Universidad Nacional del Nordeste.
Facultad de Ciencias Exactas y Naturales y Agrimensura.
Licenciatura en Sistema de Información.

DESARROLLO DE SOFTWARE MULTIPLATAFORMA UTILIZANDO BDD/TDD

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 Itati

[Presentación Preliminar | Presentación Definitiva]

Borrador v1.22.1

Año: 2014

Prologo

Este trabajo se realizo en el Departamento de Programación de la Facultad de Ciencias Exactas, Naturales y Agrimensura de la Universidad Nacional del Nordeste y forma parte del denominado “Trabajo Final de Aplicación”, con fin de cumplimentar las exigencias para la obtención del titulo de la carrera Licenciatura en Sistemas de Información. El desarrollo se hizo sobre la investigación bibliográfica de metodologías y herramientas actuales en el área de la Informática.

Luego de personalmente haber vivenciado en varios proyectos las dificultades clásicas del desarrollo de software, se me genero la motivación para indagar sobre alternativas para la creación de software de calidad que sigan algún modelo comprobado de éxito de la industria. De ello resulto la elección de las practicas 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 practica de Integración Continua.

Al mismo tiempo, y como una forma de aplicarlo en un problema real, se detecto la necesidad de un sistema para la administración de procesos de selección para premios y certámenes en Organizaciones del tercer sector; que remplace 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 solucionar la problemática, y aplicar los conceptos teóricos en un caso especifico, se propuso realizar un sistema multiplataforma utilizando tecnologías y herramientas compatibles con las practicas ágiles seleccionadas y útiles para la resolución del problema. {{ podría ir a Fundamentación?? }}

En su desarrollo colaboraron colegas, como Leandro A. Rodriguez y Simón Y. Ibalo quienes aportaron sugerencias. Y la profesora Mgter. Gladys Noemí Dapozo quien oriento y motivo 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 capitulo }}

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 capitulo }}

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 capitulo }}

Esto se hará al finalizar el informe.

Índice de contenidos

Índice de figuras

Índice de ilustraciones

[Para referencias de las imagenes en el texto usar (Véase tabla VI) o (Véase figura 12)]

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

Índice de tablas

Índice de tablas

Capitulo 1. Introducción

NOTAS APUNTES CATEDRA TFA.

[La introducción sección del TFA donde se presenta formalmente el trabajo. - Manifiesta el objetivo del TFA, las razones que motivaron su ejecución y los fundamentos que lo apoyan. - También puede incluir el estado del arte o marco teórico. Se compone de - Breve estado del arte. - Objetivo(s) – Fundamentación]

Estado del Arte.

[Estado de Arte
Da cuenta el avance de la investigación en un campo del conocimiento.
- Ofrece una visión global del área de investigación considerado
- Puede redactarse luego de realizar la búsqueda y selección bibliográfica.
- Permite conocer qué se ha hecho y evitar duplicar esfuerzos o errores superados.
- Ampliar lo expuesto en el plan de TFA

Para la redacción del estado del arte, considerar que:
- no consiste en listar los trabajos relacionados o resumirlos.
- debe dar cuenta de la comprensión del campo, las tecnologías empleadas, las técnicas, los algoritmos alternativos, entre otros.

Fundamentación.

[Para iniciar el trabajo incluir lo especificado en el proyecto de TFA. Se recomienda incorporar al menos dos antecedentes más]
[- Explicitar los conceptos teóricos que explican el problema planteado.
- Señalar los aspectos teóricos que se relacionan directamente con el problema.
- Definir los conceptos relacionados que no tienen un uso unívoco.
- Se trata de una síntesis inicial que se completará a medida que avanza la investigación.
- Los supuestos en que se basa la investigación deben derivar del contenido de este capítulo.
]

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

Capitulo 1. Introducción

1.1 Estado del Arte.

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. Sin embargo otros han querido enfatizar su adecuación como herramienta de especificación de requerimientos. Y en los últimos años se ha comenzado a avanzar 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, ATDD pone su foco en las pruebas de aceptación y BDD propone poner el foco en el comportamiento esperado por el usuario y en el modo de escribir las pruebas.

1.1.1 TDD

En 1999 Kent Beck publica “eXtreme Programmig” [5], una propuesta de ágilismo que, con base en 4 valores y 5 principios claves, proponía la utilización de 20 prácticas, ya conocidas, para hacer desarrollo de software; de entre ellas nace el Desarrollo dirigido por pruebas[1] (en ingles Test-Driven Development) o 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” [5]. Y luego profundizada en “Test Driven Development: By Example” [6] en 2002.
Básicamente, incluye tres sub-prácticas:

- **Pruebas primero** (Test-First): las pruebas se escriben antes de código que prueban. 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.

1 En ingles “Test-Driven Development”. En adelante figurará como TDD, su sigla en inglés.

- **Refactorización**[23]: 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 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 practicas se sintetizan en lo que se conoce comúnmente como el algoritmo de TDD (véase figura 1), o como Beck lo llamo “el Mantra TDD” [6]:

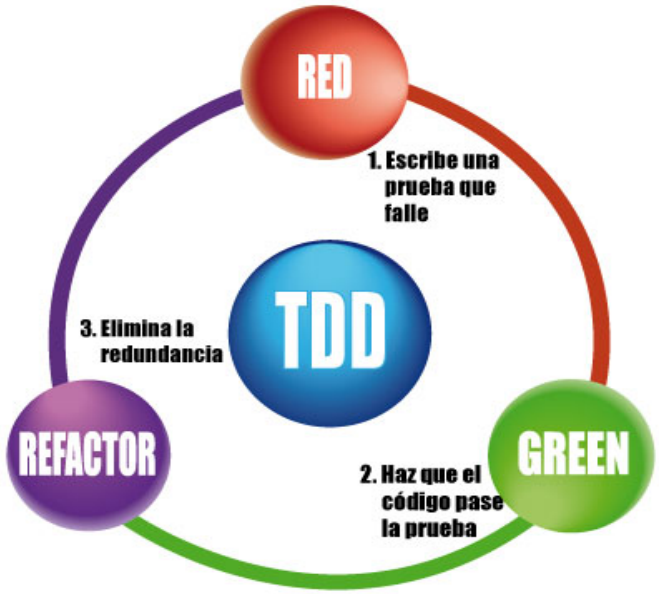


Figura 1: Esquema TDD

1. **Rojo**: se escribe una pequeña prueba que no funcionara, y posiblemente al principio no compile.
2. **Verde**: crear cel código que haga que la prueba pase rápidamente, sin importar los errores que puedas cometer.
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” [6]. 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 implican 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 pequeños funcionalidades especificas 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 practicas alternativas, que ponen el foco en otros aspectos.

