Informe Técnico TP Final
Análisis y Diseño Orientado a Objetos

Diciembre 2015 (2do. Llamado a final del equipo)

Grupo:

- **Ignacio Luna Echechuri (22763)**
- **Martín Fernández Gamen (22764)**

Índice:

1) Definiciones y consideraciones generales

El presente trabajo viene a acercar nuestra propuesta de solución para el requerimiento del TP3 final Diciembre 2015 cuyas especificaciones constan en el siguiente link: https://docs.google.com/document/d/1bVY0kAl93K-AmJxQEwWpxHzv35kRmWFCa43etBw5yWM/edit#heading=h.jjfr54i8xtfr

Nuestro producto editor de entidades JSON, llamado **"jsongen"**, permite generar archivos JSON que definan objetos (instancias de esas entidades) a partir de un archivo JSON de definición de atributos de la entidad. Para ello provee dos modos de operación bien definidos:

- Modo Consola.
- Modo Gráfico.

Requisitos previos para la Ejecución del programa:

Se requiere para la ejecución del programa la instalación previa de las siguientes herramientas:

- Java(TM) SE Runtime Environment (build 1.8.0_60 o superior para soportar la ejecución de JavaFX).
- Ant 1.9.2 o superior para el build de la aplicación.
- Ivy 2.2.0 o superior para el manejo de las dependencias.
- Clonar el repositorio de Git: https://github.com/martinffg/aydoo-final-json.git
- Correr ANT dentro del directorio donde se clonó el repositorio para buildear el código y descargar las dependencias.

Modos de Ejecución del programa:

Para correr la aplicación en ambos modos se deberán seguir los siguientes pasos:

- Luego de correr ANT para buildear el proyecto, para ejecutar el proyecto se debe acceder a la subcarpeta **build** del directorio donde se clonó el repositorio. Allí se podrá ejecutar la aplicación en ambos modos.
- Según los parámetros enviados al programa abrirá un modo o el otro.

Ejecución en Modo Consola:

Luego de correr ANT y posicionarse en el directorio BUILD del proyecto se ejecutará jsongen en modo consola con la sintaxis (la ruta debe incluir el nombre del archivo de definición):

java jsongen ***RutaArchivoDefinicion RutaYNombreObjetoSalida.json***

A modo de ejemplo, obtenemos la entidad alumno (**con todos sus valores vacíos**) haciendo:

java jsongen ***testFiles/definicion-alumno.json alumno.json***

En modo Consola la ruta default del archivo JSON de salida es: el mismo directorio de ejecución del programa, es decir, dentro del directorio build del proyecto para el ejemplo.

Ejecución en Modo Gráfico:

Luego de correr ANT y posicionarse en el directorio BUILD del proyecto se ejecutará jsongen en modo gráfico con la sintaxis (la ruta debe incluir el nombre del archivo de definición):

java jsongen ***RutaArchivoDefinicion***

A modo de ejemplo, obtenemos la entidad alumno haciendo:

java jsongen ***testFiles/definicion-alumno.json***

Este comando abrirá el formulario dinámico y permitirá editar los campos del archivo json de salida. Una vez que se hace clic o presiona ENTER en Guardar, el formulario queda bloqueado, se guarda el archivo y sólo resta cerrar el formulario dando clic en la X.

En modo Gráfico la ruta default del archivo JSON de salida es: la misma carpeta donde está el JSON de definición. Para el ejemplo será dentro de la carpeta build/testFiles/ del proyecto.

¡Atención!

Tenga en cuenta las siguientes adicionales para la ejecución de JSONGEN:

* No puede utilizarse rutas con un guión medio previo al nombre del archivo de definición. De no cumplirse, la aplicación traerá de manera errónea el nombre de la entidad.

* Tenga en cuenta que según el modo de ejecución variará la ruta de salida por default para no superponer archivos generados por distinto modo e igual nombre.

