



Universidad Nacional del Nordeste.

Facultad de Ciencias Exactas y Naturales y Agrimensura.

**Trabajo Final de Aplicación: Desarrollo de software multiplataforma
utilizando BDD, TDD e Integración Continua.**

Carrera: Licenciatura en Sistema de Información.

Alumno: Matías Mascazzini.

Profesor Orientador: Mgter. Gladys Noemí Dapozzo.

Profesor Coordinador: Mgter. Sonia Itatí Mariño.

Tribunal Evaluador:

Mettini, Aldo José.

Alfonzo, Pedro Luis.

Mariño, Sonia Itatí.

[Borrador | Presentación Preliminar | Presentación Definitiva]

Borrador v1.52.6 RC Modificación: 995

Año: 2014

Prologo

Haber vivenciado las dificultades clásicas del desarrollo de software en la ejecución de varios proyectos, me motivó para indagar sobre alternativas para la creación de software de calidad que sigan algún modelo comprobado de éxito de la industria. De ello resultó la elección de las prácticas ágiles para dirigir el desarrollo de software a través de pruebas, derivadas de la Programación Extrema (XP). En una primera instancia, con foco en las pruebas unitarias y luego con foco en las pruebas de interacción y aceptación. Finalmente la utilización de estas pruebas permitió llegar al punto de implementar la práctica de Integración Continua.

Una motivación extra fue la de estudiar, aprender y utilizar el lenguaje Ruby (con sus herramientas asociadas), conocido por ser muy utilizado en el ecosistema emprendedor internacional.

Al mismo tiempo, y como una forma de aplicar todo lo descripto anteriormente en un problema real, se detectó la necesidad de una aplicación para la administración de procesos de selección para premios y certámenes en organizaciones del tercer sector; que reemplace los procesos manuales y reduzca la utilización de papel. Esta aplicación debería abarcar desde la creación de un proceso de premiación, la votación anónima a los distintos postulantes y la publicación de resultados. Para suplir la necesidad, y aplicar los conceptos teóricos en un caso específico, se propuso realizar un sistema multiplataforma utilizando tecnologías y herramientas compatibles con las prácticas ágiles seleccionadas y útiles para la resolución del problema.

En el desarrollo de este trabajo colaboraron colegas, como Leandro A. Rodríguez y Simón Y. Ibalo quienes aportaron sugerencias y recomendaciones útiles; y la profesora Mgter. Gladys Noemí Dapozzo quien orientó y motivó para la concreción del mismo. Contó también con el apoyo de comunidades virtuales como: ComunidadTIC, y su grupo NivelR; TDDev; RubySur, en donde se iniciaron hilos de debate relacionados con el trabajo, que ayudaron a tomar algunas decisiones.

Es entonces el resultado de un largo proceso de aprendizaje, de idas y vueltas, de prueba y error; de adaptación al cambio.

Agradecimientos

Quiero dejar asentado el agradecimiento a todas aquellas personas que en mayor o menor medida colaboraron en el desarrollo de este trabajo y a lo largo de la carrera:
A mi familia, amigos, compañeros de estudio, colegas y profesores.

Dedicatoria...

Este trabajo está dedicado a ...

"La calidad es un camino sin retorno"

Resumen sintético

En la búsqueda de formas más eficientes de hacer software, en los últimos años se avanzó en los conceptos de desarrollo dirigido por pruebas, propuestos en la programación extrema. Para hacer la cosa correcta para el usuario, que le agrega valor, en primer lugar e ir haciéndola correctamente, en comparación con los métodos tradicionales que buscan hacer la cosa correctamente en primer lugar. El objetivo de este trabajo es profundizar estos conceptos y aplicarlos en un problema concreto, para evaluar sus ventajas e inconvenientes. Durante su desarrollo se logró aplicar satisfactoriamente estos conceptos, junto con la Integración Continua, siguiendo un proceso de software iterativo e incremental en una aplicación web, destinada a gestionar premios con votaciones en línea; apoyándose para alcanzar sus objetivos con una serie de herramientas: PivotalTracker, Cucumber, RSpec, GitHub, Travis-CI y Heroku. 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; también al contar con la retroalimentación, casi inmediata, de las pruebas automatizadas para realizar cambios y solucionar problemas, generando un código modificable más fácilmente.

{{ 187 palabras de 200 }}

Resumen extendido.

Hace tiempo, se empezó a pensar que el problema del desarrollo de software es hacer la cosa correcta para los usuarios, que agregue valor a su negocio, y hacerla correctamente, dentro de los plazos y presupuestos; aceptando el natural cambio en los requerimientos y respondiendo positivamente ante estos cambios.

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, utilizando pruebas unitarias fue la precursora y BDD 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 prácticas asociadas y las herramientas estas prácticas también logran “hacer la cosa correctamente”.

El objetivo general de este trabajo es profundizar el estudio sobre las pruebas del software utilizando el “desarrollo dirigido por comportamiento” o BDD (*Behaviour-Driven Development*). Para ello se realizará una revisión bibliográfica sobre este tema específico y se aplicarán los conceptos y las técnicas en un desarrollo de software concreto para llevarlos a la práctica y detectar sus ventajas. Así construir un software multiplataforma para la organización y gestión de premios y certámenes.

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

El proceso de software para el desarrollo de la aplicación fue iterativo e incremental, adaptando prácticas propuestas por la programación extrema y las propias del desarrollo dirigido por comportamiento.

Las metodologías ágiles se soportan en herramientas para poder mantener la flexibilidad necesaria ante el cambio y disminuir la deuda técnica. Las principales herramientas utilizadas son: PivotalTracker, Cucumber, RSpec, Git, GitHub, Travis-CI, Coveralls.io, Heroku.

Cabe destacar, que se utilizaron en su mayoría herramientas libres o de código abierto y en todo caso de uso gratuito.

Completadas las etapas consignadas en la metodología se lograron los siguientes resultados y artefactos de software: se seleccionó Ruby on Rails y la prácticas ágiles de

BDD y TDD junto con herramientas asociadas compatibles con Ruby. Se obtuvo la especificación de requerimientos de software y las historias 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; servidor de integración configurado y automatizado; versión de demostración de la aplicación publicada en línea, con datos de prueba. Finalmente el informe del TFA.

Se realizó la revisión bibliográfica, encontrando escasa información en español y abundante en inglés sobre la temática abordada, los conceptos y las prácticas asociadas.

Se aplicaron satisfactoriamente los conceptos teóricos 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 lo que hace la aplicación; también al contar con la retroalimentación, casi inmediata, de las pruebas automatizadas para realizar cambios y solucionar problemas generando un código modificable más fácilmente.

Este trabajo pretendió en un primer momento aplicar TDD a un caso real de un 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.

Se consideran varias líneas para trabajo futuros: como profundizar sobre las prácticas ágiles, analizar el impacto de su uso en las personas, entre otras. Se propone también continuar el desarrollo de la aplicación.

Índice de contenidos

| | |
|---|----|
| Capítulo 1. Introducción..... | 11 |
| 1.1 Estado del arte..... | 11 |
| 1.2 Objetivos del TFA..... | 22 |
| 1.3 Hipótesis..... | 22 |
| 1.4 Fundamentación..... | 23 |
| 1.6. Organización del trabajo..... | 25 |
| Capítulo 2. Metodología..... | 26 |
| 2.1 Metodología..... | 26 |
| Etapa 1: Investigación preliminar..... | 26 |
| Etapa 2: Concepción..... | 26 |
| Etapa 3: Desarrollo..... | 27 |
| Etapa 4: Documentación de la experiencia..... | 28 |
| 2.2 Proceso de software..... | 28 |
| Capítulo 3. Herramientas y/o lenguajes de programación..... | 31 |
| 3.1 Entorno de Desarrollo..... | 31 |
| 3.2 Entorno de Pruebas..... | 38 |
| 3.3 Entorno de Producción..... | 43 |
| Capítulo 4. Resultados..... | 47 |
| 4.1 Síntesis de resultados por etapa..... | 47 |
| 4.2 Proceso de Software..... | 47 |
| 4.4 Aplicando integración continua..... | 57 |
| 4.5. Despliegue de la aplicación..... | 58 |
| 4.6 Descripción del sistema..... | 59 |
| 4.7 Comentarios y discusiones del capítulo..... | 73 |
| Capítulo 5. Conclusiones y futuros trabajos..... | 79 |
| 5.2 Futuros trabajos..... | 83 |
| Referencias..... | 85 |

Índice de figuras

| | |
|--|----|
| Fig. 1: Esquema TDD. Fuente: Elaboración propia..... | 13 |
| Fig. 2: Esquema ATDD. Fuente: Elaboración propia..... | 13 |
| Fig. 3: Esquema BDD. Fuente: Elaboración propia..... | 14 |
| Fig. 4: Story Card en formato Connextra. Fuente: [5]..... | 14 |
| Fig. 5: Escenario de integración continua. Fuente: Elaboración propia..... | 19 |
| Fig. 6: Proceso de software. Fuente: Elaboración propia..... | 29 |
| Fig. 7: Primera planificación en PivotalTracker. Fuente: Elaboración propia..... | 32 |
| Fig. 8: Interfaz después de integrar con Bootstrap. Fuente: Elaboración propia..... | 36 |
| Fig. 9: Interfaz en pantalla de 364x640 píxeles. Fuente: Elaboración propia..... | 36 |
| Fig. 10: Cucumber testing stack. Fuente: [9]..... | 40 |
| Fig. 11: Mockup en baja fidelidad de formulario. Fuente: Elaboración propia..... | 49 |
| Fig. 12: Wireframe de formulario. Fuente: Elaboración propia..... | 49 |
| Fig. 13: Historias de usuario previo a priorizar. Fuente: Elaboración propia..... | 49 |
| Fig. 14: Historias de usuario en PivotalTracker. Fuente: Elaboración propia..... | 50 |
| Fig. 15: Capas de pruebas. Fuente: Elaboración propia..... | 52 |
| Fig. 16: Historia de usuario "borrar una organización". Fuente: Elaboración propia..... | 53 |
| Fig. 17: Archivo borrar_organizacion_steps.rb. Fuente: Elaboración propia..... | 53 |
| Fig. 18: Prueba falla, rojo. Fuente: Elaboración propia..... | 54 |
| Fig. 19: Spec inicial para acción "destroy" del controlador. Fuente: Elaboración propia..... | 54 |
| Fig. 20: Pruebas pendientes. Fuente: Elaboración propia..... | 55 |
| Fig. 21: Prueba unitaria para borrar organización. Fuente: Elaboración propia..... | 55 |
| Fig. 22: Ejecución fallida borrar organización. Fuente: Elaboración propia..... | 56 |
| Fig. 23: Implementación mínima. Fuente: Elaboración propia..... | 56 |
| Fig. 24: Pruebas unitarias pasan. Verde. Fuente: Elaboración propia..... | 56 |
| Fig. 25: Lista organizaciones. Fuente: Elaboración propia..... | 57 |
| Fig. 26: Mensaje confirmación. Fuente: Elaboración propia..... | 57 |
| Fig. 27: Mensaje confirmación tras eliminar dato. Fuente: Elaboración propia..... | 57 |
| Fig. 28: Pantalla principal. Fuente: Elaboración propia..... | 60 |
| Fig. 29: Inicio de sesión. Fuente: Elaboración propia..... | 60 |
| Fig. 30: Formulario creación de usuarios. Fuente: Elaboración propia..... | 60 |
| Fig. 31: Barra de navegación superior. Fuente: Elaboración propia..... | 61 |

| | |
|---|----|
| Fig. 32: Escritorio administrador. Fuente: Elaboración propia..... | 61 |
| Fig. 33: Escritorio usuario organizador. Fuente: Elaboración propia..... | 61 |
| Fig. 34: Escritorio jurado. Fuente: Elaboración propia..... | 61 |
| Fig. 35: Formulario nueva entidad organizadora. Fuente: Elaboración propia..... | 62 |
| Fig. 36: Vista reducida de interfaz CRUD. Fuente: Elaboración propia..... | 62 |
| Fig. 37: Listado compacto de procesos. Vista reducida. Fuente: Elaboración propia..... | 64 |
| Fig. 38: Formulario para crear un proceso de selección. Fuente: Elaboración propia..... | 65 |
| Fig. 39: Formulario para crear una categoría. Fuente: Elaboración propia..... | 65 |
| Fig. 40: Consulta categoría "Mejor idea". Fuente: Elaboración propia..... | 65 |
| Fig. 41: Formulario para crear un candidato. Fuente: Elaboración propia..... | 65 |
| Fig. 42: Lista de candidatos. Fuente: Elaboración propia..... | 66 |
| Fig. 43: Menú lista de categorías. Fuente: Elaboración propia..... | 66 |
| Fig. 44: Lista de electores. Vista reducida. Fuente: Elaboración propia..... | 67 |
| Fig. 45: Ventana modal de confirmación. Fuente: Elaboración propia..... | 68 |
| Fig. 46: Búsqueda elector. Vista reducida. Fuente: Elaboración propia..... | 68 |
| Fig. 47: Inicio votación. Vista reducida. Fuente: Elaboración propia..... | 69 |
| Fig. 48: Candidatos a "Mejor idea" durante votación. Fuente: Elaboración propia..... | 69 |
| Fig. 49: Candidato seleccionado para votar por él. Fuente: Elaboración propia..... | 70 |
| Fig. 50: Mensaje confirmación de emisión de voto. Fuente: Elaboración propia..... | 70 |
| Fig. 51: Mensaje tras voto registrado correctamente. Fuente: Elaboración propia..... | 70 |
| Fig. 52: Categorías deshabilitadas tras votación. Fuente: Elaboración propia..... | 70 |
| Fig. 53: Botones sociales. Fuente: Elaboración propia..... | 70 |
| Fig. 54: Escrutinio. Fuente: Elaboración propia..... | 71 |
| Fig. 55: Resultados categoría. Fuente: Elaboración propia..... | 71 |
| Fig. 56: Datos para auditoría. Fuente: Elaboración propia..... | 72 |
| Fig. 57: Estadísticas básicas de la aplicación. Fuente: Elaboración propia..... | 72 |

Índice de tablas

| | |
|--|----|
| Tabla 1: Ventajas y desventajas de Ruby. Fuente: Elaboración propia..... | 33 |
| Tabla 2: Iteraciones del proyecto. Fuente: Elaboración propia..... | 48 |

Capítulo 1. Introducción.

1.1 Estado del arte.

Desarrollo dirigido por pruebas o *Test-Driven Development* (TDD) surgió como una práctica de diseño de software orientado a objetos, basada en derivar el código de pruebas automatizadas escritas antes del mismo código. Con el correr de los años, se ha ido ampliando su uso. Se ha utilizado para poner el énfasis en hacer pequeñas pruebas de unidad que garanticen la cohesión de las clases, así como en pruebas de integración que aseguren la calidad del diseño y la separación de incumbencias. También, algunos autores han enfatizado su adecuación como herramienta de especificación de requerimientos.

En los últimos años, además se avanzó con los conceptos de TDD hacia las pruebas de interacción a través de interfaces de usuario [1]. Mientras TDD pone foco en las pruebas unitarias, el desarrollo dirigido por pruebas de aceptación o *Acceptance Test-Driven Development* (ATDD) pone su foco en las pruebas de aceptación y el desarrollo dirigido por comportamiento o *Behaviour-Driven Development* (BDD) propone poner el foco en el comportamiento esperado por el usuario y en el modo de escribir las pruebas.

1.1.1 TDD.

En 1999 Kent Beck publica “eXtreme Programmig” (en español programación extrema), una propuesta de desarrollo ágil que, con base en 4 valores y 5 principios claves, proponía la utilización de 20 prácticas para hacer desarrollo de software. De estas prácticas, nace el TDD.

TDD es una práctica iterativa de diseño de software orientado a objetos con base en pruebas automatizadas, presentada como parte de Extreme Programming (XP) en “*Extreme Programming Explained: Embrace Change*” [2]. Luego profundizada en “*Test Driven Development: By Example*” [3] en 2002.

Básicamente, incluye tres sub-prácticas:

- **Pruebas primero (Test-First):** las pruebas se escriben antes del código que se prueba. Esto permite describir un comportamiento, sin limitarse a una implementación particular, minimizando el condicionamiento del autor por lo ya construido. También da más confianza al desarrollador sobre el hecho de que el código que escribe siempre funciona.
- **Pruebas Automatizadas:** las pruebas deben estar codificadas, de manera tal que puedan ser ejecutadas en cualquier momento, sin necesidad de intervención humana. Al finalizar deben indicar si lo probado funciona bien o mal. Esto permite

liberarse del factor humano brindándonos un entorno repetible y controlado. Y del factor económico que implica realizar este tipo de pruebas manualmente.

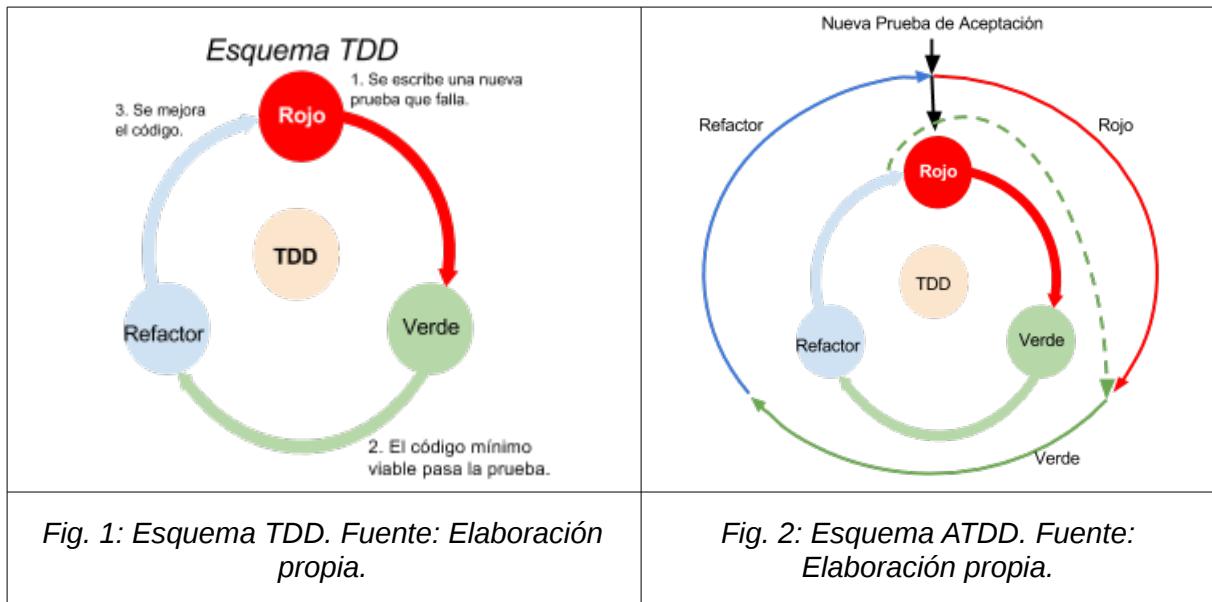
- **Refactorización:** (en inglés *refactoring*) un vez que el código pasa las pruebas; se cambia el diseño del programa sin cambiar la funcionalidad, manteniendo las pruebas que aseguran el comportamiento de esa funcionalidad. La refactorización constante facilita el mantenimiento de un buen diseño a pesar de los cambios, evitando la degradación del software.

Estas prácticas se sintetizan en el algoritmo de TDD (Véase Fig. 1), o como Beck lo llamo “el mantra TDD” [3], en este algoritmo los colores indican los distintos estados:

1. **Rojo:** se escribe una pequeña prueba que no funcionará, y posiblemente al principio no compile.
2. **Verde:** se crea el código que hace que la prueba pase rápidamente, sin importar los errores cometidos.
3. **Refactorizar:** se elimina todas las duplicaciones creadas para que la prueba pase.

Hay una regla básica en TDD que dice: “Nunca escribas nueva funcionalidad sin una prueba automatizada que falle antes” [3]. Si bien no es una regla también se dice que si no se puede escribir una prueba para lo que se está por codificar, entonces no debería estar pensando en codificar. Esto implica que ninguna funcionalidad, por pequeña que sea, debería escribirse por adelantado, si no se escribe antes un conjunto de pruebas que verifiquen su validez. Las pruebas a este nivel deben ser independientes unas de otras, y poder ejecutarse individualmente. Esto genera un desarrollo incremental de la aplicación, en pequeñas funcionalidades específicas descriptas por las pruebas. La segunda regla de TDD es “eliminar la duplicación” esto es no repetir código a lo largo de la aplicación reconociendo comportamientos comunes.

Aceptada en general esta práctica ágil y entendiendo sus beneficios generales, surge un problema: las pruebas, codificadas por el mismo programador, sean éstas de unidad o de integración, ponen el foco en el diseño, y no en lo que se espera de una prueba de cliente, que se focalice en los requerimientos [1]. De allí es que fueron surgiendo propuestas de prácticas alternativas y complementarias, que ponen el énfasis en otros aspectos. En este trabajo se destacan 2 de ellas, a continuación en este capítulo.



1.1.2 ATDD.

Acceptance Test-Driven Development (ATDD), en español “Desarrollo dirigido por pruebas de aceptación”, técnica conocida también como *Story Test-Driven Development* (STDD), es una práctica de diseño de software que obtiene el producto a partir de las pruebas de aceptación. Tuvo su aparición en 2002 [3]. Se basa en la misma noción de las pruebas primero de TDD, pero en vez de escribir pruebas unitarias, que escriben y usan los programadores, se escriben pruebas de aceptación de usuarios, en conjunto con los mismos usuarios [1]. A diferencia del sentido tradicional estas pruebas se ejecutan continuamente y no al final del ciclo.

La idea es tomar cada requerimiento, en la forma de una historia de usuario (en inglés *user story*), construir varias pruebas de aceptación del usuario, y a partir de ellas construir las pruebas automáticas de aceptación, para luego escribir el código.