---

2 Beck toma la idea presentada por M. Fowler. en “Refactoring: Improving the Design of Existing Code”, Boston: Addison-Wesley, 1999.

3 Se utiliza “refactorizar” como traducción aceptada del termino en inglés “refactoring”.

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 [6] en 2002. 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.

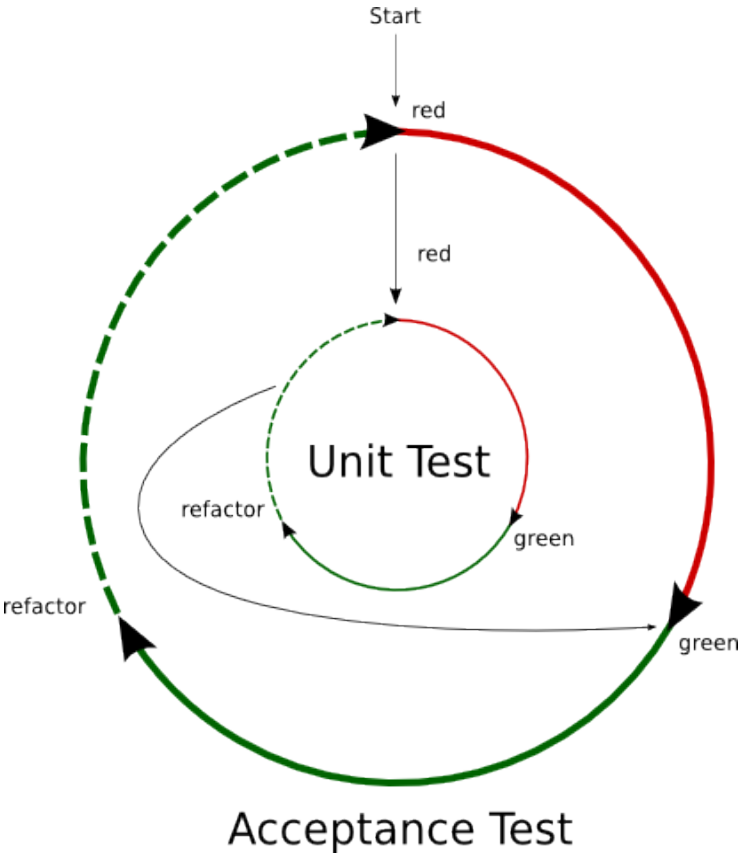


Figura 2: Esquema ATDD

La idea es tomar cada requerimiento, en la forma de una historia de usuario (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. Estas representan el requerimiento, no que lo especifican. Las pruebas de aceptación de una historia de usuario se convierten en las condiciones de satisfacción del mismo. No importa como se implementen (caja negra, blanca, integración, unitarias) lo que importa es la intención de lo que tratan de probar. Se basan en ejemplos.
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 [26.RSpecBook 28.RspecSitio.], las pruebas se siguen codificando en un lenguaje que sólo los técnicos entienden, con la diferencia que se incluye al cliente en el proceso.

1.1.2 BDD

En respuesta a las criticas que se encontraron en TDD, Dan North empezó a hablar de Behavour-Driven Development (BDD), en español Desarrollo Dirigido por Comportamiento[4], en un articulo llamado “Behavior Modification” de la revista Better Software publicado en Marzo de 2006 disponible en [7]. Según North, BDD esta pensado para hacer estas prácticas ágiles más accesibles y efectivas a los equipos de trabajo que quieren empezar con ellas. Principalmente propone que en vez de pensar en términos de pruebas, deberíamos pensar en términos de especificaciones o comportamiento. De ese modo, se las pueden 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.

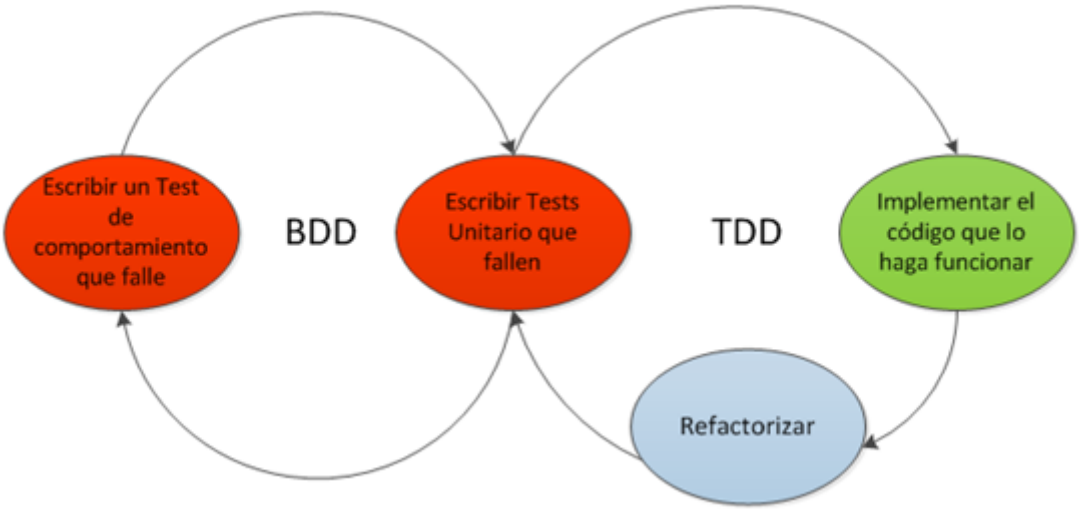


Figura 3: Esquema BDD

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 (requisitos) con el formato Connexa (“Como [X] Deseo [Y] para lograr [Z]”) más el agregado de las clausulas 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

4 En adelante figurará como BDD, su sigla en inglés.

criterios de aceptación. También se puede usar “datos tabulados”, donde se utilizan ciertas estructuras de tablas para expresar requerimientos [8]
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. [7]
Fontela [1] comenta que en el principio 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. El foco está ahora puesto decididamente en una audiencia mayor, que excede a los desarrolladores, e incluye a analistas de negocio, testers, clientes y otros interesados.

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. 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], nos 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[5].
- Usar un lenguaje único, más cerca del consumidor.
- Focalizar en la comunicación, no en las pruebas.
- Simplificar las refactorizaciones o cambios de diseño.

La crítica principal de este enfoque de BDD y ATDD viene de que si bien se pueden derivar pruebas individuales de los contratos, es imposible el camino inverso, ya que miles de pruebas individuales no pueden reemplazar la abstracción de una especificación contractual. En respuesta, Ward Cunningham en [9] 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 [10] amplía esto diciendo que los ejemplos concretos son fáciles de leer, fáciles de entender y fáciles de validar.

5 Se denomina “gold-plating”, en la jerga de administración de proyectos, a incorporar funcionalidades o cualidades a un producto, aunque el cliente no lo necesite ni lo haya solicitado.

