



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 Dapozo.

Profesor Coordinador: Mgter. Sonia Itatí Mariño.

Tribunal Evaluador:

Mettini, Aldo José.

Alfonzo, Pedro Luis.

Mariño, Sonia Itatí.

[Borrador | Presentación Preliminar | Presentación Definitiva]

Borrador v1.39.4 Modificación: 757

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 de 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 aplicar el lenguaje Ruby (con sus herramientas asociadas), conocido por ser muy utilizado en el ecosistema emprendedor internacional; pero contaba con poca documentación en español, lo que generó una dificultad extra.

Al mismo tiempo, y como una forma de aplicar todo lo descripto anteriormente en un problema real, se detectó la necesidad de un sistema 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. Este sistema debería abarcar desde la creación de un premio, la votación anónima a los distintos postulantes y la publicación de resultados. Entonces 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 la profesora Mgter. Gladys Noemí Dapozzo quien orientó y motivó para la concreción del mismo. A los que siempre estaré agradecido. En menor medida contó 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.

Prologo

[Se refiere habitualmente al: - contexto personal en que se efectuó el trabajo, - las motivaciones personales que llevaron a él, - las personas que colaboraron (incluyendo al o los directores) y - eventualmente agradecimientos, y toda otra mención personal a incluir.]

{} Estas notas se borrarán al terminar la revisión del capítulo {}

Agradecimientos

A mi familia, amigos, compañeros de estudio, colegas y profesores.

Dedicatoria...

Frase...

Resumen sintético

[Documento de longitud restringida.

Como máximo 200 palabras, según el reglamento vigente.

Se constituye de un solo párrafo.

- Deben usarse sentencias completas y evitar tecnicismos que dificulten su comprensión.
- Las oraciones deben estar redactadas con precisión, claridad y sencillez.

Debe incluir: Introducción, Fundamentación. Objetivos.

Metodología utilizada. Resultados. Conclusiones y futuros trabajos (si correspondiera).

No incluir figuras, tablas, gráficos, citas y referencias.]{{ Estas notas se borrarán al terminar la revisión del capítulo }}

Esto se hará al finalizar el informe.

Resumen extendido (máximo 3 carillas).

[Debe contener como máximo 3 páginas.

Resume el trabajo. Debe contener las siguientes secciones:

Introducción, Fundamentación. Objetivos. Metodología utilizada. Mención de las herramientas. Resultados.

Conclusiones y futuros trabajos (si correspondiera).

No incluir figuras, tablas, gráficos, citas y referencias.

]{{ Estas notas se borrarán al terminar la revisión del capítulo }}

Esto se hará al finalizar el informe.

Índice de contenidos

Capítulo 1. Introducción.....	10
1.1 Estado del arte.....	10
1.1.1 TDD.....	10
1.1.2 ATDD.....	12
1.1.2 BDD.....	13
1.1.3.1 Ejemplos como requerimientos.....	15
1.1.3.2 Historias de usuario.....	15
1.1.4 BDD vs. ATDD.....	17
1.1.5 prácticas asociadas a BDD/TDD.....	18
1.1.5.3 Interfaces gráficas de usuario en baja fidelidad.....	21
1.2 Objetivos del TFA.....	22
1.3 Hipótesis.....	23
1.4 Fundamentación.....	23
1.5 El problema de aplicación.....	24
1.6. Organización del trabajo.....	25
Capítulo 2. Metodología.....	26
Metodología.....	26
Etapa 1: Investigación preliminar.....	26
Etapa 2: Concepción.....	26
Etapa 3: Desarrollo iterativo e incremental.....	27
Etapa 4: Documentación de la experiencia.....	28
Proceso de software.....	28
Proceso iterativo general.....	28
Capítulo 3. Herramientas y/o lenguajes de programación.....	30
Entorno de Desarrollo.....	30
3.1 Pizarrón y fibra – herramientas para bosquejar.....	30
3.2 PivotalTracker – administración del proyecto y de las historias de usuario.....	30
3.3 Ruby – lenguaje de programación.....	31
3.5 Rails – framework para desarrollo web. {{ ver como achicar }}.....	32
3.6 WEBrick - servidor web de desarrollo y prueba.....	34
3.7 Bootstrap – framework front-end.....	34

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

3.8 SQLite - base de datos en desarrollo y prueba.....	35
3.9 MySQL Workbench v6.0 – modelador de datos.....	36
3.11 Git. - control de versiones de archivos y código.....	36
3.11 GitHub. - control de versiones de archivos y código.....	36
3.12 Google Docs y Libre Office – ofimática.....	37
3.13 Mozilla FireFox. - navegador web.....	37
Entorno de Pruebas.....	38
3.14 Cucumber – documentación de historias de usuario y escenarios.....	38
3.15 Capybara – librería de automatización.....	40
3.16 RSpec - pruebas unitarias.....	41
3.17 Guard – ejecución automática de pruebas locales.....	42
3.18 Coveralls – cobertura de código de las pruebas.....	42
Entorno de Producción.....	43
3.19 Travis-CI – integración continua.....	43
3.20 Heroku – servidor de aplicaciones.....	44
3.21 PostgreSQL - base de datos.....	44
Comentarios y discusiones sobre las herramientas.....	45
Capítulo 4. Resultados.....	48
Interfaces gráficas de usuario en baja fidelidad.....	49
Historias de usuario.....	50
Aplicando BDD y TDD.....	51
Ejemplo del proceso al aplicar BDD y TDD.....	53
Aplicando integración continua.....	59
Despliegue de la aplicación.....	59
Descripción del sistema.....	60
Módulo administrador.....	60
Módulo organizador.....	62
Módulo jurado.....	67
Módulo identificación.....	71
Comentarios y discusiones.....	72
{{ revisar duplicado con lo que esta en conclusiones }}.....	72
Sección 5: ESTUDIO PRELIMINAR DE IMPLEMENTACIÓN.....	79
Capítulo 5. Conclusiones y futuros trabajos.....	80
5.2 Futuros trabajos.....	84

Índice de figuras

Figura 1: Esquema TDD.....	12
Figura 2: Esquema ATDD.....	12
Figura 3: Esquema BDD.....	13
Figura 4: Story Card en formato Connextra.....	16
Figura 5: Escenario de integración continua.....	19
Figura 6: Captura de pantalla primera planificación en PivotalTracker.....	31
Figura 7: Interfaz después de integrar con Bootstrap.....	35
Figura 8: Interfaz con Bootstrap en pantalla de 364x640 píxeles.....	35
Figura 9: Cucumber testing stack.....	39
Figura 10: Mockup baja fidelidad de interfaz alta de entidad organizadora.....	50
Figura 11: Wireframe interfaz alta de entidad organizadora.....	50
Figura 12: Historias de usuario previo a priorizar.....	50
Figura 13: Historias de usuario en PivotalTracker.....	51
Figura 14: Historia de usuario "borrar una organización".....	54
Figura 15: Archivo borrar_organizacion_steps.rb.....	54
Figura 16: Prueba falla. Rojo.....	55
Figura 17: Spec inicial para acción "destroy" del controlador.....	55
Figura 18: Pruebas pendientes.....	56
Figura 19: Prueba unitaria para borrar organización.....	56
Figura 20: Ejecución fallida borrar organización.....	57
Figura 21: Implementación mínima.....	57
Figura 22: Pruebas unitarias pasan. Verde.....	57
Figura 23: Lista organizaciones.....	58
Figura 24: Mensaje confirmación.....	58
Figura 25: Mensaje confirmación tras eliminar dato.....	58
Figura 26: Escritorio administrador. Vista reducida.....	60
Figura 27: Vista reducida de interfaz CRUD.....	61
Figura 28: Estadísticas. Básicas del Sistema. Vista reducida.....	61
Figura 29: Escritorio usuario organizador. Vista reducida.....	62
Figura 30: Formulario para crear un proceso de selección.....	62

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

Figura 31: Formulario para crear una categoría.....	63
Figura 32: Lista de categorías.....	63
Figura 33: Consulta categoría "Mejor idea". Vista reducida.....	64
Figura 34: Formulario para crear un candidato.....	64
Figura 35: Lista de electores. Vista reducida.....	65
Figura 36: Ventana modal para confirmar rechazo de elector. Vista reducida.....	65
Figura 37: Búsqueda elector. Vista reducida.....	65
Figura 38: Escrutinio. Vista reducida.....	66
Figura 39: Resultados categoría. Vista reducida.....	66
Figura 40: Datos para auditoría. Vista reducida.....	67
Figura 41: Escritorio jurado. Vista reducida.....	67
Figura 42: Listado compacto de procesos. Vista reducida.....	68
Figura 43: Inicio votación. Vista reducida.....	69
Figura 44: Candidatos a "Mejor idea" durante votación.....	69
Figura 45: Candidato seleccionado para votar por él.....	69
Figura 46: Mensaje confirmación de emisión de voto.....	70
Figura 47: Mensaje tras voto registrado correctamente.....	70
Figura 48: Categorías deshabilitadas tras votación.....	70
Figura 49: Botones sociales.....	70
Figura 50: Formulario creación de usuarios.....	71
Figura 51: Inicio de sesión.....	71
Figura 52: Barra de navegación superior.....	71
Figura 53: Barra de navegación lateral.....	72

Índice de tablas

Tabla 1: Ventajas y desventajas de Ruby.....	32
Tabla 2: Iteraciones del proyecto.....	49

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. Sin embargo, 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 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 Programming” (en español programación extrema) [2], una propuesta de agilismo que, con base en 4 valores y 5 principios claves, proponía la utilización de 20 prácticas, ya conocidas, para hacer desarrollo de software. De estas prácticas, nace el TDD como se lo conoce más a menudo.

TDD es una práctica iterativa de **diseño** de software orientado a objetos con base en pruebas automatizadas, que fue presentada por Kent Beck y Ward Cunningham como parte de Extreme Programming (XP) en “Extreme Programming Explained: Embrace Change” [2]. Y 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 de 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 de 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 que, en caso de no hacerla, lo degradarían.

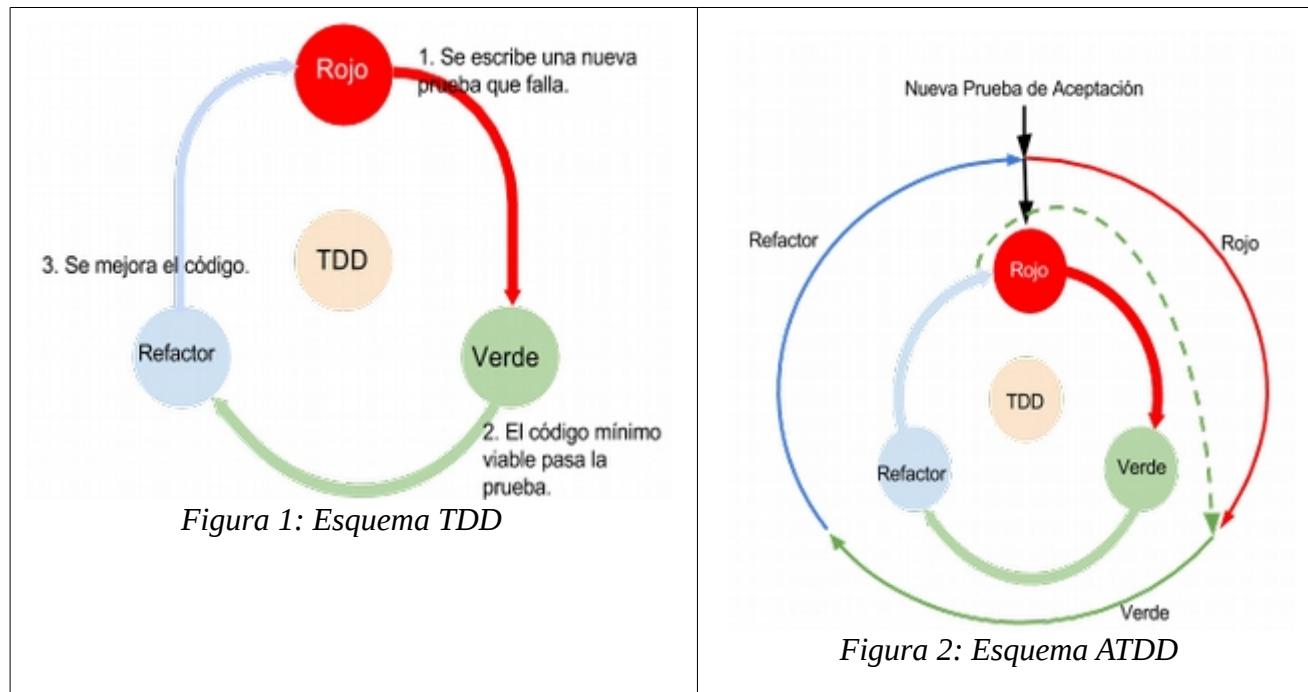
Estas prácticas se sintetizan en lo que se conoce comúnmente como el algoritmo de TDD (Véase Figura 1), o como Beck lo llamo “el mantra TDD” [3]:

1. **Rojo:** se escribe una pequeña prueba que no funcionará, y posiblemente al principio no compile.
2. **Verde:** crear el código que haga que la prueba pase rápidamente, sin importar los errores cometidos.
3. **Refactorizar:** 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]. Sin ser regla Dave Chaplin, en “Test First Programming” dice: “Si no puedes escribir una prueba para lo que estás por codificar, entonces no deberías estar pensando en codificar”. Esto implica que ninguna funcionalidad, por pequeña que sea, debería escribirse por adelantado, si no se escriben antes un conjunto de pruebas que verifiquen su validez. Las pruebas a este nivel deben ser independientes, unas de otras, y poder ejecutarse individualmente entonces se tienen que escribir una por vez. 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 breve 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 estas representan el requerimiento, no que lo especifican [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

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua 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 la 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.