Lo importante del uso de las historias de usuario es que éstas son requerimientos simples y no representan el cómo, sino solamente quién necesita qué y por qué. Por eso se dice que representan el requerimiento no su especificación [1]. Las pruebas de aceptación de una historia de usuario se convierten en las condiciones de satisfacción del mismo. No importa cómo se implementen (caja negra, blanca, integración, unitarias) lo que importa es la intención de lo que tratan de probar. Se basan en ejemplos y se escriben junto al usuario o especialistas del negocio.

ATDD permite conocer mejor cuándo se ha satisfecho un requerimiento, tanto para el desarrollador como para el cliente: simplemente hay que preguntar si todas las pruebas de aceptación de las historias de usuario están funcionando.

Si bien fue bastante aceptado, con herramientas como RSpec [4], las pruebas se siguen codificando en un lenguaje que sólo los técnicos entienden, con la diferencia que se incluye al cliente en el proceso.

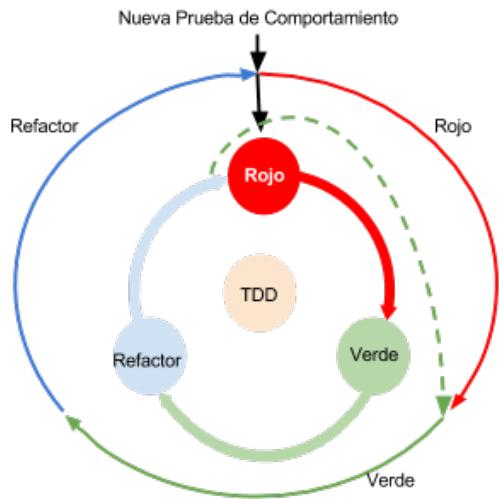


Fig. 3: Esquema BDD. Fuente: Elaboración propia.

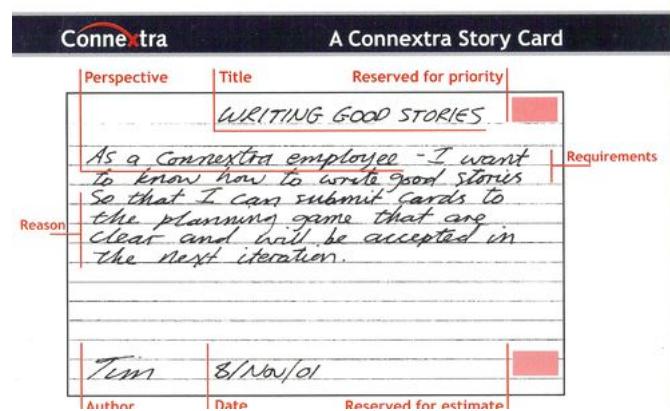


Fig. 4: Story Card en formato Connextra. Fuente: [5]

1.1.2 BDD.

Behaviour-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, y probando la interacción de objetos en escenarios. 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” de la revista Better Software publicado en Marzo de 2006 [6]. Según North, BDD está pensado para hacer estas prácticas ágiles más accesibles y efectivas a los equipos de trabajo que quieren empezar con ellas. 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.

En la filosofía de BDD, las clases y los métodos se deben escribir con el lenguaje del

negocio, haciendo que los nombres de los métodos en las pruebas puedan leerse como oraciones. Los nombres de las pruebas son especialmente útiles cuando éstas fallan y hace énfasis en el uso del verbo “debería” (en inglés *must*).

1.1.2.1 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 [7]. 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]

Así la historia 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 puede 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. [8] 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 de negocios [6].

Fontela [1], comenta 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, tanto como práctica como 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.

1.1.3 Historias de usuario.

El primer paso en un ciclo ágil, que es a menudo el más difícil, es dialogar con cada

uno de los interesados para entender los requerimientos; de este dialogo se derivan las historias de usuario.

Estas son narraciones simples y cortas, en lenguaje natural; en donde cada una describe una interacción específica entre un interesado y la aplicación [7] en el lenguaje del negocio. De forma purista se las puede definir como “un recordatorio de algo relevante que debe hablarse con el usuario” donde lo importante no son las historias en si sino la discusión que se da en torno a ellas. Si bien existen diferentes formas de escribir en lenguaje natural las historias de usuario hubo un formato que ganó popularidad que es el de una compañía llamada Connextra y de allí su nombre, que consiste en utilizar el siguiente patrón: **Como <rol> deseo <funcionalidad> para lograr <beneficio>**.

1.1.3.1 Story Card

Las historias de usuario provienen de la comunidad de interfaz humano-computadora (o sus siglas en inglés *HCI*). Propusieron tarjetas bibliográficas de 3x5 pulgadas, conocidas como “tarjetas 3x5”, y luego como “story card”, en la imagen (Véase Fig. 4) se aprecia en su formato original. Estas tarjetas contienen de una a tres oraciones (que representan la perspectiva, el requerimiento y la razón que agrega valor para el negocio); escritas en el lenguaje no técnico, pero con vocabulario del negocio en cuestión y son escritas en conjunto con los clientes y los desarrolladores. Incluyen además un título para la historia, una fecha y el autor de la misma. La razón principal para su utilización es que estas tarjetas de papel son amigables, descartables y fáciles de reordenar, de esta manera mejoran el intercambio de ideas (brainstorming) y la priorización. En líneas generales las historias de usuario deben ser testeables, lo suficientemente pequeñas para ser implementadas en una iteración y deben tener un valor para el negocio. El acrónimo en inglés SMART (**S**pecific, **M**easurable, **A**chievable, **R**elevant, y **T**imeboxed) captura las características deseables en una buena historia de usuario [7][9].

Por último, para dejar bien en claro las expectativas respecto de la funcionalidad provista por cada *user story*, al dorso de la misma se suelen escribir las condiciones de aceptación de la historia, las cuales funcionan como una especificación para quien tenga que implementarla.

1.1.3.2 Utilizando ejemplos como requerimientos.

La mayor diferencia entre las metodologías clásicas y la Programación Extrema es la forma en que se expresan los requerimientos de negocio. En XP en lugar de documentos, se consideran las historias de usuario con sus test de aceptación [10].

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 única manera de decidir los criterios de aceptación, los distintos roles del equipo se apoyan entre sí para darles forma.

1.1.4 BDD vs. ATDD.

BDD y ATDD son dos prácticas que surgieron prácticamente juntas, y si bien se lanzaron para resolver cuestiones diferentes (BDD surge como mejora de TDD, mientras que ATDD es una ampliación), llegaron a conclusiones similares.

Lo importante es que, tanto BDD como ATDD pusieron el énfasis en que no eran pruebas de pequeñas porciones de código lo que se desarrollaba, sino especificaciones de requerimientos ejecutables [11]. 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 [10].

En los dos casos, se trata de actividades que comienzan desde las necesidades del usuario, para ir deduciendo comportamientos y generando especificaciones ejecutables. Para Fontela [1], las mayores coincidencias entre BDD y ATDD están en sus objetivos generales:

- Mejorar las especificaciones.
- Facilitar el paso de especificaciones a pruebas.
- Mejorar la comunicación entre los distintos perfiles: clientes, usuarios, analistas, desarrolladores y testers.
- Mejorar la visibilidad de la satisfacción de requerimientos y del avance.
- Disminuir el gold-plating, esto es, incorporar funcionalidades o cualidades a un producto, aunque el cliente no lo necesite ni lo haya solicitado.
- Usar un lenguaje único, más cerca del consumidor.
- Focalizar en la comunicación, no en las pruebas.
- Simplificar las refactorizaciones o cambios de diseño.

BDD y ATDD se basan en diseño integral, enfocándose en el chequeo de comportamiento y en el desarrollo *top-down* [11].

Asimismo, el autor comenta que la crítica principal de este enfoque de BDD y ATDD 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 critica 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.

1.1.5 prácticas asociadas a BDD/TDD.

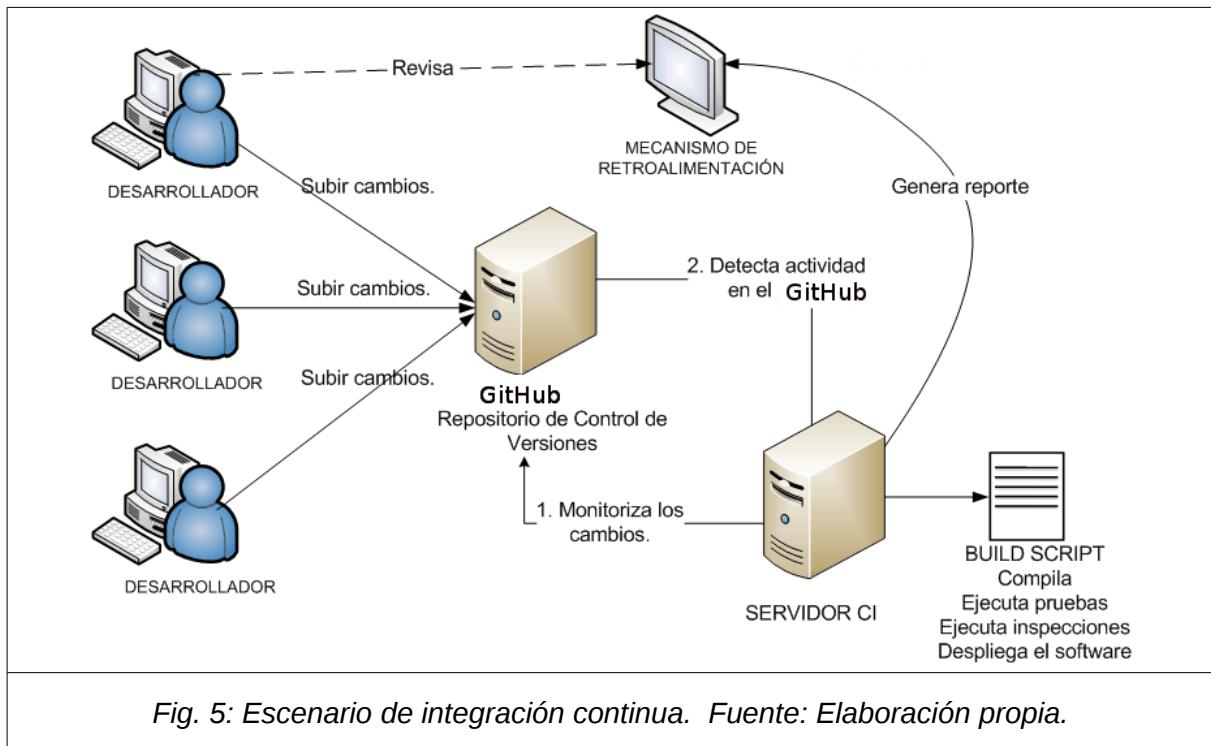
1.1.5.1 Integración Continua.

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 Fowler [12]. Consiste en automatizar y realizar, con una frecuencia al menos diaria, las tareas de compilación, ejecución de las pruebas automatizadas y despliegue de la aplicación, todo en un proceso único, emitiendo reportes ante cada problema encontrado. En la actualidad existen muchas herramientas, tanto libres como propietarias, que facilitan esta práctica.

La integración continua pretende mitigar la complejidad que el proceso de integración suele tener en las metodologías convencionales, superando, en determinadas circunstancias, la complejidad propia de la codificación. El objetivo es integrar cada cambio introducido, de tal forma que pequeñas piezas de código sean integradas de forma continua, ya que se parte de la base de que cuanto más se demore para integrar más costoso e impredecible se vuelve el proceso. Así, el sistema puede llegar a estar integrado y construido varias veces en un mismo día.

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. 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 módulo que se quiere introducir en la aplicación, se ha de regresar a la versión anterior, pues ésta ya había pasado la batería de pruebas. En concreto la integración continua consiste en disponer de un sistema automatizado, que construya el software desde las fuentes y lo valide ejecutando pruebas, más de una vez al día, obteniendo información de retroalimentación inmediatamente después de la ejecución de este proceso.

Un entorno de Integración Continua está compuesto por diferentes herramientas y prácticas en las cuales necesita apoyarse, en la imagen (Véase Fig. 5) se muestra un esquema de todas las partes funcionando juntas. El desarrollador publica su código en un repositorio, el servidor de integración continua detecta un cambio, entonces procede luego 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.



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. Cuanto más rápido se detecte un error, más fácil será corregirlo.

Actualmente, para una mejora de la calidad en la programación, se suelen realizar prácticas de CI que se basan en procesos de desarrollo continuo controlados permanentemente mediante pruebas del código.

Para implementar CI se deben llevar a cabo las siguientes sub-prácticas:

- Mantener un repositorio de código.
- Automatizar la compilación e integración.
- El programador hace sus propias pruebas y las agrega al directorio correspondiente.
- Hacer *commits* diarios, es decir aceptar los cambios diariamente.

- Ejecución automática y periódica de las pruebas añadidas.
- Administrar los resultados y enviar informes a todas las partes implicadas.

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

1.1.5.2 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.

Se debe recordar que el código que funciona es el principal aporte de valor para las metodologías ágiles, por encima de la documentación, por lo que se enfatiza en que éste sea legible y permita la comunicación entre desarrolladores. Esto da la oportunidad para revisar la aplicación de principios de diseño y reglas de estilo propias del equipo de desarrollo. Por ejemplo en el caso de una clase, se puede modificar la implementación de sus métodos sin modificar el acceso a ellos.

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.

Comenta Fontela en [13], que luego de que surgiera el término *refactoring*, en 1990, la práctica se difundió escasamente hasta su aplicación en el marco de Extreme Programming (XP) en [2]. A partir de allí se popularizó y comenzaron a surgir catálogos de problemas (“bad smells”), de refactorizaciones clásicas, de refactorizaciones hacia patrones, de refactorizaciones de modelos, y hasta de refactorizaciones de pruebas automatizadas. También los entornos de desarrollo incluyeron herramientas para favorecer ciertas refactorizaciones de catálogo.

De este modo, la refactorización aparece relacionada con BDD/TDD en dos sentidos:

- El ciclo de TDD incluye una refactorización, luego de hacer que las pruebas corran

exitosamente, para lograr un mejor diseño que el que pudo haber surgido de una primera implementación cuyo único fin es que las pruebas pasen.

- Para que la refactorización sea segura exige la existencia de pruebas automáticas escritas con anterioridad, que permitan verificar que el comportamiento externo del código refactorizado sigue siendo el mismo que antes de comenzar el proceso.

La relación entre ambas prácticas es, por lo dicho, de realimentación positiva. Así como la refactorización precisa de TDD como red de contención, TDD se apoya en la refactorización para eludir diseños triviales. Así es como a veces ambas prácticas se citan en conjunto, y se confunden sus ventajas. Por ejemplo, cuando se menciona que TDD ayuda a reducir la complejidad del diseño conforme pasa el tiempo, en realidad debería adjudicarse este mérito a la refactorización que acompaña a TDD. Sin embargo la refactorización no necesita forzosamente del cumplimiento estricto del protocolo de TDD: sólo que haya pruebas (en lo posible automatizadas), y que éstas hayan sido escritas antes de refactorizar.

Ahora bien, cuando se habla de mejorar el diseño sin cambiar el comportamiento observable, cuanto mayor sea el nivel de abstracción de las pruebas, cuanto más cerca están las pruebas de ser requerimientos, más fácil va a ser introducir un cambio de diseño sin necesidad de cambiar las pruebas.

En este sentido, 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.

1.1.5.3 Interfaces gráficas de usuario en baja fidelidad.

Tomado de la comunidad Interacción Humano-Computadora, los bocetos de baja fidelidad son una manera muy económica de explorar las interfaces de una historia de usuario. Hacerlas con lápiz y papel hace que ellas sean modificables o descartables, lo que permite a las personas involucradas participar, sobre todo a los usuarios no técnicos, de una manera más activa y no intimidante. Los guiones gráficos (en inglés *Storyboard*) capturan la interacción entre las diferentes páginas dependiendo de lo que hace el usuario. Es muy económico experimentar de esta manera, antes de pasar a los archivos HTML y CSS para crear las páginas [7]. Los wireframes son una guía visual o esquema que representa la estructura visual del sitio, su diseño y contenido [14].

Lo que se pretende con los bosquejos de las interfaces de usuario, es que sean el equivalente a las tarjetas de las historias de usuario, que sean atractivos para los interesados no técnicos y alentarlos a la prueba y error, ya que es más fácil de probar y descartar bosquejos. En [7] se recomienda utilizar los mismos elementos que en un jardín de infantes para hacer los bosquejos: papel, crayones, tijeras. Los autores llaman a esta aproximación “*Lo-Fi UI*” y a los prototipos en papel “*sketches*”. Idealmente, se hacen estos bosquejos para todas las historias de usuario que involucren una interfaz de usuario. Esto puede ser tedioso, pero eventualmente se tendrán que especificar los detalles de la interfaz de usuario cuando se creen los archivos HTML y se convierta en realidad esa interfaz. Por esto sigue siendo más fácil de cambiar en lápiz y papel que en el código.

1.2 Objetivos del TFA.

El objetivo general de este Trabajo Final de Aplicación es profundizar el estudio de los temas vinculados con la calidad del software, en particular, sobre las pruebas del software utilizando el “desarrollo dirigido por comportamiento” o BDD (*Behaviour-Driven Development*). Para ello se realizará una revisión bibliográfica sobre este tema específico y se aplicarán los conceptos y las técnicas en un desarrollo de software concreto para llevarlos a la práctica.

1.2.1 Objetivos Específicos:

- Realizar una revisión bibliográfica sobre el estado del arte, los conceptos y las técnicas que conforman el desarrollo dirigido por comportamiento (BDD) y prácticas asociadas.
- Detectar las ventajas de este enfoque, de forma empírica, mediante la aplicación de sus conceptos y técnicas en el desarrollo de una aplicación informática concreta.
- Construir un software multiplataforma para la organización y gestión de premios y certámenes; integrando los conocimientos sobre BDD para la resolución un problema concreto.

1.3 Hipótesis

Es posible adaptar y aplicar las prácticas ágiles de BDD, TDD e Integración Continua para el desarrollo de una aplicación web, que permita mejorar el proceso de desarrollo y la calidad del producto resultante.

1.4 Fundamentación

A medida que la complejidad del software fue creciendo los proyectos tuvieron mayores dificultades para cumplir plazos. Esto se llamó “La Crisis del Software”, que dio origen a la

Ingeniería del Software (IS) que pretende implementar métodos y herramientas para generar software con mayor calidad.

Desde esta perspectiva, con respecto a las pruebas, se dice "*Probar un programa es la forma más común de comprobar que satisface su especificación y hace lo que el cliente quiere que haga. Sin embargo, las pruebas sólo son una de las técnicas de verificación y validación*". Se considera a esta temática muy importante ya que la misma trata de buscar que el *producto final* sea consistente respecto del diseño inicial, además busca que llegue a manos del usuario un producto estable, carente de aquellos defectos y errores que pudieron ser detectados y corregidos antes de su distribución [15]. Sin embargo, generalmente consideran a las pruebas como una tarea que se realiza al final del proceso de desarrollo, sea este iterativo o incremental.

Según Blé [10], para proyectos que durarán años el enfoque clásico de la IS puede ser válido pero la vida de los productos es cada vez más corta y una vez en el mercado, son novedad apenas unos meses, quedando fuera de él enseguida. Las empresas se deben adaptar a este cambio constante y basar su sistema de producción en la capacidad de ofrecer novedades de forma permanente. Lo cierto es que ni los productos de software se pueden definir por completo a priori, ni son totalmente predecibles, ni son inmutables.

El problema es entonces hacer la cosa correcta para los usuarios, que agregue valor a su negocio, y hacerla correctamente (aseguramiento de la calidad), dentro de los plazos y presupuestos; aceptando el natural cambio en los requerimientos y respondiendo positivamente ante estos cambios.

En este marco, surge el agilismo como posible solución. La finalidad de los distintos métodos que componen este enfoque es reducir los problemas clásicos del desarrollo de programas, a la par de dar más valor a las personas que componen el equipo de desarrollo. Estos métodos deben permitir a los equipos desarrollar rápidamente software de calidad capaz de responder, en forma ágil y eficaz, a las necesidades de cambios que puedan surgir a lo largo de los proyectos. Su esencia es entonces la habilidad para adaptarse a los cambios.

Entre estos métodos ágiles, existen algunas prácticas en la que las pruebas pasan a ser una herramienta de diseño del código (TDD, ATDD, BDD) y, por tanto, se escriben antes que el mismo. TDD habla de que la arquitectura se forja a base de iterar y refactorizar, en lugar de diseñarla completamente de antemano. La aplicación se ensambla y se despliega en entornos de pre-producción a diario, de forma automatizada. Las baterías de tests se ejecutan varias veces al día [10]. Sin desmedro de lo anterior, fueron surgiendo alternativas

de mejoras, como BDD que propone poner el foco en generar valor para el usuario final, entonces siguiendo el principio de “las pruebas primero” se agrega una capa de pruebas de aceptación por encima de las pruebas de integración y de las unitarias para lograr hacer “la cosa correcta”, es decir un software que agregue valor al usuario final.

BDD ha ganado gran aceptación por ofrecer un camino en el cual el software funciona como se espera, y ha sido adoptado por la mayoría de los métodos de desarrollo ágiles; ya en 2010 se mencionaba esta situación en [11].

Desde su origen, BDD tiene el objetivo de integrar la verificación y la validación al tiempo que reduce los costos de garantía de la calidad del software. Utilizando la técnica de diseño con estilo de afuera hacia adentro (*outside-in*), que sigue el enfoque sistémico de diseño conocido como *top-down*, es decir empezando por la parte del software que ven los usuarios descendiendo hasta las unidades básicas. Se concentra de forma directa en la conexión entre los requerimientos de software y el artefacto que los implementa: el código.

