Desarrollo de una aplicación web utilizando Desarrollo Guiado por Comportamiento e Integración Continua

Matías Mascazzini, Mgter. Gladys Noemí Dapozo

Licenciatura en Sistemas de Información
Facultad de Ciencias Exactas y Naturales y Agrimensura. Universidad Nacional del Nordeste.
Av. Libertad 5470. Corrientes. Argentina.
matiasmasca@gmail.com, gndapozo@exa.unne.edu.ar

Resumen. En los últimos años se avanzó en los conceptos de desarrollo de software dirigido por comportamiento con el objetivo de superar las ineficiencias y dificultades del desarrollo de aplicaciones. Se busca mejorar la comunicación con el cliente para entregar valor para su negocio, haciendo la cosa correcta de forma precisa, adaptándose a los cambios que puedan surgir en el proceso y definiendo cuándo se da por finalizado el software en función de las pruebas de aceptación de cada historia de usuario. El objetivo de este trabajo es profundizar estos conceptos y aplicarlos en un problema concreto, para evaluar sus ventajas e inconvenientes. Siguiendo una metodología de 4 etapas se aplicaron satisfactoriamente estos conceptos, utilizando además Integración Continua. Aplicando un proceso de software iterativo e incremental se desarrolló una aplicación web, destinada a gestionar premios con votaciones en línea; apoyándose para alcanzar sus objetivos con una serie de herramientas. Se pudo apreciar una mejora en el proceso de desarrollo al contar con una "documentación viva" que refleja fielmente y de forma actualizada lo que hace la aplicación; y una retroalimentación, casi inmediata, surgida de las pruebas automatizadas generando un código más fácil de modificar.

Keywords: Desarrollos Ágiles, Desarrollado Dirigido por Comportamiento, Historias de usuario, Desarrollo Dirigido por Pruebas, Pruebas Unitarias, Pruebas de Aceptación, Automatización de pruebas, Integración Continua.

1 Introducción

Hace tiempo, se empezó a pensar en cómo superar las dificultades e ineficiencias en el desarrollo de software. La dificultad radica en cómo hacer la cosa correcta para los usuarios, que agregue valor a su negocio, y hacerla correctamente, dentro de los plazos y presupuestos previstos; aceptando el cambio natural en los requerimientos y

respondiendo positivamente ante estos cambios. Y además determinar cuándo está terminado el software.

En este marco, surge el agilismo para reducir los problemas clásicos del desarrollo de programas, a la par de dar más valor a las personas que componen el equipo de desarrollo.

Entre estos métodos ágiles, existen algunas prácticas en la que las pruebas pasan a ser una herramienta de diseño del código y, por tanto, se escriben antes que el mismo. De entre estas se destaca TDD (Test-Driven Development*)*, utilizando pruebas unitarias fue la precursora y BDD (Behavour-Driven Development) que propone poner el foco en generar valor para el usuario final utilizando las pruebas de aceptación y redefiniendo el dialogo con los usuarios en la búsqueda de un lenguaje ubicuo (común a todos) para lograr desde el primer momento hacer "la cosa correcta", es decir un software que agregue valor al usuario final. Con el soporte de las pruebas, las subprácticas asociadas y las herramientas, éstas prácticas también logran "hacer la cosa correctamente".

1.1. BDD

Behavour-Driven Development (BDD), en español "Desarrollo Dirigido por Comportamiento", es una práctica de diseño de software que obtiene el producto a partir de expresar las especificaciones en términos de comportamiento, de modo tal de lograr un grado mayor de abstracción. BDD pone un énfasis especial en cuidar los nombres que se utilizan, tanto en las especificaciones como en el código. De esta manera, hablando el lenguaje del cliente, se mantiene alta la abstracción, sin caer en detalles de implementación.

Surgió en respuesta a las críticas que se encontraron en TDD; Dan North empezó a hablar de BDD en un artículo llamado "Behavior Modification" en Marzo de 2006 [1]. Según North, BDD está pensado para hacer estas prácticas ágiles más accesibles y efectivas a los equipos de trabajo que quieren empezar con ellas, de allí que su esquema sea muy similar al de TDD como se aprecia en la Figura 1, agregando una capa de pruebas de comportamiento por encima de las pruebas unitarias. Principalmente propone que en lugar de pensar en términos de pruebas, se debería pensar en términos de especificaciones o comportamiento. De ese modo, se las puede validar más fácilmente con clientes y especialistas de negocio. Poner el foco en el comportamiento logra un grado mayor de abstracción, al escribir las pruebas desde el punto de vista del consumidor y no del productor.

Lo importante es que BDD puso el énfasis en que no eran pruebas de pequeñas porciones de código lo que se creaba, sino especificaciones de requerimientos ejecutables [2]. Un test de cliente o de aceptación con estas prácticas, a nivel de código, es un enlace entre el ejemplo y el código fuente que lo implementa [3].

Se trata de actividades que comienzan desde las necesidades del usuario, para ir deduciendo comportamientos y generando especificaciones ejecutables.

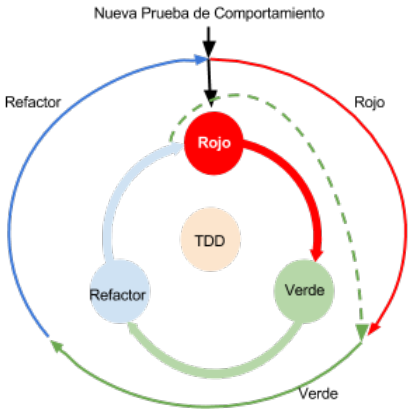

Fig. 1. Esquema BDD. (Fuente: elaboración propia)

Fontela [4] comenta que la crítica principal de este enfoque de BDD y al de ATDD (Acceptance Test-Driven Development) viene de que si bien se pueden derivar pruebas individuales de los contratos, es imposible el camino inverso, ya que miles de pruebas individuales no pueden reemplazar la abstracción de una especificación contractual. La respuesta a esta crítica fue que, si bien las especificaciones abstractas son más generales que las pruebas concretas, estas últimas, precisamente por ser concretas, son más fáciles de comprender y acordar con los analistas de negocio y otros interesados. Y que los ejemplos concretos son fáciles de leer, fáciles de entender y fáciles de validar.

Fontela [4], agrega que al inicio esta técnica se enfocaba en ayudar a los desarrolladores a practicar “un buen TDD” pero el uso de BDD, más las ideas de ATDD, han llevado a una nueva generación de BDD, como práctica y por las herramientas que utiliza. El foco está ahora puesto en una audiencia mayor, que excede a los desarrolladores, e incluye a analistas de negocio, testers, clientes y otros interesados.