* Tener en cuenta que un archivo de definición JSON que cumple con el estándar y se puede observar tanto en http://www.w3schools.com/json/json_syntax.asp como en http://json.org/ debe ir delimitado entre llaves, por ejemplo, para definición-alumno.json será:

{"campos":[

{"nombre":"nombre", "tipo":"string"},

{"nombre":"apellido", "tipo":"string"}

]**}**

Nótese: el programa soporta el estándar, pero se hace la salvedad que falta el par de corchetes en el ejemplo del mismo archivo citado en el enunciado del TP final cuyo link fue agregado al comienzo del punto 1). Se agregarán más ejemplos en la sección 6) de Pruebas.

***El programa soporta los tipos de datos simples del estándar: Number, Boolean, Null y String (cuyo valor va entre comillas). Para aumentar la robustez del programa, también soportamos los tipos Float y Double como tipos posibles.**

2) Breve descripción del flujo del programa

Durante la ejecución del programa se observa el siguiente flujo de los datos. La clase **jsongen** contiene el main y por tanto, será quien corra el flujo principal del programa.

De acuerdo con la cantidad de parámetros que se le envíe iniciará (en el caso de datos coherentes, en cuánto rutas, existencia de archivos, etc.) o bien **ModoConsola;** o bien **ModoGrafico**, a través de la clase **EjecutarModoGrafico.** Donde cada una estas clases dirigirá cada sub flujo de datos del principal.

ModoConsola utiliza **ObtenerNombre** para verificar path, nombre y extensiones de los archivos; pero más importante es aún que utiliza la clase **Controlador** como capa de control entre la UI y la capa de acceso a datos, que es dirigida por la clase **ManejadorJson.**

ManejadorJson maneja la lectura/escritura en disco, así como el parseo en la lectura de los archivos de definición utilizando la clase **ParserJson. ParserJson es** quien consume las clases de la librería Gson (framework de google para parseo de json).

Tanto **ManejadorJson** como **ParserJson** utilizan la clase **RegistroJson** como unidad atómica de dato, la cual se almacena en una estructura dinámica para el transporte entre clases (para el conjunto de registros leídos). Cada objeto RegistroJson representa un atributo del objeto Json, con tipo, nombre y valor.

De igual forma, en el **ModoConsola**, el archivo persistido posee todos los valores de los atributos vacíos, cual si fuera un template del objeto.

ModoGrafico realiza las mismas interacciones que **ModoConsola** contra las capas de Control y accede de igual manera a los datos a través de ella. La diferencia radica en que, con la estructura dinámica recibida (el contenedor de registros de la clase **RegistroJson**) llena el formulario dinámico con los campos traídos del archivo de definición json y permite al usuario su edición.

Luego, el formulario dinámico irá validando los datos ingresados en cada campo a través de **TextfieldTipos** que a su vez usará las distintas clases específicas: **TexfieldNumber**, **TextfieldBoolean**, etc.

De esta manera, una vez completado el formulario con datos validos (estos también pueden tener valores vacíos), es persistido en el archivo json cuyo nombre se toma de la definición, el formulario queda inactivo y tras cerrar la aplicación se finaliza el flujo principal de la misma. El archivo json destino contendrá los valores ingresados por el usuario en el formulario.

Se adjuntará un diagrama de las clases para una mejor visión del circuito mencionado.

3) Etapas de desarrollo y decisiones claves de diseño

En la etapa inicial, durante el análisis del problema, dado el nuevo desafío de tener que implementar una interfaz gráfica a una solución, decidimos que debíamos implementar un modelo basado en tres capas, Una capa que resuelva el acceso a los datos, una capa de Interfaz de Usuario y por último, una capa de control entre las dos primeras capas, que funcione como enlace entre ambas.

Por otro lado, fragmentado el problema en estos 3 sub problemas, decidimos utilizar librerías externas para facilitar la implementación de estas capas, fundamentalmente la de acceso a los datos y la de UI.

