromTweetPositionalExtractor.

### 6.6.3 Cliente/Servidor SQLite3

Dado la base de datos SQLite3 es un archivo, el mismo se bloquea al accederlo directamente. Para evitar esto generamos un proceso independiente que es el único que bloque el archivo, pero expone mediante DRB (remoting) la conexión a esta base para poder ser usada concurrentemente por distintos procesos.

## 6.7 Ofertas y colaboradores

El siguiente diagrama muestra el modelo de las ofertas (Offer) y sus colaboradores. Dichos colaboradores no están modelados en la demostración.

El siguiente diagrama muestra la idea de creación de los distintos objetos con la idea de poder generar objetos no válidos polimórficos a los válidos.

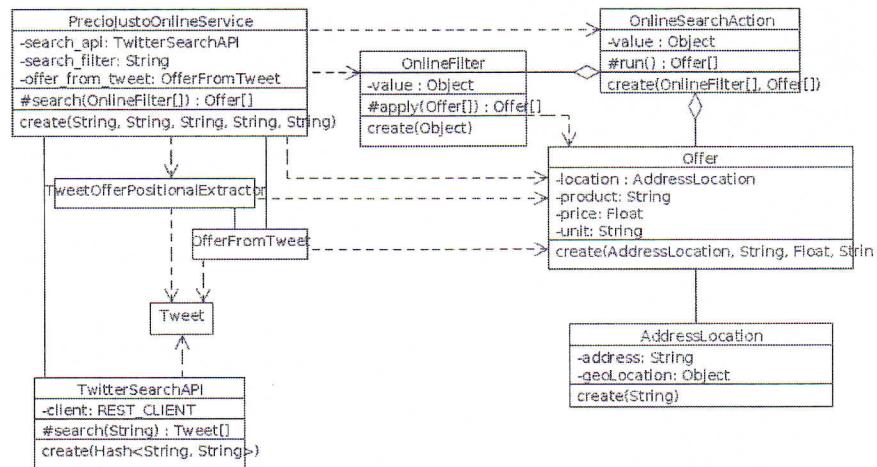