Como BDD se basa fuertemente en la automatización de las tareas de especificación y pruebas, es necesario tener las herramientas adecuadas para soportarlo. Las herramientas son necesarias para conectar el texto de los requerimientos con el código, de forma de facilitar la escritura de las pruebas y el proceso en general.

Estas prácticas ponen el foco en “hacer la cosa correcta” desde el primer momento y con el soporte de las pruebas, prácticas asociadas y herramientas para lograr “hacer la cosa correctamente”.

1.5 El problema de aplicación.

A través de la consulta a distintas instituciones, locales y nacionales, 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. Actualmente estas organizaciones realizan toda la gestión del evento usando diferentes herramientas ofimáticas, sobre todo planillas de cálculo, y en algunos casos volcando manualmente los resultados a un sistema gestor de contenidos lo que les genera un esfuerzo extra. No se encontró en el mercado un software que pueda satisfacer estas necesidades organizativas.

Este trabajo propone la construcción de una aplicación web multiplataforma que satisfaga de forma mínima y viable esas necesidades. Permitirá a diferentes instituciones y organizaciones del ámbito civil y comercial: organizar y gestionar premios y certámenes a través de Internet, con votaciones privadas y públicas, sin necesidad de conocimiento

técnico.

En el contexto del TFA, el alcance de la aplicación será la organización de premios con votación pública, postergando la administración de certámenes y otras características como trabajo futuro.

La construcción de esta aplicación web permitirá aplicar la técnica BDD/TDD en un problema real, para poder extraer conclusiones de la experiencia.

1.6. Organización del trabajo.

El presente trabajo se encuentra organizado en capítulos, de la siguiente manera:

En el **capítulo 1** se introducen los conceptos de TDD, ATDD, BDD, Integración Continua, Refactorización e Historias de Usuario. Además se exponen los objetivos principales del trabajo y se enuncia la hipótesis subyacente.

En el **capítulo 2** se explica las metodologías aplicadas en el trabajo, dentro de las cuales se abordan las metodologías ágiles de desarrollo y BDD.

En el **capítulo 3** se describen las herramientas utilizadas en el desarrollo de la aplicación, haciendo una breve reseña, marcando alguna de sus principales características y su utilización en el trabajo.

En el **capítulo 4** se describe el desarrollo de la aplicación propuesta.

Al final en el **capítulo 5** se comentan las conclusiones y se proponen algunas posibles líneas de trabajo futuras.

Capítulo 2. Metodología

En este capítulo se describe, por un lado la metodología seguida para cumplir los objetivos del presente trabajo y, por otro, la metodología para el desarrollo de la aplicación propuesta.

2.1 Metodología.

La metodología seguida consistió en 4 etapas, con sus respectivos pasos y productos (o artefactos), que se describen a continuación:

Etapa 1: Investigación preliminar.

Esta etapa se podría considerar como un estudio de pre-factibilidad; en ella se buscó obtener un conocimiento amplio de los conceptos involucrados y un panorama de las herramientas tecnológicas a utilizar.

Para cumplir con los objetivos de esta etapa, se siguieron los siguientes pasos:

Paso 1.1. Investigación documental exploratoria.

Se realizó una investigación documental exploratoria en libros de texto de la disciplina, repositorios científicos y académicos, a fin de configurar un estado del arte de los conceptos involucrados en la temática que se aborda.

Paso 1.2. Relevamiento y evaluación de las herramientas tecnológicas.

Se realizó un relevamiento y evaluación de las herramientas tecnológicas que soportan las prácticas propuestas por el BDD/TDD, a fin de seleccionar aquellas que mejor se adecuen a los objetivos del trabajo. Siguiendo lo propuesto por [10] (.NET o Python con xUnit) y por [7] (Ruby, Rails, Cucumber y RSpec), optando finalmente por lo segundo.

Etapa 2: Concepción.

En esta etapa se puso el foco en la aplicación a desarrollar con el objeto de aplicar los conceptos teóricos y utilizar las herramientas asociadas. A fin de conceptualizar esta aplicación se tuvo contacto con los posibles usuarios. Se identificó el alcance inicial del proyecto, se identificaron las entidades externas (personas y sistemas) que interactuarán con la aplicación. Esta información permitió valorar la aportación de la aplicación hacia las organizaciones.

Para cumplir con los objetivos de esta etapa, se siguieron los siguientes pasos:

Paso 2.1. Determinación y comprensión del dominio de la aplicación.

Se contactó a organizadores de premios y certámenes, indagando cómo se organizan actualmente, con el objetivo de detectar falencias y necesidades comunes.

Paso 2.2. Relevamiento de antecedentes.

Se realizaron búsquedas intensivas en Internet para buscar antecedentes, combinando diferentes palabras claves a medida que se comprendía mejor el dominio del problema.

Paso 2.3. Realización del primer taller de especificación (specification workshop).

Consistió en una entrevista con un posible futuro cliente donde se definieron las historias de usuario (Véase Fig. 13) que luego se priorizaron para su implementación posterior y algunos escenarios básicos.

Etapa 3: Desarrollo.

En esta etapa se especificó, diseñó, desarrolló y validó, una aplicación web orientada a la gestión de premios y concursos con votación electrónica en línea; mediante un proceso ágil iterativo e incremental en una serie de iteraciones.

El objetivo fue la construcción del producto mínimo viable o prototipo funcional que incluyó algunos requerimientos funcionales y no funcionales de software.

Para cumplir con los objetivos de esta etapa, se siguieron los siguientes pasos:

Paso 3.1. Configuración del ecosistema de trabajo.

Se instalaron las herramientas necesarias para trabajar con la tecnología seleccionada (Ruby, Rails) y se creó el repositorio de Código en GitHub, que se vinculó con Travis-CI y Heroku. Creando las cuentas de usuario básicas para el proyecto en cada uno de estos servicios externos.

Paso 3.2. Desarrollo del Back-end.

Se realizaron 4 iteraciones del proceso de software en las que se implementaron las historias de usuario correspondientes al módulo de administración (back-end) siguiendo estrictamente BDD/TDD. Se aplicaron conceptos tales como escenarios, pruebas de aceptación, ejecución de pruebas, automatización de pruebas, dobles de pruebas, entre otros. Durante este proceso se fueron tomando notas que luego se volcaron en el presente trabajo.

Paso 3.3. Desarrollo del Front-end.

Se realizaron 3 iteraciones, en las cuales se implementaron las historias de usuario para desarrollar los módulos del organizador del premio, para la creación de las categorías y sus candidatos. También el módulo para los jurados junto con la registración de los votos. Siguiendo un diseño de afuera hacia adentro, sin énfasis en las pruebas primero. Iterando hasta completar el prototipo funcional mínimo y viable.

Etapa 4: Documentación de la experiencia.

Se confeccionó el informe del Trabajo Final de Aplicación, siguiendo la estructura propuesta en su reglamento.

2.2 Proceso de software.

Para el desarrollo de la aplicación se siguió la metodología ágil 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 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.

2.2.1 Proceso iterativo general.

Para cada iteración (en inglés *sprint*) se realizó el mismo proceso de software, que se ilustra en la imagen (Véase Fig. 6), y se describe a continuación:

- A. Taller de Especificación.
- B. Entrevista con el cliente, donde se obtuvieron:
 - 1. Historias de usuario.
 - 2. Prototipo de GUI en baja fidelidad (*Lo-fi UI mockup*).
 - 2.1. Prototipos rápidos de las páginas (*wireframes, mockup*).
 - 2.2. Guiones gráficos (*storyboard*) de la navegación entre páginas.
- C. Taller de planificación.
 - 1. Priorización de historias de usuario.
 - 2. Selección de historias de usuarios a realizarse en la iteración o *Iteration Plan*.
 - 3. Definición de tareas asociadas a las historias seleccionadas.
 - 4. Documentación en herramienta PivotalTracker.
- D. Desarrollo de la iteración.
- E. Demostración y lanzamiento de avance (en inglés *small release*).

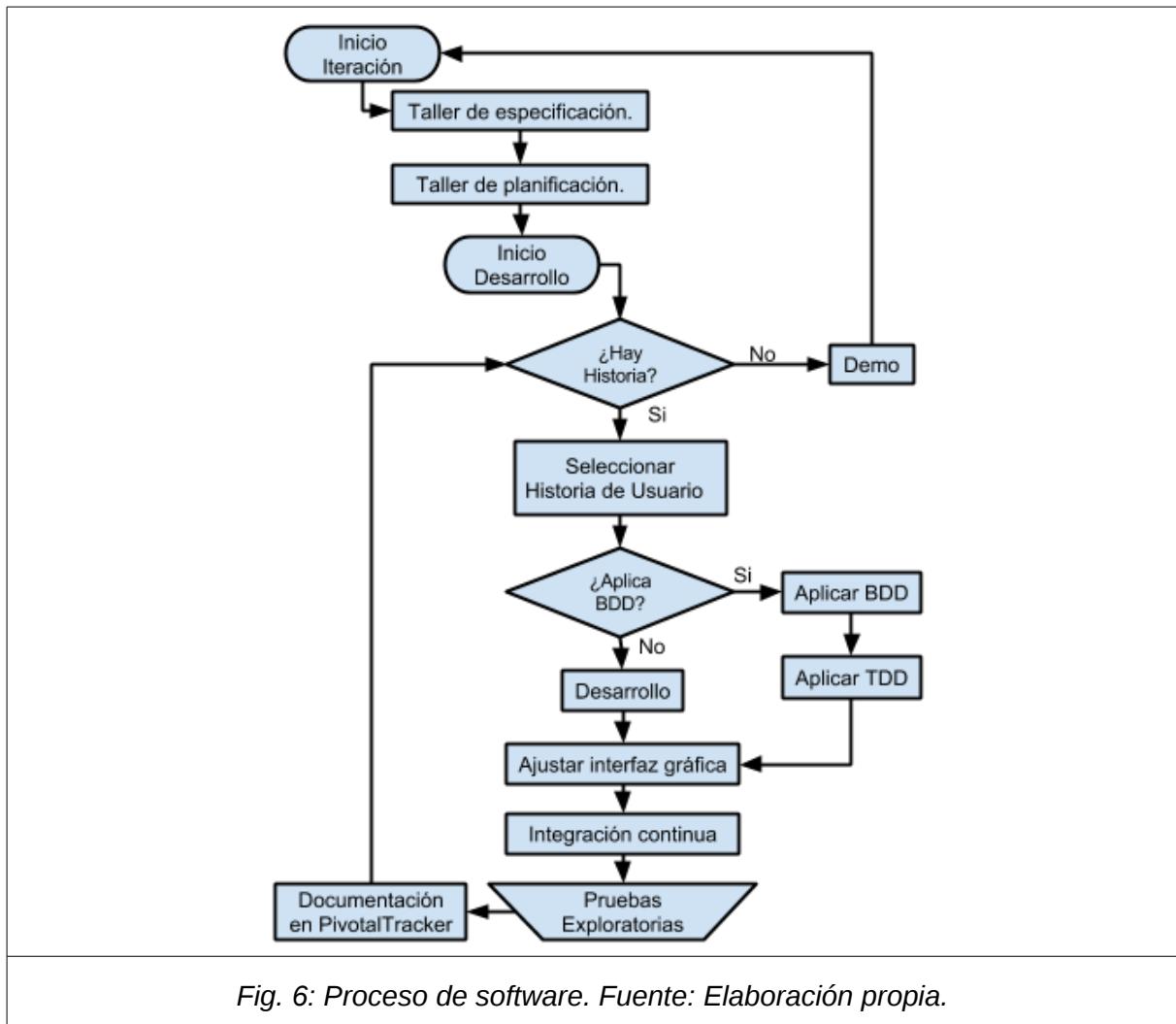


Fig. 6: Proceso de software. Fuente: Elaboración propia.

Del diálogo con los interesados primero se derivaron las historias de usuario. Como es una aplicación web que involucra a usuarios finales, se diseñaron las interfaces de gráficas de usuario (GUI). Primero como dibujos de las páginas web, de baja fidelidad, que luego se combinaron con los *storyboards*, para analizar la navegación entre ellas, antes de crear estas interfaces en HTML.

2.2.2 Subproceso para cada historias de usuario.

Para la implementación de cada historia de usuario de la iteración se realizó un proceso de 5 pasos, variando en la intensidad de aplicar “las pruebas primero”. En las actividades del paso A1 se aplico estrictamente BDD y TDD, luego en las actividades del paso A2 se omitió la confección de las pruebas primero:

Paso A1. Actividades desarrolladas con BDD.

1. Creación de escenarios para la historia de usuario seleccionada.
2. Obtención de ejemplos, definición de camino normal y camino alternativo.

3. Automatización de pruebas de aceptación e integración.
4. Aplicación de TDD con RSpec.
5. Creación de interfaz de usuario básica.

Paso A2. Actividades sin aplicación de BDD.

1. Selección de la historia de usuario.
2. Diálogo con usuario sobre posibles escenarios.
3. Creación interfaz de usuario básica.
4. Desarrollo.

Paso B. Refinamiento de interfaz de usuario.

Paso C. Integración Continua (Travis-CI, Heroku).

Paso D. Pruebas exploratorias y correcciones.

Paso E. Documentación de avance en PivotalTracker.

En A1 se utilizó Cucumber para convertir las narraciones informales en pruebas de aceptación e integración.

Capítulo 3. Herramientas y/o lenguajes de programación

Las herramientas utilizadas se clasificarán según su uso en 3 tipos de entornos de trabajo: Desarrollo, Pruebas y Producción. En el título de cada una se sigue el patrón Nombre – función de la herramienta en el proyecto.

3.1 Entorno de Desarrollo.

A continuación se describen las herramientas utilizadas en el entorno de desarrollo (este incluye actividades de análisis, diseño, programación) las que se utilizaron siguiendo un diseño de afuera hacia adentro.

3.1.1 Pizarrón y fibra - herramientas para bosquejar.

La utilización del pizarrón “ecológico” y fibra al agua, permitió un ahorro significativo en el uso de papel. Sirvió para mejorar la comunicación con el cliente, ya que al ser un borrador que ambos, analista y cliente, podían ver y compartir al mismo tiempo, y a sabiendas, que esa información sería borrada, permitió un mejor entendimiento y facilitó la creatividad.

3.1.2 PivotalTracker - administración del proyecto y de las historias de usuario.

PivotalTracker (<http://www.pivotaltracker.com/>) [16], es una herramienta para la administración ágil de proyectos, que permite la colaboración de los miembros de un equipo en tiempo real mediante un pizarra compartida, donde se priorizan las tareas. Una de sus características es que ayuda a conocer fácilmente la velocidad del equipo; utilizando para ello el nivel de dificultad de cada historia y el tiempo que se demoró en completarla.

Se lo utilizó para la documentación de las historias de usuario y para la administración del proyecto, si bien esta herramienta está orientada al marco de trabajo Scrum, resultó compatible con el trabajo realizado; en ella se definió la pila de producto (en inglés *backlog*) y se priorizaron las historias de usuario, resultado de los talleres de especificación. Esta herramienta sirvió para responder fácilmente la pregunta “¿Y ahora qué sigue?” cuando se está trabajando en una iteración. Además permitió documentar las tareas realizadas en las historias más importantes y en algunos casos adjuntar los *wireframes* o *mockups* de las GUI.

En la imagen (véase Fig. 7) se pueden apreciar las diferentes pizarras, que proporciona automáticamente la herramienta, con las historias de usuario priorizadas:

- *Current*: para las historias de la iteración actual.
- *Backlog*: pila de producto, son las historias, asignadas a diferentes iteraciones según su dificultad estimada. Representan las funcionalidades que debe tener la aplicación

resultado.

- *Icebox*: son historias de usuario que aún no se han asignado a ninguna iteración.
- *EPICS*: allí figuran las historias de usuario “épicas”, estas son aquella que están compuestas por varias historias de usuario, agrupadas para generar una característica.

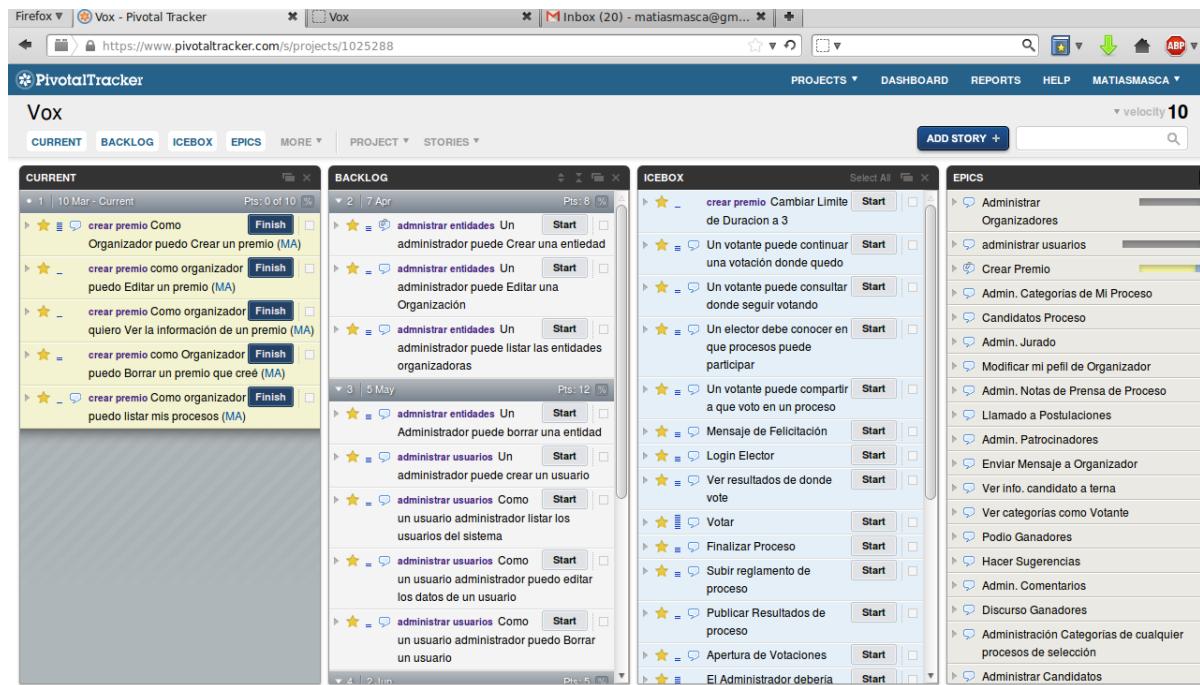


Fig. 7: Primera planificación en PivotalTracker. Fuente: Elaboración propia.

3.1.3 Ruby - lenguaje de programación.

Se utilizó el lenguaje de programación Ruby, en su versión 1.9.3, para codificar la aplicación web desarrollada. Ya que Ruby es un lenguaje de programación orientado a objetos, creado como un lenguaje amigable al programador su sintaxis está enfocada en incrementar la productividad de las personas y no en la eficiencia de la máquina, esta última busca realizarla de manera transparente para el programador. Según sus creadores Ruby es “Un lenguaje de programación dinámico y de código abierto enfocado en la simplicidad y productividad. Su elegante sintaxis se siente natural al leerla y fácil al escribirla.” [17]. Este lenguaje se adapta perfectamente a la práctica de TDD, incluyendo pruebas unitarias de forma nativa. En la siguiente tabla (Véase Tabla 1) se mencionan algunas ventajas y desventajas del lenguaje:

Tabla 1: Ventajas y desventajas de Ruby. Fuente: Elaboración propia.

| Ventajas | Desventajas |
|---|---|
| Es fácilmente integrable (en módulos) | Débilmente tipado. Tipos dinámicos pueden traer problemas inesperados. |
| Ofrece flexibilidad (se lo puede modificar, quitando partes o redefiniéndolas) | No soporta polimorfismo de funciones (sobrecarga) |
| Ofrece simplicidad (hacer más con menos) | Se puede modificar fácilmente las Librerías Estándar de Ruby |
| Es cercano al lenguaje natural | Hilos de Ruby son “green threads” programados y ejecutados por una máquina virtual. |
| Posee una amplia librería estándar. | Problemas de compatibilidad hacia atrás |
| Productividad | Ruby más lento que lenguajes compilados |
| Permite desarrollar soluciones a bajo Costo. | No tiene soporte completo de Unicode, pero sí de UTF-8. |
| Es software libre. El intérprete y las bibliotecas están licenciados bajo las licencias libres y de código abierto GPL y Licencia pública Ruby. | |
| Es multiplataforma. | |
| Soporta pruebas unitarias. | |

3.1.5 Rails - framework para desarrollo web. {{ ver como achicar }}

Ruby on Rails (rubyonrails.org) [18], en adelante Rails, es un framework o entorno de programación escrito en Ruby para el desarrollo de aplicaciones Web; se lo utilizó en su versión 4.0.1. Las razones para elegir Rails sobre otros frameworks, fueron:

1. Proporciona un stack completo de tecnologías Web (todo en uno)
2. Solidez y madurez. Lo usan empresas como Twitter, Groupon, GitHub.
3. Pensado en la productividad (enfatiza convención sobre configuración).
4. Su comunidad es grande y su licencia es libre.

Según sus creadores “Ruby on Rails es un entorno de desarrollo web de código abierto que está optimizado para la satisfacción de los programadores y para la productividad sostenible. Permite escribir un buen código evitando repeticiones y favoreciendo la convención antes que la configuración.”[18]. Rails, implementa 3 patrones de diseño: “Model-View-Cotroller” o MVC para la aplicación como un todo, “Active Record” para los modelos, basados en una base de datos relacional formando la capa de persistencia. Y

“Template View” para la construcción de las páginas HTML. Además para soportar la arquitectura MVC se basa en 3 módulos: ActiveRecord para crear los modelos, ActionView para crear las vistas y ActionController para crear los controladores. Además Rails incorpora otros módulos no mencionados aquí.