1.1.3 Practicas asociadas a BDD/TDD.

Integración Continua

La integración continua (del inglés “Continuous Integration” o simplemente CI), es una práctica introducida por Beck en XP [5] y luego muy difundida a partir del trabajo de Fowler [9]. 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 modulo que se quiere introducir en el 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 nuestro software desde las fuentes y lo valide ejecutando pruebas, más de una vez al día, obteniendo información de retroalimentación inmediatamente después de la ejecución de este proceso.

Un entorno de Integración Continua está compuesto por diferentes herramientas y prácticas en las cuales necesita apoyarse, en la imagen (Véase figura figura 4) 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.

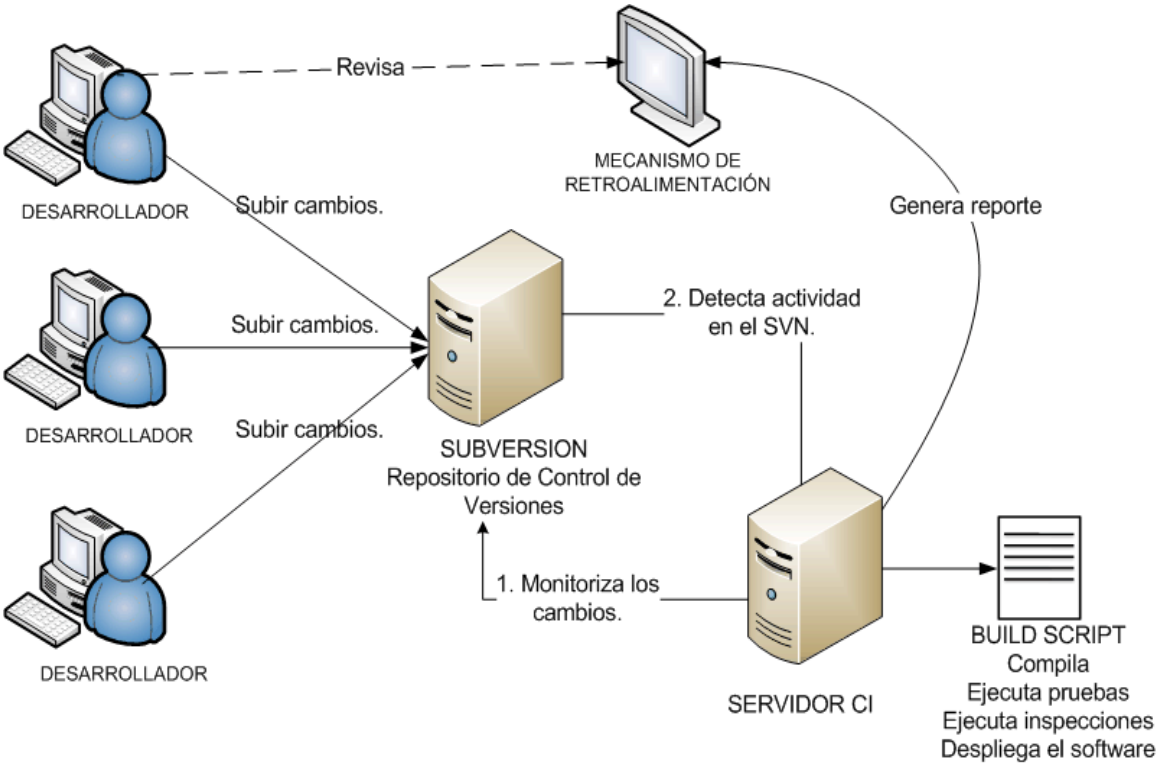


Figura 4: Escenario de Integración Continua

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 practicas 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[6] diarios.
- Ejecución automática y periódica de las pruebas añadidas.
- Administrar los resultados y enviar informes a todas las partes implicadas.

6 Commit: es la acción de comprometer un cambio, en un entorno transaccional. Generalmente no se la traduce.

La relación entre BDD/TDD e integración continua viene de la propia definición de la CI, ya que para poder realizarla debimos haber construido pruebas automatizadas previas a la integración.

Refactorización.

Del inglés Refactoring[7] [8], 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 esta 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. Recordemos 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 estamos cambiando código que funciona por otro que no sabemos si funcionará igual que antes. Entonces una practica 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 [11], 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 [5]. 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.

Segun Fontela [1], la refactorización aparece relacionada con BDD/TDD en dos sentidos:

---

7 El termino aparece por primera vez en el trabajo de William F. Opdyke, R. Johnson, “Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems”, Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPPA), 1990, ACM.

8 En el marco de este trabajo, se utiliza el término en inglés (“refactoring”) solamente para el nombre de la práctica metodológica. Tanto el sustantivo que describe un cambio particular (“refactorización”) como el verbo que describe la acción de realizar el cambio (“refactorizar”) se utilizarán en castellano.

- 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 queremos facilitar la refactorización, tendríamos 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 trabajamos con un buen diseño orientado a objetos, que trabaje contra interfaces y no contra implementaciones. Y en algún punto, probablemente en las pruebas unitarias, debamos admitir cambios a las mismas.

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 (*Behavour-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 practica.

Objetivos Específicos:

- Realizar una revisión bibliográfica sobre el estado del arte, los conceptos y las técnicas que conforman el desarrollo dirigido por comportamiento (TDD) y prácticas asociadas.
- Detectar las ventajas de este enfoque, mediante la aplicación de sus conceptos y técnicas en el desarrollo de una aplicación informática concreta.
- Construir un software multiplataforma que permita a una institución organizar premios y certámenes, gestionando a través de este todos los procesos

involucrados; integrando los conocimientos sobre el desarrollo dirigido por pruebas para la resolución del problema.

1.3 Hipótesis

Es posible adaptar y aplicar las practicas ágiles de BDD, TDD e Integración Continua en una aplicación web, construida con el lenguaje Ruby, en el contexto del trabajo final carrera.

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 llamo “La Crisis del Software” entonces surgió la Ingeniería del Software (IS). Desde esta perspectiva, con respecto a las pruebas, dicen *“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*” [4].

Sommerville [4] considera a esta temática como 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 la consideran una tarea que se realiza al final del proceso de desarrollo, sea este iterativo o incremental.

Para proyectos que durarán años esto puede ser valido pero la vida de los productos es cada vez más corta. 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.

Posteriormente surgió el Ágilismo como posible solución. La finalidad de los distintos métodos que componen el ágilismo es reducir los problemas clásicos de los programas, a la par de dar más valor a las personas que componen el equipo de desarrollo. Entre estos existe una practica 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. La esencia del ágilismo es la habilidad para adaptarse a los cambios. 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. [2] Sin desmedro de lo anterior, fueron surgiendo alternativas de mejoras, BDD propone poner en 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.
Estas practicas ágiles ponen el foco en “hacer la cosa correcta” desde el primer momento y con el soporte de las pruebas “hacer la cosa correctamente”.