1.1.2 Las historias de usuario en BDD

Buscando generar un lenguaje único entre el equipo de desarrollo y el cliente; North propone simplemente estructurar la forma de describir las historias de usuario con el formato Connextra: “Como <rol> deseo <funcionalidad> para lograr <beneficio>”, o alguna variante compatible; más el agregado de las cláusulas Given-When-Then (Dado, Cuando, Entonces) para ayudar a estructurar la explicación de una funcionalidad, en la definición de los escenarios y hacer posible la ejecución de los criterios de aceptación. Este formato captura: el interesado o su rol, el objetivo del interesado para la historia y la tarea a realizar [5]. De esta manera una historia de usuario al inicio tendrá una estructura como la siguiente:

Característica: [Nombre de la historia]

Como un [rol en la aplicación]

Para lograr [para alcanzar algún objetivo concreto]

Deseo [hacer alguna tarea o funcionalidad]

Luego la historia de usuario queda completa con los escenarios y sus ejemplos, que representan los test de aceptación del cliente y determinan cuando una funcionalidad está completa. También se pueden usar "datos tabulados", en los cuales se utilizan ciertas estructuras de tablas para expresar requerimientos y reglas de negocio, ya que las reglas de negocio cambian menos que las interfaces gráficas. Estas tablas posibilitan describir datos de entrada y de salida [6].

La visión de North era tener una herramienta que pueda ser usada por analistas de sistemas y testers para capturar los requerimientos de las historias en un editor de textos y desde ahí generar el esqueleto para las clases y sus comportamientos, todo esto sin salir de su lenguaje del negocio [1].

1.1.3 Utilizando ejemplos como requerimientos

La mayor diferencia entre las metodologías clásicas y la Programación Extrema (XP) es la forma en que se expresan los requerimientos de negocio. En XP [7] en lugar de documentos, se consideran las historias de usuario con sus test de aceptación [3].

Los tests de aceptación o de cliente, de las historias de usuario, son las condiciones escritas de que el software cumple los requerimientos de negocio que el cliente demanda.

El trabajo del analista de negocio se transforma para reemplazar páginas de requerimientos escritos en lenguaje natural, por ejemplos ejecutables surgidos del consenso entre los distintos miembros del equipo, incluido por supuesto el cliente.

En BDD la lista de ejemplos de cada historia, se escribe en una reunión (taller de especificación) que incluye a responsables del producto, desarrolladores, responsables de calidad y otros interesados. Todo el equipo debe entender qué es lo que hay que hacer y por qué, para concretar el modo en que se certifica que el software lo hace. Como no hay una única manera de decidir los criterios de aceptación, los distintos roles del equipo se apoyan entre sí para darles forma.

1.2 prácticas asociadas a BDD

1.2.1 Integración Continua

La integración continua (del inglés "Continuous Integration" o simplemente CI), es una práctica introducida en XP y luego muy difundida a partir del trabajo de Fow-

ler [8]. Consiste en automatizar y realizar, con una frecuencia al menos diaria, las tareas de compilación desde las fuentes, ejecución de las pruebas automatizadas y despliegue de la aplicación, todo en un proceso único, obteniendo información de retroalimentación inmediatamente después de la ejecución de este proceso.

Para que esta práctica sea viable es imprescindible disponer de una batería de test, preferiblemente automatizados, de tal forma, que una vez que el nuevo código está integrado con el resto del sistema se ejecute toda la batería de pruebas [9]. Si son pasadas todas las pruebas, se procede a la siguiente tarea del despliegue de la aplicación. En caso contrario, aunque no falle el nuevo código que se quiere introducir en la aplicación, se debe regresar a la versión anterior, pues ésta ya había pasado la batería de pruebas.

En la actualidad existen muchas herramientas, tanto libres como propietarias, que facilitan esta práctica [4].

Un entorno de CI está compuesto por diferentes herramientas y prácticas en las cuales necesita apoyarse, en la Figura 2 se muestra un esquema de todas las partes funcionando juntas. El desarrollador publica su código en un repositorio, el servidor de CI es informado de un cambio, entonces procede a ejecutar una serie de pasos para realizar el despliegue, entre ellos ejecutar las pruebas antes y después del despliegue, y luego informar al desarrollador el resultado del procedimiento.

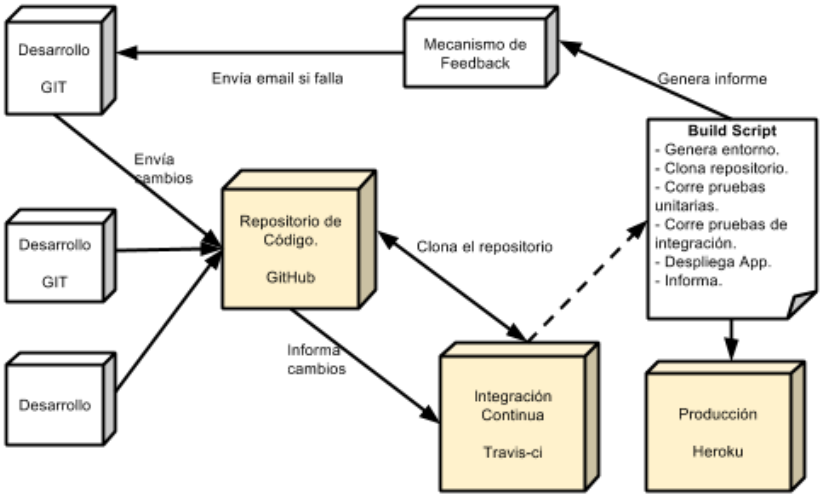

Fig. 2. Escenario de integración continua. (Fuente: elaboración propia)

CI es una práctica que a pesar de no corregir los errores ayuda a la detección de los mismos de manera temprana en el desarrollo de software, haciéndolos más fáciles de encontrar y eliminar. Cuánto más rápido se detecte un error, más fácil será corregirlo.

La relación entre BDD y TDD con la CI viene de la propia definición de la CI, dado que esta requiere pruebas automatizadas previas a la integración.

1.2.2 Refactorización

Es una práctica que busca mejorar la calidad del código, su diseño interno, sin modificar el comportamiento observable (requerimiento). Se aplica sobre el código original o el modificado.

Uno de los objetivos de la refactorización es la mejora del diseño después de la introducción de un cambio, con vistas a que las sucesivas modificaciones hechas a una aplicación no degraden progresivamente su diseño. Se pretende reestructurar el código para eliminar duplicaciones, mejorar su legibilidad, simplificarlo y hacerlo más flexible de forma que se faciliten los posteriores cambios.