Rails enfatiza en la aplicación de dos principios de desarrollo: i) No te repitas (en inglés *Don't Repeat your self* o *DRY*); ii) Convención sobre configuración (en inglés *Convention Over Configuration* o *CoC*).

i) **No te repitas**, significa que la información se debe colocar en un único e inequívoco lugar. En Rails, por ejemplo en las vistas HTML se utilizó el concepto de “partials” para reutilizar vistas compartidas. Las validaciones de datos, se hacen solo en los Modelos siempre antes de cada operación y los filtros ayudan a este fin en los controladores. Seguir este principio ayudo a escribir menos líneas de código y a facilitar las modificaciones.

ii) **Convención sobre configuración**, es un paradigma de diseño que automatiza algunas configuraciones basado en el nombre de las estructuras de datos y variables. Esto significa que el framework Rails hará una serie de suposiciones basadas en la convención y usos normales sobre el nombre de algunas variables y las estructuras de datos asociadas. De esta manera es que el desarrollador debería preocuparse por los casos anormales; para esto el framework hace uso de la reflexión y la meta-programación de Ruby. Para ser conciso, no repetitivo y aumentar la productividad, Rails hace un uso extensivo de la meta-programación (permite a los programas modificarse a sí mismos en estructura o comportamiento mientras se ejecutan) y reflexión (es la habilidad de un programa de examinar los tipos y las propiedades de sus objetos mientras se ejecuta) de Ruby, según explican en [7].

Rails también define una estructura particular de archivos y carpetas para organizar el código de nuestra aplicación junto con sus archivos asociados, la que por su extensión no se describe.

3.1.5.1 Gemas.

Se utilizó Rails con el agregado de algunas librerías externas, llamadas gemas, las cuales figuran en el archivo */CODIGO/vox/Gemfile*, del CD que acompaña este informe. De las cuales se destaca:

- **Devise.** Se utilizó esta gema por sus mecanismos, estándar de facto, para la registración y autenticación de usuarios. La misma incluye unas Vista, que fueron adaptadas a esta aplicación.

- **Chartkick.** Se utilizó esta gema para implementar los gráficos de Columna y Torta, que se utilizaron para mostrar una estadística básica y para mostrar el escrutinio.
- **Bootstrap-sass.** Se utilizó esta gema para integrar Boostrap 3 a Rails, utilizando el pre-procesador CSS Sass.
- **BetterErrors.** Se utilizó esta gema para remplazar los mensajes de error de Rails, en el entorno de Desarrollo, para facilitar la depuración de errores.

3.1.6 WEBrick - servidor web de desarrollo y prueba.

Se utilizó este servidor web en el entorno de desarrollo, ya que es parte de la librería estándar de Ruby y provee un servidor web HTTP básico y HTTPS con autenticación, entre otras características. Pero no debe ser utilizado en entornos de producción ya que no fue diseñado para soportar alta concurrencia de usuarios, es mono-hilo y monoproceso. Sin embargo este ayudo a que rápidamente se pueda probar los resultados del desarrollo, si la necesidad de instalar un servidor web adicional. Se puede consultar más información sobre WEBrick en [19].

3.1.7 Bootstrap - framework front-end.

Se utilizó el framework front-end Boostrap [20], en su versión 3, para la implementación de las vistas HTML de la aplicación. Bootstrap, es un conjunto de herramientas para el diseño interfaces de usuario (front-end) web de código libre; es sencillo y ligero, puede bastar con un fichero CSS (bootstrap.css) y uno JavaScript (bootstrap.js). Está basado en los últimos estándares de desarrollo para la web: HTML5, CSS3 y JavaScript/Jquery. Incluye todos los elementos necesarios para una interfaz de usuario web (iconos, botones, listas, tablas, mensajes, áreas de texto, etc.). Se basa en un sistema de columnas, que por defecto permite un ancho de 940px o 1200px cuando es responsive, según lo consultado en [21]. Se lo utilizó pre-procesado con Sass (*Syntactically Awesome Stylesheets*) con la utilización de la gema **Bootstrap-sass**. En la imagen (Véase Fig. 8) se puede apreciar su aplicación en la interfaz de back-end para editar una entidad organizadora de procesos de selección en la sistema.

Fig. 8: Interfaz después de integrar con Bootstrap.
Fuente: Elaboración propia.

Fig. 9: Interfaz en pantalla de 364x640 píxeles.
Fuente: Elaboración propia.

Además la utilización de este framework permitió obtener un diseño responsivo, que posibilita que el contenido de la interfaz se adapte a casi cualquier tamaño de pantalla dinámicamente, como se aprecia en la imagen (Véase Fig. 9), de esto resulta la posibilidad de usar la aplicación desde computadoras con diferentes tamaños de pantalla o desde dispositivos móviles que tengan un navegador web incorporado. Al tiempo que se obtuvo un diseño agradable, con aspecto profesional, que se mantuvo homogéneo en toda la aplicación, con muy poco esfuerzo y conocimiento técnico.

3.1.8 SQLite - base de datos en desarrollo y prueba.

SQLite (sqlite.org) [22] es un motor de base de datos minimalista, contenido en una librería (de aproximadamente 500Kb programada en C) que se integra con las aplicaciones; donde la base de datos se almacena en un simple archivo, alojado junto con la aplicación que lo utiliza [23]. Cabe aclarar que SQLite es un proyecto de código abierto de dominio público.

En Ruby y en Rails, su utilización se hace mediante una gema, en este caso la gema **sqlite3**. La utilización de SQLite permitió tener una base de datos liviana para los entornos de Desarrollo y Prueba, sin la necesidad de instalar un servidor de bases de datos. Al ser compatible con el estándar SQL, en este caso no hizo falta hacer adaptaciones al momento de pasar a la base de datos de producción, donde sí se utiliza un servidor de base de datos como se menciona más adelante en este capítulo.

3.1.9 MySQL Workbench v6.0 - modelador de datos.

MySQL Worckbench [24] es una herramienta visual unificada, para arquitectos de base de

datos, desarrolladores y administradores de base de datos. MySQL Workbench provee modelado de datos, desarrollo SQL y herramientas para configuración del servidor, administración de usuarios, copia de respaldo y más. Es una aplicación de código abierto. Se utilizó esta herramienta visual para el modelado de datos, en su versión 6, para diseñar el modelo lógico inicial de la base de datos que utiliza la aplicación. Y para la documentación de la estructura final de las tablas.

3.1.10 Git - control de versiones de archivos y código.

Git es un sistema distribuido de control de versiones, libre y de código abierto, diseñado para manejar todo tipo de archivos, con rapidez y eficiencia. Fue creado para colaborar en el mantenimiento del núcleo del sistema operativo Linux y reemplazar a un sistema llamado BitKeeper. Según Chacon en [25] “*Git es tremadamente rápido, muy eficiente con grandes proyectos, y tiene un increíble sistema de ramificación (branching) para desarrollo no lineal.*”. Por las cualidades citadas se utilizó Git para hacer el versionado local de archivos del proyecto, en la máquina de desarrollo y luego se lo vinculó con el servicio GitHub, que se describe a continuación.

3.1.11 GitHub - control de versiones de archivos y código.

GitHub [26] es un servicio web para alojar repositorios de código, con el agregado de funcionalidades para la administración de código; compatible con Git y Subversion. GitHub es un servicio que gira en torno a los usuarios; esto significa que, el acceso a los proyectos se hace por nombre de usuario, por ejemplo para este trabajo: <https://github.com/matiasmasca/vox> siendo matiasmasca el usuario. No existe una versión auténtica de ningún proyecto, lo que permite a cualquiera de ellos pueda ser movido fácilmente de un usuario a otro en el caso de que el primer autor lo abandone. Si bien GitHub es una compañía comercial, que cobra por las cuentas que tienen repositorios privados para albergar proyectos públicos de código abierto, cualquiera puede crear una cuenta gratuita [25]. Para el presente trabajo se vinculó el repositorio local Git con una cuenta de GitHub [26], para tener acceso remoto al mismo, publicar el código fuente y para desde allí hacer el despliegue (*deploy*) de la aplicación en el servicio Heroku, que se describe más adelante. Se utilizó este servicio en línea para evitar tener que montar un servidor Git, tarea que escapa a los objetivos del trabajo.

Cabe destacar que además se vinculó el repositorio de GitHub con:

- **Coveralls.io** (<https://coveralls.io/>)[27] es un servicio para medir la cobertura de código, se describe más adelante.

- **Travis-CI** (<https://travis-ci.org/>)[28] es un servicio de integración continua y despliegue, para realizar la práctica ágil de “integración continua”. Se describe más adelante junto con las herramientas de despliegue en producción.

Estas integraciones con servicios de terceros fueron posibles gracias a que GitHub implementa el concepto de “WebHook”, que envía mensajes HTTP Post ante cambios en el repositorio a los servicios vinculados.

A modo de comentario, para el archivo “readme” del repositorio, se tuvo que usar un tipo especial de formato de archivos llamado Markdown, lo que permite que al consultar el repositorio desde un navegador, se encuentre un texto de descripción con cierto formato básico.

3.1.12 Google Docs y Libre Office – ofimática.

Para la redacción de los documentos relacionados con el proyecto se utilizó Docs de la suite ofimática en línea Google Docs, que es una versión en línea de LibreOffice; además se utilizó Google Drawings para digitalizar las interfaces en baja fidelidad y realizar alguna de las figuras utilizadas en el informe. Al ser un servicio en la nube permitió tener disponibilidad en todo momento y actuó de resguardo (backup) de la información.

Luego para la redacción de este informe se utilizó Writer, en su versión 4.2., un procesador de texto muy completo parte del paquete ofimático LibreOffice que además de software libre permitió la exportación del documento a varios formatos y trabajar con herramientas como las referencias cruzadas.

3.1.13 Mozilla FireFox. - navegador web.

Es un navegador web de código abierto, creado y mantenido por la fundación Mozilla. Se lo utilizó, en su versión 32, para hacer las búsquedas bibliográficas, descargar los programas utilizados y su documentación; como así también durante el desarrollo y para hacer las pruebas exploratorias.

3.2 Entorno de Pruebas

A continuación se describen las herramientas utilizadas en el entorno de pruebas, el mismo se utilizaba en paralelo con la programación.

3.2.1 Cucumber – documentación de historias de usuario y escenarios.

Cucumber (www.cukes.info) [29] es una herramienta de comunicación y colaboración pensada para soportar el proceso de BDD (behavior-driven design); que ejecuta descripciones funcionales en texto plano como pruebas automatizadas. Estas pruebas

interactúan directamente con el código de la aplicación, pero están escritas en un medio y un lenguaje que cualquier persona puede entender. La idea general es que tanto el equipo técnico como los involucrados no técnicos trabajen juntos para escribir estos textos y lograr un lenguaje común; según se explica en [30]. El lenguaje que entiende Cucumber se llama “Gherkin”, representa la forma en que deben ser escritos estos textos. Las ventajas que hacen de Cucumber una buena elección son: se puede escribir una especificación en más de 40 idiomas. Se puede usar etiquetas (TAG) para organizar y agrupar los escenarios, y finalmente se puede integrar fácilmente con las librerías de Ruby.

Escribiendo estas pruebas antes que el código, se pueden eliminar muchos mal entendidos mucho antes de que estos lleguen al código fuente. Las pruebas de aceptación escritas en este estilo se convierten en más que solo pruebas, ellas son “Especificaciones Ejecutables”, mencionan en [30].

3.2.1.1 Documentación viva.

Las pruebas de Cucumber comparten los beneficios de los documentos de especificación tradicionales en que pueden ser escritos y leídos por interesados del negocio, pero estas tienen una ventaja distintiva que es que pueden ser ejecutadas en una computadora en cualquier momento y decirnos que tan precisas son con respecto a lo desarrollado. En la práctica esto significa que en vez de escribir una vez la documentación y que esta gradualmente pierda validez, se convierte en una cosa viva que refleja el estado real del proyecto.

3.2.1.2 Fuente de verdad.

Además estas pruebas se convierten en la fuente de verdad definitiva acerca de lo que el sistema hace. Teniendo un simple lugar donde ir por esta información, ahorra mucho tiempo; que habitualmente se pierde en mantener sincronizada documentos de requerimientos, pruebas y código. También ayuda a construir la verdad dentro del equipo, ya que todos tienen la misma versión de la verdad y ya no más una parte de ella.

3.2.1.3 Cómo funciona.

Cucumber es una herramienta de línea de comando, cuando se ejecuta con el comando “cucumber”, lee las especificaciones escritas en archivos de texto plano llamados “features” (características), y con extensión *.feature; los examina en busca de escenarios a probar, y ejecuta estos escenarios contra el sistema. Cada escenario es una lista de pasos, Cucumber trabaja a través de estos. Para que Cucumber pueda entender estos archivos, ellos tienen que seguir una serie de reglas básicas de sintaxis. El nombre de estas reglas lo

han llamado “Gherkin”. Además junto con los archivos `*.feature`, se le da a Cucumber un conjunto de “definiciones de pasos” (en inglés *step definitions*) que mapean cada oración escrita en lenguaje natural con un paso (en inglés *step*) escrito en código Ruby, que describe la acción que ese paso debe realizar.

Las definiciones de los pasos se las registra también en archivos de texto plano cuyos nombres terminan en `_steps.rb`. Estos pasos son algunas líneas de código Ruby que interactúan con el código de la aplicación. Normalmente se vincula con alguna librería de automatización de procesos para que interactúe con el sistema real; en este caso la librería para automatización utilizada fue Capybara, que se describe más adelante.

Si el código Ruby del paso se ejecuta sin errores, Cucumber procede al próximo paso del escenario. Si se termina el escenario sin que ninguno de estos pasos haya provocado un error, se marca el escenario como “pasado” y pasa al siguiente escenario dentro del archivo `*.feature`. Si alguno de los pasos dentro del escenario falla, entonces se marca el escenario como “fallido” y se continua con el siguiente escenario. A medida que se ejecutan los escenarios, Cucumber imprime en la pantalla cuáles pasos pasaron y cuáles no, indicando el número de línea donde no está pasando. Utilizando una herramienta para colorear la consola, se le puede asignar los colores verde, amarillo, azul y rojo. Cuando todas las pruebas pasan en la interfaz queda todo el registro de un color verde y de allí su curioso nombre.

Los archivos de pasos se ubican en la carpeta “`step_definitions`”, sub-carpeta de “`feature`”, por ejemplo en este caso la ruta es: `CODIGO/vox/features/feature/step_definitions`, dentro del CD que acompaña este informe. La jerarquía que sigue Cucumber, desde los archivos `*.features` hasta la librería de automatización, está ilustrada en la imagen (Véase Fig. 10) extraída de [30] donde se aprecia cómo se desciende gradualmente.

3.2.2 Capybara – librería de automatización.

Capybara [31] es una herramienta de automatización, escrita en Ruby, que permite simular



Fig. 10: Cucumber testing stack.

Fuente: [9]

las interacciones del usuario con la aplicación través del navegador web; siguiendo los pasos que se describen en las definiciones de pasos de Cucumber. Y permite utilizar diferentes “Drivers” para realizar estas interacciones. Sus principales beneficios son:

- Funciona por defecto en aplicaciones Rails o Rack.
- API intuitiva, que imita el lenguaje del usuario.
- Cambios de back-end, se puede cambiar la forma en que se ejecutan las pruebas; desde un proceso en segundo plano que simula un navegador hasta hacerlo en un navegador real, sin tener que cambiar las pruebas.
- Sincronización, no hay que esperar manualmente a que terminen procesos asíncronos.
- Se lo puede utilizar con Cucumber, RSpec y MiniTest.

Su DSL (*domain-specific language*), en español *lenguaje de propósito específico*, provee selectores para: navegación, clic en botones y enlaces, interactuar con formularios, hacer consultas y búsquedas en el código de la página, trabajar con ventanas, ejecutar scripts, responder a mensajes modales, macheo y depuración. Ayudándonos en la lectura de las propiedades del DOM (*Document Object Model*), en español *Modelo de Objetos del Documento*.

Su utilización para el proyecto, en conjunto con su DSL, helpers y selectores, ayudaron a la productividad para la creación y ejecución de las pruebas de aceptación e integración. Por ejemplo, un selector que resultó particularmente útil fue “`save_and_open_page`” que abre en un navegador web con el estado actual del DOM, lo que ayudo a entender el proceso y como se ejecutan las pruebas en el navegador. Para su utilización hubo que agregar las gemas “capybara” y “cucumber-rails” al gemfile del proyecto.

Manejadores (Drivers): Capybara puede utilizar una variedad de navegadores y “headless drivers” (manejadores de cabeceras HTTP) entre ellos: RackTest (utilizado por defecto), Selenium, Capybara-webkit, Poltergeist. Los que realizarán las interacciones de diferentes maneras y con diferentes limitaciones. RackTest (escrito en Ruby) por ejemplo corre en segundo plano, sin utilizar una interfaz de usuario, lo que permite utilizarlo en máquinas virtuales, pero está limitado para la ejecución de JavaScript y tiene que ejecutarse en el mismo lugar donde está la aplicación.

3.2.3 RSpec - pruebas unitarias.

RSpec (<http://www.rspec.info>) [4] es una herramienta para aplicar la práctica ágil de BDD para el lenguaje Ruby. Está diseñado para hacer de TDD una experiencia agradable y

productiva. Se caracteriza por tener:

- Una herramienta de línea de comandos, muy completa. El comando: rspec.
- Descripciones textuales de ejemplos y grupos, con la gema [rspec-core](#).
- Reportes flexibles y personalizables.
- Lenguaje de expectativas extensible, con la gema [rspec-expectations](#).
- Un framework para dobles de prueba integrado, con la gema [rspec-mocks](#).

RSpec provee un DSL para especificar el comportamiento de los objetos. Adaptando la metáfora de describir el comportamiento de la manera en que lo se haría si se está hablando con el cliente o con otro desarrollador. Técnicamente, RSpec es una meta-gema de Ruby, que integra las gemas: rspec-core, rspec-expectations y rspec-mocks, estas pueden funcionar de forma independiente. Está escrito en Ruby y su código fuente está disponible públicamente. Fue creada por Steven Baker en 2005. Al igual que Cucumber las pruebas se escriben en archivos de texto plano siguiendo cierto formato, pero con extensión .rb y se guardan en la carpeta llamada “spec”.

Para el presente trabajo se utilizó RSpec, en su versión 2, para realizar las pruebas unitarias y de Integración del proyecto; siguiendo la práctica de TDD.

3.2.4 Guard – ejecución automática de pruebas locales.

Guard (<http://guardgem.org/>) [32], esta herramienta de línea de comandos que permite monitorizar cambios en los archivos de un directorio y ejecutar una acción en consecuencia, programando unos script en un archivo llamado Guardfile, escrito en su *DSL llamado “Guard DSL”* con base en Ruby; dicho archivo se debe ubicar en el directorio raíz del proyecto, en este caso se encuentra en /CODIGO/vox/Guardfile, del CD que acompaña este informe. Además Guard puede enviar una notificación al sistema operativo para informar del éxito o fracaso de las pruebas (mediante la utilización de una gema llamada Listen).

Se utilizó esta herramienta para ejecutar las pruebas automáticamente durante el desarrollo, con la ayuda de las gemas 'guard-cucumber', 'guard-rspec' y 'guard-rails'. Que permiten crear archivos Guardfile estándar ejecutando unos comandos (por ejemplo "guard init rspec" o "guard init rails"). Para ejecutarlo, una vez instalado, basta abrir una consola, ubicarse en el directorio de trabajo y ejecutar el comando: "guard"

3.2.5 Coveralls – cobertura de código de las pruebas.

Coveralls.io (<https://coveralls.io/>) [27] es un servicio para medir la cobertura de código que realizan las pruebas; posee una versión gratuita para proyectos de código libre que estén publicados en GitHub. Es agnóstico en términos del lenguaje de programación que evalúa o

del servidor de integración continua que se utilice. Este servicio hace un seguimiento de la cobertura del código y lleva estadísticas sobre los cambios, mostrando en una interfaz web sencilla; para el presente trabajo se puede observar el resultado ingresando a la siguiente dirección web: <https://coveralls.io/r/matiasmasca/vox>

Su integración al proyecto fue realmente sencilla, sólo se necesitó crear una cuenta en el servicio, vincularla con el repositorio en GitHub desde el sitio web [26]; esta vinculación es el único requisito que tiene el servicio Coveralls. Y luego se agregó la gema “coveralls” al archivo *Gemfile*. Finalmente se agregaron dos líneas de código en la configuración de las pruebas, en Cucumber: require 'coveralls' y en RSpec: Coveralls.wear!

Este servicio permite saber el grado de cobertura de las pruebas lo que permitió mantenernos en unos márgenes aceptables. Cabe aclarar que de las herramientas mencionadas del entorno de pruebas, esta fue la única que se usó exclusivamente cuando la aplicación ya estaba vinculada con en el servidor de integración continua [28].

3.3 Entorno de Producción

3.3.1 Travis-CI – integración continua.

Travis-CI. (<https://travis-ci.org/>)[28] es un servicio de integración continua y despliegue automatizado, para realizar la práctica ágil de “integración continua”. Está integrado con GitHub y ofrece soporte para más de 15 lenguajes de programación; además permite ejecutar las pruebas sobre diferentes versiones de una tecnología y encriptar archivos que tengan información sensible. Cabe destacar que es un servicio gratuito para proyectos de código abierto, con opciones pagas para proyectos privados.

Para su utilización, hay que crear una cuenta en el servicio. Luego vincularla con el repositorio en GitHub desde el sitio web [26]. Y finalmente se debe agregar un archivo llamado “travis.yml” en raíz del repositorio, este archivo contiene la configuración del servicio y es allí donde se configuran las diferentes opciones que se desea se ejecuten ante cada cambio en el repositorio vinculado. Para este proyecto dicho archivo se encuentra disponible en <https://github.com/matiasmasca/vox/blob/master/.travis.yml>

Dado los alcances del trabajo, para conocer los detalles técnicos de cómo realizar la integración continua para proyectos escritos en Ruby se puede consultar la guía de Travis-CI para Ruby en [33].