Como, ya en 2010, mencionaban en [10], 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. Desde su origen, BDD tiene el objetivo integrar la verificación y la validación en una técnica de diseño con estilo de afuera hacia adentro (outside-in), es decir, empezando por la parte del software que ven los usuarios hasta las unidades básicas, al tiempo que reduce los costos de garantía de la calidad del software. Mientras se concentra de forma directa en la conexión entre los requisitos 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 requisitos con el código, de forma de facilitar la escritura de las pruebas y el proceso en general.

El problema de la comunicación entre el Equipo de Desarrollo y los Usuarios.
{{ Mencionar la importancia de los Ejemplos, en el libro de Carlos Ble o de Gojko Adzic, para dirigir la especificación del comportamiento esperado }}
“gap” entre el cliente y el equipo de desarrollo. El teléfono descompuesto.

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 detecto la necesidad de utilizar algún tipo de

software para gestionar todo el proceso organizativo de estos eventos, incluyendo sus procesos de votación. Actualmente estas 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 buscará aplicar la técnica BDD/TDD para comprobar sus beneficios en un problema real mediante la construcción de una aplicación multiplataforma; esta 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.

Este trabajo pretendió aplicar TDD a un caso real de un sistema web y termino encontrando en BDD una herramienta ideal para la comunicación con los clientes, que permitió aplicar la practicar de Integración Continua.

1.6. Organización del Trabajo.
El presente trabajo se encuentra organizado en capítulos, de los cuales en el capítulo 1 se expone se introduce en el marco teórico. Además se exponen los objetivos principales del trabajo junto con la hipótesis.

En el capítulo 2 se explica la metodologías aplicacdas en el trabajo, dentro de las cuales se abordan las metodologías ágiles de desarrollo y ... . Además se explica ...

En el capítulo 3 se enumeran las herramientas que se han usado para el desarrollo del sistema propuesto para resolver el problema de aplicación, haciendo una breve reseña de estas, marcando alguna de sus principales caracteristicas y su utlización en el trabajo.

En el capítulo 4 se muestran los resultados obtenido mediante el software, en el cual se describe junto a cada una de sus partes como ser el analisis, el modelo de datos que utiliza y una revisión de las principales partes del mismo.

Al final en el capítulo 5 se comentan algunas conclusiones a las que se ha llegado. Además se proponen algunos posibles lineamientos para continuar.

=== === ===
En este capítulo se podrá incluir:

Breve estado del arte

Objetivo(s)

Fundamentación

NOTAS APUNTES CATEDRA TFA.

[Para iniciar el trabajo incluir lo especificado en el proyecto de TFA. Se recomienda incorporar al menos dos antecedentes más]

[- Explicitar los conceptos teóricos que explican el problema planteado.

- Señalar los aspectos teóricos que se relacionan directamente con el problema.

- Definir los conceptos relacionados que no tienen un uso unívoco.

- Se trata de una síntesis inicial que se completará a medida que avanza la investigación.

- Los supuestos en que se basa la investigación deben derivar del contenido de este capítulo.

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

[Metodología Ágil]

[Practicas de Programación Extrema]

La mayor diferencia entre las metodologías clásicas y la Programación Extrema es la forma en que se expresan los requisitos de negocio; donde en lugar de documentos, son ejemplos ejecutables. [2]

Los tests de aceptación o de cliente son el criterio escrito de que el software cumple los requisitos de negocio que el cliente demanda. Son ejemplos escritos por los dueños de producto. Es el punto de partida del desarrollo en cada iteración, la conexión perfecta entre Scrum y XP; allá donde una se queda y sigue la otra. [2]

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

En ATDD la lista de ejemplos (tests) de cada historia, se escribe en una reunión que incluye a dueños de producto, desarrolladores y responsables de calidad. 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 sotfware 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.
[Historias de Usuario]

[BDD]

“In both approaches there are three commonly used styles: tables that take inputs and check outputs from the system, tables that query the system and check collections of values against the ones obtained from the system under test, and tables that support workflows.”
Decision Tables: The first set of tables take input values to the system, execute something in the system, and then check some values from the system against the ones provided.
Query Tables: datos de ejemplo. are used after gathering data from the system and receiving a list of different things stored in the system.
Script Tables: You can use script tables to express workflows in your system. You can also use script tables to combine decision tables with query tables in one test.

“One of the mantras in behavior-driven development is outside-in development.” [8]

[TDD]

Practicas asociadas a TDD/BB.

- [Refactoring]
- [Integración Continua]

[Scrum] para la administración del proyecto.

[Patrón MVC] {{ Hay que ver hasta donde se justifica hablar de esto }}

BDD

Un test de cliente o de aceptación con estos frameworks, a nivel de código, es un enlace entre el ejemplo y el código fuente que lo implementa. [2]