El problema con las refactorizaciones es que tienen su riesgo, ya que se está cambiando código que funciona por otro que no se sabe si funcionará igual que antes. Una buena práctica es escribir pruebas automatizadas antes de modificar el código, que verifiquen su funcionamiento y ejecutándolas luego de las modificaciones para ver si continúan pasando. De este modo estas pruebas funcionaran como una red de seguridad ante fallos [10].

Si se quiere facilitar la refactorización, se tendría que contar con pruebas a distintos niveles, en línea con lo que sugieren BDD y ATDD. Las pruebas de más alto nivel no deberían cambiar nunca si no es en correspondencia con cambios de requerimientos. Las pruebas de nivel medio podrían no cambiar, si se trabaja con un buen diseño orientado a objetos, que trabaje contra interfaces y no contra implementaciones. Y en algún punto, probablemente en las pruebas unitarias, se debe admitir cambios a las mismas.

1.1 Objetivo

El objetivo general de este trabajo fue profundizar el estudio sobre las pruebas del software, utilizando el “desarrollo dirigido por comportamiento” o BDD (Behavour-Driven Development). Para ello se realizo una revisión bibliográfica sobre este tema específico y se aplicaron los conceptos y las técnicas en un desarrollo de software concreto, un software multiplataforma para la organización y gestión de premios y certámenes, a partir del cual se evaluaron ventajas y desventajas de su aplicación.

1.2 El problema de aplicación

A través de la consulta a distintas instituciones que organizan eventualmente premios y certámenes, se detectó la necesidad de utilizar algún tipo de software para

gestionar el proceso organizativo de estos eventos, incluyendo sus procesos de votación.

Este trabajo propone la construcción de una aplicación web que satisfaga de forma mínima y viable esas necesidades. En el contexto del TFA (Trabajo Final de Aplicación), el alcance de la aplicación será la organización de premios con votación pública, postergando la administración de certámenes y otras características como líneas futuras de trabajo.

La construcción de esta aplicación web permitirá poner en práctica las técnicas asociadas a BDD, TDD e Integración Continua en un problema real, para poder extraer conclusiones de la experiencia.

1.3 Metodología

La metodología seguida incluyó cuatro etapas: investigación preliminar, concepción de la aplicación, desarrollo de la aplicación y documentación de la experiencia.

Para la construcción de la aplicación web se desarrollaron los primeros módulos siguiendo BDD y TDD, luego se completo el desarrollo de la aplicación sin aplicar los anteriores pero siguiendo las historias de usuarios.

1.3.1 Proceso de software

Para el desarrollo de la aplicación se siguió la metodología ad-hoc ágil con base en la conocida como Programación Extrema adaptada para un trabajo unipersonal, por lo cual no se realizaron algunas de sus prácticas propuestas donde intervienen varias personas de un mismo equipo. Se utilizó esta metodología como base ya que la práctica ágil de TDD fue propuesta en este marco y como BDD puede ser entendida como una ampliación de TDD, ambas encuadran bien en XP.

2 Resultados

Completadas las etapas consignadas en la metodología se seleccionó el framework Ruby on Rails [11] y las prácticas ágiles BDD y TDD junto con herramientas asociadas compatibles con el lenguaje de programación Ruby [12]. También se obtuvieron los siguientes artefactos de software: las historias de usuario y los prototipos de interfaces gráficas de usuario. Luego de la etapa de desarrollo se obtuvo un prototipo funcional para premios (aplicación web), el repositorio público del código, las pruebas automatizadas, la medición automatizada de la cobertura de pruebas, el servidor de integración configurado y automatizado, y la versión de demostración de la aplicación publicada en línea, con datos de prueba. Finalmente el informe del TFA.

2.1 Proceso de Software

Dado que se siguió un proceso de desarrollo de software iterativo e incremental, para el desarrollo de la aplicación se realizaron siete iteraciones. En la tabla 2 se describen los rangos de fechas y la velocidad medida en puntos de historia de cada una de las iteraciones.

Hasta la cuarta iteración se siguió la práctica de BDD de forma pura. Se observa que desde la segunda iteración en adelante, a medida que se fue familiarizando con la metodología y con las tecnologías utilizadas se fue ganando en velocidad. Otro aumento de velocidad se observa a partir de la quinta iteración, debido a la cantidad de historias de usuario completadas siguiendo las prácticas de diseño top-down pero sin poner tanto énfasis en las pruebas primero. Sin embargo esta velocidad está sesgada por el desconocimiento de la tecnología a utilizar, esto provocó errores en la estimación del esfuerzo para las historias de usuario. Esta situación, coincide con Humphrey [13] que advierte sobre la dificultad de hacer estimaciones con proyectos totalmente nuevos que no tienen un punto de comparación previa.

Tabla 1. Iteraciones del proyecto. Fuente: Elaboración propia.

Iteración	Fecha Inicio	Fecha Cierre	Puntos completados o Velocidad
1ra	17 Marzo	14 de Abril	15
2da	14 de Abril	12 de Mayo	6
3ra	12 de Mayo	9 de Junio	12
4ta	9 de Junio	7 de Julio	18
5ta	7 de Julio	4 de Agosto	24
6ta	4 de Agosto	1 de Septiembre	26
7ma	1 de Septiembre	1 de Octubre	40

El proceso de desarrollo fue apoyado por herramientas, necesarias para mantener la flexibilidad ante el cambio y disminuir la deuda técnica. Las principales fueron: PivotalTracker [14], Cucumber [15], RSpec [16], Ruby on Rails [11], Git [17], GitHub [18], Travis-CI [19], Coveralls.io [20] y Heroku [21]. Cabe destacar, que se utilizaron en su mayoría herramientas libres o de código abierto y en todo caso de uso gratuito.

Como parte del proceso, y dado que se siguió un enfoque de afuera hacia adentro (del inglés outside-in), propuesto en [5], se empezó por el bosquejo de las pantallas con wireframes y mockups y la interacción entre ellas. Para continuar luego con el comportamiento esperado descripto a través de las historias de usuario, con sus esce-

narios y ejemplos. En el apéndice 1 se describe la experiencia de su utilización en el TFA.

2.1.2 Historias de usuario

Se utilizaron las historias de usuario para la captura de requerimientos y comunicación con el cliente, utilizando el formato "Connextra" adaptado a papeles de colores de 10x10 cm, que se pegaron sobre un pizarrón. Una vez definidos, y con una primera priorización, se digitalizaron en la herramienta PivotalTracker para su documentación y seguimiento, aunque también se siguió utilizando en papel como una guía visual del trabajo a realizar y realizado.