La utilización de este servicio permitió ejecutar las pruebas en un ambiente controlado y repetible antes de pasar a la etapa de despliegue, que además también la realizó.

3.3.2 Heroku – servidor de aplicaciones.

Heroku (www.heroku.com) [34] es una plataforma de aplicaciones en la nube, servicio de plataforma como servicio (*platform as a service* o PaaS), que provee soporte para varios lenguajes (Ruby, PHP, Node.js, Python, Java, Clojure, Scala) y permite realizar el despliegue de una aplicación bajo demanda abstrayéndose sobre la plataforma que está detrás (Sistema operativo, runtime, almacenamiento, comunicaciones, virtualización, etc.). Esto quiere decir que la aplicación se ejecutará en un entorno virtualizado de forma transparente para el desarrollador y para el usuario. Heroku se hará cargo, automáticamente y bajo demanda, de construir, desplegar, ejecutar y escalar la aplicación [35]. Para el caso de Rails utiliza internamente un servidor web llamado Unicorn. La arquitectura de complementos permite a los desarrolladores customizar el servicio con el uso de más de 100 paquetes de servicios de terceros que puede agregar según sus necesidades [35]. La utilización de este servicio, en su versión gratuita, permitió desplegar la aplicación en un entorno controlado y repetible, de forma sencilla, rápida y sin costo económico. Además de permitió la vinculación con el servidor de integración continua, Travis-CI, sin mayor esfuerzo; esta combinación hizo posible tener una versión del sistema publicada desde el primer momento.

3.3.3 PostgreSQL - base de datos.

PostgreSQL (<http://www.postgresql.org/>) [36] es un sistema de gestión de bases de datos objeto-relacional (en inglés *object-relational database management system* u *ORDBMS*), de código libre. Utiliza un modelo cliente/servidor y usa *multiprocesos* en vez de *multihilos* para garantizar la estabilidad del sistema. PostgreSQL funciona muy bien con grandes cantidades de datos y una alta concurrencia de usuarios accediendo a la vez al sistema, según sus creadores en [36].

Se utilizó este motor de base de datos en el entorno de producción, el esquema de la base de datos tuvo que ser compatible con él. Si bien la utilización de este motor no fue una decisión tomada, ya que es lo que estaba disponible gratuitamente en el servidor de producción, no se encontró una razón que justifique su remplazo por otro motor, por ejemplo MySQL. Cabe aclarar que la base de datos, utilizada gratuitamente en Heroku tiene un límite que permite hasta 10.000 registros.

3.4 Comentarios y discusiones sobre las herramientas.

Las metodologías ágiles se soportan en herramientas para poder mantener la flexibilidad necesaria ante el cambio y disminuir la deuda técnica; de allí la longitud de este capítulo y la cantidad de ellas. Cabe destacar que, siempre que se pudo, se trató de utilizar herramientas

libres o en su defecto de código abierto y en todo caso de uso gratuito para evitar temas legales de las licencias de uso.

Las herramientas utilizadas facilitaron en gran medida el desarrollo del presente trabajo, logrando buenos niveles de productividad. Sobre todo aquellas que automatizaron tareas monótonas y repetitivas.

Gracias a ellas he logrado desarrollar el sistema propuesto, sin mayores dificultades y en un tiempo aceptable.

Las herramientas, como Cucumber, RSpec y BetterErrors ayudaron a encontrar exactamente el lugar donde está fallando el código o la prueba. Facilitando el proceso de “debugging” y contribuyendo a que lleguen menos errores a producción.

Al ser un proyecto de código abierto, existen una serie de servicios profesionales a los que se puede tener acceso sin costo, por una política generalizada en los ambientes de software libre. Ellos por ejemplo son: GitHub, Travis-ci, Coveralls.io, PivotalTracker, entre otros.

3.4.1 Sobre Cucumber.

- i. A medida que se avanza en el proyecto, se van sumando “features” y sus “steps definition”, se puede apreciar como la reutilización de pasos se vuelve útil y necesaria; además acelera notablemente el proceso de crear los features y sus pasos.
- ii. No se pudo detectar la falta de algunas features hasta avanzado en el proceso. Por ejemplo la feature de "mostrar categoría" para el administrador se la detectó cuando estaba haciendo el caso particular de "mostrar categoría" para el usuario organizador. Es decir que pueden haber comportamientos sin probar y no detectarse esa falta.
- iii. A medida que el software crece en complejidad, se hace cada vez más difícil mantener el mapa mental de todas las rutas posibles y los estados que tiene que tener el sistema en cada una de estas rutas. Aquí es cuando los escenarios y las pruebas automatizadas empiezan a evidenciar sus beneficios de liberar la carga mental y controlar esas posibles rutas y estados.
- iv. Cuando se empieza a usar los "Esquemas de Escenarios" para probar los casos extremos, se aprecia el poder de la herramienta, para probar comportamiento deseado sobre muchos casos de prueba. Se ahorra tiempo y código repetitivo

3.4.2 Documentación viva.

- v. Hubo cierta clase de información que no resultó fácil de encontrar o ver rápidamente;

como los distintos valores que puede asumir un dato (de tipo selección). Por ejemplo para "Estado" de una solicitud de admisión a un proceso: aprobado, pendiente o rechazado; si no hay una prueba que los muestre, una forma seria utilizar las "tablas de ejemplos" de Cucumber, pero en definitiva debe existir al menos 1 prueba que los utilice de esa forma. En algunos casos fueron agregados como comentarios, de la forma "#PO: ...", dejando notas de lo conversado con el Product Owner sobre los posibles valores de un dato.

- vi. En general hay que tener en cuenta que esta documentación, no es sólo para el programador actual, es para todo el equipo, es para el equipo que venga después, es para el cliente.

3.4.3 Sobre RSpec.

- Las pruebas por defecto en RSpec, generadas por Rails, son realmente frágiles se rompieron con frecuencia ante cambios en la estructura.
- Probar el comportamiento de la vista (GUI), en el caso donde había asociaciones de datos, se volvió un poco engorroso y difícil en RSpec comparado en el primer intento de la prueba de la misma funcionalidad con Cucumber.
- Si se trabaja en conjunto con Cucumber, no hay que duplicar esfuerzo y probar las vistas en un solo lugar.

3.4.4 Sobre Guard.

En algunas oportunidades, mientras se encontraba activo el monitoreo se detectaba un incremento notable en el uso de recursos de la máquina que provocaban un recalentamiento del hardware o una enlentecimiento general, entonces se optaba por no usarla cuando la máquina de desarrollo era una computadora portátil. Sin embargo la experiencia de recibir una retroalimentación, casi, inmediata sobre el código recién modificado, ayudaba a seguir con mayor velocidad la técnica TDD o BDD. Luego existen técnicas de optimización y herramientas para acelerar la ejecución de los test, pero la explicación de estas escapan al presente trabajo.

Capítulo 4. Resultados

En este capítulo se exponen los resultados alcanzados durante la realización del presente trabajo. En primer lugar se comentan los artefactos de cada etapa y sobre el proceso iterativo realizado y algunas de sus características. En segundo lugar, se comenta sobre la aplicación de los conceptos teóricos con sus prácticas asociadas; mostrando un ejemplo de todo el proceso realizado por cada historia de usuario. Sobre el final se comenta sobre la práctica de integración continua y como se realizó el despliegue de la aplicación. Y finalmente se realiza un recorrido por la aplicación lograda.

4.1 Síntesis de resultados por etapa.

A modo de síntesis, al completarse las etapas consignadas en la metodología del trabajo se lograron los siguientes resultados y artefactos de software:

Etapa 1 – Investigación Preliminar: del relevamiento realizado se selecciono el lenguaje Ruby, la práctica ágil BDD y las herramientas asociadas compatibles con Ruby: Cucumber, Capybara y RSpec. También se seleccionó el framework Ruby on Rails.

Etapa 2 - Concepción: se obtuvieron el documento ERS (Especificación de Requerimientos de Software) véase anexo I. Y las Historias de usuario.

Etapa 3 - Desarrollo: se obtuvo el prototipo funcional para premios (aplicación web); el repositorio público de código; las pruebas automatizadas, la medición de cobertura de pruebas automatizada; se configuro el servidor de CI, automatizando el despliegue.

Etapa 4 - Documentación: se obtuvo el Informe TFA.

4.2 Proceso de Software.

Dado que se siguió un proceso de desarrollo de software iterativo e incremental, para el desarrollo de la aplicación se realizaron 7 iteraciones. En la tabla 2 se describen los rangos de fechas y la velocidad medida en puntos de cada una de las iteraciones.

Como se explico antes, hasta la iteración 4 se siguió la práctica de BDD de forma pura. De 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 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; se sobre evalúo y algunas tareas que se pensaba a priori que iban a ser más

difíciles de lo que fueron la primera vez que se aplicó la técnica. Y otras a posteriori llevaron hasta incluso el triple de lo que se pensaba, por ejemplo 1 semana de trabajo para poder subir la una imagen y hacer la prueba, tarea que tenía una estimación de 2 días. Esta situación ya que fue prevista por W. Humphrey en [37] donde advierte la dificultad de hacer estimaciones sobre proyectos totalmente nuevos que no tienen un punto de comparación previa.

Tabla 2: Iteraciones del proyecto. Fuente: Elaboración propia.

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

Como parte del proceso, y dado que se siguió un enfoque de afuera hacia adentro (del inglés outside-in), propuesto en [7], se empezó por el bosquejo de las pantallas con *wireframes* y *mockups*, la interacción entre ellas. Continuando con el comportamiento esperado describiéndolo en las historias de usuario, con sus escenarios y ejemplos. A continuación se describe la experiencia de su utilización en el presente trabajo.

4.2.1 Prototipos de interfaces gráficas de usuario (GUI).

Luego de obtener una visión general del problema a solucionar, en conversaciones con el cliente, y al ser la solución propuesta un sistema web, se procedió a crear los prototipos rápidos de interfaces de usuario. Para ello se utilizó un pizarrón, o en algunos casos lápiz y papel, 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 estos elementos permitió analizar las propiedades y navegación de la aplicación de manera rápida y económica.

En la imagen (Véase Fig. 11) se aprecia el bosquejo para la interfaz de “alta de una entidad organizadora”, que luego se digitalizó utilizando Drawing (Véase Fig. 11).

Entidad Organizadora.

Nombre: []

Dirección: []

Descripción: []

Sitio Web: []

E-mail: []

Logo (URL): []

[SAVE]

| Entidad Organizadora | |
|----------------------|---|
| Nombre | Abcdefghi |
| Domicilio | Abcdefgh |
| Descripción | TEXT |
| Sitio Web | http://www.sitioentidadorganizadora.com.ar/ |
| email | usuario@correo.com.ar |
| Logo | URL |
| Guardar | |

Fig. 11: Mockup en baja fidelidad de formulario. Fig. 12: Wireframe de formulario. Fuente: Elaboración propia.

4.2.2 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 10x10cm, que se pegaron sobre un pizarrón, como se ve en la imagen (Véase Fig. 13). 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.



Fig. 13: Historias de usuario previo a priorizar. Fuente: Elaboración propia.

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>. Una vez allí puede activar las diferentes pizarras haciendo clic en las palabras de la izquierda: *Current, Backlog, Icebox*,

Done y Epics como se aprecia en la imagen (Véase Fig. 14) donde están activadas.

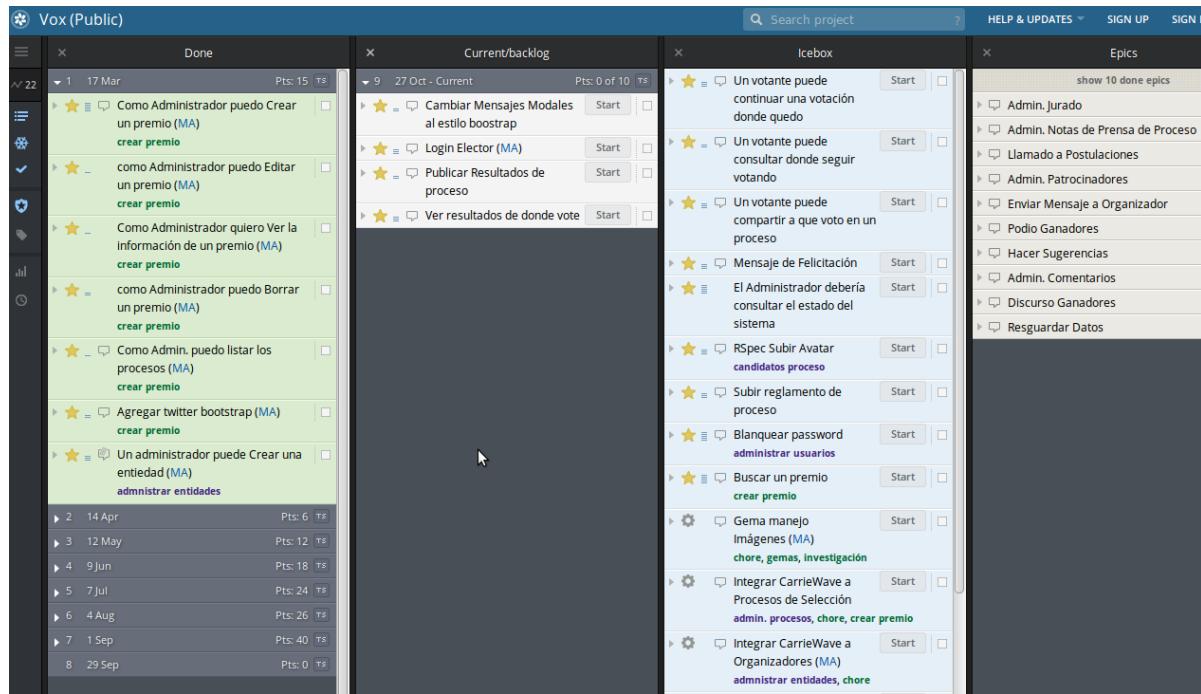


Fig. 14: Historias de usuario en PivotalTracker. Fuente: Elaboración propia.

4.3 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 satisfactoria y luego se da un ejemplo del proceso seguido para aplicar estas técnicas en una historia de usuario.

Se aplicaron estos 2 conceptos para los módulos de administración (back-end) del usuario administrador, es decir a las funcionalidades que realiza el administrador de la aplicación. Utilizando las herramientas específicas: Cucumber y Capybara para BDD; RSpec para TDD y Guard para ejecutar 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 incluyó las historias de usuario elementales: crear, editar, borrar, listar y mostrar. Con sus “caminos felices” y casos extremos.

A nivel de código se implementaron las restricciones de seguridad, con la integración de la gema Devise y la utilización de filtros a nivel de controlador.

Para el primer módulo que se realizó no se utilizó “scaffold”, herramienta que incluye Rails para generar estructuras de código estándar. La razón principal de esta decisión fue porque se quería entender el funcionamiento del framework antes de pasar a la utilización de las

herramientas que ahorran trabajo.

4.3.1 Aplicando BDD.

Las historias de usuario implementadas, junto con sus escenarios, están disponibles en la carpeta CODIGO/vox/features/, del CD que acompaña este informe. Un ejemplo de estas se puede apreciar en la imagen (Véase Fig. 15).

Para cada archivo “*.feature” se tuvo que automatizar la ejecución de la prueba en los archivos de definición de pasos. En la subcarpeta “step_definitions”, ubicada en “CODIGO/vox/features/step_definitions/” del CD que acompaña este informe, se encuentran estos archivos, con extensión “*_step.rb”. Cada uno de ellos vincula los pasos la historia de usuario (archivo *.feature) con un código ejecutable por la herramienta Cucumber y librerías complementarias. En la imagen (Véase Fig. 16) se puede apreciar uno de estos archivos de definición de pasos.

De las entrevistas con los clientes se obtuvieron ejemplos que se utilizaron para ilustrar comportamiento esperado por la aplicación. Utilizar el lenguaje del cliente ayudó a mantener alta la abstracción, 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.

4.3.1 Aplicando TDD.

Para aplicar la práctica de TDD se utilizó la herramienta RSpec. Con ella se escribieron las pruebas unitarias para las distintas capas de la aplicación, como se ilustra en la imagen (Véase Fig. 15).

Estas se encuentran en la carpeta “spec”, ubicada en /CODIGO/vox/spec del CD que acompaña este informe, y dentro de ella organizadas en las diferentes sub-carpetas:

/spec/controllers Las pruebas unitarias referidas a los controladores, del modelo MVC.

/spec/models Las pruebas unitarias referidas a los modelos.

/spec/helpers Las pruebas unitarias relativas a los helpers de la aplicación.

/spec/routing Las pruebas para probar las diferentes rutas de la aplicación.

/spec/views Las pruebas para algunas vistas, ya que luego se optó por utilizar Cucumber y probar las vistas desde allí, para no duplicar el esfuerzo.

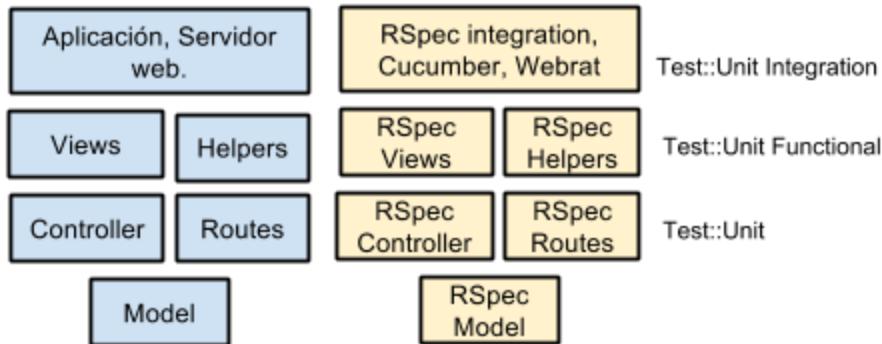


Fig. 15: Capas de pruebas. Fuente: Elaboración propia.

Las pruebas se escribieron en archivos de texto plano con extensión “*_spec.rb”.

Cabe aclarar que ni al aplicar BDD o TDD se utilizaron dobles de prueba, en su defecto se utilizó la estrategia de acceso directo al modelo con datos de prueba estáticos; que si bien puede resultar ineficiente, a largo plazo, para el presente trabajo no se justificó la complejidad extra del uso de dobles de prueba.

En la imagen (Véase Fig. 21), donde además de apreciar como son los archivos de las pruebas, se puede ver un objeto 'organizer' que es creado de la clase 'Organizer' con atributos estáticos que están en la variable 'valid_attributes' definida al inicio del archivo.

Una vez aplicada BDD y TDD con todas las historias de usuario del módulo back-end de administración y habiendo extraído algunas conclusiones de la experiencia se completó el resto de las funcionalidades mínimas de la aplicación con la utilización en menor medida de estas prácticas para reducir los tiempos y dejar una aplicación funcional en su forma mínima viable.

4.3.3 Ejemplo del proceso al aplicar BDD y TDD.

A modo de ejemplo a continuación, se ilustrará 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 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. En la imagen (Véase Fig. 16) se muestra como se ve 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. 16: Historia de usuario "borrar una organización". Fuente: Elaboración propia.

Paso 2: Crear pasos. Automatizar.

El siguiente paso consiste 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.

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 4 pasos del escenario de la Fig. 16, solo se tuvo que automatizar uno nuevo, como figura en la imagen (Véase Fig. 17). 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. 17: 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 la prueba falla porque no tiene una implementación asociada, como se ve en la imagen (Véase Fig. 18), la prueba falla porque no encuentra un método en el controlador asociado a la acción "destroy".

```

Escenario: borrar premio
  Dado existe una Organización llamada "ACME" con domicilio en "Av. Siempre Viva 742" # features/borrar_organizacion.feature:10
  Y que estoy en la pantalla de Administración de Organizaciones # features/step_definitions/borrar_organizacion_steps.rb:3
  Cuando hago click en Borrar para "ACME" # features/step_definitions/borrar_organizacion_steps.rb:3
    The action 'destroy' could not be found for OrganizersController (AbstractController::ActionNotFound)
    ./features/step_definitions/borrar_proceso_steps.rb:3:in `^hago click en Borrar para "(.*?)"$/'
    features/borrar_organizacion.feature:13:in `Cuando hago click en Borrar para "ACME"'
    Entonces se borra la Organización "ACME" # features/step_definitions/borrar_organizacion_steps.rb:3

Failing Scenarios:
cucumber features/borrar_organizacion.feature:10 # Scenario: borrar premio

1 scenario (1 failed)
4 steps (1 failed, 1 skipped, 2 passed)
0m1.429s
[Finished in 12.3s with exit code 1]
[cmd: ['/home/comunidadadic/.rvm/bin/rvm-auto-ruby -S cucumber features/borrar_organizacion.feature -l14']]
[dir: /home/comunidadadic/vox]
[path: /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games]

```

Fig. 18: 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. Para ello se crearon los archivos en la carpeta /spec que interpretará la herramienta RSpec. En este caso se accedió al archivo “organizers_controller_spec.rb” (disponible en el CD que acompaña este informe en /CÓDIGO/vox/spec/controllers/organizers_controller_spec.rb) y se creó la descripción del comportamiento esperado, como se ve en la imagen (Véase Fig. 19).

```

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

```

Fig. 19: Spec inicial para acción “destroy” del controlador. Fuente: Elaboración propia.

Paso 6: Ejecutar. Prueba pendiente.