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

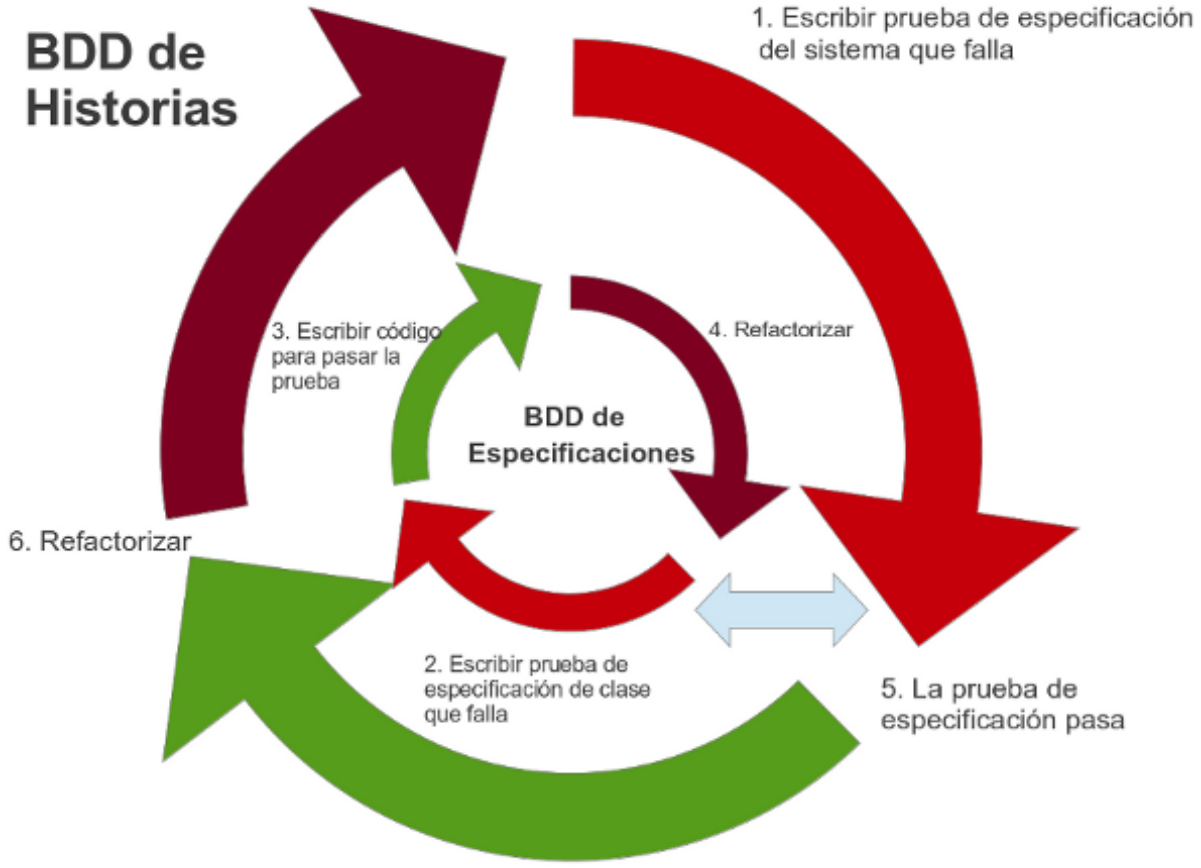


Figura 5: Esquema BDD Completo.

[Patrón MVC]

[Historias de Usuario]

The first step in the Agile cycle, and often the most difficult, is a dialogue with each of the stakeholders to understand the requirements. We first derive *user stories*, which are short narratives each describing a specific interaction between some stakeholder and the application. The *Cucumber* tool turns these stylized but informal English narratives into

acceptance and integration tests. As SaaS usually involves end-users, we also need a user interface. We do this with *low-fidelity (Lo-Fi)* drawings of the Web pages and combine them into *storyboards* before creating the UI in HTML.

User stories came from the Human Computer Interface (HCI) community. They developed them using 3-inch by 5-inch (76 mm by 127 mm) index cards, known as "3-by-5 cards." (We'll see other examples of paper and pencil technology from the HCI community shortly.) These cards contain one to three sentences written in everyday nontechnical language written jointly by the customers and developers. The rationale is that paper cards are nonthreatening and easy to rearrange, thereby enhancing brainstorming and prioritizing. The general guidelines for the user stories themselves is that they must be testable, be small enough to implement in one iteration, and have business value. Section 5.2 gives more detailed guidance for good user stories.

This user story format was developed by the startup company Connextra and is named after them

Feature name

As a [kind of stakeholder],
So that [I can achieve some goal],
I want to [do some task]

Summary of BDD and User Stories

BDD emphasizes working with stakeholders to define the behavior of the system being developed. Stakeholders include nearly everyone: customers, developers, managers, operators, ...

User stories, a device borrowed from the HCI community, make it easy for nontechnical stakeholders to help create requirements.

3x5 cards, each with a user story of one to three sentences, are an easy and nonthreatening technology that lets *all* stakeholders brainstorm and prioritize features.

The Connextra format of user stories captures the stakeholder, the stakeholder's goal for the user story, and the task at hand.

[SaaS Book]

Capitulo 2. Metodología

NOTAS APUNTES CATEDRA TFA.

[- Contiene una formulación esquemática del procedimiento general desarrollado para probar las hipótesis o realizar el trabajo.

- Mencionar y describir las etapas y sus fases aplicadas en el desarrollo del TFA.

- En trabajos publicados en el campo de la informática y la computación, los autores indican los pasos que persiguen para lograr los resultados.

- Pueden especificarse las técnicas de recolección y análisis de datos.]

[Para iniciar el trabajo incluir lo especificado en el proyecto de TFA. Si correspondiera los avances logrados y expuestos recientemente] {{ Estas notas se borrarán al terminar la revisión del capitulo }}

Etapa 1: Investigación preliminar.

1.1. Se realizará 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.

1.2. Además, se realizará un relevamiento y evaluación de las herramientas tecnológicas que soportan las prácticas propuestas por el TDD, a fin de seleccionar aquellas que mejor se adecuen a los objetivos del trabajo.

Etapa 2: Diseño y desarrollo de una aplicación.

Se diseñará y desarrollará una aplicación de la vida real, orientada a la gestión de certámenes y concursos. La aplicación tendrá algunos módulos desarrollados mediante TDD y otros módulos, con funcionalidades similares, siguiendo técnicas tradicionales. Se aplicarán conceptos tales como casos de prueba, ejecución de pruebas, automatización de pruebas, dobles de pruebas, entre otros.

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

2.1 Determinar y comprender el dominio de la aplicación.

2.1.1 Contactar organizadores de premios y certámenes, indagando cómo se organizan actualmente. (Detectar necesidades).

2.1.2 Investigar qué servicios y sistemas similares existen en el mercado (Local, Nacional, Hispano e Internacional).

Producto: documento ERS (Especificación de Requisitos de Software).

3 Construcción del Producto Mínimo Viable. (Prototipo Funcional)

3.1 Construir un producto mínimo viable, incluye algunos requisitos funcionales y no funcionales de software; con la aplicación de la técnica de TDD (desarrollo guiado por pruebas)

3.1.1 Confeccionar prototipo de interfaces (mockup)

3.1.2 Confeccionar historias de usuario sobre requisitos para modulo B.

3.1.3 Codificar modulo siguiendo TDD.

3.1.4 Ejecutar pruebas, medir tasa de fallo y errores.

3.2 Conclusiones:

- Comparar experiencias de desarrollo (variables cualitativas y cuantitativas)
- Despliegue de la aplicación.

Producto: prototipo funcional (sistema web)

– “Specification Workshops” para levantar las historias.

Proceso:

Procesos Incrementales: entregan versiones,

Procesos Iterativos: la primera versión tiene todos los requisitos y luego se van mejorando.

[Metodología Ágil]

TDD y enfoques metodológicos

TDD nació con un enfoque metodológico que era el de XP, por otro lado bastante laxo. Sin embargo, esto ha ido evolucionando, y en esa evolución mucho tuvo que ver la historia de las herramientas y técnicas particulares.

Fowler [15] propone hacer lo que él llama “diseño de afuera hacia adentro” (Outside-in Software Development). En esta metodología, se parte de la interfaz de usuario y se pasa a la capa media, y así sucesivamente. El problema de esta solución es que requiere muchos objetos ficticios aún no desarrollados.