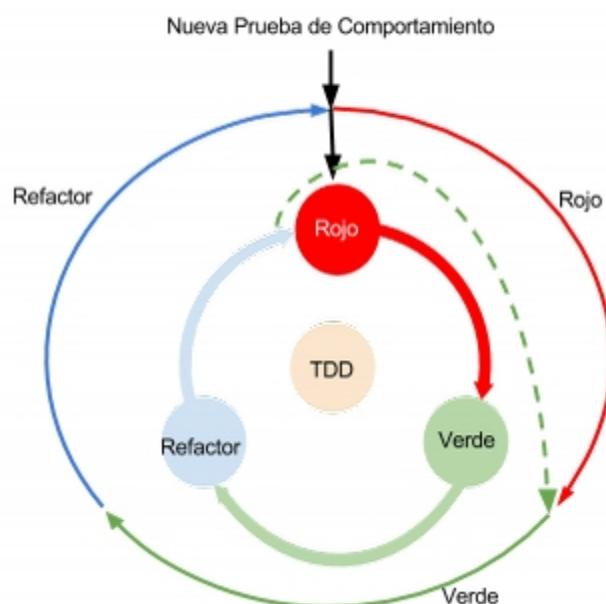


Figura 3: Esquema BDD

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 disponible en [5]. Según North, BDD está pensado para hacer estas prácticas ágiles más accesibles y efectivas a los equipos de trabajo que quieren

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

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, propuesta inicialmente por North, 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”).

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 variación 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; todo esto en la palma de una mano [6]. Tendrán una estructura como la siguiente:

Característica: [Nombre]

Como un [rol en el sistema]

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”, donde se utilizan ciertas estructuras de tablas para expresar requerimientos y reglas de negocio, mencionan en [7], ya que las reglas de negocio cambian menos que las interfaces gráficas; estas tablas posibilitan describir datos de entrada y de salida.

La visión de North era tener una herramienta que pueda ser usada por Analistas de Sistemas y Testers para que ellos puedan 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 [5].

Fontela [1], comenta que al inicio el foco de esta técnica estaba 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.

1.1.3.1 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 [8].

Los tests de aceptación o de cliente, de las historias de usuario, son el criterio escrito 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 (nuestro idioma), 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 dueños de 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.

Las historias de usuario son el punto de partida del desarrollo en cada iteración, esto genera la conexión de XP con otros métodos ágiles como Scrum allá donde una se queda y sigue la otra [8].

1.1.3.2 Historias de usuario.

El primer paso en un ciclo ágil, que es a menudo el más dificultoso, es dialogar con cada uno de los interesados para entender los requerimientos; de este dialogo se derivó las historias de usuario (en inglés user stories).

Estas son unas narraciones simples y cortas, en lenguaje natural; en donde cada una describe una interacción específica entre un interesado la aplicación [6] 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

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

que se da entorno 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, ya desaparecida, compañía llamada Connextra y de allí su nombre, que consiste en utilizar el siguiente patrón: Como <rol> quiero <funcionalidad> para <beneficio>

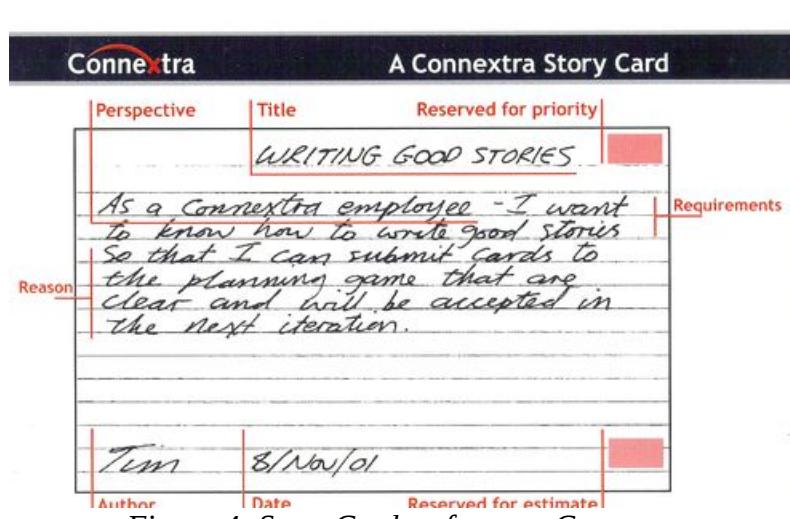


Figura 4: Story Card en formato Connextra

Story Card

Las historias de usuario provienen de la comunidad de Interfaz humano-computadora (o sus siglas en inglés HCI). Ellos las desarrollaron usando tarjetas bibliográficas de 3x5 pulgadas (76 mm x 127 mm), conocidas como “tarjetas 3x5”, y luego “story card”, en la imagen (Véase Figura 4) se aprecia una de ellas 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 por 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 (Specific, Measurable, Achievable, Relevant, y Timeboxed) captura las características deseables en una buena historia de usuario [6][9].

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

1.1.4 BDD vs. ATDD.

ATDD y BDD son dos prácticas casi contemporáneas, la primera surgida en 2002 y la segunda empezada a desarrollar en 2003. Y si bien se lanzaron para resolver cuestiones diferentes (BDD surge como mejora de TDD, mientras que ATDD es una ampliación), llegaron a conclusiones para nada disjuntas.

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 [10]. 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 [8].

En los dos casos, se trata de actividades que empiezan de lo general y las necesidades del usuario, para ir deduciendo comportamientos y generando especificaciones ejecutables [1]. Fontela [1], dice que 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* [10].

Fontela [1], 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

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

especificación contractual. En respuesta, Ward Cunningham expresa, 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. Lasse Koskela amplía esto diciendo 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 por Beck en XP [2] y luego muy difundida a partir del trabajo de Fowler [11]. 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 asociada en las metodologías convencionales, superando, en determinadas circunstancias, la propia complejidad 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 espere para integrar más costoso e impredecible se vuelve el proceso. Así, el sistema puede llegar a ser 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.

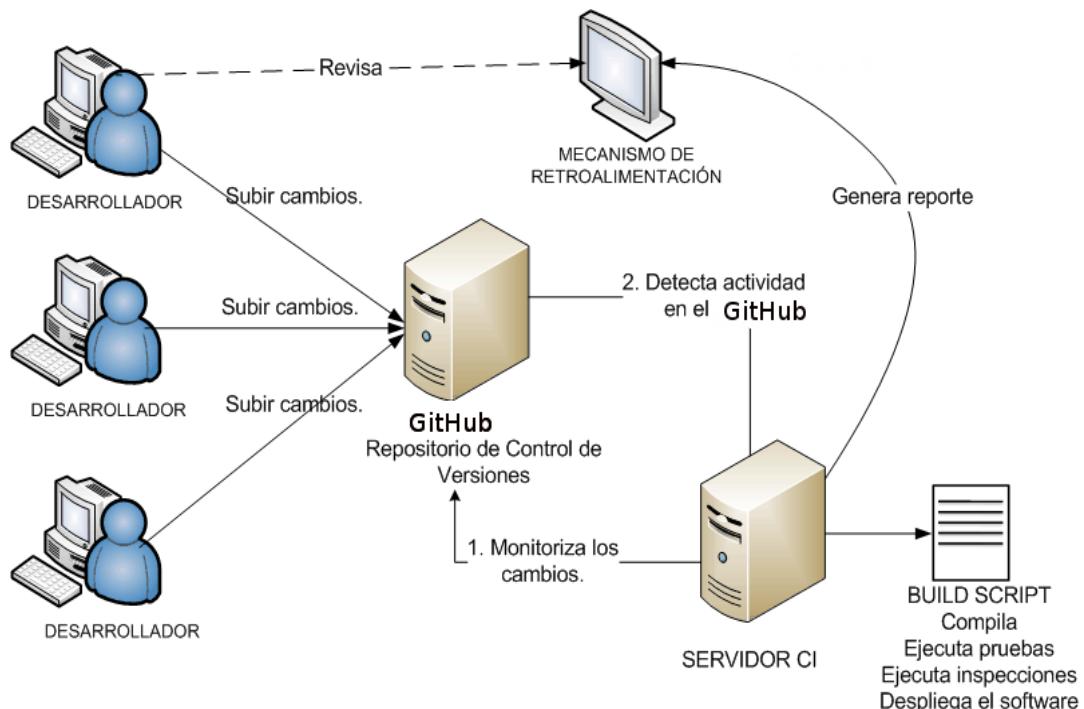


Figura 5: Escenario de integración continua

Un entorno de Integración Continua está compuesto por diferentes herramientas y prácticas en las cuales necesita apoyarse, en la imagen (Véase Figura 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, generalmente utilizando algún “hook”, entonces procede a ejecutar una serie de pasos para realizar el despliegue, entre ellos ejecutar las pruebas tanto antes como 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 detención 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 Integración Continua que se basan en unos procesos de desarrollo continuos controlados permanentemente mediante pruebas del código.

Para implementar CI se debe 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 comprometer 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, ya que para poder realizarla se debió haber construido pruebas automatizadas previas a la integración.

1.1.5.2 Refactorización.

Del inglés Refactoring, es una práctica que busca mejorar la calidad del código, su diseño interno, sin modificar el comportamiento observable (requerimiento). Esta se aplica una vez que ya está codificado, el código que se desea modificar.

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 la calidad de 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 (como SOLID, única responsabilidad, etc.) 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. Entonces una práctica para que estas sean seguras es escribir pruebas automatizadas antes de modificar el código, que verifiquen su funcionamiento. Ejecutándolas luego de las modificaciones para ver si continúan pasando y así estas pruebas funcionaran como una red de seguridad ante fallos.

Comenta Fontela en [12], que luego de que surgiera el término refactoring, en un trabajo de Opdyke y Johnson de 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.

Según Fontela [1], 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 sea 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, sugieren en [6]. Los wireframe son una guía visual o esquema que representa la estructura visual del sitio, su diseño y contenido [13].

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 estos sean atractivo para los interesados no técnicos y alentarlos a la prueba y error, ya que es más fácil de probar y descartar bosquejos. En [6] se recomienda utilizar los mismos elementos que en un jardín de infantes para hacer los bosquejos: papel, crayones, tijeras. Ellos llaman a esta 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 tendrá que especificar los detalles de la interfaz de usuario cuando se creen los archivos HTML y se convierta en realidad esa interfaz. Entonces esto sigue siendo más fácil de cambiar en lápiz y papel que en el código [6].

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 “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.

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

1.3 Hipótesis

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

1.4 Fundamentación

A medida que un software crece y se vuelve más complejo los proyectos no llegan a buen puerto, o lo hacen muy tarde, esto se llamó “La Crisis del Software”, debido a la cual surgió la Ingeniería del Software (IS).