El detalle de las tareas realizadas y las historias completadas, según su prioridad, se encuentran documentadas en la herramienta PivotalTracker y pueden ser consultadas en línea en https://www.pivotaltracker.com/projects/1025288.

2.1.3 Aplicación de conceptos: BDD y TDD

Dado que aplicar las prácticas de BDD y TDD es parte de los objetivos del trabajo a continuación se comenta sobre su aplicación y luego en el apéndice 2 se da un ejemplo del proceso seguido para aplicar estas técnicas en una historia de usuario en particular.

Se aplicaron los conceptos BDD y TDD en los módulos de administración (back-end) del usuario administrador, es decir a las funcionalidades que realiza el administrador de la aplicación. Se utilizaron las herramientas específicas: Cucumber [15] y Capybara [22] para BDD, Rspec [16] para TDD y Guard [23] para ejecutar localmente todas las pruebas continuamente. Este back-end está compuesto por los siguientes módulos: Procesos de selección, Organización, Usuarios, Categorías, Candidatos.

Para cada uno de ellos se incluyeron las historias de usuario elementales: crear, editar, borrar, listar y mostrar. Con sus "caminos felices" y casos extremos elementales.

2.1.3.1 Aplicando BDD

Las historias de usuario junto con sus escenarios se implementaron con la herramienta Cucumber [15] en archivos "*.feature". Estos están disponibles en la carpeta "features" en [24]. Un ejemplo de historia de usuario en la herramienta se puede apreciar en la Figura 5 del apéndice 2.

De las entrevistas con los clientes se obtuvieron ejemplos que se utilizaron en las historias de usuario para ilustrar comportamiento esperado por la aplicación. Utilizar el lenguaje del cliente ayudó a mantener alta la abstracción durante el proceso de educción, sin caer en detalles de implementación. Por ello los archivos *.feature pueden ser leídos por cualquier persona involucrada en el proyecto y entender de qué se trata esa historia de usuario o incremento funcional.

2.1.3.2 Aplicando TDD

Para aplicar la práctica de TDD se utilizó la herramienta Rspec [16]. Con ella se escribieron las pruebas unitarias para las distintas capas de la aplicación.

Las pruebas se escribieron en archivos de texto plano con extensión "*_spec.rb". Se encuentran disponibles en la carpeta "spec", ubicada en [24], y dentro de ella organizadas en las diferentes sub-carpetas.

En la Figura 10 del apéndice 2 se puede apreciar como lucen las pruebas unitarias con esta herramienta.

2.1.4 Aplicación de integración continua

Para aplicar CI se utilizó el servicio Travis-ci [19]. Se realizaron más de 200 integraciones automatizadas. En el apéndice 3 se ilustra cómo se realizo cada una de ellas.

2.2 Despliegue de la aplicación

El despliegue automatizado de la aplicación se realizó en la plataforma Heroku [21] y actualmente se puede acceder a él desde: http://tfa-vox.herokuapp.com.

El despliegue de la aplicación permitió tener disponible el sistema para los usuarios desde etapas muy tempranas y recibir feedback sobre el comportamiento del mismo junto con pequeñas solicitudes de cambio, que se fueron agregando en las siguientes iteraciones.

En el apéndice 4 se dan detalles de cómo acceder a la demostración en línea de la aplicación.

3 Conclusión

Con respecto a los objetivos fijados en el trabajo:

La revisión bibliográfica realizada detectó, escasa información en español y abundante en inglés sobre la temática abordada, los conceptos y las prácticas asociadas.

Se aplicaron satisfactoriamente los conceptos, métodos y herramientas asociados a BDD y TDD a una aplicación específica, construyendo una aplicación web, multiplataforma, para la organización de premios.

Se pudo apreciar una mejora en el proceso de desarrollo al contar con una "documentación viva" que refleja fielmente y de forma actualizada el comportamiento esperado de la aplicación. Otra mejora fue contar con la retroalimentación, casi inme-

diata, de las pruebas automatizadas para realizar cambios y solucionar problemas generando un código más fácil de modificar.

Además, durante el desarrollo del trabajo, se pudieron comprobar algunas de las ventajas que ofrece BDD y TDD, en concordancia con lo expuesto en [4], estas son:

- Clarifican los criterios de aceptación de cada requerimiento a implementar.
- Generan una especificación ejecutable de lo que el código hace.
- Brindan una serie de pruebas de regresión completa.
- Facilitan la detección y localización de errores.
- Indican cuándo se debe detener la programación, disminuyendo el gold-plating y características innecesarias.
- Facilitan el proceso de incorporar nuevas características y realizar modificaciones que mejoren el código ya existente.
- Una ventaja de BDD, que no tiene TDD, es que se puede incorporar a proyectos ya avanzados o para el mantenimiento de sistemas legados.

Este trabajo pretendió en un primer momento aplicar TDD a un caso real de una aplicación web y terminó encontrando en BDD una herramienta ideal para la comunicación con los clientes, que permitió incluso aplicar la práctica de integración continua.

A continuación se exponen una serie de conclusiones sobre algunas de las temáticas específicas mencionadas en el trabajo:

3.1 Conclusiones específicas.

3.1.1 Sobre las técnicas aplicadas

Si se considera la aplicación de estas prácticas se debe tener presente que TDD, aunque se enfoca en las pruebas, se realiza una actividad de diseño de software y no es un mero proceso de pruebas. Una situación similar se da en el caso de BDD, donde por más que se realicen pruebas automatizadas, su objetivo es la comunicación entre el equipo de desarrollo y los usuarios, mediante las historias de usuario, para entregar funcionalidades que agreguen valor al negocio.

La aplicación de estas prácticas ágiles en el desarrollo de la aplicación propuesta resultó al principio algo complejo debido a la inexperiencia, pero con la práctica y el avance del proyecto se fue simplificando naturalmente. Es recomendable prever el tiempo extra que lleva el proceso de aprendizaje y sus costos económicos asociados.

Se recomienda entonces planificar la realización de “spikes” como indican Blé [3] y Beck [7]. Esto es realizar pequeños experimentos, aparte del proyecto, para indagar y experimentar con la tecnología a utilizar y familiarizarse con ella.

En cambio con BDD resultó más fácil todo el proceso de aprendizaje y aplicación al trabajo, dado que como se tiene en cuenta el comportamiento esperado desde el punto de vista del usuario, generalmente resultaron ser acciones simples a comprobar.