[15] Martin Fowler, “Mocks Aren´t Stubs”,
http://martinfowler.com/articles/mocksArentStubs.html , como estaba en junio de 2011.

La ventaja de cualquiera de los enfoques top-down es que podemos atacar de a un requerimiento por vez, lo cual está muy alineado con las ideas de los métodos ágiles, que buscan cerrar cuanto antes pequeñas funcionalidades, en forma incremental.

Outside-in Software Development: A Practical Approach to Building Successful Stakeholder-based Products

[Patrón MVC]
[BDD]
[TDD]
[Integración Continua]

PROCESO ITERATIVO REALIZADO.

a. Dialogo con el cliente.
b. Lo-fi UI mockup.
    - Prototipo paginas (Wireframes, mockup).
    - Storyboard: navegación entre paginas.
c. User stories & scenarios.
d. BDD Behavior-driven Desing / user stories.
c. Test-first dev. (unit. / funct.)
d. Measure Velocity (Pivotaltracker)
e. Deploy. (Travis, heroku)
f. pruebas exploratorias.

Capitulo 3. Herramientas y/o lenguajes de programación

NOTAS APUNTES CATEDRA TFA.

[Describir sintéticamente las herramientas y/o lenguajes de análisis y programación empleados en la realización del trabajo, si correspondiere.

- Indicar en cada herramienta o lenguaje de programación su aplicación en el desarrollo del TFA.

- Si se incluyen ejemplos, deberán ser relevantes al problema planteado.

- En general, este capítulo no debería ser un manual de uso de la herramienta.

- Incluir referencias.]

[Mencionar sintéticamente las herramientas / lenguajes de programación utilizados y fundamentar su uso en el TFA]

{{ **TO-DO**: reducir los textos innecesariamente extensos en Herramientas }}

Capitulo 3. Herramientas y/o lenguajes de programación

Las metodología seleccionada se apoya en herramientas para alcanzar sus objetivos, de allí la longitud de este capitulo. Las herramientas utilizadas se clasificaran según su uso en 3 tipos de entornos de trabajo: Desarrollo, Pruebas y Producción. En el titulo de cada una se sigue el patrón Nombre – Función 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.

Historias de Usuario. PostIt. - Captura de requisitos. Herramienta de Análisis.

{{ mandar a metodología }}

Las historias de usuario son unas narraciones cortas, en lenguaje natural; en donde cada una describe una interacción especifica entre un usuario y la aplicación. Se las utilizaron como herramienta para la captura de requisitos. Estas se escribieron en formato “Connextra” en papeles de colores de 10x10cm, que se pegaron sobre un el pisaron, como se ve en la imagen (Fig. 1). 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.

Figura 6: Historias de usuario previo a priorizar

Wireframe, Mockup y Storyboard - Interfaces de Baja Fidelidad. Herramienta de Análisis.

{{ mandar a metodología }}

Siguiendo los pasos de un diseño de afuera hacia adentro, luego de obtener una visión general del problema a solucionar 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 pagina o Wireframe y los mockup) en baja fidelidad; el uso de estos elementos permitió analizar las propiedades y navegación de la aplicación.

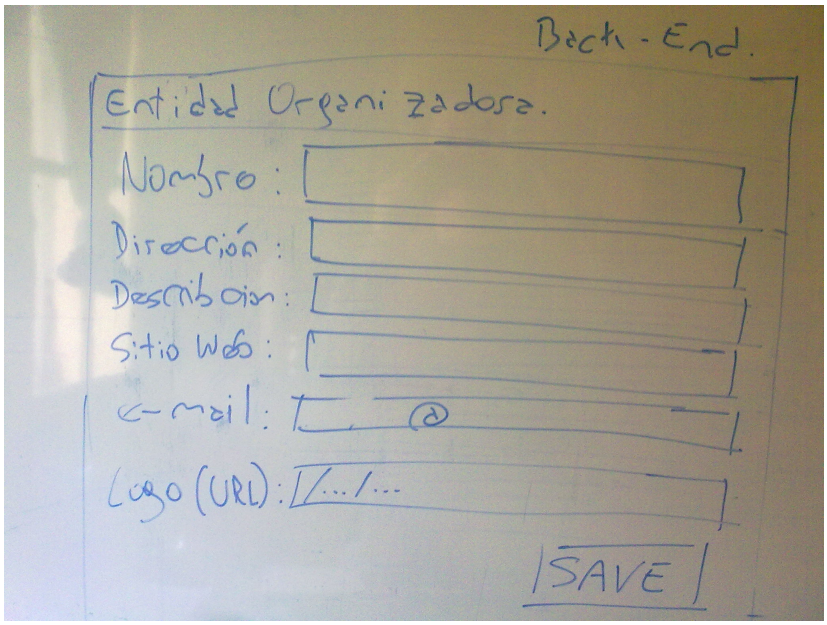


Figura 7: Wireframe interfaz alta de entidad organizadora

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.

Wireframe: es una técnica de diseño que consiste en generar una guía visual o esquema que representa la estructura visual del sitio, su diseño y contenido. [**20**. Diseño]

Los Guiones gráficos (Storyboard) capturan la interacción entre las diferentes paginas 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 paginas. [SaaS Book.]

Lo que se quiere con los bosquejos de las interfaces de usuario, es que sean el equivalente a las tarjeta de 3x5; que esto sea atractivo para los interesados no técnicos y alentarlos a la prueba y error, ya que es más fácil de probar y descartar bosquejos. Se recomienda utilizar los mismo elementos que en un jardín de infantes para hacer los bosquejos "UI mockups": papel, crayones, tijeras. Ellos llaman a esta esta aproximación de baja tecnología "Lo-Fi U" y a los prototipos en papel "sketches".

Idealmente, se hacen "sketches" 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 UI 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. [SaaS Book.]