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”* [14]. Sommerville [14], 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. 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 C. Blé [8], 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 ese 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 [8]. 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 [10].

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 han organizado premios y certámenes en el pasado, 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. Se buscó un software que pueda satisfacer estas necesidades organizativas y no se encontró uno puntual que las satisfaga, en español, como servicio en Internet.

Este trabajo propone la construcción de una aplicación web multiplataforma que satisfaga de forma mínima y viable esas necesidades. Tendrá como fin gestionar la organización de premios, certámenes y votaciones en general, permitiendo el seguimiento de sus procesos administrativos. 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 para trabajos futuros.

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 introduce en el marco teórico a los conceptos de TDD, ATDD, BDD, Integración Continua y Refactorización. Además se exponen los objetivos principales del trabajo junto con la hipótesis.

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 del sistema, 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 algunas conclusiones a las que se ha llegado y se proponen algunas posibles líneas de trabajo futuras.

Capítulo 2. Metodología

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

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 [8] (.NET o Python con xUnit) y por [6] (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, para 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 del sistema 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 servicios y sistemas similares existentes en el mercado.

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 taller de especificación (specification workshop).

Consistió en una reunión de trabajo con un posible futuro cliente donde se definieron las historias de usuario (Véase Figura 12) que luego se priorizaron para su implementación posterior y algunos escenarios básicos.

Etapa 3: Desarrollo iterativo e incremental.

{} explicar con Gráfico que son 3 bucles anidados el proceso {}

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; en 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. **Los pasos realizados en cada iteración se verán con más profundidad en la sección Proceso de software.** (pag. 28), más adelante en este capítulo y en el Capítulo 4. Resultados (pag. 48).

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

Paso 1. Configuración ecosistema de trabajo.

Se instalaron las herramientas necesarias para trabajar con la tecnología seleccionada (Ruby, Rails, SublimeText) 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 2. Desarrollo del Back-end.

Se implementaron las historias de usuario, según su prioridad, correspondientes al módulo back-end de administración 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.

- 2.1. Codificación de Ejemplos siguiendo BDD en Ruby con Cucumber.
- 2.2. Codificación de Ejemplos siguiendo metodología TDD en Ruby con RSpec.
- 2.3. Diseño de Interfaces; integrando con Boostrap.

Iterando los sub-pasos 1 a 3 hasta completar la codificación del BackEnd

Paso 3. Desarrollo del Front-end.

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

Luego se implementaron las historias de usuario, según su prioridad, desarrollaron los módulos del organizador del premio/votación, 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 la codificación del front-end logrando un prototipo funcional mínimo y viable.

Etapa 4: Documentación de la experiencia.

Se confeccionó el informe del Trabajo Final de Aplicación, siguiendo la estructura y propuesta en su reglamento, siguiendo el orden secuencial de cada uno de los capítulos.

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 el marco de ella misma y como BDD puede ser entendida como una ampliación de TDD, ambas encuadran bien en XP.

Proceso iterativo general.

Para cada iteración (sprint), en total 7, se realizó el mismo proceso que se describe a continuación:

A. Taller de Especificación.

Reunión de dialogo con el cliente, donde se obtuvo:

1. Creación de historias de usuario.
2. Creación de prototipo de GUI en baja fidelidad (Lo-fi UI mockup).
 - 2.1. Creación de los prototipos rápidos de las páginas (wireframes, mockup).
 - 2.2. Creación de guiones gráficos (storyboard) de la navegación entre páginas.

B. Taller de planificación.

1. Priorización de historias de usuario.
2. Selección de historias de usuarios a realizarse en la iteración (sprint) o Iteration Plan.
3. Definición de tareas asociadas a las historias seleccionadas.
4. Documentación en herramienta PivotalTracker

C. Desarrollo de la iteración.

D. Demostración (small release).

Del diálogo con los interesados primero se derivaron las historias de usuario. Como es una aplicación web que involucra a usuarios finales, también se necesitó hacer las interfaces de gráficas de usuario (GUI). Se las hizo primero como dibujos de las páginas web, de baja fidelidad, y se las combinó con los guiones gráficos “storyboards”, para analizar la navegación entre ellas, antes de crear estas interfaces en HTML.

Desarrollo de la iteración.

Luego para la implementación de cada historia de usuario de la iteración, se realizó un proceso similar, variando en la intensidad de aplicar las pruebas primero (pasos A1 y A2):

A1. Cuando se aplicó BDD.

1. Creación de escenarios, para las historias de usuario seleccionada.
2. Obtención de ejemplos, camino normal y camino alternativo.
3. Automatización de pruebas de aceptación e integración.
4. Aplicar TDD con RSpec.
5. Crear interfaz de usuario básica.

A2. Cuando no se aplicó BDD.

1. Selección de la historia de usuario.
 2. Diálogo con usuario sobre posibles escenarios.
 3. Crear interfaz de usuario básica.
 4. Desarrollo.
- B. Refinamiento de interfaz de usuario.
- C. Integración Continua (Travis-CI, Heroku).
- D. Pruebas exploratorias y correcciones.
- E. Documentación de avance en PivotalTracker. Sirvió para medir la velocidad (Measure Velocity)

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.

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 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.

3.2 PivotalTracker – administración del proyecto y de las historias de usuario.

PivotalTracker (<http://www.pivotaltracker.com/>) [15], 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 wireframe o mockup de las GUI.

En la imagen (véase Figura 6) 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.

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

- 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.

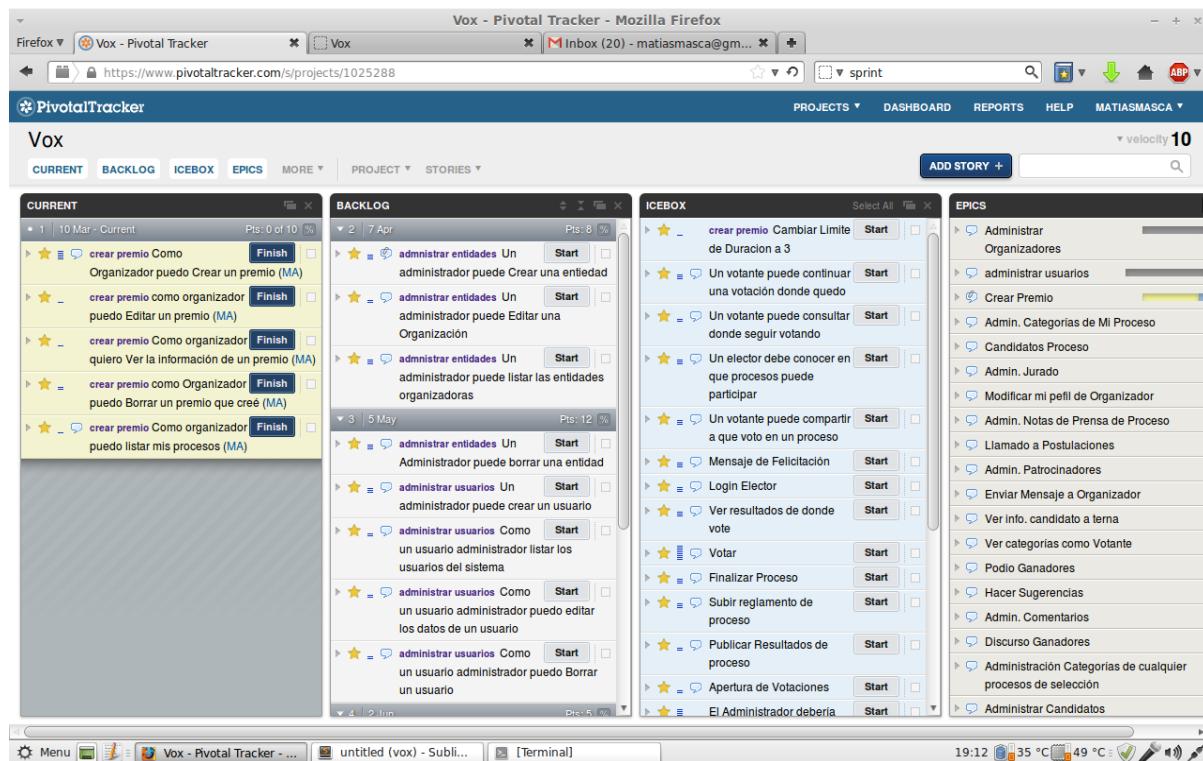


Figura 6: Captura de pantalla primera planificación en PivotalTracker

3.3 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.” [16]. Este lenguaje se adapta perfectamente a la práctica de TDD, incluyendo pruebas unitarias de

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua forma nativa. En la siguiente tabla (Véase Tabla 1) se mencionan algunas ventajas y desventajas del lenguaje:

Tabla 1: Ventajas y desventajas de Ruby

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.	

3.5 Rails – framework para desarrollo web. {{ ver como achicar }}

Ruby on Rails (rubyonrails.org) [17], 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. Te permite escribir un buen código evitando que te repitas y favoreciendo la convención antes que la configuración.”[17]. 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 [6].

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 figura en un anexo (Véase anexo II). {{ ¿Se puede omitir este anexo? }}

Gemas.

Se utilizó Rails con el agregado de algunas librerías externas, llamadas gemas, las cuales figuran en el archivo `/Gemfile`. 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.

3.6 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, como sucede con otros lenguajes interpretados para la web como PHP. Puede consultar más información sobre WEBrick en [18].

3.7 Bootstrap – framework front-end.

Se utilizó el framework front-end Boostrap [19], 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 [20].

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

Se lo utilizó pre-procesado con Sass (*Syntactically Awesome Stylesheets*) con la utilización de la gema **Bootstrap-sass**. En la imagen (Véase Figura 7) se puede apreciar su aplicación en la interfaz de back-end para editar una entidad organizadora de procesos de selección en la sistema.

VOX: CMS para Procesos de Selección

Inicio Escritorio

Identificado: donramon@chavo.mx.

Editar perfil | Cerrar sesión

Usuario

Mi Perfil

Cerrar sesión

Estadística

Estadísticas

Editando Organización

ID Usuario

Nombre de la Organización

Dirección

Sitio web

correo electrónico

Isologotipo:

Guardar cambios

Mostrar | Atrás

VOX: CMS para Procesos de Selección

Usuario

Mi Perfil

Cerrar sesión

Estadística

Estadísticas

Editando Organización

ID Usuario

4

Figura 7: Interfaz después de integrar con Bootstrap

Figura 8: Interfaz con Bootstrap en pantalla de 364x640 píxeles

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 Figura 8), 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.

3.8 SQLite - base de datos en desarrollo y prueba.

SQLite (sqlite.org) [21] 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 [22]. 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.

3.9 MySQL Workbench v6.0 – modelador de datos.

MySQL Worckbench [23] es una herramienta visual unificada, para arquitectos de base de datos, desarrolladores y administradores de base de datos. MySQL Worckbench 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.

3.11 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 [24] “*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 vínculo con el servicio GitHub, que se describe a continuación.

3.11 GitHub. - control de versiones de archivos y código.

GitHub [25] 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 [24]. Para el presente trabajo se vinculó el repositorio local Git con una cuenta de GitHub [25], 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/>)[26] es un servicio para medir la cobertura de código, se describe más adelante.
- **Travis-CI** (<https://travis-ci.org/>)[27] 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.

3.12 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.

3.13 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.

Entorno de Pruebas

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

3.14 Cucumber – documentación de historias de usuario y escenarios.

Cucumber (www.cukes.info) [28] 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 [29]. 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 [29].

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.

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.

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”, 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

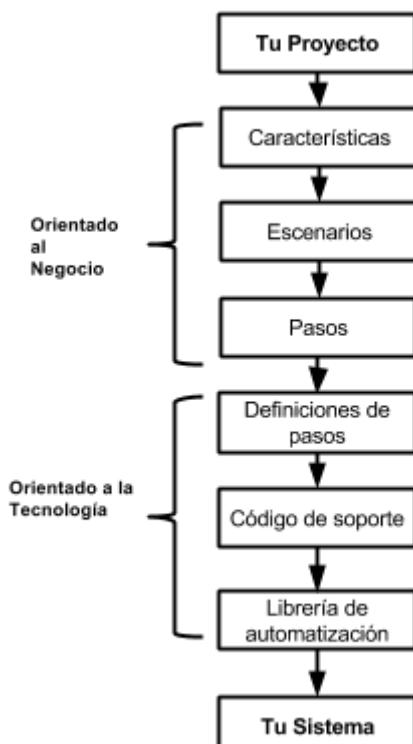


Figura 9: Cucumber testing stack.