También se presentaron casos donde resultó muy fácil salirse de la técnica, de las pruebas primero, cuando la funcionalidad a agregar parecía pequeña. Sobre todo porque se piensa que se gana en velocidad (producción); lo cual es cierto en el corto plazo, pero cuando haya cambios, más adelante, aparecerán los problemas clásicos del desarrollo de software. Además si se abandona la técnica por un momento y el programador agrega funcionalidades sin hacer antes las pruebas, la denominada “documentación viva” sufre el mismo problema que la documentación tradicional y ya no reflejará el comportamiento esperado de la aplicación.

Otra cuestión a considerar es la aparente sobrecarga mental de tener que idealizar el comportamiento y conceptualizar las pruebas antes que el código, sobre una actividad que de por sí es intelectual intensiva. Se comprobó que existe una sobrecarga, al tener que programar acciones para escribir las pruebas, de forma distinta, a las del código de producción. Por ejemplo para probar la subida de un archivo, el framework Rails proporciona herramientas para hacer de esta tarea sencilla, parametrizando una llamada a una función. Pero las pruebas, desde Cucumber/Capybara y luego desde RSpec, para probar este mismo comportamiento resultaron complicadas y consumieron un tiempo excesivo. Sin embargo, son los beneficios que proporcionan éstas técnicas los que justifican las dificultades y esfuerzos extras. El punto a favor aquí, es que se pone el foco sobre el problema que se está tratando de resolver con los ejemplos concretos a resolver; evitando problemas como la parálisis del analista.

3.1.2 Sobre las historias de usuario

Respecto a las historias de usuario, un aspecto clave es la definición de los escenarios para cada una de ellas. De allí surge la cantidad de esfuerzo a realizar para que la historia sea aceptada. Si bien no es condición sine qua non el conocimiento de conceptos generales de como validar y verificar software, ayudarán al momento de acordar los escenarios con el responsable del producto y otros stakeholder. Para este fin los autores consultados recomiendan la participación de todo el equipo de desarrollo en los talleres de requerimientos.

3.1.3 Sobre las pruebas

Se pudo comprobar que a mayor impacto del cambio, mayor capacidad de adaptación al cambio proveen las pruebas. En aquellas situaciones donde por cambios naturales en los requerimientos, que impactaban internamente en toda la lógica de la aplicación, las pruebas ayudaban a realizar el cambio sin que al terminar el proceso se

generen una cantidad significativa de errores o reclamos por parte de los usuarios dado que el comportamiento esperado seguía siendo el mismo.

Con la existencia de estas herramientas, y su relativa facilidad de integración en el flujo de trabajo, se puede decir que las pruebas creadas por el mismo desarrollador se vuelven una condición exigible en desarrollos profesionales. Por lo tanto su enseñanza formal debería empezar a considerarse también.

3.1.3.1 Sobre las pruebas de comportamiento.

Se pudo apreciar cómo las pruebas de comportamiento con Cucumber ofrecen, de manera simple y precisa, la posibilidad de definir y comprobar el funcionamiento esperado por los usuarios de la aplicación. Además, cuando surgen cambios, por ejemplo en una actualización, en lenguajes como Ruby y herramientas como Rails donde generalmente hay problemas de compatibilidad hacia atrás, esto resulta muy útil para el mantenimiento temprano de la aplicación.

Un problema que se detectó es que las pruebas de comportamiento pueden no detectar errores, es decir, la función evaluada devuelve verdadero, incluso cuando un camino lógico es incorrecto, por esta razón no reemplazan a otro tipo de pruebas.

Generalmente, estas pruebas de comportamiento fallan primero, o más rápidamente, que las unitarias; ya que en la interacción entre partes se pueden dar fallas o problemas que no fueron contemplados a más bajo nivel.

3.1.4 Sobre errores

Una clave al aplicar las técnicas de BDD y TDD es que se tiene que aprender a leer los mensajes de error e identificar de qué tipo son, si es una prueba que naturalmente falla (por la falta del código a evaluar) o si es un error de compilación, de sintaxis o de otro tipo. Luego de aplicar éstas técnicas, durante algunas iteraciones, se vuelve natural leer los errores con más tranquilidad. La facilidad de detectar los errores se incrementa a medida que el desarrollador se acostumbra a la modalidad de trabajo.

Si bien se pudo comprobar que con la utilización de las pruebas de aceptación automatizadas se redujeron, o previnieron, ciertos errores del camino normal de ejecución y los casos extremos contemplados en estas pruebas, lo que fue mejorando la calidad del producto presentado en cada iteración. Ésta mejora no implica la no existencia de fallas en el producto, será tarea de los testers, miembros del equipo realizar otro tipo de pruebas que busquen evidenciar las fallas lo antes posibles en el proceso de desarrollo.

Los errores pueden y estarán presentes aunque se utilicen las prácticas de BDD y TDD porque las pruebas generadas se refieren a los comportamientos esperados, tanto de la aplicación resultante como de sus comportamientos internos. Como generalmente en su concepción no se tuvieron presentes algunos caminos alternativos, casos extremos o incluso factores externos, los errores se harán presentes. La responsabilidad de las prácticas es limitada.

3.1.6 Sobre la integración continua

La experiencia real al aplicar CI en el trabajo demostró que supone un estilo diferente de desarrollar, lo cual implica un cambio de filosofía para el que es necesario un período de adaptación. Si bien no es excluyente se recomienda la utilización de las pruebas automatizadas proveen BDD y TDD.

Además permite detectar posibles problemas durante el despliegue, de manera temprana, que pueden tener soluciones muy complejas a posteriori. Y permite un ahorro significativo de esfuerzo al automatizar tareas repetitivas.

3.1.7 Sobre refactorización

El proceso seguido de las pruebas primero, en TDD y BDD, es como tejer laboriosamente una red de seguridad, nodo por nodo, pero al final queda una red donde el desarrollador se siente muy seguro modificando el código, saltando al vacío de modificar algo que funciona como se espera pero que podría funcionar mejor.

En general se puede afirmar que es mejor refactorizar un código que tiene un comportamiento esperado, a uno que no se sabe si tiene el comportamiento esperado pero que funciona bien. Y esa certeza la dan indefectiblemente las pruebas.