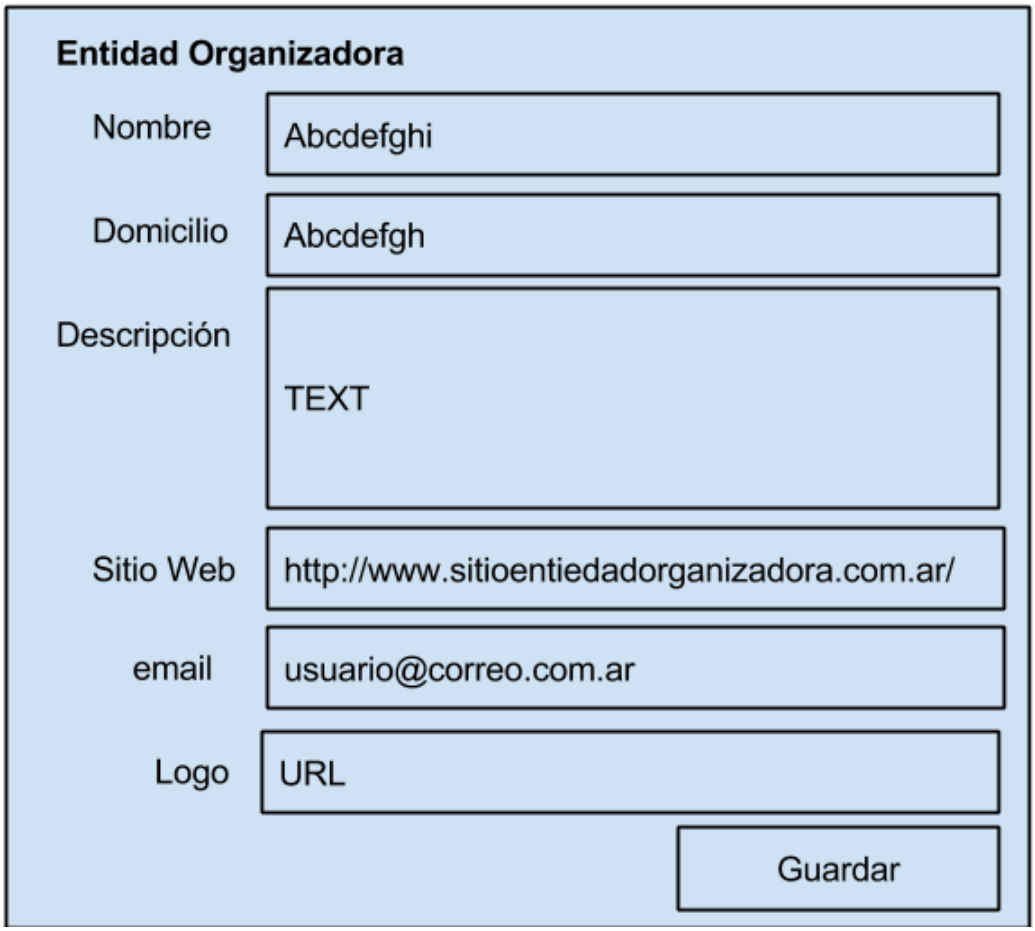


Figura 8: mockup baja fidelidad de interfaz alta de entidad organizadora

3.1 PivotalTracker – Administración y priorización de Historias de Usuario.

PivotalTracker (http://www.pivotaltracker.com/) [19.PivotalTracker], 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 la historia y el tiempo que se demoro en completarla.
Se lo utilizo para la documentación de las historias de usuario y para la administración del proyecto aplicando la metodología Scrum; en ella se definió el backlog y priorizaron las historias de usuario. Esto sirvió para responder fácilmente la pregunta “¿Y ahora que sigue?

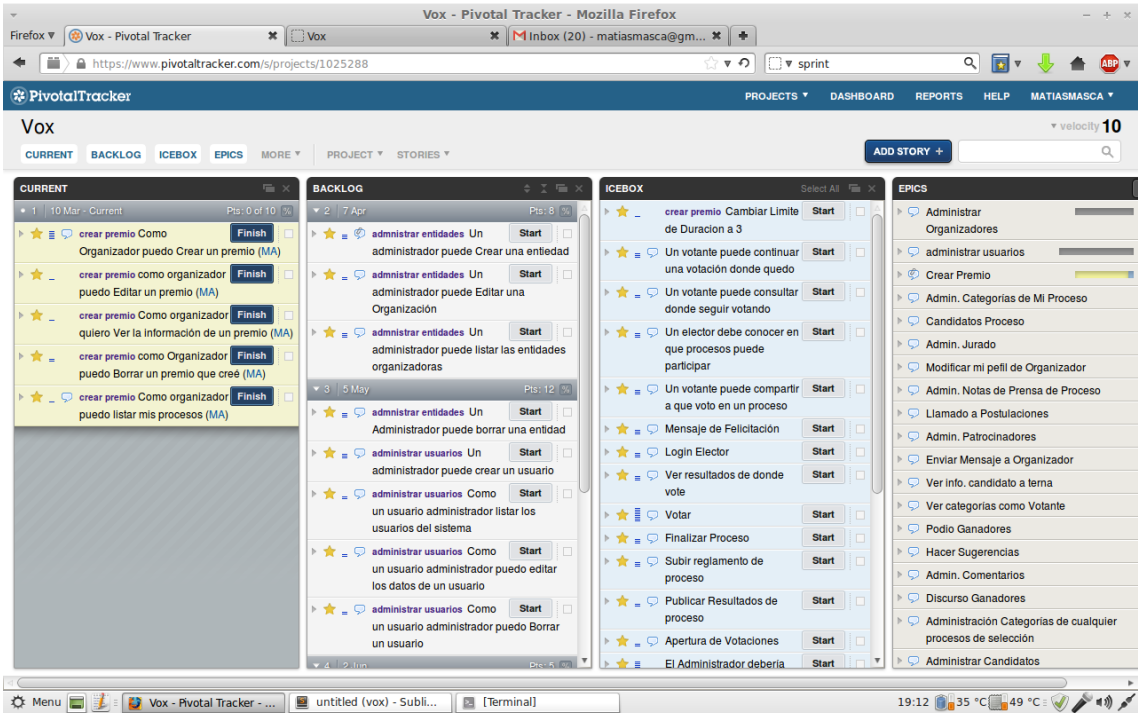


Figura 9: Captura de pantalla primera planificación en PivotalTracker

3.2 Pizarrón y fibra – herramientas para bosquejar.

La utilización del pinzaron “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 podíamos ver y compartir al mismo tiempo, y a sabiendas, que esa información sería borrada, permitió un mejor entendimiento y facilito la creatividad.

3.3 Ruby – Lenguaje de programación.

Se utilizo 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, es decir, 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.” [6].

Ventajas de Ruby:

- Es fácilmente integrable (en módulos)
- Ofrece flexibilidad (se lo puede modificar, quitando partes o redefiniéndolas)
- Ofrece simplicidad (hacer más con menos)
- Es cercano al lenguaje natural
- Posee una amplia librería estándar.
- Productividad
- Permite desarrollar soluciones a bajo Costo.
- Es software libre. El intérprete y las bibliotecas están licenciadas de forma dual (inseparable) bajo las licencias libres y de código abierto GPL y Licencia pública Ruby.
- Es multiplataforma.

Desventajas de Ruby:

- Débilmente Típado. Tipos dinámicos pueden traer problemas inesperados.
- No soporta polimorfismo de funciones (sobrecarga)
- Se puede modificar fácilmente las Librerías Estándar de Ruby
- Hilos de Ruby son “green threads” programados y ejecutados por una máquina virtual.
- Problemas de compatibilidad hacia atrás
- Ruby más lento que lenguajes compilados
- No tiene soporte completo de Unicode, pero si de UTF-8.