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”, dentro de donde estén almacenados los archivos feature, por ejemplo en este caso la ruta relativa es ..//feature/step_definitions dentro del proyecto. La jerarquía que sigue Cucumber, desde los archivos features hasta la librería de automatización, esta ilustrada en la imagen (Véase Figura 9) extraída de [29] donde se aprecia cómo se desciende gradualmente.

3.15 Capybara – librería de automatización.

Capybara [30] es una herramienta de automatización, escrita en Ruby, que permite simular 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 ayudó 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.

3.16 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.

3.17 Guard – ejecución automática de pruebas locales.

Guard (<http://guardgem.org/>) [31], 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. 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”

3.18 Coveralls – cobertura de código de las pruebas.

Coveralls.io (<https://coveralls.io/>) [26] 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 [25]; 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 [27].

Entorno de Producción

3.19 Travis-CI – integración continua.

Travis-CI. (<https://travis-ci.org/>)[27] 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 [25]. 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 [32].

Procedimiento automatizado.

Con cada integración (en inglés build) se instalaban todas las librerías necesarias para ejecutar la aplicación Rails, se generaba la base de datos desde cero, luego se cargaban los datos de prueba; seguido a esto se ejecutaban las pruebas tanto de Interacción como unitarias. Si todos estos procesos se ejecutaban sin errores (exit 0), el servicio procedía a realizar el despliegue de la aplicación en el servidor de Heroku [33], subiendo los archivos necesarios, ejecutando las instalaciones de librerías necesarias, concluyendo con la configurando 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 se puede consultar en línea en <https://travis-ci.org/matiasmasca/vox>, también se puede acceder a un historial de integraciones que se realizaron desde la plataforma. En casos de fallo se suspende el deploy y envía un email informando que el proceso ha 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>

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.

3.20 Heroku – servidor de aplicaciones.

Heroku (www.heroku.com) [33] 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 [34]. 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 [34].

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 permitir 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.

3.21 PostgreSQL - base de datos.

PostgreSQL (<http://www.postgresql.org/>) [35] 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 [35].

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

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua que permite hasta 10.000 registros.

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.

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 cargar

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

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 sería 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.

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.

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 enfriamiento 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

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

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.

Al completarse las etapas consignadas en la metodología del trabajo se lograron los siguientes resultados y artefactos de software:

Etapa 1: Seleccionado lenguaje Ruby, Seleccionada práctica ágil BDD y herramientas asociadas compatibles con Ruby: Cucumber, Capybara y RSpec, seleccionado el framework Ruby on Rails.

Etapa 2: documento ERS (Especificación de Requerimientos de Software) véase [anexo XX](#). Historias de usuario.

Etapa 3: prototipo funcional para premios (sistema web); repositorio público de código; pruebas automatizadas; servidor de integración configurado.

Etapa 4: Informe TFA.

Proceso de Software.

Para el desarrollo de la aplicación se realizaron 7 iteraciones, como se aprecia en la tabla (Véase tabla 2).

Hasta la iteración 4 se siguió la práctica de BDD de forma pura; en la tabla (Véase tabla 2) se nota un aumento en la velocidad, 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. Luego se aprecia un aumento de la velocidad, 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 [36] donde advierte la dificultad de hacer estimaciones sobre proyectos totalmente nuevos que no tienen un punto de comparación previa.

Tabla 2: Iteraciones del proyecto

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

Se siguió un enfoque de afuera hacia adentro (del inglés outside-in), propuesto en [6], este sigue el enfoque sistémico de diseño *top-down*. Entonces se empezó por las pantallas, la interacción entre ellas. Pasando por el comportamiento esperado descrito en las historias de usuario, con sus escenarios y ejemplos.

Interfaces gráficas de usuario en baja fidelidad.

Siguiendo los pasos de un diseño de afuera hacia adentro, 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 (los esquemas de página o wireframe y los mockup) en baja fidelidad; el uso de estos elementos permitió analizar las propiedades y navegación de la aplicación. En la imagen (Véase Figura 10) se aprecia el bosquejo para la interfaz de alta de una entidad organizadora, que luego se pasa en limpio en la siguiente imagen (Véase Figura 11), más adelante se muestra esta interfaz implementada en la aplicación web (Véase Figura 7).

Este mockup es una captura de pantalla de un formulario de captura de datos. Se titula "Entidad Organizadora." y contiene los siguientes campos:

- Nombre: [campo vacío]
- Dirección: [campo vacío]
- Descripción: [campo vacío]
- Sitio Web: [campo vacío]
- E-mail: [campo vacío]
- Logo (URL): [campo vacío]

Al final del formulario hay un botón que dice "/SAVE/".

Figura 10: Mockup baja fidelidad de interfaz alta de entidad organizadora

Este wireframe muestra la estructura de un formulario para la "Entidad Organizadora". Los campos y sus tipos de datos son:

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

En la parte inferior derecha del wireframe hay un botón que dice "Guardar".

Figura 11: Wireframe interfaz alta de entidad organizadora

Historias de usuario.

Para el presente trabajo se utilizaron las historias de usuario para la captura de requerimientos y comunicación con el cliente. Estas se escribieron en formato "Connextra" (Véase Figura 4) adaptado a papeles de colores de 10x10cm, que se pegaron sobre un el pizarrón, como se ve en la imagen (Véase Figura 12). Una vez definidos y con una primera priorización se digitalizaron en la herramienta PivotalTracker, para su documentación y seguimiento, aunque también se las siguió utilizando en papel como una guía visual del trabajo a realizar y realizado.



Figura 12: Historias de usuario previo a priorizar

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

Los detalles 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 izquierda: Current, Backlog, Icebox, Done y Epics como se aprecia en la imagen (Véase Figura 51).

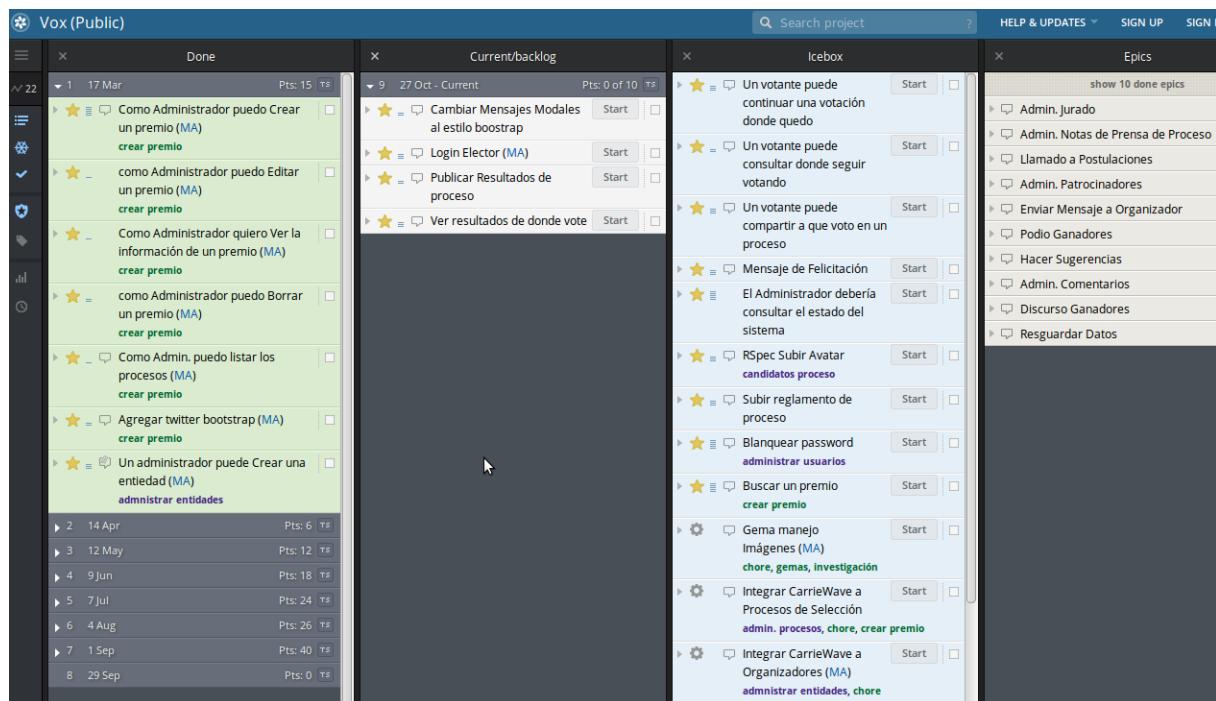


Figura 13: Historias de usuario en PivotalTracker

Aplicando BDD y TDD.

Se aplicó satisfactoriamente la práctica de BDD y TDD, para los módulos de administración (back-end) del usuario administrador. Utilizando las herramientas específicas: Cucumber y Capybara para BDD; RSpec para TDD y Guard para ejecutar todas las pruebas continuamente. El back-end esta compuesto por los siguientes módulos:

- Procesos de selección.
- Organización.
- Usuarios.
- Categorías.
- Candidatos.

Cada uno de ellos incluyó las historias de usuario elementales: *crear, editar, borrar, listar y mostrar*. Con sus “caminos felices” y casos extremos.

Aplicando BDD.

Las historias de usuario implementadas, junto con sus escenarios, se las puede encontrar en la carpeta `/features/`, al momento de escribir este informe, está compuesto por 30 archivos. Para cada una de ellas en la subcarpeta “`step_definitions`” se encuentran los archivos de automatización de las pruebas, cada uno de ellos vincula los pasos con un código ejecutable por la herramienta Cucumber y librerías complementarias.

En BDD se utilizaron ejemplos en las conversaciones para ilustrar comportamiento esperado por la aplicación. Hablar el lenguaje del cliente ayuda a mantener alta la abstracción, sin caer en detalles de implementación. Por ello los archivos `.feature` pueden ser leídos por cualquier persona intervenirte en el proyecto y entender de qué se trata.

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

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

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

Aplicando TDD.

Las pruebas unitarias escritas para el presente trabajo se encuentran en `/spec` y dentro de ella en las diferentes carpetas:

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

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

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

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

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

/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.

La primera impresión resultó en una aparente sobrecarga de trabajo, cuando se repite el tipo de pruebas entre Cucumber y RSpec, que evalúan el mismo comportamiento esperado de la aplicación. Pero en el caso de TDD (RSpec), que es de más bajo nivel, también se deberían evaluar con pruebas 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.

En el caso del framework Rails, es condición necesaria conocer el patrón MVC y su funcionamiento e implementación en esta herramienta, para entender cómo deberían funcionar las cosas. Cuando se entiende como fusionan las cosas o como debería funcionar es más fácil luego crear las pruebas primero antes que el código, porque uno sabe a dónde quiere llegar, donde se van a estar presentando los errores y el código que uno le gustaría tener (en la jerga "Code we wish we had" o CWWWH); uno sabe en qué pasos intermedios puede ir definiendo cosas que aún no existe. Por ejemplo al crear una acción en un modelo, uno puede ir partiendo de la vista, el controlador y llegar con pruebas hasta el método del modelo.

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 mencionadas de los módulos de back-end del administrador.

Paso 1: Crear archivo feature.

```
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"
```

Figura 14: Historia de usuario "borrar una organización"

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 Figura 14) se muestra como se ve este archivo en el editor de texto SublimeText. En este caso se estaba usando la etiqueta @wip del inglés “work in progress” para poder ejecutar más fácilmente esa prueba en particular en Cucumber.

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.

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

Lo

Figura 15: Archivo borrar_organizacion_steps.rb.

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 Figura 14, solo se tuvo que automatizar uno nuevo, como Figura en la imagen (Véase Figura 15). 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.