Referencias

1. D. North. (2006). Introducing BDD [Online]. Disponible: http://dannorth.net/introducing-bdd/, Consultado el: May. 8, 2013.
2. H. Lopes Tavares, “A tool stack for implementing Behaviour-Driven Development in Python Language”, Instituto Federal Fluminense (IFF), Campos dos Goytacazes, Brasil. 2010. cap. 1, pp. 1-2.
3. C. Blé Jurado. “Diseño Ágil con TDD”, Primera Edición. Madrid, España: iExpertos, 2010. cap. 1, pp. 33-52-, cap. 3 pp. 67.
4. C. Fontela, “Estado del arte y tendencias en Test-Driven Development”, Facultad de Informática, Universidad Nacional de La Plata, La Plata, Argentina, 2011. cap. 1 y 2, pp. 07-22.
5. D. Patterson y A. Fox, “*Engineering Software as a Service: An Agile Approach Using Cloud Computing”*, Primera Edición. San Francisco, LA, USA: Strawberry Canyon LLC, 2014. cap. 5 pp 355-360.
6. M. Gärtner, “ATDD by Example: A Practical Guide to Acceptance Test-Driven Development”, Primera Edición. Boston, MA, USA: Addison-Wesley Professional, 2012. cap. 9 pp 131-133.
7. K. Beck, “Extreme Programming Explained: Embrace Change”, Segunda Edición. Boston, MA, USA: Addison-Wesley Professional, 1999. cap 7 pp. 30 cap 8 pp. 35.
8. M. Fowler. (2006). Continuous Integration [Online]. Disponible: http://martinfowler.com/articles/continuousIntegration.html, Consultado el: May. 8, 2013.
9. P. Rodríguez González, "Estudio de la aplicación de metodologías ágiles para la evolución de productos software", Facultad de informática Universidad Politécnica de Madrid, Madrid, España, 2008. pag. 16.

10. C. Fontela, “Cobertura entre pruebas a distintos niveles para refactorizaciones más seguras”, Facultad de Informática, Universidad Nacional de La Plata, La Plata, Argentina, 2013. cap. 1 pp. 10.
11. D. Hansson, “Ruby on Rails,” [Online]. Disponible: http://rubyonrails.org/. Consultado el: Sep. 23, 2014.
12. Ruby, “Lenguaje de programación Ruby”. Disponible: https://www.ruby-lang.org/es/. Consultado el: Sep. 23, 2014.
13. W. Humphrey, “Software Estimating” en “PSP: A Self-Improvement Process for Software Engineers”, Primera Edición. Boston, MA, USA: Pearson Education, Inc., Massachusetts, 2005. cap. 5 pp. 84.
14. Pivotal Labs, Inc., San Francisco, CA, USA, “PivotalTracker,” Disponible: http://www.pivotaltracker.com/. Consultado el: Sep. 28, 2014.
15. Cucumber Ltd., Cairndow, Argyll, Scotland, “Cucumber,” Disponible: http://cukes.info/. Consultado el: Sep. 30, 2014.
16. RSpec Development Team, USA, “RSpec,” Disponible: http://rspec.info/. Consultado el: Otc. 01, 2014.
17. Software Freedom Conservancy Inc., NY, USA, “Git”, Disponible: http://git-scm.com/. Consultado el: Sep. 26, 2014.
18. GitHub Inc., San Francisco, CA, USA, “GitHub,” Disponible: https://github.com/. Consultado el: Sep. 26, 2014.
19. Travis CI GmbH, Berlin, Germany, “Travis CI,” Disponible: https://travis-ci.org/. Consultado el: Sep. 29, 2014.
20. Lemur Heavy Industries LLC., Venice, CA, USA, “Coveralls.io,” Disponible: https://coveralls.io/. Consultado el: Sep. 26, 2014.
21. Heroku Inc., San Francisco, CA, USA, “Heroku,” Disponible: https://www.heroku.com/about. Consultado el: Oct. 10, 2014.
22. Elabs AB, Göteborg, Sweden, “Capybara,” Disponible: http://jnicklas.github.io/capybara/. Consultado el: Oct. 01, 2014.
23. T. Guillaume, La Chaux-de-Fonds, Switzerland, “Guard,” Disponible: http://jnicklas.github.io/capybara/. Consultado el: Oct. 01, 2014.
24. M. Mascazzini. (2014). Repositorio público de código del TFA [Online]. Disponible: https://github.com/matiasmasca/vox/tree/master/, Consultado el: Apr. 13, 2015.

Apéndice 1: Prototipos de interfaces gráficas de usuario (GUI)

Luego de obtener una visión general del problema a solucionar, al ser la solución propuesta un sistema web, se procedió a crear los prototipos rápidos de interfaces de usuario, sugeridos en [5]. En la Figura 3 se aprecia el bosquejo para la interfaz de "alta de una entidad organizadora", que luego se digitalizó utilizando LibreOffice Draw (Véase Figura 4).

Para ello se utilizó un pizarrón o papel y lápiz, para crear los bocetos de las interfaces de usuarios en baja fidelidad (los esquemas de página o wireframes y los mockups o maquetas); el uso de éstos elementos permitió analizar las propiedades y navegación de la aplicación de manera rápida y económica.

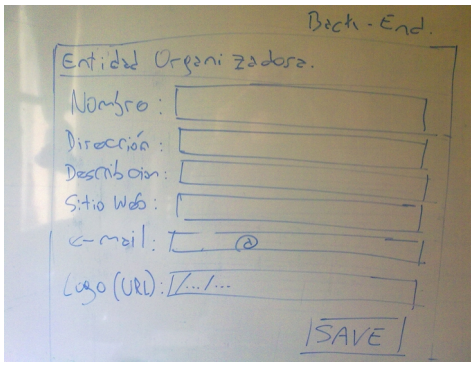

Fig. 3. Mockup en baja fidelidad de formulario (Fuente: elaboración propia)

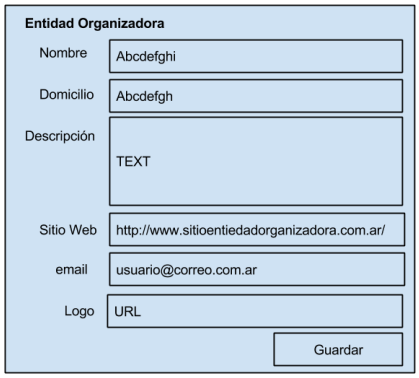

Fig. 4. Wireframe de formulario (Fuente: Elaboración propia)

Apéndice 2: Ejemplo del proceso al aplicar BDD y TDD

A modo de ejemplo a continuación se ilustra el proceso seguido para la historia de usuario "Borrar organización", con el soporte en imágenes capturadas de la pantalla. De manera similar, se siguió este proceso para todas las historias de usuario de los módulos que componen el back-end del perfil de usuario administrador.

Paso 1: Crear archivo feature.