Figure 6: Diagrama de clases de servicio online

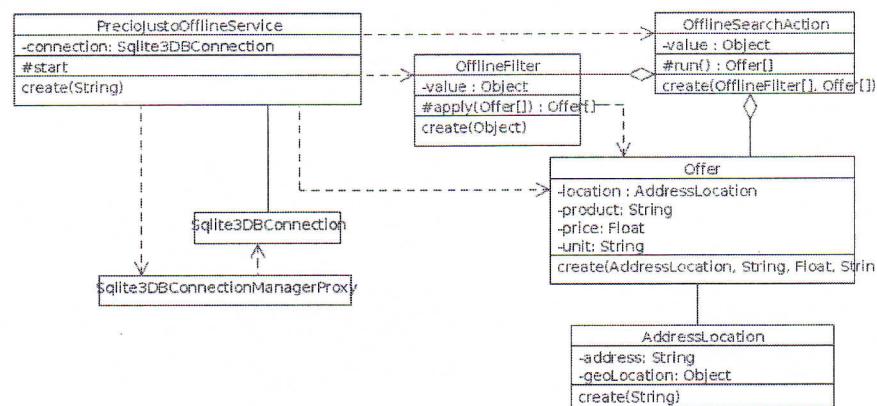


Figure 7: Diagrama de clases de servicio offline

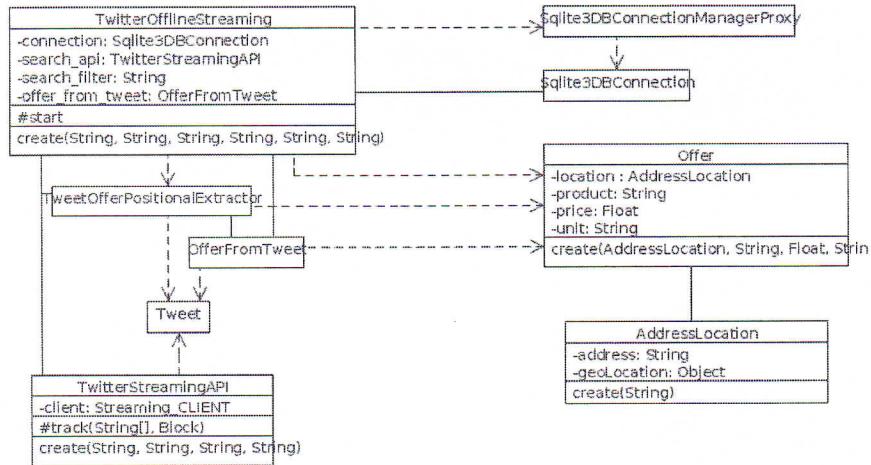


Figure 8: Diagrama de clases de streaming de twitter

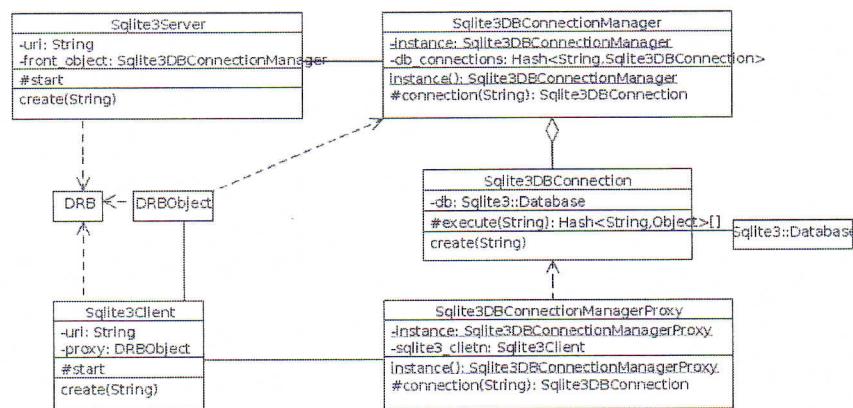


Figure 9: Diagrama de clases de BD SQLite3

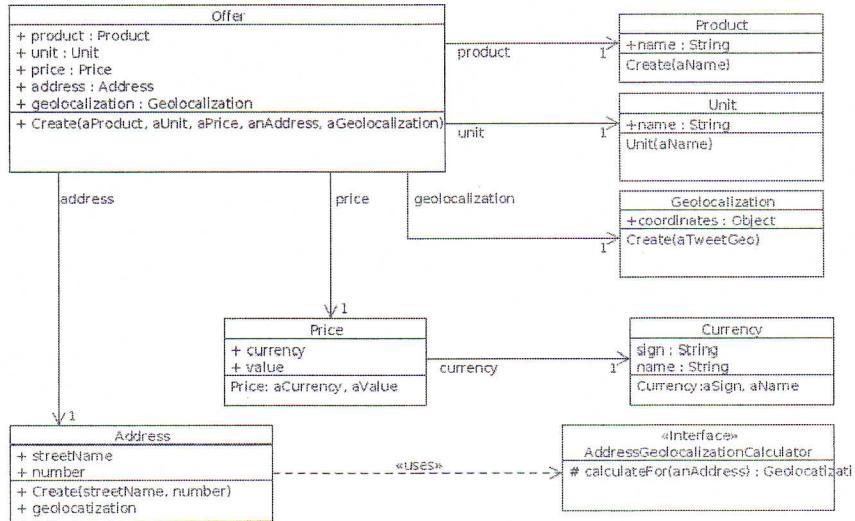


Figure 10: Diagrama de clases de offer y colaboradores

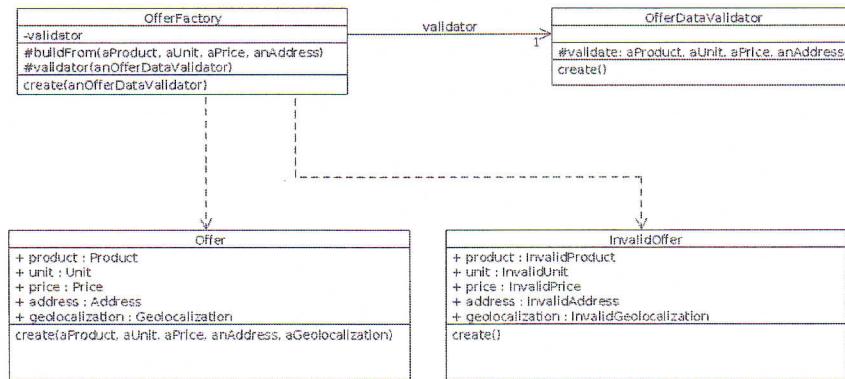


Figure 11: Diagrama de clases de creación de offer

*Informe más auto-contenido.  
para presentar y justificar<sup>13</sup> el diseño a secundaria.*

*I  
RE-  
ENTREGAR.*

*¿Cómo se obtiene info de Twitter?  
¿Cómo se validan las en Twitter?  
¿Cómo se obtienen y muestran los resultados?*