Si se ejecutan las pruebas unitarias tal cual estaban en el 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 imagen (Véase Fig. 20).

```
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. 20: 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 imagen (Véase Fig. 21) para este caso se crea previamente un objeto “organizer” que actúa como doble de acción para la prueba, generando un registro en cada ejecución. 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. 21: Prueba unitaria para borrar organización. Fuente: Elaboración propia.

Paso 8: Ejecutar pruebas unitarias. Fallan.

Al ejecutar las pruebas del paso anterior estas fallan, como se capturo en la imagen (Véase Fig. 22) ya que aún no se ha creado la implementación de ese método en el código de la aplicación en sí misma.

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. 22: 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 que pase esa prueba. La práctica indica que primero hay que hacer el mínimo código que pase esa prueba y luego ir refactorizando. En la imagen (Véase Fig. 23) se captura el código final 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. 23: Implementación mínima. Fuente: Elaboración propia.

Paso 10: Pruebas pasan.

Al ejecutarse las pruebas, luego de la refactorización, se puede apreciar (Véase Fig. 24) como estas continúan pasando sin fallas.

```
.....
Finished in 0.42024 seconds
7 examples, 0 failures

Randomized with seed 40504
```

Fig. 24: Pruebas unitarias pasan. Verde. Fuente: Elaboración propia.

Paso 11: Prueba exploratoria.

Si bien no es parte de las prácticas BDD o de TDD, finalmente 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 imagen (Véase Fig. 25) se ve la interfaz, luego de ser estilizada con Bootstrap, para esta historia.

Lista de Organizaciones

| Nombre Organización | Domicilio | Sitio web | Correo electrónico | |
|---|---|------------------------|--------------------|--|
| ACME | Desierto de Arizona | www.acme.org | contact@acme.org | <button>Mostrar</button> <button>Editar</button> <button>Borrar</button> |
| AMPAS (Academy of Motion Picture Arts and Sciences) | 8949 Wilshire Boulevard Beverly Hills, California 90211 | http://www.oscars.org/ | contact@oscars.org | <button>Mostrar</button> <button>Editar</button> <button>Borrar</button> |

[Crear nueva Organización](#)

Fig. 25: Lista organizaciones. Fuente: Elaboración propia.

Luego en la siguiente imagen (Véase Fig. 26) el mensaje de confirmación antes de borrar el registro.

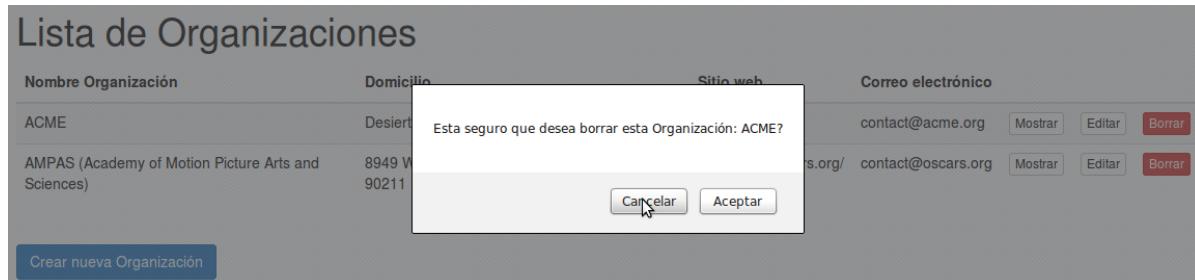


Fig. 26: Mensaje confirmación. Fuente: Elaboración propia.

Luego de eliminar el registro, se le muestra al usuario otro mensaje de confirmación (Véase Fig. 27).



Fig. 27: Mensaje confirmación tras eliminar dato. Fuente: Elaboración propia.

Paso 12: Iterar.

Se continúa con la siguiente historia de la iteración.

4.4 Aplicando integración continua.

Para aplicar la integración continua se utilizó el servicio Travis-ci. Se realizaron más de 210 integraciones automatizadas. En cada una de ellas se realizó el siguiente procedimiento:

4.4.1 Procedimiento automatizado.

Con cada integración (en inglés *build*) se instalaron todas las librerías necesarias para ejecutar la aplicación Rails, se generó la base de datos desde cero, 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 (exit 0), el servicio procedió a realizar el despliegue de la aplicación en el servidor de Heroku [34], subiendo los archivos necesarios, ejecutando las instalaciones de librerías necesarias, concluyendo con la configuración de la base de datos con sus datos de prueba (*seed.rb*) y luego reiniciando la aplicación, para dejarla lista y funcionando.

La bitácora de esta operación para cada uno de las integraciones se puede consultar en línea en <https://travis-ci.org/matiasmasca/vox>.

En caso de fallo se suspendió el despliegue (en inglés *deploy*) y se envió un email informando que el proceso había fallado. Por ejemplo se puede consultar, en línea, el registro de los pasos automatizados realizados por el servicio para la integración número 240 en: <https://travis-ci.org/matiasmasca/vox/builds/39428531>.

Una vez que se tuvo un conjunto de pruebas, la tarea de agregar la integración continua al flujo del trabajo resultó bastante sencilla, sólo hubo que dar de alta una cuenta en el servicio Travis-ci y configurar un archivo "travis.yml". Luego se configuró una cuenta en el servidor de aplicaciones Heroku y se agregó algunos datos al archivo "travis.yml". Sin conocimiento previo esta tarea llevó 8 horas.

4.5. Despliegue de la aplicación.

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

En este entorno se utiliza una base de datos PostgreSQL.

Si bien se realizaron algunas experiencias manuales, para entender cómo realizar el despliegue con las recomendaciones de [35], luego se automatizó y delegó esta tarea a Travis-ci. En el anexo II se describen los detalles técnicos de como se realiza el despliegue de la aplicación en Heroku.

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.

4.5.1. 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 3 perfiles usuarios:

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

Organizador: usuario: unemail@gocom ; clave: clave12345

Una vez identificado en el sistema se le recomienda, presionar "ACME Prize", en el menú de la izquierda, para ver un proceso de selección con datos de prueba.

Para ver el escrutinio, de ese proceso, primero debe "Cerrar las Votaciones", para ello debe ingresar en: http://tfa-vox.herokuapp.com/selection_processes/1/edit?cambiar_estado=cerrar y presione "Guardar".

Luego si ingresar a: http://tfa-vox.herokuapp.com/selection_processes/1/voter/results podrá ver los resultados.

Jurado:

Usuario: jurado1@gocom ; clave: clave12345

Usuario: jurado2@gocom ; clave: clave12345

Usuario: jurado3@gocom ; clave: clave12345

4.6 Descripción del sistema.

En esta sección se describen las funcionalidades de la aplicación definida para aplicar las prácticas y conceptos descritos como parte de los objetivos del trabajo. Cabe destacar, que en el trabajo se abordan algunas funcionalidades básicas a modo de ejemplo, y se espera completar el resto de las mismas como trabajo futuro.

VOX, es un sistema gestor de contenidos web para premios y votaciones. Permite a una organización administrar la información relativa a uno o más premios y votaciones, con foco en los nominados/candidatos, junto con la administración del padrón de electores participantes, votación anónima y la posterior consulta de resultados (escrutinio). Permite automatizar algunas tareas que se realizan de forma manual y compartir la misma información para todos los participantes.

El acceso al sistema está basado en 3 perfiles de usuarios registrados: Administrador, Organizador y Jurado. Siendo el primero el único que se debe crear de forma manual al desplegar el sistema.

A continuación se muestra la interfaz principal de la aplicación y se describen las funcionalidades principales de la aplicación web , desde la perspectiva del usuario (Véase Fig. 28).



Fig. 28: Pantalla principal. Fuente: Elaboración propia.

4.6.1 Identificación de usuarios.

4.6.1.1 Inicio de sesión.

Para poder utilizar la aplicación y participar de los procesos, los usuarios deberán identificarse, para ello presiona el botón “Iniciar sesión” e ingresan sus respectivos usuarios y contraseñas, en el formulario que se muestra en la imagen (Véase Fig. 29).

4.6.1.2 Registro.

Todos los usuarios deben pasar por un proceso de registro, completando el formulario que se muestra en la imagen (Véase Fig. 30), donde se le solicitan 3 datos básicos: nombre de usuario, email y contraseña.

Iniciar Sesión

Correo electrónico

Clave

Remember me

Iniciar sesión

Registrarse

[¿Olvidó su clave?](#)

Crea tu cuenta de Vox

Datos Obligatorios:

| | | |
|---------------------------------|----------------------|--|
| Nombre de Usuario | <input type="text"/> | Ingrese el nombre que utilizará para identificarse en el sistema. |
| Dirección de correo electrónico | <input type="text"/> | Ingrese una dirección de correo electrónico válida, nos comunicaremos con usted. |
| Clave de acceso | <input type="text"/> | |
| Confirme su clave de acceso | <input type="text"/> | |

Puede completarlos más tarde:

| | | |
|---------------------|----------------------|--|
| Nombre/s | <input type="text"/> | Ingrese su nombre o nombres, para personalizar algunas partes del sistema. |
| Apellido | <input type="text"/> | Ingrese su apellido/s., para personalizar algunas partes del sistema. |
| Usuario en Facebook | <input type="text"/> | Ingrese la URL de su perfil de facebook. |
| Usuario en Twitter | <input type="text"/> | @ Ingrese su usuario de twitter, sin el @. |

Opciones:

Fig. 29: Inicio de sesión. Fuente: Elaboración propia.

Fig. 30: Formulario creación de usuarios. Fuente: Elaboración propia.



Fig. 31: Barra de navegación superior. Fuente: Elaboración propia.

A excepción del administrador, que se crea al hacer el despliegue. Solo este usuario puede cambiar el nivel de privilegios de los demás usuarios; ya que por defecto se registran con el nivel de *jurados*.

4.6.2 Menú escritorio.

Una vez identificado en el sistema se le mostrará el “escritorio” del usuario, este consiste en una serie de botones que dan acceso a las principales funcionalidades del usuario. En la parte superior de la pantalla, en la barra negra, aparece la palabra “Escritorio” como se ve en la imagen (Véase Fig. 31) al presionarla podrá siempre volver a su escritorio de usuario.



Fig. 32: Escritorio administrador. Fuente: Elaboración propia.



Fig. 33: Escritorio usuario organizador. Fuente: Elaboración propia.

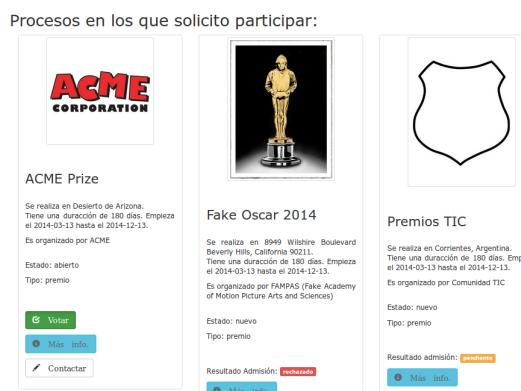


Fig. 34: Escritorio jurado. Fuente: Elaboración propia.

4.6.2.1 Escritorio del administrador.

Es la interfaz desde la que este usuario, puede modificar o crear los datos que utiliza la aplicación, consiste en una botonera como se ve en la imagen (Véase Fig. 32). Para crear

nuevos datos puede presionar los botones de la derecha y se le mostrarán formularios de creación de datos. Para cada opción de la izquierda se le mostrará una tabla o interfaz CRUD (del inglés **C**reate, **R**ead, **U**pdate and **D**elete), que sirven para visualizar y modificar los datos, son similares a la que se muestra en la imagen (Véase Fig. 36), en esta tanto al presionar el botón 'editar' como el botón 'nuevo', le mostrarán un formulario para crear o modificar los datos.

La única restricción es que no tiene acceso a las tablas relativas a los votos, ya que no se detectó una razón válida para que exista la necesidad de modificar lo relativo a un voto.

4.6.2.2. Escritorio del organizador.

Al igual que el administrador este usuario tiene un “escritorio” desde donde puede acceder a las opciones más utilizadas, como se aprecia en la imagen (Véase Fig. 33) y un menú lateral que le brinda acceso directo a otras opciones.

Este es el módulo principal del sistema web, desde aquí un usuario con privilegios de *organizador*, puede crear y modificar todo lo relativo a los *procesos de selección (premios)* de su organización.

4.6.2.2.2 Mi organización.

Da acceso al formulario para modificar los datos de la organización asociada al usuario como se aprecia en la imagen (Véase Fig. 35), en caso de no tener asociada una organización el botón dirá 'Crear Organización' y le mostrar un formulario igual al utilizado para modificar (Véase Fig. 35).

Nueva entidad organizadora

Podrá asociar los procesos de selección que crea a esta Organización, luego se hará referencia a ella como Organizador y no a su usuario.

Del Usuario: 15

Nombre de la Organización

Dirección

Sitio web

correo electrónico

Isologotipo:

Al subir la imagen declaras que no infringes derechos de autor y que tienes permiso para utilizar la Imagen que estas subiendo.

Browse... No file selected.

Guardar cambios

Fig. 35: Formulario nueva entidad organizadora. Fuente: Elaboración propia.

4.6.2.2.2 Mis procesos.

Lista de procesos de selección

| Nombre del Proceso | Lugar | Duración | Fecha Inicio | Fecha Cierre | Tipo | Estado | |
|----------------------|---|----------|--------------|--------------|------|---------|--|
| Fake Oscar 2014 | 8949 Wilshire Boulevard Beverly Hills, California 90211 | 180 | 2014-03-13 | 2014-12-13 | 1 | nuevo | <input type="button"/> Mostrar <input type="button"/> Editar <input type="button"/> Borrar |
| Grammy 2014 | Av. Siempre Viva 742 | 180 | 2014-03-13 | 2014-12-13 | 1 | nuevo | <input type="button"/> Mostrar <input type="button"/> Editar <input type="button"/> Borrar |
| Premios TIC | Corrientes, Argentina | 180 | 2014-03-13 | 2014-12-13 | 1 | nuevo | <input type="button"/> Mostrar <input type="button"/> Editar <input type="button"/> Borrar |
| Miss Multiverso 2014 | Los Angeles, CA, USA | 364 | 2014-03-13 | 2014-12-13 | 2 | nuevo | <input type="button"/> Mostrar <input type="button"/> Editar <input type="button"/> Borrar |
| ACME Prize | Desierto de Arizona | 180 | 2014-03-13 | 2014-12-13 | 1 | cerrado | <input type="button"/> Mostrar <input type="button"/> Editar <input type="button"/> Borrar |

Nuevo proceso de selección

Fig. 36: Vista reducida de interfaz CRUD. Fuente: Elaboración propia.

Desde el botón “Mis Procesos” (Véase Fig. 33) se puede acceder a un listado de todos los procesos (interfaz CRUD) de la organización asociada al usuario, como se ve en la imagen (Véase Fig. 36), y no a los de otras organizaciones.

4.6.2.2 Mis categorías.

Símil a “Mis procesos” pero para las categorías asociadas a los procesos de la organización del usuario.

4.6.2.3 Escritorio del jurado.

A diferencia de los anteriores, el escritorio del jurado muestra directamente los procesos de selección en los que solicito participar, en la imagen (Véase Fig. 34) se ve que solicito participar en 3 de ellos.

En esta versión del sistema sólo usuarios registrados pueden votar y se los ha llamado jurados. Por defecto todo usuario registrado es de tipo jurado. Cada jurado deberá postularse para cada proceso de selección en los que quiera participar y luego deberá ser aceptado por el organizador del proceso.

Para cada proceso de selección se muestra una descripción, su estado y tipo. Continuando con el estado relativo al usuario. En caso de que pueda votar aparecerá el botón en Verde con la palabra 'votar', como figura en la imagen (Véase Fig. 34), si fue rechazado aparecerá una etiqueta en rojo y naranja si aún está pendiente de consideración.

4.6.3 Menú inicio.

Estando identificado en el sistema, al ingresar a este menú se le presenta al usuario una tabla con los procesos de selección disponibles. Para cada uno de estos figura su nombre, lugar, duración, tipo y estado. Además para cada proceso aparece un botón naranja “Solicitar autorización” donde el usuario puede solicitar participar de ese proceso y en caso de que el usuario pueda votar aparecerá el botón verde “Votar”, como figura en la imagen (Véase Fig. 37), finalmente un botón celeste “Más info.” para acceder a más información sobre ese proceso.

| Nombre del Proceso | Lugar | Duración | Fecha Inicio | Fecha Cierre | Tipo | Estado | Solicitar autorización | Votar | Más info. |
|--------------------|---|----------|--------------|--------------|------|---------|------------------------|---|---|
| ACME Prize | Desierto de Arizona | 180 | 2014-03-13 | 2014-12-13 | 1 | abierto | Solicitar autorización |  |  |
| Fake Oscar 2014 | 8949 Wilshire Boulevard Beverly Hills, California 90211 | 180 | 2014-03-13 | 2014-12-13 | 1 | nuevo | Solicitar autorización |  |  |
| Grammy 2014 | Av. Siempre Viva 742 | 180 | 2014-03-13 | 2014-12-13 | 1 | nuevo | Solicitar autorización |  |  |
| Premios TIC | Corrientes, Argentina | 180 | 2014-03-13 | 2014-12-13 | 1 | nuevo | Solicitar autorización |  |  |

Fig. 37: Listado compacto de procesos. Vista reducida. Fuente: Elaboración propia.

4.6.4 Procesos de selección.

El proceso de selección es la funcionalidad implementada para gestionar un evento: crea un proceso, define las categorías, permite incorporar los candidatos y controla tanto la apertura como el cierre de las votaciones, además controlar el padrón de electores (jurados).

4.6.4.1 Crear un proceso de selección.

El usuario organizador puede crear procesos de selección, asociados a la organización que representa que previamente tuvo que haber creado.

Tras presionar el botón “Nuevo proceso de selección” de su escritorio, se le mostrará un formulario de creación como el de la imagen (Véase Fig. 38). Entonces deberá proveer información básica y útil para los interesados en ese nuevo proceso: una descripción general, fechas límite, el lugar donde se realiza, un tipo de proceso y el estado.

Tipo: con esta lista se define si el proceso de selección será un premio, un certamen o una votación. * En la versión actual solo se trabaja con premios y votaciones.

Estado: el estado de un proceso sirve para determinar su visibilidad e iniciar y terminar las votaciones. Puede tener 3 estados: nuevo, abierto y cerrado.

Por defecto un proceso tiene estado “nuevo”. Abierto significa que se puede votar en él y cerrado que ya terminó y no se puede votar más en él.

4.6.4.2. Categorías de un proceso.

Las categorías de un proceso, son las n-uplas de candidatos a competir para ganar esa categoría.

Un usuario organizador puede crear categorías vinculadas al proceso seleccionado. Los datos solicitados son un nombre, una descripción y el número de plazas (la cantidad de candidatos que podrá tener), como se puede apreciar en el formulario de la imagen (Véase Fig. 39).

Luego podrá ir agregando los candidatos directamente, al seleccionar la categoría y presionar los botones “Agregar candidato” como se ven en la imagen (Véase Fig. 40)

Nuevo proceso de selección

Nombre del proceso
ingrese el nombre del proceso, que verá el público.

Lugar
ingrese el lugar donde se realizará el proceso.

Duración
ingrese la cantidad de días
días.

Fecha de inicio
21 octubre 2014

Fecha cierre
21 octubre 2014

Tipo
premio

Estado
nuevo

Organiza: ACME
• Desierto de Arizona Más información

[Guardar cambios](#)

Nueva categoría

Las categorías, forman las n-uplas de los procesos (por ejemplo una terna de nominados, tiene 3 plazas).

Pertenece al proceso: ACME Prize

• Más información

Nombre categoría
ingrese un nombre para la categoría.

Descripción
ingrese una descripción para la categoría, que ayude a determinar los candidatos que la incluyen.

Nro. de plazas
ingrese la cantidad de candidatos que podrá tener la categoría.

[Guardar Cambios](#)

[Ir a Categorías](#)

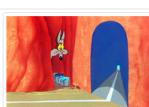
Fig. 38: Formulario para crear un proceso de selección. Fuente: Elaboración propia.

Fig. 39: Formulario para crear una categoría. Fuente: Elaboración propia.

A medida que se van agregando los candidatos, se irán mostrando en la categoría y se irán quitando los botones “Agregar candidato”; esto se aprecia en la parte inferior derecha de la imagen (Véase Fig. 40) donde ya se habían agregado 3 candidatos y todavía faltan agregar dos para completar la categoría.

Nombre: Mejor idea
Descripción: TOP5. Mejores ideas para aplicar en el desierto de Arizona
Nro. de plazas: 5
Proceso de selección: ACME Prize

Candidatos/Postulantes:



Tunel falso

Pintar un tunel falso sobre una pared, y luego redirigir las líneas del tráfico hacia él.

[Mostrar](#) [Editar](#)



Comida con hierro

Colocar comida con pelotitas de hierro, luego usar un imán grande para atrapar a la presa.

[Mostrar](#) [Editar](#)



Piedra gigante

Atar una piedra gigante con una cuerda, elevarla 3 metros y poner comida debajo.

[Mostrar](#) [Editar](#)

Agregar Candidato

Agregar Candidato

Nuevo candidato

Los candidatos, forman parte de las n-uplas de los procesos (por ejemplo una terna de nominados, tiene 3 candidatos).

Pertenece al proceso: ACME Prize

Para la categoría: Mejor idea

Postulante / Nominado
ingrese el nombre (o una identificación) para el postulante, que verá el público.

BIOs
ingrese una descripción sobre el postulante (candidato, nominado) para que vea el público.

Avatar usuario:
 Examinar... No se seleccionó un archivo.

[Guardar Cambios](#)

[Ir a Candidatos](#)

Fig. 40: Consulta categoría "Mejor idea". Fuente: Elaboración propia.

Fig. 41: Formulario para crear un candidato. Fuente: Elaboración propia.

Lista de Candidatos

| Nombre | Bios | Url Avatar | Cod. Categoría | |
|-------------------|--|---|----------------|---------------------------|
| Tunel falso | Pintar un tunel falso sobre una pared, y luego redirigir las líneas del tráfico hacia él. | /images/uploads/avatars/candidatos/21.png | 5 | Mostrar Editar Eliminar |