El primer paso consiste en documentar la historia de usuario en un archivo de texto plano con extensión "*.feature"; escrito en formato Connextra, con el agregado de

la descripción de sus escenarios. Primero el denominado “camino feliz” y luego los casos extremos elementales a contemplar por la aplicación.

En la Figura 5 se muestra cómo luce este archivo en un editor de texto. En este caso se estaba usando la etiqueta @wip del inglés “work in progress” para poder ejecutar más fácilmente esta prueba en particular en Cucumber.

```
Característica: borrar una organización
  Con la finalidad de borrar un premio que ya no se utiliza
  como un usuario registrado de una organización
  Quiero poder eliminar un premio en el sistema

#Camino feliz
# Borrar desde listado
@wip
Escenario: borrar premio
  Dado existe una Organización llamada "ACME" con domicilio en "Av. Siempre Viva 742"
  Y que estoy en la pantalla de Administración de Organizaciones
  Cuando hago click en Borrar para "ACME"
  Entonces se borra la Organización "ACME"
```

Fig. 5. Historia de usuario "borrar una organización" (Fuente: elaboración propia)

Paso 2: Crear pasos. Automatizar.

El siguiente paso se basa en traducir cada línea del archivo *.feature en pasos ejecutables por una herramienta, es decir la automatización de estos pasos desde la perspectiva de un usuario. Aquí es donde se escriben y automatizan las pruebas de aceptación.

Para cada archivo “*.feature”, del paso anterior, se automatizó la ejecución de los escenarios en archivos de definición de pasos (step definitions), ubicados en “/features/step_definitions/” en [24], con extensión “*_step.rb”. Cada uno de ellos vincula los pasos de los escenarios de una historia de usuario (archivo *.feature) con un código ejecutable por la herramienta Cucumber [15] y librerías complementarias. En la Figura 6 se puede apreciar uno de estos archivos de definición de pasos.

Lo interesante aquí es la reutilización que se puede hacer de estos pasos, ya que en general un usuario realiza una misma serie de pasos y cambia solo en algunas condiciones particulares. Nótese como de los cuatro pasos del escenario de la Figura 5, solo se tuvo que automatizar uno nuevo, como se aprecia en la Figura 6. El resto de los pasos ya estaban automatizados con anterioridad y estarán en otros archivos. Sin embargo no se debe perder el enfoque de que esta debe ser una herramienta de comunicación con los usuarios.

```
Entonces(/^se borra la Organización "(.*?)"$/) do |item_borrado|
  find("#organizer-list").should have_no_content(item_borrado)
end
```

Fig. 6. Archivo borrar_organizacion_steps.rb (Fuente: elaboración propia)

Paso 3: Ejecutar las pruebas. GUI mínima.

Al ejecutar las pruebas en este punto naturalmente fallan porque no se tiene una GUI básica en la cual ejecutarlas. Siguiendo los mockups se crean las interfaces gráficas mínimas para que la prueba pueda ser ejecutada.

Paso 4: Ejecutar. Prueba falla.

En este punto al ejecutar los pasos de la feature (prueba) fallan naturalmente porque no tienen una implementación asociada, como se ve en la Figura 7, la ejecución marca un fallo en la prueba porque no encuentra un método en el controlador asociado a la acción “destroy”.

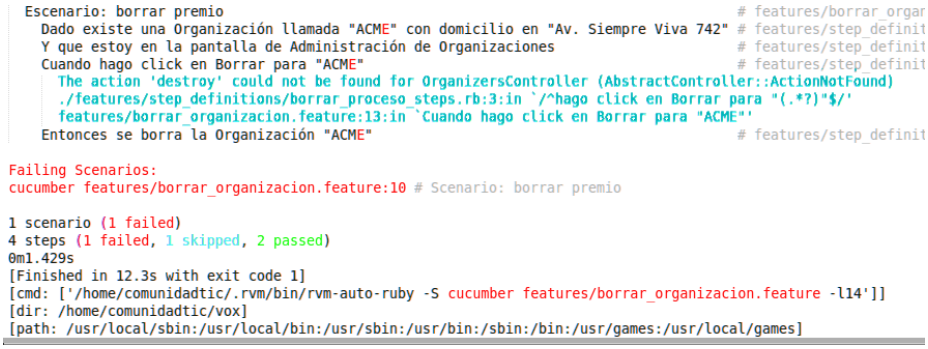

Fig. 7. Prueba falla, rojo (Fuente: elaboración propia)

Paso 5: Pruebas unitarias. Crear spec

Siguiendo la práctica, se necesita crear una implementación que pase la prueba anterior pero antes de ello se tienen que crear las pruebas a nivel unitario. Se crearon los archivos, que interpretará la herramienta Rspec, en la carpeta “/spec”. En este caso se accedió al archivo “organizers_controller_spec.rb” (disponible en el CD que acompaña este informe en /CODIGO/vox/spec/controllers/organizers_controller_spec.rb) y se creó la descripción del comportamiento esperado, como se ve en la Figura 8.

```
describe " DELETE Destroy" do
  it "muestra un mensaje de confirmación"
  it "borra el registro"
  it "redirecciona al index"
end
```

Fig. 8. Spec inicial para acción “destroy” del controlador (Fuente: elaboración propia)

Paso 6: Ejecutar. Prueba pendiente.

Al ejecutar las pruebas del paso anterior, la herramienta RSpec indicará que las pruebas están pendientes de implementación y no se ejecutaron, obteniendo una respuesta como se aprecia en la Figura 9.

```
Pending:
  OrganizersController  DELETE Destroy muestra un mensaje de confirmación
    # Not yet implemented
    # ./spec/controllers/organizers_controller_spec.rb:53
  OrganizersController  DELETE Destroy borra el registro
    # Not yet implemented
    # ./spec/controllers/organizers_controller_spec.rb:54
  OrganizersController  DELETE Destroy redirecciona al index
    # Not yet implemented
    # ./spec/controllers/organizers_controller_spec.rb:55

Finished in 0.28669 seconds
8 examples, 0 failures, 3 pending
```

Fig. 9. Pruebas pendientes (Fuente: elaboración propia)

Paso 7: Escribir pruebas unitarias.

Se escriben las pruebas unitarias que ejecutará la herramienta RSpec. Como se puede apreciar en la Figura 10 para este caso en cada ejecución se define e inicializa un objeto “organizer” que actúa como doble de acción para la prueba, generando un registro con datos de prueba. Luego este registro es borrado como parte de la prueba.

```
describe "DELETE Destroy" do
  it "borra el registro organizer solicitado" do
    organizer = Organizer.create! valid_attributes
    expect {
      delete :destroy, {:id => organizer.to_param}, valid_session
    }.to change(organizer, :count).by(-1)
  end
  it "redirecciona al index de organizer" do
    organizer = Organizer.create! valid_attributes
    delete :destroy, {:id => organizer.to_param}, valid_session
    response.should redirect_to(organizer_url)
  end
end
```

Fig. 10. Prueba unitaria para borrar organización (Fuente: elaboración propia)

Paso 8: Ejecutar pruebas unitarias. Fallan.