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 “mockup” 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 Figura 16), 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/step_definition
  Y que estoy en la pantalla de Administración de Organizaciones # features/step_definition
  Cuando hago click en Borrar para "ACME" # features/step_definition
    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_definition

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/comunidadtic/.rvm/bin/rvm-auto-ruby -S cucumber features/borrar_organizacion.feature -l14']]
[dir: /home/comunidadtic/vox]
[path: /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/usr/games:/usr/local/games]
```

Figura 16: Prueba falla. Rojo.

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” y se creó la descripción del comportamiento esperado, como se ve en la imagen (Véase Figura 17).

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

Figura 17: Spec inicial para acción “destroy” del controlador.

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 Figura 18)

```
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
```

Figura 18: Pruebas pendientes.

Paso 7: Escribir pruebas unitarias.

Se escriben las pruebas unitarias que ejecutará la herramienta RSpec. Como se puede apreciar en la imagen (Véase Figura 19) 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
```

Figura 19: Prueba unitaria para borrar organización.

Paso 8: Ejecutar pruebas unitarias. Fallan.

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

```

Figura 20: Ejecución fallida borrar organización.

Al ejecutar las pruebas del paso anterior estas fallan, como se capturo en la imagen (Véase Figura 20) ya que aún no se ha creado la implementación de ese método en el código de la aplicación en sí misma.

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 Figura 21) 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

```

Figura 21: Implementación mínima.

Paso 10: Pruebas pasan.

Al ejecutarse las pruebas, luego de la refactorización, se puede apreciar (Véase Figura 22) como estas continúan pasando sin fallas.

be

```

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

Randomized with seed 40504

```

Figura 22: Pruebas unitarias pasan. Verde.

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 Figura 23) se ve la interfaz, luego de ser estilizada con Boostrap, 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](#)

Figura 23: Lista organizaciones.

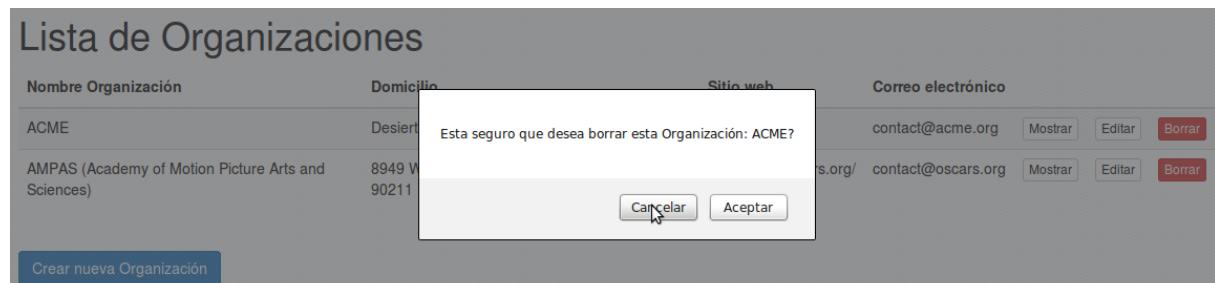


Figura 24: Mensaje confirmación.

Luego en la siguiente imagen (Véase Figura 24) el mensaje de confirmación antes de borrar el registro.

Luego de eliminar el registro, se le muestra al usuario otro mensaje de confirmación (Véase Figura 25), que también se podría incluir en la prueba de aceptación como se hizo para la historia “borrar mi proceso” (Véase archivo features/borrar_mi_proceso.feature)

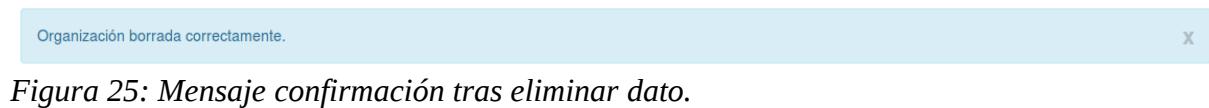


Figura 25: Mensaje confirmación tras eliminar dato.

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.

Aplicando integración continua.

Para practicar la integración continua se utilizó el servicio Travis-ci. Desde donde se realizaron, al momento de iniciar la redacción del informe, un total de 210 integraciones automatizadas. En cada uno de ellos se realizó el procedimiento detallado en herramientas, sección 3.19 (pag. 43). {{ revisar me parece que no describe }}

- Agregar la "Integración Continua".

Una vez se tuvo un conjunto de pruebas, esta tarea 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". Si conocimiento previo esta tarea llevó 8 horas.

- A mayor impacto del cambio, mayor defensa proveen las pruebas.

Al integrar la gema Devise, esto impactó en varias partes del sistema, provocando lógicamente muchos errores. Estos errores nunca fueron vistos por los usuarios finales del sistema porque Travis-ci no hacia el despliegue en Heroku ya que fallaban las pruebas.

- Falso positivo.

En Travis-ci una prueba fallaba (Rojo) y no hacia el deploy; pero la prueba fallaba por un error en ella misma, al corregirla se verificó que el sistema funcionaba bien.

Despliegue de la aplicación.

El despliegue automatizado de la aplicación se realiza en la plataforma Heroku y actualmente se puede acceder a él desde: <http://tfa-vox.herokuapp.com> {{ ¿Pasar esto a referencias? ¿solo agregar la referencia? O dejar como esta? }} utiliza una base de datos PostgreSQL. Si bien se realizaron algunas experiencias manuales, para entender cómo realizar el despliegue, con las recomendaciones de [34], luego se automatizó y delegó esta tarea a Travis-ci.

Esto 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

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua cambio, que se fueron agregando en las siguientes iteraciones.

{Evaluar agregar un Anexo para mostrar los pasos de cómo crear cuenta en Heroku y hacer el despliegue automatizado }{{hacer anexo}}

Descripción del sistema.

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, la votación anónima y la posterior consulta de resultados (escrutinio). Permite automatizar algunas tareas que se realizan de forma manual.

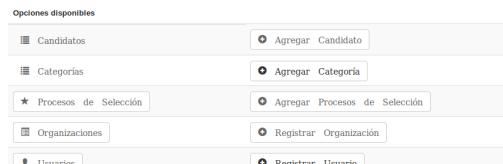
El acceso al sistema está basado en 3 perfiles de usuarios registrados: Administrador, Organizador y Jurado. Siendo el primero el único que se debe crear de forma manual al desplegar el sistema.

A continuación se describen las características principales del sistema web desarrollado durante el presente trabajo, desde la perspectiva del usuario.

Módulo administrador.

Este módulo consiste en una interfaz back-end para que un usuario, con estos privilegios, pueda modificar los datos, por solicitud de algún usuario en particular o por detectar algún mal funcionamiento. Actuará como un soporte y control de las operaciones del sistema web.

A través del menú “escritorio” tiene acceso a las opciones más comunes, mediante una botonera y menús de navegación laterales (Véase Figura 26). Desde allí tiene acceso a interfaces CRUD de todo el sistema, es decir a aquellas que sirven para visualizar y modificar los datos (Véase Figura 27); menos a las tablas Figura 26: Escritorio administrador. Vista reducida. relativas a los votos, ya que no se detectó una razón válida para que exista la necesidad de modificar lo relativo a un voto. Además de la posibilidad de acceder al formulario para crear nuevos registros. También tiene acceso a estadísticas básicas sobre la utilización del



Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

sistema, como se aprecia en la imagen (Véase Figura 28), para la creación de los gráficos se utilizó la gema **Chartkic** (véase Herramientas 3.5), lo que permitió de manera muy sencilla obtener unos gráficos de calidad satisfactoria.

A nivel de código se implementaron estas restricciones de seguridad, con la integración de la gema **Devise** (véase Herramientas 3.5) y la utilización de filtros a nivel de controlador.

Solo este usuario puede cambiar el nivel de privilegios de los demás usuarios; ya que por defecto se registran con el nivel de *jurados*.

Cabe destacar que este fue el primer módulo que se realizó y en su mayoría fue realizado siguiendo estrictamente las prácticas relacionadas a BDD. No utilizando “scaffold”, herramienta que incluye Rails para generar estructuras de código estándar; la razón principal de esta decisión fue porque se estaba aprendiendo a utilizar el framework y se quería entender su funcionamiento antes de pasar a la utilización de sus herramientas que ahorran trabajo.

Lista de procesos de selección

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

Nuevo proceso de selección

Figura 27: Vista reducida de interfaz CRUD.

Procesos

1. Premio - 2. Certamen 3. Votación

A pie chart titled "Procesos" showing the distribution of three types of processes. The categories are labeled "premio" (blue), "certamen" (red), and "votación" (grey). The "premio" slice is 85.7% (labeled "85.7%"), the "certamen" slice is 14.3% (labeled "14.3%"), and the "votación" slice is 0% (labeled "0%").

Categoría	Porcentaje
premio	85.7%
certamen	14.3%
votación	0%

A horizontal bar chart titled "Procesos" showing the count of processes for each category. The categories are "premio", "certamen", and "votación". The values are 6 for premio, 1 for certamen, and 0 for votación. The x-axis ranges from 0 to 8.

Categoría	Cantidad
premio	6
certamen	1
votación	0

Figura 28: Estadísticas. Básicas del Sistema. Vista reducida

Módulo organizador.

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* de su organización.

Al igual que el administrador, tiene un “escritorio” desde donde puede acceder a las opciones más utilizadas, como se aprecia en la imagen (Véase Figura 29) y un menú lateral dinámico que le brinda acceso directo a otras opciones.

Usuario Organizador

- Mis Categorías
- Agregar Categoría
- Mis procesos
- Nuevo proceso de selección
- Mi Organización
- Modificar mi Organización
- Mi Perfil
- Editar Perfil
- Cerrar sesión

Figura 29: Escritorio usuario organizador. Vista reducida.

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

Figura 30: Formulario para crear un proceso de selección.

Procesos.

El usuario puede crear procesos de selección asociados a la organización que representa.

Un proceso de selección provee información básica y útil para los interesados en él: una descripción general, fechas límite, el lugar donde se realiza, un tipo de proceso y el estado. Como se puede apreciar en la imagen (Véase Figura 30)

Tipo: se define si el proceso de selección será para un premio, un certamen o una votación.

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

Para el alcance del presente trabajo solo se trabaja con premios y votaciones, cuya diferencia es solo es una cuestión semántica ya que sus procesos son idénticos.

Estado de un proceso.

El estado de un proceso, sirve para iniciar y terminar las votaciones. Por defecto un proceso tiene estado “nuevo”, para que los jurados puedan votar deberá cambiarlo manualmente a “abierto”; de igual manera para terminar el proceso de votación deberá cambiarlo a “cerrado”. Para ello se facilitan 2 enlaces “abrir elecciones” y “cerrar elecciones” desde la descripción general del proceso.

Desde el botón “Mis Procesos” (Véase Figura 29) 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 Figura 27), y no a los de otras organizaciones.

Categorías.

Un usuario organizador puede crear categorías vinculadas al proceso seleccionado; ellas conformaran las n-uplas de candidatos a competir en la categoría. Los datos solicitados son un nombre, una descripción y el número de plazas, como se puede apreciar en el formulario de la imagen (Véase Figura 31).

Desde aquí el usuario podrá configurar cuantos candidatos por categoría (Nro. de plazas) para luego ir agregándolos directamente al seleccionar la categoría, como se ven en la imagen (Véase Figura 33)

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

Categorías: ACME Prize

Mejor idea	3 / 5
Mejor Caída	3 / 3
Mejor Trampa	5 / 5
Mejor Cohete	3 / 3
Candidatos a Mejor Wile E. Coyote	9 / 5

+ Nueva Categoría

Figura 31: Formulario para crear una categoría.

Figura 32: Lista de categorías.

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

Lista de Categorías.

Tanto en el menú lateral, como en la descripción del proceso, aparecerá un listado de la categorías de ese proceso con una etiqueta de color a la derecha; como se aprecia la imagen (Véase 32).

X / Y: Una referencia visual, que recuerda al usuario si la categoría está completa (verde), incompleta (amarillo) o excedida (rojo). Donde X representa la cantidad de candidatos asociados e Y la cantidad de plazas de la categoría.

A medida que se van agregando los candidatos, se irán mostrando en la vista de la misma categoría y se irán quitando los botones “Agregar candidato”; esto se aprecia en la imagen (Véase Figura 33) donde ya se habían agregado 3 candidatos y todavía faltaban dos. Y se irá actualizando la lista de categorías antes mencionada.