Para el acceso a datos, investigando acerca de cómo realizar un parseo que soporte el estándar Json, decidimos utilizar la librería GSON v.2.5, una librería de Google, que por lo que pudimos investigar, contaba con mucho uso a nivel profesional, lo que nos permite, aumentar la confiabilidad al diseño con su uso. Además el soporte de Google en el desarrollo de nuevas facilidades nos permitiría poder reemplazar fácilmente este módulo por nuevas actualizaciones y mejoras, es decir, nos da una mayor flexibilidad al diseño.

También esta se pudo utilizar gracias a la ayuda de Nicolás Paez, ya que nos ayudó a configurar el build.xml para poder utilizarla en el trabajo práctico.

El concepto de la separación de capas y sus clases, nos dan una mayor mantenibilidad al diseño por estar todo bien modularizado y distribuidas claramente las responsabilidades de las clases. A futuro también nos permitiría seguir agregándole funcionalidades al diseño, haciéndolo también extensible.

Por lo tanto, optamos por atacar en principio la capa de acceso a datos, construimos la clase RegistroJson, que utilizamos como unidad de datos compleja. A partir de allí armamos la clase ManejadorJson que inicialmente soportaba tanto lógica de parseo como el control de lectura/escritura de los archivos JSON según se requiriera.

Decidimos que dicha clase ManejadorJSON tenía una responsabilidad que no le correspondía y por lo tanto creamos la clase ParserJson que maneja de aquí en más el parseo de los archivos de entrada y es responsable de completar la estructura dinámica de datos y proveerla a ManejadorJson. También es responsable de utilizar las librerías de GSON. Vimos que este cambio fue muy bueno pues bajó el alto acoplamiento y responsabilidad, mejoró la modularización y simplificó el diseño.

Respecto a la UI decidimos utilizar JavaFX, una librería nativa de Java 1.8 de la cual no teníamos conocimientos previos. Lo tomamos como un gran desafío profesional, un aprendizaje general, pues hasta el momento no habíamos utilizado librerías externas en proyectos Java universitarios. Es decir, asumimos un gran riesgo, en pos de aprender cosas nuevas.

Conscientes que ambas capas necesitaban interfaces claras, terminamos de definir la capa de Control, que precisamente viene a ocupar dicho lugar en el diseño, además de precisamente validar se cumplan las reglas del negocio también. Articular este juego de responsabilidades.

Pero no fue nada sencillo, hicimos TDD durante todo el proyecto hasta encontrarnos con el problema de la interfaz gráfica, dónde no contamos con suficiente información para poder definir una cantidad considerable de pruebas unitarias, además de su complejidad natural. Por lo tanto, decidimos trabajar en un entorno de testing mientras aprendíamos a utilizar la librería JavaFX, hacer muchos test exploratorios hasta dar con la funcionalidad deseada e ir sumando aquellos tests unitarios necesarios.

Pero más allá de este contratiempo propio de avanzar sobre lo desconocido, decidimos que el diseño sea robusto también, en la medida que soporte el ingreso de datos estándar simples, además de sus subtipos. Por ejemplo, surgió a duda sobre si manejar tipos integer, float o number e incluso null. Se decidió soportarlos todos.

Finalmente se logró en el funcionamiento del producto un muy buen rendimiento, además, sobre todo a partir de la interfaz gráfica de dar una usabilidad mucho mayor, mejorando así la experiencia del usuario y por supuesto, nuestra propia satisfacción, por la tarea realizada.

4) Problemas encontrados y soluciones aplicadas

El primer gran problema que se nos presentó fue la decisión de utilizar librerías externas, cosa que nunca antes habíamos hecho, por lo que la única manera de resolverlo fue investigar mucho al respecto, lo que nos llevó a la decisión de utilizar GSON y JavaFX como explicamos en puntos anteriores. Aún así tuvimos que seguir leyendo y trabajarlo mucho para sacarlo adelante.