Al ejecutar las pruebas del paso anterior estas fallan ya que aún no se ha creado la implementación de ese método en el código de la aplicación en sí misma, situación que se puede apreciar en la Figura 11.

```
Failures:

  1) OrganizersController DELETE Destroy redirecciona al index de organizer
     Failure/Error: delete :destroy, {:id => organizer.to_param}, valid_session
     AbstractController::ActionNotFound:
       The action 'destroy' could not be found for OrganizersController
     # ./spec/controllers/organizers_controller_spec.rb:61:in `block (3 levels)
in <top (required)>'

  2) OrganizersController DELETE Destroy borra el registro organizer solicitado
     Failure/Error: delete :destroy, {:id => organizer.to_param}, valid_session
     AbstractController::ActionNotFound:
       The action 'destroy' could not be found for OrganizersController
     # ./spec/controllers/organizers_controller_spec.rb:56:in `block (4 levels)
in <top (required)>'
     # ./spec/controllers/organizers_controller_spec.rb:55:in `block (3 levels)
in <top (required)>'

Finished in 0.27492 seconds
7 examples, 2 failures
```

Fig. 11. Ejecución fallida borrar organización (Fuente: elaboración propia)

Paso 9: Escribir mínima implementación.

Una vez que se tiene un soporte en pruebas unitarias, se está en condiciones de escribir el código de la aplicación. La práctica indica que primero hay que hacer el mínimo código que pase esa prueba y luego ir refactorizando. En la Figura 12 se puede apreciar el código fuente luego de una primera refactorización.

```
def destroy
  @organizer.destroy
  respond_to do |format|
    format.html { redirect_to selection_process_url, notice: 'Organización borrada correctamente.' }
    format.json { head :no_content }
  end
end
```

Fig. 12. Implementación mínima (Fuente: elaboración propia)

Paso 10: Pruebas pasan.

Al ejecutarse las pruebas, luego de la refactorización, se puede apreciar (Véase Figura 13) como estas continúan pasando sin fallas.

```
.......

Finished in 0.42024 seconds
7 examples, 0 failures

Randomized with seed 40504
```

Fig. 13. Pruebas unitarias pasan (Fuente: elaboración propia)

Paso 11: Prueba exploratoria.

Si bien no es parte de las prácticas BDD o de TDD, al terminar la primera versión se realizaba una prueba exploratoria sobre la interfaz de usuario, siguiendo los escenarios, para comprobar visualmente el funcionamiento y para detectar posibles errores de situaciones no contempladas en los escenarios de la historia. en la Figura 14 se ve la interfaz, luego de ser estilizada con Boostrap, para esta historia.

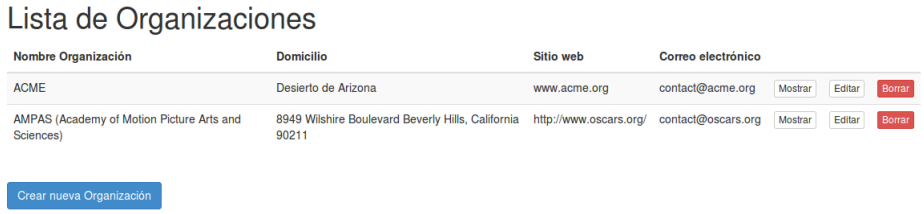

Fig. 14. Lista organizaciones (Fuente: elaboración propia)

Al presionar el botón “eliminar”, de la tabla, se muestra el mensaje de confirmación, como se aprecia en la Figura 15.

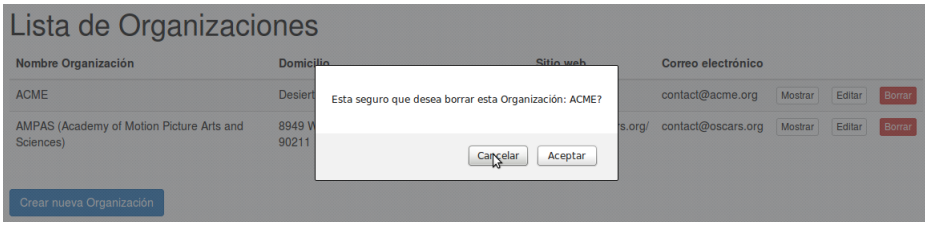

Fig. 15. Mensaje confirmación (Fuente: elaboración propia)

Luego de eliminar el registro, se muestra al usuario otro mensaje de confirmación (Véase Figura 16).

Fig. 16. Mensaje confirmación tras eliminar dato (Fuente: elaboración propia)

Paso 12: Iterar.

Se continúa con la siguiente historia de la iteración.

Apéndice 3: Procedimiento automatizado de Integración.

En cada una de las 200 integraciones se realizó el siguiente procedimiento:

Con cada integración (en inglés build) se instalaron todas las librerías necesarias para ejecutar la aplicación Rails, se generó la base de datos, luego se cargaron los datos de prueba; seguido a esto se ejecutaron las pruebas tanto de interacción como unitarias. Si todos estos procesos se ejecutaron sin errores, el servicio procedió a realizar el despliegue de la aplicación en el servidor de Heroku [21], subiendo los archivos necesarios, ejecutando las instalaciones de librerías necesarias, concluyendo con la configuración de la base de datos con sus datos de iniciales y luego reiniciando la aplicación, para dejarla lista y funcionando. En caso de fallo se suspendió el despliegue y se envió un email informando que el proceso había fallado.

La bitácora de cada una de las integraciones, donde se incluye cada operación realizada, se puede consultar en línea en https://travis-ci.org/matiasmasca/vox.

Apéndice 4: Demostración en línea

Para utilizar la versión de demostración de la aplicación, debe ingresar con un navegador web a la dirección: http://tfa-vox.herokuapp.com/. Luego para identificarse puede utilizar los siguientes datos de acceso, para los tres perfiles usuarios:

Administrador - usuario: donramon@chavo.mx; clave: clave12345

Organizador - usuario: unemail@go.com; clave: clave12345

Jurado – usuarios:

jurado1@go.com; clave: clave12345

jurado2@go.com; clave: clave12345

jurado3@go.com; clave: clave12345