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 <small>Pintar un tunel falso sobre una pared, y luego redirigir las líneas del tráfico hacia él.</small> Mostrar Editar	 Comida con hierro <small>Colocar comida con pelotitas de hierro, luego usar un imán grande para atrapar a la presa.</small> Mostrar Editar	 Piedra gigante <small>Atar una piedra gigante con una cuerda, elevarla 3 metros y poner comida debajo.</small> Mostrar Editar
--	---	--

[Agregar Candidato](#) [Agregar Candidato](#)

Figura 33: Consulta categoría "Mejor idea". Vista reducida.

Nuevo candidato

Las candidatos, forman parte de las n-uplas de los procesos (por ejemplo una tema 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:
 No se seleccionó un archivo.

[Guardar Cambios](#)

[Ir a Candidates](#)

Figura 34: Formulario para crear un candidato.

Candidatos.

Un usuario organizador puede crear candidatos para una categoría seleccionada. Como se aprecia en la imagen (Véase Figura 34) se informa para que proceso y que categoría se agregaría el candidato. Luego los datos a ingresar serán el nombre del candidato, una descripción y la posibilidad de adjuntar una imagen que se utilizará como avatar del candidato.

Padrón

Una vez creada una organización y creado un proceso de selección, el usuario organizador

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

podrá administrar el padrón de electores de ese proceso. Se accede a través del menú 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 Figura 35) se puede ver un listado con 8 electores (jurados) registrados; de los cuales 5 fueron aprobados, 1 rechazado y 2 esperan por aprobación (los ítems 6 y 7).

Para a **Lista de Electores**

Nombre	email	Cod. Proceso	Estado	Acción
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

Figura 35: Lista de electores. Vista reducida.

Esta seguro que desea RECHAZAR el elector: Gregorio Perez para este padrón?

Esc... Una...

Seleccione un usuario: Micaëla Acuña

Agregar seleccionado

Buscar por email: un

Ir al Padrón

En una primera vista, como se aprecia en la imagen (Véase Figura 35), se observa la visualización de los electores registrados. Una vez que se ha elegido el elector a rechazar, se muestra una confirmación de acción. Una vez aceptada, se actualiza la lista y se reflejan los cambios.

uestra

en el nivel

en la

de 88

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua interfaz.



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.

Resultados categoría.

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.

Para sumar transparencia al proceso, tanto cuando se muestran los resultados de cada, como al final del escrutinio se muestran los datos en crudo sobre los cuales se hicieron los cálculos; En el caso de querer ver todo el proceso, se accede con el último botón “ver votos”.

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

En la imagen (Véase Figura 40) se aprecian los datos para la categoría “Mejor cohete”, son los datos que figuran a continuación de los mostrados en la Figura 39.

Datos para auditoría:

Recuento votos categoría: [[{"id": "22", 3}, {"id": "23", 2}, {"id": "24", 2}]]

```
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">]>
```

Figura 40: Datos para auditoría. Vista reducida.

Procesos de Selección en VOX

Se listarán los procesos de selección, de la plataforma, para que puedas participar de ellos o ver los resultados.

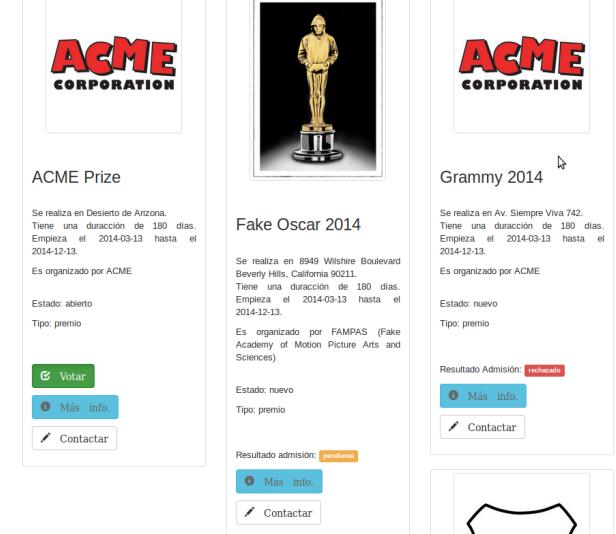


Figura 41: Escritorio jurado. Vista reducida.

Módulo jurado.

Este es el módulo para realizar las votaciones, 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. Pero deberá postularse para cada proceso de selección en los que quiera participar y luego deberá ser aceptado por el organizador del proceso.

Listado procesos.

Ingresando al menú “Escritorio” o estando identificado en el sistema. Este tipo de usuario puede ver un listado de los procesos de selección creados en el sistema.

Para cada proceso de selección figura una descripción, su estado y tipo. Continuando con el estado relativo al usuario; En caso de poder votar aparecerá el botón en Verde, como figura en la imagen (Véase Figura 41), si fue rechazado aparecerá una etiqueta en rojo y naranja si aún está pendiente de consideración.

Ingresando desde el menú “Inicio”, podrá ver una tabla con todos los procesos de forma más compacta que desde el escritorio, como se aprecia en la Figura 42.

Postulación como jurado.

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

Ingresando por el menú “Inicio”, el usuario puede ver una tabla (Véase Figura 42) 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.

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	Votar	Más info.
Fake Oscar 2014	8949 Wilshire Boulevard Beverly Hills, California 90211	180	2014-03-13	2014-12-13	1	nuevo	Solicitar autorización		Más info.
Grammy 2014	Av. Siempre Viva 742	180	2014-03-13	2014-12-13	1	nuevo	Solicitar autorización		Más info.
Premios TIC	Corrientes, Argentina	180	2014-03-13	2014-12-13	1	nuevo	Solicitar autorización		Más info.

Figura 42: Listado compacto de procesos. Vista reducida

Votar.

Una vez que el usuario es aceptado como jurado por el organizador del proceso de selección, el botón “votar” se activará en el “escritorio” como se aprecia en la Figura 41 o aparecerá activado como en la Figura 42 mostrándose de color verde. Al presionarlo, el sistema se redirigirá a la página de votación. En esta mostrará la información básica del proceso de selección seleccionado, como se aprecia en la imagen (Véase Figura 43), una descripción con las fechas claves, información sobre el organizador, una forma de contactarlo. Y el logotipo del organizador.

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, que se aprecian en la imagen (Véase Figura 43).

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

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 Wile E. Coyote Mejor Cohete
 Mejor Trampa Mejor Caída Mejor idea

Figura 43: Inicio votación. Vista reducida.

Candidatos a Mejor Wile E. Coyote Mejor Cohete Mejor Trampa Mejor Caída
 Mejor idea

Candidatos a Mejor idea



Piedra gigante

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

[Piedra gigante](#)



Comida con hierro

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

[Comida con hierro](#)



Tunel falso

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

[Tunel falso](#)

Votar seleccionado

Figura 44: Candidatos a "Mejor idea" durante votación.

Una vez que presione sobre alguno de ellos, se desplegará la información sobre los candidatos de esa categoría. En la imagen (Véase Figura 44) se ven los candidatos luego de presionar sobre la categoría “Mejor idea”, 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 cambiara su color indicando la selección, como se ilustra en la imagen (Véase Figura 45). Seguido a esto para emitir el voto deberá presionar el botón verde inferior “Votar seleccionado”.

Candidatos a Mejor idea



Piedra gigante

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

[Piedra gigante](#)



Comida con hierro

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

[Comida con hierro](#)



Tunel falso

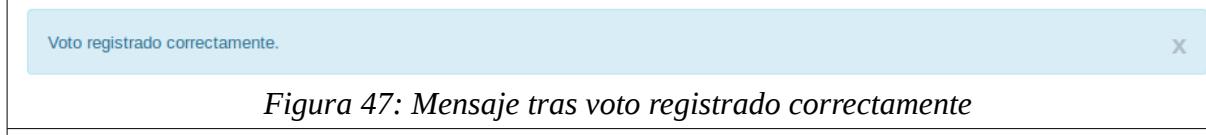
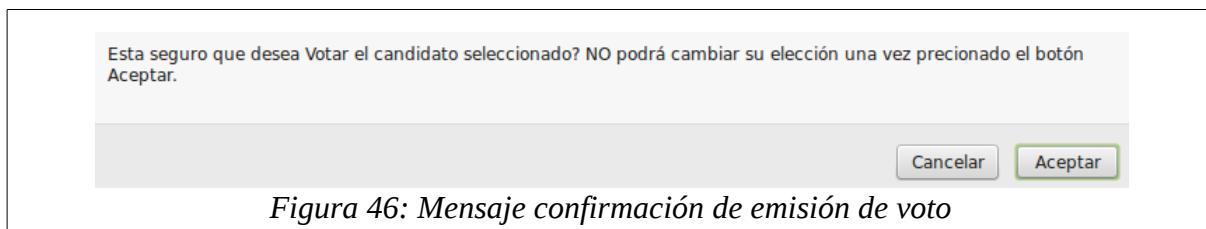
Pintar un tunel falso sobre una pared,
y luego redirigir las líneas del tráfico hacia él.

[Tunel falso](#)

Votar seleccionado

Figura 45: Candidato seleccionado para votar por él.

Debido a que el voto es anónimo e irreversible se le pide al usuario que confirme la operación, como se puede apreciar en la imagen (Véase Figura 46) el mensaje de confirmación.



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 muestra al usuario un mensaje como el de la imagen (Véase Figura 47).

A medida que vaya emitiendo su voto en cada categoría estos se irán deshabilitando, hasta que al finalizar quedarán todas deshabilitadas, con un ícono con tilde, como se ve en la imagen (Véase Figura 48) y si aunque un usuario mal intencionado habilitase un botón por código el voto no se registrará ya que se hacen controles previos.

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 Figura 49)

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.

Módulo identificación.

Inicio de sesión.

Para poder participar de los procesos del sistema web los usuarios deberán identificarse, ingresando sus respectivos usuarios y contraseñas.

Registración.

A excepción del administrador, todos los usuarios deben pasar por un proceso de registración donde se le solicitan 3 datos básicos: nombre de usuario, email y contraseña.

Crea tu cuenta de Vox

Datos Obligatorios:

Nombre de Usuario	<input type="text" value="Ingrese el nombre que utilizará para identificarse en el sistema"/>
Dirección de correo electrónico	<input type="text" value="Ingrese una dirección de correo electrónico valida, nos comu"/>
Clave de acceso	<input type="text"/>
Confirme su clave de acceso	<input type="text"/>

Puede completarlos más tarde:

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

Botones:

- Crear cuenta** (botón azul)
- Iniciar Sesión** (botón gris)

Iniciar Sesión

Correo electrónico

unemail@go.com

Clave

••••••••••

Remember me

Iniciar sesión (botón azul)

Registrarse (botón gris)

[¿Olvidó su clave?](#)

Figura 50: Formulario creación de usuarios.

Figura 51: Inicio de sesión.



Barra de navegación.

- Barra superior.

En esta versión, como se puede observar en la imagen (Véase Figura 52) la barra de navegación tiene dos menús “Inicio” y “Escritorio”, que permiten al usuario dirigirse a dos lugares concretos de visita frecuente. Luego a la derecha se dispone la información sobre el usuario identificado, junto con la opción para finalizar la sesión o modificar sus datos personales.

- Barra lateral.

A la izquierda de la interfaz gráfica se presenta una barra vertical de navegación, cuyos contenidos cambian de forma dinámica dependiendo de la acción que realiza el usuario. En la imagen (Véase Figura 53) se ve la primera vista que tiene para un usuario organizador, que al momento de ingresar al sistema ya tenía creados 2 procesos.

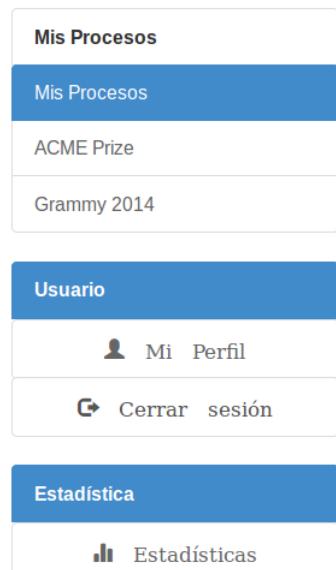


Figura 53: Barra de navegación lateral.