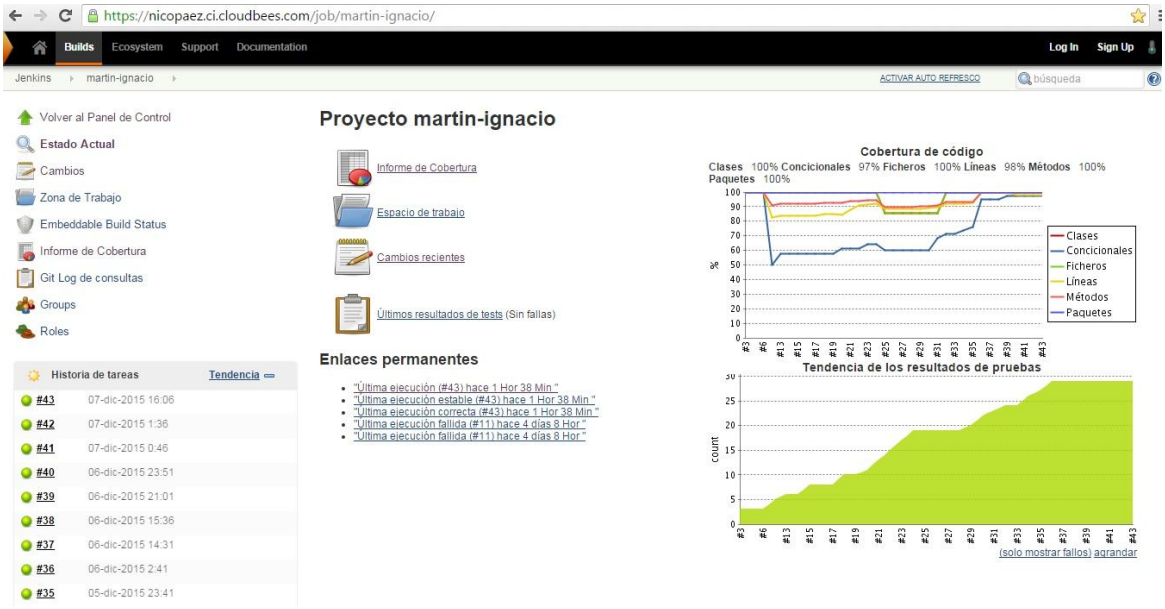
El segundo problema que encontramos fue el integrar dichas librerías al entorno JAVA. Allí surgieron muchas dudas, idas y vueltas entre avanzar con Ant-Ivy o Maven que permitía una mejor integración. Pero según se nos solicitó, dado que con Maven no corría completo Jenkins, tuvimos que volver el proyecto a Ant-Ivy. Esto nos trajo bastantes dolores de cabeza pues la librería GSON fue diseñada específicamente para Maven según el manifiesto de Google.

Aún así, tras muchas pruebas leyendo sobre la forma de trabajo de Ant e Ivy tuvimos inconvenientes, los elevamos a la cátedra y esto nos aportó una nueva visión. Habíamos comprendido mal el funcionamiento de Ant-Ivy. Profundizamos más los temas y pudimos sacarlos adelante.