| Comida con hierro | Colocar comida con pelotitas de hierro, luego usar un imán grande para atrapar a la presa. | /images/uploads/avatars/candidatos/20.png | 5 | Mostrar Editar Eliminar |
| Piedra gigante | Atar una piedra gigante con una cuerda, elevarla 3 metros y poner comida debajo. | /images/uploads/avatars/candidatos/19.jpg | 5 | Mostrar Editar Eliminar |
| Caida 2 | Alas quemadas por rayo | /images/uploads/avatars/candidatos/27.jpg | 4 | Mostrar Editar Eliminar |

Fig. 42: Lista de candidatos. Fuente: Elaboración propia.

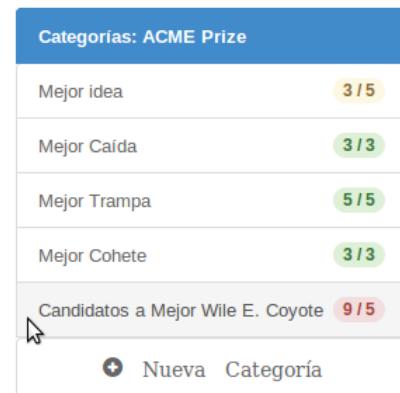


Fig. 43: Menú lista de categorías. Fuente: Elaboración propia.

4.6.4.2.1 Menú lista de categorías.

Para cada proceso se muestra un listado de las categorías que lo forman, tanto en el menú de navegación lateral como en la descripción del proceso.

Junto con el nombre de la categoría se muestra una etiqueta de color a la derecha. Si la categoría está completa será de color verde, si incompleta será amarilla y si está excedida en la cantidad de candidatos, será roja; como se aprecia la imagen (Véase 43). Además se agrega una referencia numérica de 2 números "X/Y", donde X representa la cantidad de candidatos asociados actualmente a la categoría e Y la cantidad de plazas de la categoría.

4.6.4.3. Candidatos.

Un usuario organizador puede crear candidatos para una categoría, que debe estar previamente seleccionada. Luego ir presionando los botones "Agregar candidato" y completando el formulario, como el representando en la imagen (Véase Fig. 41). En el mismo se informa para qué proceso y qué categoría se agregará el candidato. Para el alta de un candidato los datos a ingresar serán: el nombre, una descripción y una imagen (que se utilizará para representar al candidato).

4.6.4.3.1 Listado de candidatos.

Otra forma de acceder a los candidatos es a través del menú "Candidatos" de la barra de navegación lateral, este muestra una grilla (interfaz CRUD) con los candidatos asociados al proceso de selección actual, como se muestra en la imagen (Véase Fig. 42)

4.6.4.4. Padrón

El padrón es el listado de usuarios registrados que pueden emitir su voto en el proceso de selección asociado al padrón. Este listado es administrado por el usuario organizador dueño

del proceso.

Se accede a través del menú de navegación lateral “Padrón”. Desde allí podrá controlar quien puede votar en el proceso seleccionado, autorizando o rechazando las peticiones de participación como jurado.

En la imagen (Véase Fig. 44) se puede ver el padrón, representado como un listado. En este ejemplo hay 8 electores (jurados) asociados: de los cuales 5 fueron aprobados, 1 rechazado y 2 esperan por aprobación (los ítems 6 y 7).

4.6.4.4.1 Aprobar o rechazar electores.

Para aprobar o rechazar la solicitud de participación de un jurado, el usuario organizador solo debe presionar sobre el botón verde con la tilde para aceptar, o el botón naranja con la X para rechazar. Luego deberá confirmar la operación desde el mensaje modal que se le presentará en ambos casos con los datos del elector, como se aprecia en la imagen (Véase Fig. 45).

4.6.4.4.2 Agregar elector.

Para agregar un elector, deberá presionar el botón “Aregar Elector”. Luego se le mostrará una interfaz desde donde podrá agregar seleccionado desde una lista o buscando los usuarios registrados, en la aplicación, por su correo electrónico (que es un dato único), como se aprecia en la imagen (Véase Fig. 46).

Lista de Electores

| Nombre | email | Cod. Proceso | Estado | |
|-------------------|-----------------|--------------|-----------|----------|
| Gregorio Martinez | unemail@go.com | 1 | aprobado | Eliminar |
| Ramon Ortega | beto@ortega.com | 1 | aprobado | Eliminar |
| Gregorio Acuña | jurado1@go.com | 1 | aprobado | Eliminar |
| Gregory Acuña | jurado2@go.com | 1 | aprobado | Eliminar |
| Ramon Izidor | unemail1@go.com | 1 | | Eliminar |
| Gregorio Delmonte | unemail2@go.com | 1 | | Eliminar |
| Gregorio Perez | unemail3@go.com | 1 | rechazado | Eliminar |
| Micaëla Acuña | jurado3@go.com | 1 | aprobado | Eliminar |

+Agregar Elector

Fig. 44: Lista de electores. Vista reducida. Fuente: Elaboración propia.

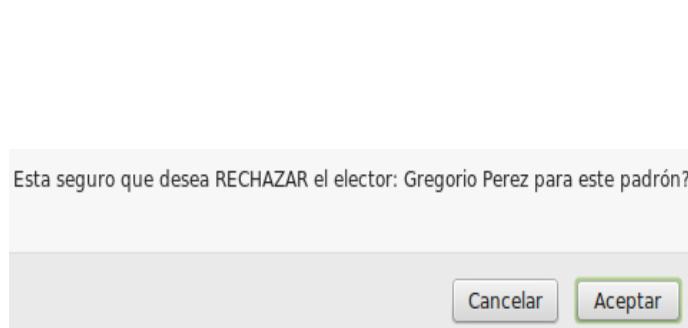


Fig. 45: Ventana modal de confirmación. Fuente: Elaboración propia.

Fig. 46: Búsqueda elector. Vista reducida. Fuente: Elaboración propia.

4.6.5 Votación.

Es la forma en que un elector hace manifiesta su preferencia por un candidato de una categoría, por única vez de forma anónima e irreversible.

4.6.5.1. Votar.

Si el usuario es un jurado podrá ingresar a la votación desde su escritorio, presionando el botón verde “Votar” que aparecerá oportunamente sobre los procesos en los que solicito participar. Como se aprecia en a la izquierda de la imagen (Véase Fig. 34).

Sino también podrá ingresar desde el menú “Inicio”; allí podrá ver una grilla con todos los procesos, como se aprecia en la imagen (Véase Fig. 37), donde también aparecerá un botón “Votar” de color verde, cuando el usuario este habilitado en el padrón del proceso.

4.6.5.2 Postulación como jurado.

Ingresando por el menú “Inicio”, el usuario puede ver una tabla (Véase Fig. 37) con todos los procesos en los que se podría llegar a participar y presionando el botón “solicitar autorización” puede postularse para ser aceptado como jurado en ese proceso de selección.

4.6.5.3. Emitir voto.

Una vez presionado el botón verde “Votar”, la aplicación se redirigirá hacia la página de votación. En esta mostrará la información básica del proceso de selección indicado: una descripción con las fechas claves, información sobre el organizador, una forma de contactarlo y el logotipo del organizador, como se aprecia en la imagen (Véase Fig. 47).

Luego el usuario tendrá que ir seleccionado la categoría en la que desea emitir un voto, presionando sobre los botones naranja con el nombre de la categoría; en la parte inferior de la imagen (Véase Fig. 47) se ven estos botones previos a realizar el voto.

ACME Prize

Se realiza en Desierto de Arizona.
Tiene una duración de 180 días. Empieza el 2014-03-13 hasta el 2014-12-13.

Es organizado por ACME

Tipo: premio

Estado: abierto

[Más info.](#) [Contactar](#)



Candidatos a Mejor idea

| | | |
|---|---|--|
| Piedra gigante Atar una piedra gigante con una cuerda, elevarla 3 metros y poner comida debajo. i&#9658; Piedra gigante | Comida con hierro Colocar comida con pelotitas de hierro, luego usar un imán grande para atrapar a la presa. i&#9658; Comida con hierro | Tunel falso Pintar un tunel falso sobre una pared, y luego redirigir las líneas del tráfico hacia él. i&#9658; Tunel falso |
|---|---|--|

Votar seleccionado

Fig. 47: Inicio votación. Vista reducida. Fuente: Fig. 48: Candidatos a "Mejor idea" durante Elaboración propia.

Una vez que presione sobre alguno de ellos, la pantalla cambiará y se desplegará la información sobre los candidatos de la categoría seleccionada. En la imagen (Véase Fig. 48) se ven los candidatos luego de presionar sobre la categoría “Mejor idea” y antes de que seleccione alguno de ellos.

Para emitir su voto sobre la categoría, el usuario, deberá seleccionar uno de los candidatos, presionando sobre alguno de los botones con el nombre del candidato. Entonces cambiará su color indicando la selección, como se ilustra en la imagen (Véase Fig. 49) donde está seleccionado el segundo candidato.

Luego de seleccionar un candidato, para emitir el voto deberá presionar el botón verde “Votar seleccionado” que se ubica al final de la lista de candidatos, como se aprecia en la imagen (Véase Fig. 49).

Debido a que el voto es anónimo e irreversible se le pide al usuario que confirme la operación a través de un mensaje en pantalla, como se puede apreciar en la imagen (Véase Fig. 50).

Candidatos a Mejor idea

| | | |
|---|---|--|
| Piedra gigante Atar una piedra gigante con una cuerda, elevarla 3 metros y poner comida debajo. i&#9658; Piedra gigante | Comida con hierro Colocar comida con pelotitas de hierro, luego usar un imán grande para atrapar a la presa. i&#9658; Comida con hierro | Tunel falso Pintar un tunel falso sobre una pared, y luego redirigir las líneas del tráfico hacia él. i&#9658; Tunel falso |
|---|---|--|

Votar seleccionado

Fig. 49: Candidato seleccionado para votar por él. Fuente: Elaboración propia.

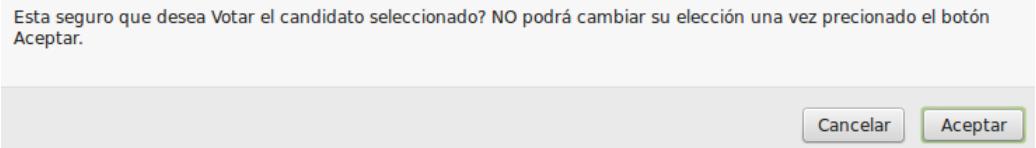


Fig. 50: Mensaje confirmación de emisión de voto. Fuente: Elaboración propia.



Fig. 51: Mensaje tras voto registrado correctamente. Fuente: Elaboración propia.



Fig. 52: Categorías deshabilitadas tras votación. Fuente: Elaboración propia.



Fig. 53: Botones sociales. Fuente: Elaboración propia.

Luego de la confirmación, el voto se registra anónimamente y se deshabilita la categoría para que ya no pueda volver a votar; El sistema registra en que categorías emitió voto el usuario pero no registra a quién voto. Se renueva la pantalla, se muestra al usuario un mensaje de confirmación como el de la imagen (Véase Fig. 51).

A medida que vaya emitiendo su voto en cada categoría estas se irán deshabilitando, hasta que al finalizar quedaran todas deshabilitadas, con un ícono con tilde, como se ve en la imagen (Véase Fig. 52).

4.6.5.4. Compartir.

Al pie de página, el usuario puede seleccionar diferentes botones para compartir su participación en el proceso de selección en diferentes redes sociales, como se puede apreciar en la imagen (Véase Fig. 53)

4.6.5.5 Ver resultados.

En esta edición, los resultados serán privados para el organizador. En futuras versiones se podrá implementar y mejorar la forma de mostrar los resultados al jurado y al público.

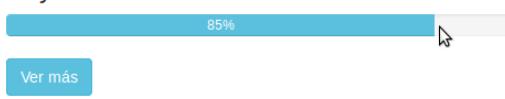
Una vez cerrado un proceso de selección se habilitará la visualización del escrutinio y el usuario podrá ver los resultados.

En una primera vista, como se aprecia en la imagen (Véase Fig. 54), el escrutinio muestra unas barras de progreso, que representan el grado de avance de las votaciones; a nivel general, arriba en azul, y por cada categorías a medida que se desplace hacia abajo en la interfaz.

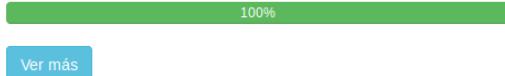
Escrutinio



Categoría: Candidatos a Mejor Wile E. Coyote



Categoría: Mejor Cohete



Categoría: Mejor Cohete

100%

Ver más

Puesto Nro. 1 | ACME Giant Firecrackers | Obtuvo 3 votos.
Puesto Nro. 2 | ACME Rocket Sled | Obtuvo 2 votos.
Puesto Nro. 3 | ACME Missile-Bombs | Obtuvo 2 votos.

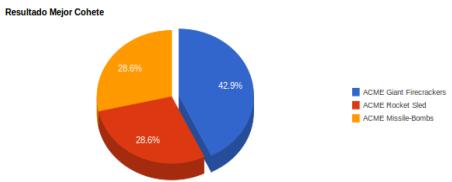
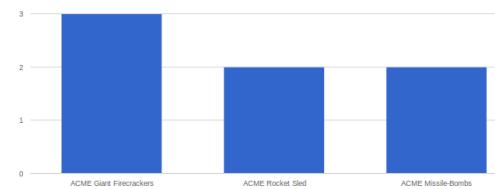


Fig. 54: Escrutinio. Fuente: Elaboración propia.

Fig. 55: Resultados categoría. Fuente: Elaboración propia.

Estas barras tendrán diferentes colores, según sea el caso:

- Azul o celeste: proceso incompleto, falta la emisión de algún voto.
- Verde: proceso completo, han emitido voto todos los electores.
- Rojo: sobran o faltan votos, es decir no coincide la cantidad de votos con la cantidad de electores habilitados en el padrón. Esto se puede dar en el caso que se agreguen o quiten electores, una vez finalizado el proceso.

Luego para ver los resultados de cada categoría el usuario organizador debe presionar el botón "ver más" de cada una de ellas.

Ver más.

Allí se despliegan y presentan los datos relativos a la categoría en varios formatos: texto, gráfico de barras y gráfico de torta.

Datos para Auditoría.

Se muestran los datos en crudo sobre los cuales se hicieron los cálculos. En el caso de querer ver esto mismo para todo el proceso, se accede con el botón “ver votos” al final del escrutinio.

En la imagen (Véase Fig. 56) se aprecian los datos en crudo para la categoría “Mejor cohete”, estos datos son los mostrados en los gráficos de la Fig. 55.

Ver Votos.

Al presionar este botón, muestra los todos datos en crudo sobre los cuales se realizaron los cálculos de todo el proceso de selección.

| <p>Datos para auditoría:</p> <pre>Recuento votos categoría: [[{"id": "22", "votos": 3}, {"id": "23", "votos": 2}, {"id": "24", "votos": 2}]]</pre> <pre>Votos categoría:#<ActiveRecord::Relation [#<Ballot id: 2, selection_process_id: 1, category_id: 2, candidate_id: 22, digital_signature: nil, created_at: "2014-10-21 13:38:45", updated_at: "2014-10-21 13:38:45">, #<Ballot id: 7, selection_process_id: 1, category_id: 2, candidate_id: 23, digital_signature: nil, created_at: "2014-10-22 11:51:09", updated_at: "2014-10-22 11:51:09">, #<Ballot id: 11, selection_process_id: 1, category_id: 2, candidate_id: 23, digital_signature: nil, created_at: "2014-10-22 11:52:14", updated_at: "2014-10-22 11:52:14">, #<Ballot id: 15, selection_process_id: 1, category_id: 2, candidate_id: 22, digital_signature: nil, created_at: "2014-10-22 11:53:34", updated_at: "2014-10-22 11:53:34">, #<Ballot id: 19, selection_process_id: 1, category_id: 2, candidate_id: 22, digital_signature: nil, created_at: "2014-10-22 11:54:45", updated_at: "2014-10-22 11:54:45">, #<Ballot id: 23, selection_process_id: 1, category_id: 2, candidate_id: 24, digital_signature: nil, created_at: "2014-10-22 11:55:35", updated_at: "2014-10-22 11:55:35">, #<Ballot id: 26, selection_process_id: 1, category_id: 2, candidate_id: 24, digital_signature: nil, created_at: "2014-10-22 11:56:50", updated_at: "2014-10-22 11:56:50">]]></pre> | <p>Procesos</p> <p>1. Premio - 2. Certamen 3. Votación</p> <table border="1"> <thead> <tr> <th>Categoría</th> <th>Cantidad</th> <th>Porcentaje</th> </tr> </thead> <tbody> <tr> <td>premio</td> <td>6</td> <td>85.7%</td> </tr> <tr> <td>certamen</td> <td>1</td> <td>14.3%</td> </tr> </tbody> </table> | Categoría | Cantidad | Porcentaje | premio | 6 | 85.7% | certamen | 1 | 14.3% |
|---|--|------------|----------|------------|--------|---|-------|----------|---|-------|
| Categoría | Cantidad | Porcentaje | | | | | | | | |
| premio | 6 | 85.7% | | | | | | | | |
| certamen | 1 | 14.3% | | | | | | | | |
| <p>Fig. 56: Datos para auditoría. Fuente: Elaboración propia.</p> | <p>Fig. 57: Estadísticas básicas de la aplicación. Fuente: Elaboración propia.</p> | | | | | | | | | |

4.6.5.6 Abrir y cerrar votaciones.

Para que los jurados puedan votar el organizador deberá abrir las elecciones manualmente, para ello deberá cambiar manualmente el estado del proceso a “abierto”; de igual manera para terminar el proceso de votación deberá cambiarlo a “cerrado”.

Esto se hace editando el proceso, cambiando el estado y luego guardando los cambios; para ello se facilitan 2 enlaces “abrir elecciones” y “cerrar elecciones” desde el panel del proceso para el organizador (Ir a “Mis procesos” y luego a una opción “Mostrar proceso” de un proceso de la grilla).

4.6.6 Estadísticas.

Desde el menú de navegación, a la izquierda, los usuarios también tiene acceso a estadísticas básicas sobre la utilización del sistema, que son una serie de gráficos (de torta

y de barras) como se aprecia en la imagen (Véase Fig. 57).

4.7 Comentarios y discusiones del capítulo.

En el presente capítulo se han descripto los resultados logrados en este trabajo, los que permiten demostrar que los objetivos planteados inicialmente se han cumplido en su totalidad, obteniendo un resultado satisfactorio.

4.6.1 Sobre las historias de usuario.

Su utilización en el proceso de software fue guiando la actividad de desarrollo, de modo que se tuvo la respuesta a "¿Y ahora qué sigue?" al saber que se tenía que pasar todos los escenarios, y sus pasos, planteados en la historia de usuario para que la funcionalidad sea aceptada por el cliente.

4.6.2 Sobre la aplicación de BDD.

Las pruebas de integración, de comportamiento con Cucumber, dieron la posibilidad de comprobar el funcionamiento del sistema en todo momento, que permitió evaluar el impacto de un cambio.

Por lo general las pruebas de comportamiento (cukes) fallaron primero, o más rápidamente, que las unitarias y de más bajo nivel (RSpec); ya que en la interacción entre partes se produjeron fallas o problemas que no habían sido contemplados a nivel unitario.

4.6.3 Sobre la aplicación de TDD.

Al igual que con las pruebas de integración, las pruebas unitarias generadas tras la aplicación de TDD dieron la posibilidad de comprobar el funcionamiento del sistema en todo momento, resultando muy útil al momento de evaluar el impacto de un cambio, ya sea por la refactorización o por cambios en otras partes de la aplicación.

La primera impresión, tras aplicar TDD, fue que generaba una aparente sobrecarga de trabajo; por ejemplo cuando se repite el tipo de pruebas entre Cucumber y RSpec, que evalúan el mismo comportamiento esperado de la aplicación. Pero al crear pruebas que evalúan otro tipo de interacciones y alternativas que no tienen que ver con el comportamiento observado por el usuario sino el comportamiento esperado del código bajo estudio; como ser interacciones entre clases, interfaces entre capas y utilización de sistemas de terceros se verificó la diferencia entre ambos enfoques.

Una dificultad con respecto a TDD fue pensar primero en la prueba, e imaginar cómo es el

código que se espera obtener (en la jerga "Code we wish we had" o CWWWH) en una nueva tecnología o herramienta. Es de esperarse que al trabajar sobre una tecnología conocida por el equipo de desarrollo esta dificultad no se presente o sea de menor impacto.

4.6.4 Sobre las pruebas

La fragilidad de las pruebas, que significa que estas dejan de funcionar ante cambios en el código que evalúan, se hizo notar a medida que se avanzó con la aplicación de BDD, TDD y cuando se incluyeron nuevas funcionalidades a la aplicación. Por momentos se notó el esfuerzo doble, de mantener las pruebas y mantener el código; esto provocaba que no se notaría el avance inmediato en nuevas funcionalidades para el usuario, situación que también se ve reflejada en la tabla de iteraciones (Véase tabla 2) con la disminución de la velocidad.

Cuando se integró la gema Devise, las pruebas incluso rotas ayudaron a detectar errores de comportamiento. Por ejemplo en el panel del administrador que no se mostraba.

4.6.5 Sobre los errores.

Al momento de producirse los errores, o encontrarse defectos, las herramientas utilizadas ayudaron a encontrar exactamente el lugar donde está fallando el código o la prueba y facilitaron la corrección del mismo.

Algunas pruebas ayudaron a encontrar errores en etapas tempranas. Por ejemplo un error de ortografía que no se había localizado, en un mensaje, hasta avanzado el proyecto (error sistemático) se presentó en una prueba que comprobaba un mensaje. Este error nunca llegó a producción.

4.6.5.1 Riesgo de errores.

Aunque las pruebas de aceptación (ejemplos) cubran las situaciones normales y casos extremos, aun se pueden presentar errores. Por ejemplo en la aplicación no se prueba una ruta a un ID que no existe (http://tfa-vox.herokuapp.com/selection_processes/9999) y esto genera un error 404: "We're sorry, but something went wrong.". Esto es porque las pruebas generadas con BDD y TDD no cubre un escenario donde un usuario malintencionado cambia manualmente el ID del recurso en la dirección web o URL.

Al avanzar con el desarrollo se pudo detectar errores y detalles del comportamiento que se escapaban a las pruebas. Por ejemplo, el nombre de clase de una tabla en HTML de una de las vistas, no es un error que el usuario pueda observar a simple vista, pero podría generar

errores internos, o errores de personalización, que pasen desapercibidos a simple vista. Entonces, si bien se utilizan pruebas los errores pueden permanecer ocultos ya que la metodología es un proceso de diseño y no de verificación y validación.

Otra situación que se presentó fue cuando la herramienta que se usa para realizar las pruebas no leía las "variables de sesión de usuario", entonces las pruebas fallaban porque no se podía leer estas variables. Para que las pruebas pasen se podía evitar esos caminos lógicos que contemplen esas variables de sesión o se pusieron excesivas condiciones lógicas, finalmente se logró solucionar el inconveniente agregando lógica al motor de pruebas. Es decir se tuvo que realizar un esfuerzo extra no previsto.

4.6.5.2 Errores ante cambio.

Los problemas asociados a los cambios en la estructura interna de la aplicación estuvieron presentes durante el presente trabajo, como se comenta a continuación:

Por ejemplo al crear el modulo “procesos de selección” (selection_process) se produjó una equivocación humana en la creación y se puso el nombre en plural: *selection_processes* en lugar de *selection_process*, por la forma que trabaja el framework utilizado esto tuvo un impacto en todo lo relacionado a esa entidad, en las 3 capas: modelo, controlador y vista; Y las pruebas asociadas. Una vez detectado, para subsanar la omisión se tuvo que cambiar a singular: *selection_process*. *En ese momento, naturalmente*, los errores empezaron a aparecer y las pruebas a fallar. El saber cuáles pruebas fallaban y cuáles no, ayudo a medir el impacto de cambio introducido y a solucionar también los problemas en otras partes del sistema asociadas, como ser la configuración de las rutas. En este caso se evidenció la fragilidad de las pruebas ante cambios en el código: en Cucumber fueron 25 pruebas rotas y en RSpec fueron más de 46. Afecto a otros modelos asociados como por ejemplo *organizer*.