Comentarios y discusiones.

{{ revisar duplicado con lo que esta en conclusiones }}

Resultó ser un proceso que en un principio resultó algo complejo, pero con la práctica y el avance del mismo proyecto se fue simplificando. No se debe perder el punto clave de que, aunque se utilizan pruebas, es una práctica de diseño de software y en el caso de BDD además es una herramienta de comunicación.

La definición de los escenarios para cada historia de usuario es la clave, 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 los “product owner” o responsables del producto y otros interesados. Para este fin los autores recomiendan la participación de todo el equipo de desarrollo en los talleres de requerimientos.

En general aplicar estas técnicas resultó en primera instancia muy difícil por el desconocimiento de las herramientas y de la tecnología utilizada; luego a medida que se avanzó fue resultando más fácil. C. Blé [8] comenta sobre aplicar TDD con una tecnología desconocida y recomienda realizar pequeños experimentos, aparte del proyecto, a los que se los llama “spike” (es un pequeño programa que se escribe para

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

indagar en la herramienta, explorando su funcionalidad) para experimentar con la tecnología a utilizar y familiarizarnos con ella. Con respecto a TDD resulta difícil pensar primero en la prueba, e imaginar cómo es el código que se espera obtener en una nueva tecnología o herramienta. En cambio con BDD resulto más fácil, ya que se tiene en cuenta el comportamiento esperado y no como como se alcanza.

Resulta muy fácil salirse de la técnica, de las pruebas primero, cuando la funcionalidad a agregar parece 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.

La fragilidad de las pruebas se hace notar a medida que se avanza con la técnica, e incluye nuevas funcionalidades al sistema. Por momentos se notó el esfuerzo doble, de mantener las pruebas y mantener el código; no se notaba 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.

Otra cuestión 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. 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 resultó 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 paralices de

Desde el punto de vista del cliente, se estima que este, va a valorar menos las pruebas que se hagan después de mostrar algo que funciona, por eso resulta mejor su integración como parte del proceso previo a la entrega.

El proceso fue guiando actividad de programar, 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.

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.

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. {{Revisar si puede quedar esta última oración o es chocante}}

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 de realizar en el sentido de la detección de los problemas, por las mismas razones que la estimación, se requiere conocimiento sobre la tecnología utilizada, sobre buenas prácticas y principios de diseño de software para aprovecharla. En algunos casos luego de completar la historia de usuario, pasando las pruebas de aceptación, se pasa a la siguiente característica (feature) desperdiando parte del esfuerzo de haber creado esa red de seguridad.

Resultó que refactorizar ayudó a encontrar errores más rápido. Por ejemplo: se estaba refactorizando un mensaje de error en la vista y se creó un helper; cuando se estaba aplicando el helper a Organizadores, se escribió mal el nombre del modelo @organizer y al ejecutar las pruebas se presentaron los errores; entonces al revisar en exactamente ese punto se notó que estaba mal esa variable y que lo correcto era @organizers. En el mismo caso aparecieron errores en las pruebas por el cambio de texto en el mensaje que se mostraba, ya que se pasó de un mensaje específico a uno general.

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.

Este proceso seguido de las pruebas primero es como tejer laboriosamente una red de seguridad, nodo por nodo, pero al final queda una red donde uno se siente muy seguro modificando el código, saltando al vacío de modificar algo que funciona como se espera

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua pero que podría funcionar mejor.

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.

Uno de los módulos se creó sin utilizar las facilidades que incluye este framework, esto se hizo con la intención de entender cómo funcionaba internamente antes de avanzar con las funcionalidades del sistema. 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 que surgen las principales ventajas de utilizar 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.

Luego si se utilizan las facilidades del framework para un módulo, con la propuesta de andamiaje (del inglés *scaffold*) que incorpora, al momento de hacer las pruebas unitarias se enfoca el esfuerzo en los cambios que se realicen para adaptar a las reglas del negocio y no sobre lo que se considera habitual. Se entiende que el framework hace bien su trabajo y no es necesario testearlo, es un esfuerzo extra que no se justifica.

Sobre los errores.

- Hay que aprender a leer los errores e identificar de qué tipo son. Uno termina aprendiendo a leerlos más tranquilo. La tranquilidad al leer un mensaje de error se incrementa a medida que uno se acostumbra a la forma de trabajar.
- 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).
- Si los casos de prueba (ejemplos) no cubren las situaciones, pueden haber errores allí. Por ejemplo no se prueba una ruta a un ID que no existe y generar un error 404: "We're sorry, but something went wrong."
- 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.

- Riesgo: si la herramienta que se usa para realizar las pruebas, no soporta una funcionalidad por ejemplo "Variables de sesión de usuario", entonces al fallar porque no puede leer estas variables no se tomaran esos caminos lógicos, que no contemplen esas variables, o se pondrán excesivas condiciones lógicas.
- Si bien hay pruebas, los errores pueden permanecer ocultos, ya que la metodología es un proceso de diseño y no de prueba. Pero al avanzar uno puede detectar errores o detalles del comportamiento que se podían estar escapado. Por ejemplo el nombre de una 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, errores de personalización que pasen desapercibidos a simple vista.

Errores ante cambio.

Al crear los Procesos de Selección (selection_process), se produjo una equivocación en la creación y se puso en plural: selection_processes, lo cual impacto en todo lo relacionado a esa entidad, en las 3 capas: modelo, controlador y vista. Y las pruebas asociadas.

Para unificar criterios tuve que cambiar a singular el nombre y los errores empezaron a brotar, saber que pruebas fallaban ayudo a medir el impacto del cambio y a solucionar problemas en otras partes del sistema asociadas, como ser la configuración de las rutas. Se evidencio la fragilidad de las pruebas ante cambios en el código: en cucumber fueron 25 pruebas rotas y en TDD (Rspec) fueron más de 46. Afecto a otros modelos asociados por ejemplo Organizer.

Lo interesante de este caso es encontrar 1 a 1 los puntos de fallo en la aplicación, cuando los errores informados por el intérprete no lo hacen de una forma tan clara y precisa.

- A priori, es difícil medir el impacto del cambio. Que las pruebas fallen ante el cambio no significa del todo que este rota por completo; paso que luego de corregir algo para otra prueba, las pruebas dejaron de estar rotas.

- En "módulos" que no tenían pruebas, se volvió a tener el problema de que la plataforma tenía fallas ante ciertos cambios, pero no teníamos forma de saber hasta que alguien de alguna manera probaba ese camino y se producía el fallo. Es decir sin la alerta temprana de las pruebas ¿Cuál es la alternativa?
- Cuando hubo que cambiar una bandera de Permisos (autorización) de "is_organizer?" a "is_owner?", en toda la aplicación, las pruebas ayudaron a comprobar que todo siga funcionando bien luego del cambio.

#Falsos Positivos.

- Falsos positivos desde BDD, uno no sabe si realmente está probando bien, puede tener falso positivos. Por ejemplo tenía que hacer Clic para borrar, pero no hacia el clic y borraba el registro entonces daba Verde el siguiente paso.
- Daba verde pero no estaba probando lo que tenía que probar.

Por ejemplo: en la features "listado_entidades_organizadoras.feature" 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 creaba. Si bien estaba bien la funcionalidad, la prueba era un falso positivo porque en caso de 1 registro sólo la prueba tendría que fallar.

- en Travis, CI, una prueba daba Rojo y no hacia el deploy; pero la prueba fallaba por un error en ella misma y el sistema andaba bien.

Dificultades

- Con el desconocimiento de la tecnología la **estimación** del esfuerzo en Scrum, se sobre evalúa, algunas tareas se pensaban a priori que iban a ser más difíciles de lo que fueron en la primera vez que se aplicó la técnica. Y otras a posteriori llevaron el triple de lo que se pensaba. 1 semana de trabajo para poder subir la una imagen y hacer la prueba.

- Se hace difícil elegir una Gema, que solucione un problema particular, sin conocer la tecnología. Por ejemplo una gema que maneje lo relacionado con las imágenes que subirán los usuarios del sistema.
- ¿Donde poner el foco en los test? ¿testear absolutamente todo? con el uso de Framework ¿vale la pena testear lo que hace el framework?
- Es muy difícil, al no conocer y es más fácil cuando ya se conoce. [[De una discusión de Ágiles: Esta conclusión fue prevista por Watts S. Humphrey en su método PSP [36], donde advierte que dicho método no se debe usar para hacer cosas nuevas y que solo se debe usar cuando se hacen cosas conocidas. Al parecer se puede generalizar a todo método de construcción de software.]] Carlos Blé, habla de esto y habla de hacer spike antes para experimentar.
- Cuando se tenía que indagar sobre como agregar cierta gema, 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 extra.
- **Creación** de pruebas: Al tratar de hacer un test sobre algo que no estoy entendiendo bien, y se vuelve muy dificultoso, no encuentro ejemplos de casos similares. Es probable que este encarando mal la solución. Por ejemplo cree un application_helper para mostrar un mensaje en una tabla, y

Documentación Viva.

- No es fácil de ver o encontrar cierta clase de información como los Valores que puede asumir un Dato (de tipo selección). Por ejemplo para "Estado" de una solicitud de admisión a un proceso: 1. aprobado, 2. pendiente. 3. rechazado.

Si no hay una prueba que los muestre, por ejemplo como las tablas de ejemplos de cucumber, pero debe existir al menos 1 prueba que los utilice de esa forma. En algunos casos los agregue como comentarios "#PO: ..." dejando notas de lo conversado con el Product Owner.

- Esta documentación, no es para el programador actual, es para todo el equipo, es para el que venga después, es para el cliente.

Sección 5: ESTUDIO PRELIMINAR DE IMPLEMENTACIÓN.

{} esto al final no se hizo... {}

Resultado. Análisis

[- Si el TFA incluye relevamiento de datos, incluir una sección de análisis de los resultados.

- Para brindar objetividad científica, aplicar correctamente las herramientas estadísticas.
- En el informe, en la sección Metodología describir las técnicas estadísticas aplicadas en el análisis, con su correspondiente referencia.

]

Capítulo 5. Conclusiones y futuros trabajos

NOTAS APUNTES CÁTEDRA TFA.

- [- Manifiestan lo más destacado de los resultados del TFA.]
- El número de conclusiones depende de la importancia del tema, los aspectos encontrados y lo comprobado.
- Evitar que las conclusiones sean un resumen de cada capítulo.
- La redacción debe ser clara, concreta, directa y enfática.]

{} En la conclusión deben estar como se cumplieron los objetivos {} {} chequear los objetivos {}

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.

Se puede apreciar una mejora en el proceso de desarrollo al contar con una “documentación viva” que refleja fielmente y de forma actualizada lo que hace el sistema.

El procedimiento, bajo BDD y TDD, permite que el código sea modificable más fácilmente que si no se tiene ese soporte de pruebas.

[18][Steve Freeman, Nat Pryce, “Growing Object-Oriented Software, Guided by Tests”, Addison-Wesley Professional, 2010.]

Freeman y Pryce, se puede afirmar que “TDD es una técnica que combina pruebas, especificación y diseño en una única actividad holística”, y que tiene entre sus objetivos iniciales el acortamiento de los ciclos de desarrollo.

Las ventajas que ofrece TDD, de nuevo según Freeman y Pryce [18], son:

- Clarifica los criterios de aceptación de cada requerimiento a implementar.
- Al hacer los componentes fáciles de probar, tiende a bajar el acoplamiento entre los mismos.
- Nos da una especificación ejecutable de lo que el código hace.
- Nos da una serie de pruebas de regresión completa.

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

- Facilita la detección de errores mientras el contexto está fresco en nuestras mentes.
- Nos dice cuándo se debe dejar de programar, disminuyendo el gold-plating y características innecesarias.

Conclusiones CI.

- La experiencia real al aplicar Integración Continua en proyectos de software demuestra que supone un estilo diferente de desarrollar, lo cual implica un cambio de filosofía para el que es necesario un período de adaptación.
- Se deben introducir paulatinamente las pruebas dentro de la construcción para identificar las principales áreas donde existen mayores errores.
- La Integración Continua permite detectar de manera temprana posibles problemas de integración que pueden tener soluciones muy complejas a posteriori.
- A medida que se van aplicando estas prácticas en el desarrollo cotidiano se irá conformando el entorno de Integración Continua encaminado a la obtención de un producto de mayor calidad.