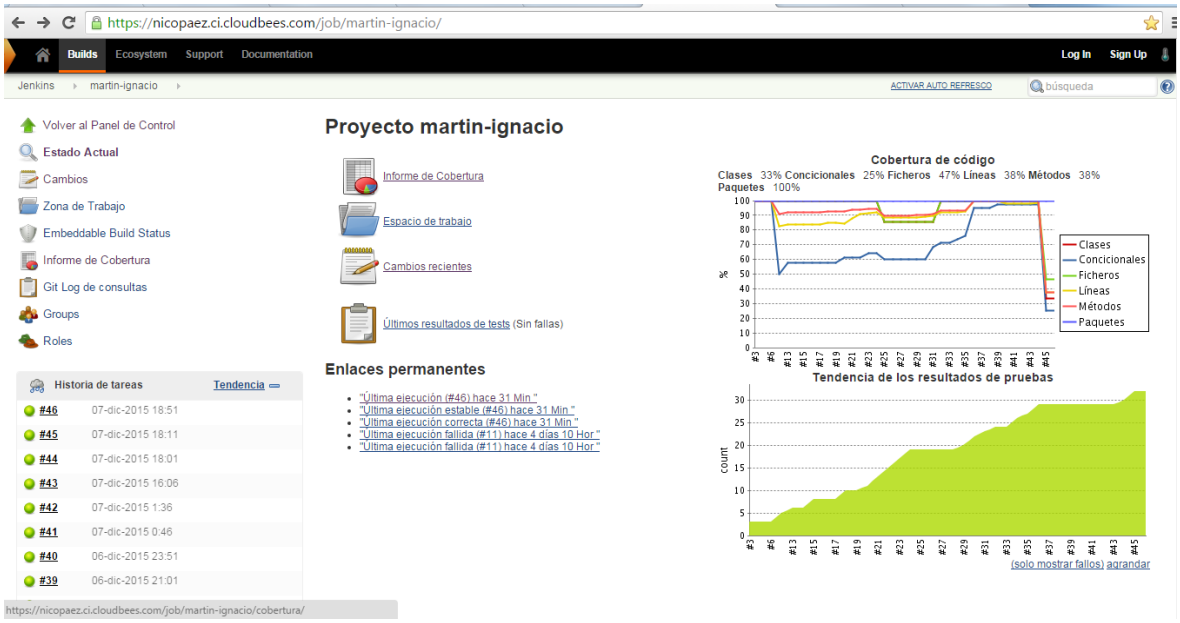
Otro problema surgido fue la versión de Java utilizada. JavaFX utiliza Java 1.8.0.40 en adelante, por lo tanto tuvimos que investigar cómo hacer correr satisfactoriamente los tests en Travis y en Jenkins (actualizar versión Java de Travis). Tarea que tras arduas jornadas pudimos lograr satisfactoriamente.

Otro problema que encontramos con Travis fue la ejecución de los tests para el entorno gráfico, por el uso de threads. Tuvimos que investigar también en la documentación de Travis: https://docs.travis-ci.com/user/gui-and-headless-browsers/ y se modificó el travis.yml para poder correr el test **modoGraficoTest**, tal como requería el link que consultamos.

Tuvimos bastantes problemas para testear el modo gráfico, lo que impactó al subir en el repositorio el código que lo realiza que bajaran notablemente nuestras métricas pasando de casi un 100% de cobertura a un 33%. Adjunto una captura de las métricas de Jenkins previo a subir las clases de la UI:



Luego de subir las clases de la UI Jenkins quedó así:



En definitiva, dichos tests unitarios de la UI fueron reemplazados por pruebas exploratorias en la fase de desarrollo, quedando tal vez como deuda técnica seguir aprendiendo respecto a pruebas unitarias para UI, tarea según lo aprendido, no demasiado sencilla.

Los reportes a nivel local nos dan como resultado una cobertura del 70% de la aplicación, pero hemos notado que en la ejecución de Jenkins lanza la siguiente excepción:

Exception in thread "Thread-1" java.lang.UnsupportedOperationException: Unable to open DISPLAY

Por lo que no puede correr el test de ModoGrafico para corroborar su correcto funcionamiento. El mismo problema hemos notado en servidores Debian sin modo gráfico instalado. En cambio Travis, pudimos configurar travis.yml para que este pueda ejecutarse sin problema agregando las siguientes líneas:

before_install:

- "/sbin/start-stop-daemon --start --quiet --pidfile /tmp/custom_xvfb_99.pid --make-pidfile --background --exec /usr/bin/Xvfb -- :99 -ac -screen 0 1280x1024x16"

5) Atributos de calidad aplicados en el Diseño.

- **Rendimiento:** ambos modos presentan un rendimiento muy positivo durante las pruebas realizadas.