Lo interesante de este caso fue encontrar 1 a 1 los puntos de fallo en la aplicación fácilmente, cuando los mismos errores informados por el intérprete del lenguaje no lo hacen de una forma tan clara y precisa, por lo menos en el servidor de desarrollo utilizado.

Otro ejemplo es cuando se tuvo que cambiar un método de autorización, se cambió de “*is_organizer?*” a “*is_owner?*”, en toda la aplicación, otra vez las pruebas ayudaron a comprobar que todo siga funcionando bien luego del cambio.

A priori, resultó difícil medir el impacto del cambio. Que las pruebas fallen ante el cambio no significa, del todo, que estas estén rotas por completo; en algunas ocasiones sucedió que luego de corregir cierta parte del código, por una prueba que fallaba, otras pruebas que no

funcionaban por el impacto del primer cambio, volvieron a funcionar.

En la segunda parte del proceso de software seguido "módulos" que no tenían pruebas (módulos del front-end), se volvió a tener el problema de que la aplicación tenía fallas ante ciertos cambios, pero no había forma de saber hasta que alguien de alguna manera probaba ese camino y se producía el fallo. Es decir sin el alerta temprano de las pruebas ¿Cuál es la alternativa para evitar que las fallas lleguen al usuario sin recurrir a un enfoque clásico? queda planteada la inquietud. Acostumbrarse a tener una retroalimentación inmediata por las pruebas y luego salirse de esa retroalimentación, sirvió para reforzar la idea de los beneficios de aplicar las técnicas.

4.6.5.3 Falsos Positivos.

De la experiencia con BDD, se pueden tener falso positivos. Por ejemplo se tenía que hacer clic para borrar, pero la prueba no hacia el clic y borraba el registro, entonces la prueba daba verde y continuaba con el siguiente paso. Esta daba verde pero no estaba probando lo que tenía que probar, fue hasta avanzado el proceso que se notó esta falencia al estar haciendo refactorizaciones de las pruebas.

Otro ejemplo de falso positivo por un mal diseño de la prueba, fue cuando: en la *features* "listado_entidades_organizadoras.feature" se tenía una tabla con 1 sola entrada en lugar de 2 entradas, entonces todo estaba en verde, pero en realidad no estaba probando que tenía que tener 2 registros en la tabla. Estaba probado que las tablas coincidan en contenidos, entre la tabla de ejemplos y la tabla creada. Si bien estaba bien la funcionalidad, la prueba era un falso positivo porque en caso de 1 registro sólo la prueba tendría que fallar.

Un problema que se detectó es que las pruebas de comportamiento pueden pasar incluso cuando un camino lógico es incorrecto, por ejemplo una bifurcación lógica que daba falso en situaciones que se esperaba sea verdadero. Entonces, si la prueba de Integración solo comprueba que la información se muestre sin analizar su contexto y no se prueban todos los caminos, la falla puede seguir estando presente sin ser detectada. En cambio la prueba de más bajo nivel debería probar necesariamente esa bifurcación y comprobar ambos caminos en varias pruebas distintas.

Otro ejemplo, fue cuando no se hacía el despliegue de la aplicación en Travis-ci por una prueba que fallaba; luego se detectó que la prueba fallaba por un error en ella misma y que la aplicación andaba bien.

4.6.6 Sobre la integración continua.

Delegar la tarea de desplegar, esto es: ejecutar las pruebas, crear un entorno limpio, actualizar los archivos de la aplicación, actualizar la base de datos, iniciar la nueva versión; en un servicio que lo haga de forma automatizada produjo un ahorro significativo de esfuerzo y tiempo. Además fue una experiencia muy enriquecedora para el desarrollador.

Al integrar la gema Devise en la aplicación, este cambio impactó en varias partes del sistema provocando lógicamente muchos errores. Estos errores nunca fueron vistos por los usuarios finales de la aplicación porque Travis-ci no hacia el despliegue en Heroku ya que fallaban las pruebas. Una vez solucionados los usuarios tuvieron nuevas funcionalidades funcionando, como se espera que lo hagan, en armonía con las anteriores.

Aunque se tuvo situaciones de falsos positivos, como se comentó anteriormente (Cfr. pág. 78), fueron muy pocas y no por un error del servicio utilizado.

4.6.7 Sobre la refactorización.

Como parte de las prácticas de BDD y TDD, se aplicaron refactorizaciones al código cuando fue posible. En general este paso resultó difícil porque requiere conocimiento sobre la tecnología utilizada, sobre buenas prácticas y principios de diseño de software para aprovecharla.

4.6.8 Sobre la utilización del framework Rails.

Para entender mejor cómo funciona el framework, antes de avanzar con las funcionalidades del sistema, uno de los módulos se creó sin utilizar las herramientas que este incluye para la construcción rápida de aplicaciones. De esta experiencia se puede decir que efectivamente al no utilizar el framework se percibe el incremento de trabajo, en tiempo y esfuerzo. De allí surge la principal ventaja de utilizarlo como una herramienta para realizar tareas comunes y repetitivas.

Para aplicar TDD, en el caso del framework, se verificó que es condición necesaria conocerlo en primer lugar y segundo lugar conocer el patrón MVC, su funcionamiento e implementación en esta herramienta, para entender cómo deberían funcionar las cosas. Por ejemplo al crear una acción en un modelo, se puede partir de la vista, continuar con el controlador y llegar con pruebas hasta el método del modelo.

En un principio al seguir TDD se escribieron pruebas unitarias que evaluaban el comportamiento normal de funcionalidades que provee el framework luego se entendió que esto generaba un esfuerzo injustificado ya que se debería suponer que el framework hace

bien su trabajo y el esfuerzo debería de ponerse en la producción que realiza el desarrollador para ajustar la aplicación a las reglas de negocio del problema que trata de resolver.

4.6.9 Dificultades

Durante la ejecución surgieron algunas dificultades, que se comentan a continuación:

Dificultades con la estimación del esfuerzo: debido al desconocimiento de la tecnología la estimación, para cada historia de usuario, las estimaciones no fueron precisas. Las tareas pensadas como difíciles consumieron menor cantidad de tiempo y las más sencillas llevaron el triple de lo que se pensaba. Por ejemplo llevo 1 semana de trabajo para lograr hacer la funcionalidad de subir una imagen para el logotipo de la organización, siguiendo TDD; en este caso el esfuerzo se dio en seleccionar técnicamente como se solucionaría la funcionalidad, hacer alguna aproximación, luego hacer las pruebas primero y luego desarrollar la funcionalidad en la aplicación.

Dificultad para las decisiones técnicas: se hizo difícil elegir las gemas a utilizar debido a no tener un conocimiento profundo de la tecnología. Por ejemplo una gema que maneje lo relacionado con las imágenes que subirán los usuarios del sistema.

Al no poseer el conocimiento técnico adecuado resultaba difícil contestar estas interrogantes ¿Dónde poner el foco en los test? ¿Testear absolutamente todo?, luego con el uso de framework ¿Vale la pena testear lo que hace el framework?

Esfuerzo extra: un ejemplo del esfuerzo extra que derivó de la aplicación de BDD y TDD fue cuando se tenía que indagar sobre como agregar cierta gema (librería), no solo había que entender el funcionamiento básico de esta y como agregarla al proyecto, sino también había que averiguar cómo hacer las pruebas en Cucumber y en RSpec lo cual consumía su tiempo y esfuerzo extra. Esto último costaba más generalmente que el esfuerzo inicial de comprender como funciona y como integrarla al proyecto.

Capítulo 5. Conclusiones y futuros trabajos

Con respecto a los objetivos fijados al inicio del trabajo:

Se realizó la revisión bibliográfica, encontrando 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 inmediata, de las pruebas automatizadas para realizar cambios y solucionar problemas generando un código modificable más fácilmente.

Además, durante el desarrollo del trabajo, se pudo comprobar algunas de las ventajas que ofrece BDD y TDD, estas son:

- Clarifican los criterios de aceptación de cada requerimiento a implementar.
- En TDD, al hacer los componentes fáciles de probar tiende a bajar el acoplamiento entre los mismos.
- 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.
- Un gran 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 un sistema 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 destacan una serie de conclusiones sobre algunas de las temáticas específicas mencionadas en el trabajo:

Sobre las técnicas aplicadas.

Si se considera la aplicación de estas prácticas se debe mantener siempre presente que en

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.

En general la dificultad provenía del desconocimiento de las herramientas y de la tecnología utilizada, se verificó entonces la necesidad de planificar las realización de spikes como recomienda Blé [10] y Beck [2]. Esto es al utilizar una tecnología desconocida se recomienda realizar pequeños experimentos, aparte del proyecto, a los que se los llama “spike” 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, ya que se tiene en cuenta el comportamiento esperado desde el punto de vista del usuario, que generalmente son acciones simples y no como se implementa internamente.

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 en el corto plazo (lo inmediato) es cierto, 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 de des-actualización que la documentación tradicional y ya no reflejará el comportamiento esperado del sistema.

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 verificó que existe una sobrecarga, de tener que programar acciones de forma distinta, para escribir las pruebas, 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. Pero la prueba desde Cucumber/Capybara y luego RSpec, para probar este comportamiento resultado complicada y un consumo de tiempo excesivo. Sin embargo son los beneficios que proporcionan estas 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.

Sobre las historias de usuario.

Respecto a las historias de usuario, un aspecto clave es la definición de los escenarios para cada historia de usuario. De allí surge el 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, estos 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.

Sobre la aplicación de BDD.

Las de comportamiento con Cucumber, ofrecen de una manera muy simple y precisa la posibilidad de comprobar el funcionamiento del sistema si surgen cambios, por ejemplo en una actualización del framework Rails. En lenguajes como Ruby y herramientas como Rails donde generalmente hay problemas de compatibilidad hacia atrás, esto resulta muy útil para el mantenimiento de la aplicación.

Un problema que se detectó es que las pruebas de comportamiento pueden pasar incluso cuando un camino lógico sea incorrecto, como se comentó anteriormente (Cfr. pág. 78).

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.

Sobre las pruebas.

Se pudo comprobar que a mayor impacto del cambio, mayor defensa 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 ya 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.

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 estas técnicas, durante algunas iteraciones, se vuelve natural leerlos con más tranquilidad. La facilidad de detectar los errores se incrementa a medida que el desarrollador se acostumbra a la forma de trabajar.

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. Esto 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 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, pero cuando en su concepción no se tuvieron presentes los caminos alternativos, casos extremos o incluso factores externos, los errores se harán presentes. La responsabilidad de las prácticas es limitada.

Sobre la integración continua.

La experiencia real al aplicar integración continua 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.

La integración continua permite detectar de manera temprana posibles problemas de integración que pueden tener soluciones muy complejas a posteriori. Además permite un ahorro significativo de esfuerzo al automatizar tareas repetitivas.

A medida que se van aplicando las prácticas ágiles propuestas en el desarrollo cotidiano se irá conformando el entorno de integración continua dirigido a la obtención de un producto de mayor calidad.

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.

La refactorización es un tema amplio, del que existe bibliografía específica, que excede a los alcances del presente trabajo y se recomendará profundizar en el futuro.

Sobre la utilización del framework Rails.

De esta experiencia realizada surge que la principal ventaja es su utilización como una herramienta para realizar tareas comunes y repetitivas. El framework libera al desarrollador de pensar y consumir energía en algunas decisiones triviales o que están fuera de su alcance, e incluso que están fuera de los alcances económicos de un proyecto normal. Por ejemplo, la comunicación con la base de datos, la comunicación entre capas, la seguridad, los helpers de cada capa, validaciones comunes, etc. Su utilización genera un incremento en la productividad del desarrollador que le permite enfocarse en las cuestiones del negocio solamente.

5.2 Futuros trabajos.

Se consideran como las líneas futuras de trabajo derivadas del presente trabajo las siguientes:

- Profundizar sobre la práctica ágil de refactoización, pudiendo aplicar diferentes tipos de ella a este trabajo.
- Utilizar herramientas o servicios para analizar el código fuente y refactorizar donde sea posible, como por ejemplo CodeClimate o CodeReview.
- Realizar un análisis del impacto en los recursos humanos al aplicar estas técnicas.
- confeccionar una metodología para su implementación en una empresa.
- realizar la experiencia de aplicar las técnicas a sistemas legacy (legacy code), por ejemplo sobre los módulos que actualmente no tienen pruebas.
- Comparar el enfoque Top-down, con el Button-up aplicando TDD/BDD
- aplicar técnicas y herramientas para acelerar y optimizar la ejecución de los test en todos sus niveles.

Con respecto a la aplicación generada, se propone:

- Agregar funcionalidades en lo relativo a los Certámenes.
- Mejorar la visibilidad y la forma en que se muestran los resultados.

- Agregar un sistema de plantillas, para la creación de los eventos.
- Mejorar la visualización del estado de un proceso, las tareas realizadas y las que faltan realizar.
- Mejorar el diseño gráfico y la experiencia de usuario.
- Agregar controles extra, por ejemplo que sólo cuando todas las categorías estén en verde se pueda abrir una elección.
- Notificar a usuarios por email, ante cambios en estado de un proceso.

Se utilizará la aplicación en la próxima edición de los Premios TIC de la organización ComunidadTIC, sin fecha estipulada.

Referencias

1. C. Fontela. "Estado del arte y tendencias en Test-Driven Development", Facultad de Informática, Universidad Nacional de La Plata, La Plata, Argentina, 2011. cap.
2. K. Beck, "Extreme Programming Explained: Embrace Change", Segunda Edición. Boston, MA, USA: Addison-Wesley Professional, 1999.
3. K. Beck, "Test Driven Development: By Example", Primera Edición. Boston, MA, USA: Addison-Wesley Professional, 2002.
4. RSpec Development Team, USA, "RSpec," Disponible: <http://rspec.info/>. Consultado el: Otc. 01, 2014.
5. R. Devies. (2006). Connextra User Story 2001: ConnextraStoryCard. [Online]. Disponible: http://agilecoach.typepad.com/photos/connextra_user_story_2001/connextrastorycard.html, Consultado el: Nov. 25, 2013.
6. D. North. (2006). Introducing BDD [Online]. Disponible: <http://dannorth.net/introducing-bdd/>, Consultado el: May. 8, 2013.
7. D. Patterson y A. Fox, "*Engineering Software as a Service: An Agile Approach Using Cloud Computing*", Primera Edición. San Francisco, CA, USA: Strawberry Canyon LLC, 2014.
8. M. Gärtner, "ATDD by Example: A Practical Guide to Acceptance Test-Driven Development", Primera Edición. Boston, MA, USA: Addison-Wesley Professional, 2012.
9. D. Chelimsky, "The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends", Primera Edición. Dallas, TX, USA: The Pragmatic Programmers LLC., 2011.
10. C. Blé Jurado. "Diseño Ágil con TDD", Primera Edición. Madrid, España: iExpertos, 2010.
11. H. Lopes Tavares, G. Guimarães Rezende, V. Mota dos Santos, R. Soares Manhães y R. Atem de Carvalho, "A tool stack for implementing Behaviour-Driven Development in Python Language", Núcleo de Pesquisa em Sistemas de Informação, Instituto Federal Fluminense (IFF), Campos dos Goytacazes, Brasil. 2010.

12. M. Fowler. (2006). Continuous Integration [Online]. Disponible:
<http://martinfowler.com/articles/continuousIntegration.html>, Consultado el: May. 8, 2013.
13. C. Fontela, "Cobertura entre pruebas a distintos niveles para refactorizaciones más seguras", Facultad de Informática, Universidad Nacional de La Plata, La Plata, Argentina, 2013.
14. D. Brown, "Communicating Design: Developing Web Site Documentation for Design and Planning", Segunda Edición. Berkeley, CA, USA: New Riders. 2011.
15. I. Sommerville, "Verificación y Validación" en "Ingeniería del Software" Novena Edición. Madrid, España: Pearson Educación S.A., 2005.
16. Pivotal Labs, Inc., San Francisco, CA, USA, "PivotalTracker," Disponible:
<http://www.pivotaltracker.com/>. Consultado el: Sep. 28, 2014.
17. Ruby, "Lenguaje de programación Ruby". Disponible: <https://www.ruby-lang.org/es/>. Consultado el: Sep. 23, 2014.
18. D. Hansson, "Ruby on Rails," [Online]. Disponible: <http://rubyonrails.org/>. Consultado el: Sep. 23, 2014.
19. Neurogami, Scottsdale, AZ., USA, "WEBrick," Disponible: <http://ruby-doc.org/stdlib-2.1.1/libdoc/webrick/rdoc/WEBrick.html>. Consultado el: Sep. 26, 2014.
20. Bootstrap, San Francisco, CA, USA, "Bootstrap.". Disponible: <http://getbootstrap.com/>. Consultado el: Sep. 9, 2014.
21. J. Spurlock, "Bootstrap", Primera Edición, Sebastopol, CA, USA: O'Reilly Media, Inc., . 2013.
22. SQLite Consortium, Charlotte, NC, USA, "SQLite,". Disponible:
<http://www.sqlite.org/about.html>. Consultado el: Sep. 25, 2014.
23. M. Owens, "The Definitive Guide to SQLite", Segunda Edición, Berkeley, CA, USA: Apress, 2010.
24. Oracle Corporation, Redwood, CA, USA, "MySQL Workbench," Disponible:
<http://www.mysql.com/products/workbench/>. Consultado el: Sep. 25, 2014.
25. S. Chacon, "Pro Git", Primera Edición. New York, NY, USA: Apress, 2009.

26. GitHub Inc., San Francisco, CA, USA, "GitHub," Disponible: <https://github.com/>. Consultado el: Sep. 26, 2014.
27. Lemur Heavy Industries LLC., Venice, CA, USA, "Coveralls.io," Disponible: <https://coveralls.io/>. Consultado el: Sep. 26, 2014.
28. Travis CI GmbH, Berlin, Germany, "Travis CI," Disponible: <https://travis-ci.org/>. Consultado el: Sep. 29, 2014.
29. Cucumber Ltd., Cairndow, Argyll, Scotland, "Cucumber," Disponible: <http://cukes.info/>. Consultado el: Sep. 30, 2014.
30. M. Wynne y A. Hellesøy, "*The Cucumber Book: Behaviour-Driven Development for Testers and Developers*", Primera Edición. Raleigh, NC, USA: Pragmatic Programmers, LLC., 2012.
31. Elabs AB, Göteborg, Sweden, "Capybara," Disponible: <http://jnicklas.github.io/capybara/>. Consultado el: Oct. 01, 2014.
32. T. Guillaume, La Chaux-de-Fonds, Switzerland, "Guard," Disponible: <http://jnicklas.github.io/capybara/>. Consultado el: Oct. 01, 2014.
33. Travis CI GmbH, Berlin, Germany, "Travis CI, Building a Ruby Project" Disponible: <http://docs.travis-ci.com/user/languages/ruby/> Consultado el: Sep. 29, 2014.
34. Heroku Inc., San Francisco, CA, USA, "Heroku," Disponible: <https://www.heroku.com/about>. Consultado el: Oct. 10, 2014.
35. A. Hanjura, "Heroku Cloud Application Development: A comprehensive guide to help you build, deploy, and troubleshoot cloud applications seamlessly using Heroku", Primera Edición, Birmingham, Reino Unido: Packt Publishing Ltd., 2014.
36. PostgreSQL Global Development Group, CA, USA, "PostgreSQL," Disponible: <http://www.postgresql.org/>. Consultado el: Sep. 25, 2014.
37. W. Humphrey, "Software Estimating" en "PSP: A Self-Improvement Process for Software Engineers", Primera Edición. Boston, MA, USA: Pearson Education, Inc., Massachusetts, 2005.