<http://guide.agilealliance.org/guide/bdd.html>

// Notas mentales. Conclusiones Preliminares.

Sobre Errores.

- Hay que aprender a leer los errores e identificar de qué tipo son. Uno termina aprendiendo a leerlos más tranquilo. La tranquilidad al leer un mensaje de error se incrementa a medida que uno se acostumbra a la forma de trabajar.
- 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).
- Si los casos de prueba (ejemplos) no cubren las situaciones, pueden haber errores allí. Por ejemplo no se prueba una ruta a un ID que no existe y generar un error 404: "We're sorry, but something went wrong."
- 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.

- Si bien hay pruebas, los errores pueden permanecer ocultos, ya que la metodología es un proceso de diseño y no de prueba. Pero al avanzar uno puede detectar errores o detalles del comportamiento que se podían estar escapado. Por ejemplo el nombre de una 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, errores de personalización que pasen desapercibidos a simple vista.

// Errores ante cambio.

Al crear los Procesos de Selección (selection_process), me había equivocado en la creación y puse en plural: selection_processes, lo cual impactó en todo lo relacionado a esa entidad, en las 3 capas: modelo, controlador y vista. Y las pruebas asociadas.

Para unificar criterios tuve que cambiar a singular el nombre y los errores empezaron a brotar, saber que pruebas fallaban ayudó a medir el impacto del cambio y a solucionar problemas en otras partes del sistema asociadas, como ser la configuración de las rutas. Se evidenció la fragilidad de las pruebas ante cambios en el código: en cucumber fueron 25 pruebas rotas y en TDD (Rspec) fueron más de 46. Afectó a otros modelos asociados por ejemplo Organizer.

Lo interesante de este caso es encontrar 1 a 1 los puntos de fallo en la aplicación, cuando los errores informados por el intérprete no lo hacen de una forma tan clara y precisa.

- A priori, es difícil medir el impacto del cambio. Que las pruebas fallen ante el cambio no significa del todo que esté rota por completo; paso que luego de corregir algo para otra prueba, las pruebas dejaron de estar rotas.

- En "módulos" que no tenían pruebas, se volvió a tener el problema de que la plataforma tenía fallas ante ciertos cambios, pero no se tuvo forma de saber hasta que alguien, de alguna manera, probó ese camino y se producía el fallo. Es decir sin la alerta temprana de las pruebas ¿Cuál es la alternativa?

- Cuando hubo que cambiar una bandera de Permisos (autorización) de "is_organizer?" a "is_owner?", en toda la aplicación, las pruebas ayudaron a comprobar que todo siga

#Falsos Positivos.

- Falsos positivos desde BDD, uno no sabe si realmente está probando bien, puede tener falso positivos. Por ejemplo tenía que hacer clic para borrar, pero no hacia el clic y borraba el registro entonces daba Verde el siguiente paso.
- Daba verde pero no estaba probando lo que tenía que probar.

Por ejemplo: en la features "listado_entidades_organizadoras.feature" 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 creaba. Si bien estaba bien la funcionalidad, la prueba era un falso positivo porque en caso de 1 registro sólo la prueba tendría que fallar.

- en Travis-CI, CI, una prueba daba Rojo y no hacia el deploy; pero la prueba fallaba por un error en ella misma y el sistema andaba bien.

Dificultades

- Con el desconocimiento de la tecnología la **estimación** del esfuerzo en Scrum, se sobre evalúa, algunas tareas se pensaban a priori que iban a ser más difíciles de lo que fueron en la primera vez que se aplicó la técnica. Y otras a posteriori llevaron el triple de lo que se pensaba. 1 semana de trabajo para poder subir la una imagen y hacer la prueba.
- Se hace difícil elegir una Gema, que solucione un problema particular, sin conocer la tecnología. Por ejemplo una gema que maneje lo relacionado con las imágenes que subirán los usuarios del sistema.
- Donde poner el foco en los test? testear absolutamente todo? con el uso de Framework ¿vale la pena testear lo que hace el framework?
- Es muy difícil, al no conocer y es más fácil cuando ya se conoce. [[De una discusión de Ágiles: Esta conclusión fue prevista por Watts S. Humphrey en su método PSP [36], donde

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

advierte que dicho método no se debe usar para hacer cosas nuevas y que solo se debe usar cuando se hacen cosas conocidas. Al parecer se puede generalizar a todo método de construcción de software.]] Carlos Blé, habla de esto y habla de hacer Spike antes para experimentar.

- Cuando se tenía que indagar sobre como agregar cierta gema, 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 extra.

- **Creación** de pruebas: Al tratar de hacer un test sobre algo que no estoy entendiendo bien, y se vuelve muy dificultoso, no encuentro ejemplos de casos similares. Es probable que este encarando mal la solución. Por ejemplo cree un application_helper para mostrar un mensaje en una tabla, y

5.2 Futuros trabajos.

NOTAS APUNTES CÁTEDRA TFA.

{{ Estas notas se borrarán al terminar la revisión del capítulo }}

[En algunos casos, se pueden incluir futuros trabajos o líneas de desarrollo.]

- Utilizar herramientas para revisar el código y refactorizar, como CodeClimate y CodeReview.
- Profundizar sobre la práctica de Refactoring y aplicar diferentes tipos de ella a este trabajo.
- análisis del impacto en los recursos humanos al aplicar estas técnicas.
- Una metodología para su implementación en una empresa.
- Aplicar las técnicas a sistemas legacy (legacy code).
- Comparar el enfoque Top-down, con el Button-up aplicando TDD/BDD
- Técnicas y herramientas para acelerar y optimizar la ejecución de los test.

Con respecto a la aplicación generada:

- Agregar funcionalidades, agregar lo relativo a los Certámenes.
- Agregar un sistema de plantillas, para los eventos.
- Mejorar el diseño gráfico.

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

- 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.

Referencias {{ revisar como se escribe la Edición }}

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. 1 y 2, pp. 07-22.
2. K. Beck, "Extreme Programming Explained: Embrace Change", Boston: Addison-Wesley Professional, 1999. cap 7 pp. 30 cap 8 pp. 35. {{ faltan edición }}
3. .K. Beck, "Test Driven Development: By Example", Primera Edición. Boston: Addison-Wesley Professional, 2002. cap. 31-32 cap. 0. (prefacio). {faltan qué hojas}
4. RSpec Development Team, USA, "RSpec," Disponible: <http://rspec.info/>. Consultado el: Otc. 01, 2014.
5. D. North. (2006). Introducing BDD [Online]. Disponible: <http://dannorth.net/introducing-bdd/>, Consultado el: May. 8, 2013.
6. D. A. Patterson and A. Fox, *Engineering Software as a Service: An Agile Approach Using Cloud Computing*, Primera Edición. Strawberry Canyon LLC, 2014. cp. 5 pp 355-360. {{ faltan ciudad de la editorial }}
7. M. Gärtner, "ATDD by Example: A Practical Guide to Acceptance Test-Driven Development" Boston: Addison-Wesley Professional, 2012. cap. 9 pp 131-133. {{ faltan qué edición }}
8. C. Blé Jurado. "Diseño Ágil con TDD", Primera Edición. Madrid, España: iExpertos, 2010, cap. 1, pp. 33-52-, cap. 3 pp. 67.
9. D. Chelimsky, "The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends", Primera Edición. The Pragmatic Programmers, LLC. , 2010. pag. 25. {{ faltan la ciudad de la editorial }}
10. H. Lopes Tavares, G. Guimarães Rezende, V. Mota dos Santos, R. Soares Manhães, 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. cap. 1, pp. 1.
11. M. Fowler. (2006). Continuous Integration [Online]. Disponible: <http://martinfowler.com/articles/continuousIntegration.html>, Consultado el: May. 8, 2013.

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

12. C. Fontela, "Cobertura entre pruebas a distintos niveles para refactorizaciones más seguras", Facultad de Informática, Universidad Nacional de La Plata, La Plata, Argentina, 2013. cap. 1, pp. 10.
13. D. M. Brown, "Communicating Design: Developing Web Site Documentation for Design and Planning", Segunda Edición. New Riders. 2011. {{ falta pagina y ciudad de la editorial }}
14. I. Sommerville, "Verificación y Validación" en "Ingeniería del Software" Novena Edición. Madrid, España: Pearson Educación S.A., 2005, cap. 22, sec. 5, pp. 470.
15. Pivotal Labs, Inc., San Francisco, CA, USA, "PivotalTracker," Disponible: <http://www.pivotaltracker.com/>. Consultado el: Sep. 28, 2014.
16. Ruby, "Lenguaje de programación Ruby,". Disponible: <https://www.ruby-lang.org/es/>. Consultado el: Sep. 23, 2014.
17. D. H. Hansson, "Ruby on Rails," [Online]. Disponible: <http://rubyonrails.org/>. Consultado el: Sep. 23, 2014.
18. Neurogami, Scottsdale, AZ., USA, "WEBrick," Disponible: <http://ruby-doc.org/libdoc/2.1.1/webrick/rdoc/WEBrick.html>. Consultado el: Sep. 26, 2014.
19. Boostrap, San Francisco, CA, USA, "Boostrap,". Disponible: <http://getbootstrap.com/> . Consultado el: Sep. 9, 2014.
20. J. Spurlock, "Bootstrap", Primera Edición, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472. 2013. {{ faltan paginas }}
21. SQLite Consortium, Charlotte, NC, USA, "SQLite,". Disponible: <http://www.sqlite.org/about.html>. Consultado el: Sep. 25, 2014.
22. M. Owens, "The Definitive Guide to SQLite", Apress, Berkeley, CA, 2010. {{ falta Nro de Edicion y Paginas }}
23. Oracle Corporation, Redwood, CA, USA, "MySQL Worckbench," Disponible: <http://www.mysql.com/products/workbench/>. Consultado el: Sep. 25, 2014.
24. S. Chacon, "Pro Git", Primera Edición. Apress, 2009. {{ faltan paginas y ciudad de la editorial}}
25. GitHub Inc., San Francisco, CA, USA, "GitHub," Disponible: <https://github.com/>. Consultado el: Sep. 26, 2014.

Desarrollo de software multiplataforma utilizando BDD, TDD e Integración Continua

26. LEMUR Heavy Industries LLC., Venice, CA, USA, "Coveralls.io," Disponible: <https://coveralls.io/>. Consultado el: Sep. 26, 2014.
27. Travis CI GmbH, Berlin, Germany, "Travis CI," Disponible: <https://travis-ci.org/>. Consultado el: Sep. 29, 2014.
28. Cucumber Ltd., Cairndow, Argyll, Scotland, "Cucumber," Disponible: <http://cukes.info/>. Consultado el: Sep. 30, 2014.
29. M. Wynne and A. A. Hellesøy, "*The Cucumber Book: Behaviour-Driven Development for Testers and Developers*", Primera Edición. Pragmatic Programmers, LLC., 2012. {{ faltan paginas y ciudad de la editorial }}
30. Elabs AB, Göteborg, Sweden, "Capybara," Disponible: <http://jnicklas.github.io/capybara/>. Consultado el: Oct. 01, 2014.
31. T. Guillaume-Gentil, La Chaux-de-Fonds, Switzerland, "Guard," Disponible: <http://jnicklas.github.io/capybara/>. Consultado el: Oct. 01, 2014.
32. 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.
33. Heroku Inc., San Francisco, CA, USA, "Heroku," Disponible: <https://www.heroku.com/about>. Consultado el: Oct. 10, 2014.
34. A. Hanjura, "Heroku Cloud Application Development: A comprehensive guide to help you build, deploy, and troubleshoot cloud applications seamlessly using Heroku", Primera Edición, Packt Publishing Ltd., Birmingham, UK, 2014. pp. 8-12. {{ faltan capitulo }}
35. PostgreSQL Global Development Group, CA, USA, "PostgreSQL," Disponible: <http://www.postgresql.org/>. Consultado el: Sep. 25, 2014.
36. W. S. Humphrey, "Software Estimating" en "PSP: A Self-Improvement Process for Software Engineers", Primera Edición. Pearson Education, Inc., Massachusetts, 2005 ch.5 pp. 84.