- **Usabilidad**: el modo consola es muy sencillo de utilizar, además se suma el formulario de edición muy bien realizado facilitando la tarea al usuario notablemente.

- **Confiabilidad**: Todas las entidades desarrolladas por nosotros están debidamente probadas y aquellas librerías externas cuentan con el soporte constante de Google (Gson) y Java 1.8 (Oracle).

- **Flexibilidad**: El diseño presentado es ampliamente flexible y adaptable a nuevos requerimientos del usuario.

- **Mantenibilidad**: La corrección de potenciales bugs es muy sencilla, ya que la modularización permite ir atomizando cada vez más las tareas y haciendo que cada punto de falla esté más restringido.

6) Pruebas realizadas a la aplicación.

Para realizar pruebas sobre la aplicación incluimos en el proyecto la carpeta **testFiles** con archivos de definición JSON y también allí pueden encontrarse salidas de las entidades ejecutadas, también en formato json (y sin definición en el nombre del archivo).

Detectamos que en los ejemplos de definición citados en la consigna faltaban los corchetes que engloban al tipo JSON (de acuerdo al estándar), por lo tanto tuvimos que readaptar estos archivos y los mismos funcionan correctamente.

Ejemplo ejecución del modo consola de la aplicación:

java jsongen testFiles/definicion-producto.json producto.json (genera un json de valores vacíos)

definicion-producto.json

```
{"campos":[
        {"nombre":"identificador", "tipo":"String"},
        {"nombre":"detalle", "tipo":"String"}
]}
```

producto.json

```
{
```

"identificador":"",

"detalle":""

}

Ejemplo ejecución del modo gráfico de la aplicación:

java jsongen testFiles/definicion-persona.json (abre formulario, permite editar y guardar cambios)

definicion-persona.json

{"campos":[

{"nombre":"nombre", "tipo":"string"},

{"nombre":"apellido", "tipo":"string"},

{"nombre":"casado", "tipo":"boolean"},

{"nombre":"edad", "tipo":"integer"}

]}

persona.json

{

"nombre":"Juan",

"apellido":"Perez",

"casado":true,

"edad":18

}

7) Conclusiones

Creemos que el trabajo práctico nos planteó un desafío muy grande, pues nos enfrentó con requerimientos habituales del mercado laboral pero que en el ámbito universitario hasta el momento no nos había tocado experimentar.

Amén de esto, el equipo decidió redoblar la apuesta y el esfuerzo por superarnos. Sabíamos nuestras limitaciones y decidimos seguir adelante con el único objetivo de aprender y mejorar como profesionales.

En líneas generales el trabajo nos abrió la cabeza, en conjunto con todo el aprendizaje recibido durante la materia y su correlativa, Ingeniería de Software. Toda esa experiencia profesional sin dudas nos animó a salir adelante, a salir de nuestra zona de confort e ir por nuevas experiencias de desarrollo.

El nivel de exigencia fue alto, acorde al mercado laboral y creo que eso nos hizo poner en foco, con algunos tropiezos durante la cursada, presentación de trabajos y tareas semanales, pero tras haber recorrido todo el camino, todo se pone en perspectiva y podemos decir que somos mejores profesionales.

El ser mejores profesionales impacta directamente en la calidad de nuestros productos. Sin dudas hoy vemos cosas que simplemente antes no veíamos por el sesgo propio del programador, nuestras mentes están mejo preparadas para concebir al producto como un todo integral y no sólo como unas cuantas líneas de código, que pasan tests y que necesita el usuario.

Esto impacta también en nosotros como personas, hemos crecido y nos sentimos más responsables en una industria cada día más amplia y desafiante.

Y para volver a foco de nuestro producto, creemos que pudimos plasmar este crecimiento en el crecimiento del producto, con más atributos de calidad, pero sobre todo, con una mayor experiencia profesional que lo respalda, la nuestra.

Atentamente, el equipo de desarrollo:

Ignacio Luna Echechuri y

Martín Fernández Gamen