3.4 JavaScript – Lenguaje de Programación.

JavaScript® (generalmente abreviado como JS) es un lenguaje de programación interpretado, orientado a objetos, conocido por su utilización como lenguaje de script para páginas web que se ejecutan directamente en el navegador del usuario, pero también es usado en entornos sin navegador, tales como node.js o Apache CouchDB. Es un lenguaje script multi-paradigma, basado en prototipos, dinámico, soporta estilos de programación funcional, orientada a objetos e imperativa.
El JavaScript estándar es ECMAScript, que se baso en el JavaScript original, trabajo de Rendan Eich para Netscape y parte del Jscript de Microsoft. A partir de 2012, todos los navegadores web modernos soportan completamente ECMAScript 5.1. Los navegadores

web más antiguos soportan por lo menos ECMAScript 3. Una sexta revisión del estándar está en proceso.
[7]
Si bien su uso directo en el trabajo fue mínimo, es la base técnica para las interacciones enriquecidas (efectos de transición, mensajes de alerta) generadas en la aplicación, que son proporcionadas por el framework Boostrap y la gema Chartkick, que se describen más adelante.

3.5 Rails – Framework.

Se utilizo el lenguaje de programación Ruby junto con el framework Ruby on Rails (rubyonrails.org) en su versión 4.0.1.; en adelante Rails.

Rails es un framework o entorno de programación escrito en Ruby para el desarrollo de aplicaciones Web.

Las razones para elegir Rails sobre otros frameworks:

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. " [8]

Rails, implementa 3 patrones de diseño: "Model-View-Cotroller" 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 paginas HTML

Rails enfatiza en la aplicación de dos principios de desarrollo:

i) No te repitas (Don't Repeat your self o DRY).

ii) Convención sobre configuración (Convention Over Configuration o CoC)

i) *No te repitas*, significa que la información se debe colocar en un único e equivoco lugar. En Rails, por ejemplo en las vistas HTML se utilizo 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 lineas 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 s*ignifica 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 programador debería preocuparse por los casos anormales; para esto el framework hace uso de la reflexión y la meta-programación de Ruby.

Rails también define una estructura particular para organizar el código de nuestra aplicación.

Directorio/Archivo	Propósito
Gemfile	Lista de librerías (gemas) utilizadas por la aplicación.
Rakefile	Contiene comandos para automatizar algunas tareas de mantenimiento y desarrollo.
app/controllers	Contiene el código de los Controladores.
app/view	Contiene las plantillas (templates) de las vistas
app/view/layouts	Contiene una pagina plantilla que utilizan todas las vistas de la aplicación.
app/helpers	Contiene el código de los Helpers de las vistas.
app/assets	Contiene archivos estáticos (JavaScript, imágenes, hojas de estilo)
config	Archivos con información básica de configuración
config/environments	Configuraciones para ejecutar la aplicación en diferentes entornos: desarrollo, producción, etc.
config/database.yml	Configuración de la base de datos utilizada, en cada entorno.
config/routes.rb	Mapeo de los URIs a acciones en los controladores
db	Archivos que describen el esquema de la base de datos
db/development.sqlite3	Archivo que contiene la base de datos SQLite de desarrollo
db/test.sqlite3	Archivo que contiene la base de datos SQLite utilizada en las pruebas
db/migrate/	Archivos que describen los cambios en la base de datos, permite deshacerlos.
db/schema.rb	Contiene el esquema final de la base de datos, luego de las migraciones.
db/seed.rb	Contiene registros que se pueden insertar en la base de datos al instalarla. En este caso contiene datos de prueba.
doc/	Documentación generada.
lib/	Código adicional compartido en la aplicación.
log/	Archivos de bitácora, registran eventos de la aplicación.
public/	Directorio de archivos públicos, contiene archivos y recursos (asset) estáticos.
script/	Herramientas de desarrollo que no hacen a la aplicación.
tmp/	Archivos temporales, mantenidos durante la ejecución del servidor (como cache, pid, y archivos de sesión)

Tabla 1: Estructura de directorios de Rails

[Tabla Adaptación de Rails Guides. Fuente: http://guides.rubyonrails.org/getting_started.html {{ mandar a un Anexo.}}]

Para ser conciso, no repetitivo y aumentar la productividad, Rails hace un uso extensivo de la meta-programación (permite a los programas modificare a si mismo 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 [Saasbook]

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. Rails incorpora otros módulos no mencionados aquí.

Se utilizo Rails con el agregado de algunas librerías externas, llamadas gemas, las cuales figuran en el archivo /Gemfile. De las cuales destaco:

- **Devise**. Se utilizo 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.
- **Chartkic**. Se utilizo 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 utilizo esta gema para integrar Boostrap 3 a Rails, utilizando el pre-procesador CSS Sass.
- **BetterErrors**. Se utilizo esta gema para remplazar los mensajes de error de Rails, en el entorno de Desarrollo, para facilitar la depuración de errores.

 # Comentarios experiencia con Cucumber. {{donde integrar estos}}

- El **framework** libera de pensar, y consumir energía, en algunas decisiones triviales o que están fuera del alcance del desarrollador promedio, e incluso que están fuera de los alcances económicos de un proyecto normal. La comunicación con la BD, la comunicación entre capas, la seguridad, los helpers, validaciones, la integración con servicios de terceros, etc. etc.

- Al no usar el **framework** se percibe el incremento de trabajo en tiempo y esfuerzo; esto se vivencio con los primeros módulos de back-end, cuando se estaba tratando de comprender a fondo la tecnología a utilizar en el proyecto y no se hizo uso de algunas características del framework.

3.6 WEBrick - Servidor web de desarrollo y prueba.

Se utilizo 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; este provee HTTP 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.
Más información en http://ruby-doc.org/stdlib-2.1.1/libdoc/webrick/rdoc/WEBrick.html

3.7 Bootstrap – framework front-end.

Se utilizo el framework front-end Boostrap [10], en su versión 3, para la implementación de las vistas HTML de la aplicación. Se lo utilizo pre-procesado con Sass (*Syntactically Awesome Stylesheets*) con la utilización de la gema **Bootstrap-sass.**

El framework 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) . Esta basado en los últimos estándares de desarrollo de 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 responsivo. [11. Bootstrap]
En la siguiente imagen se puede apreciar su aplicación en la interfaz de back-end para editar una entidad Organizadora de procesos de selección en la sistema.

Figura 10: Interfaz alta de entidad organizadora con Bootstrap en pantalla de 364x640 píxeles

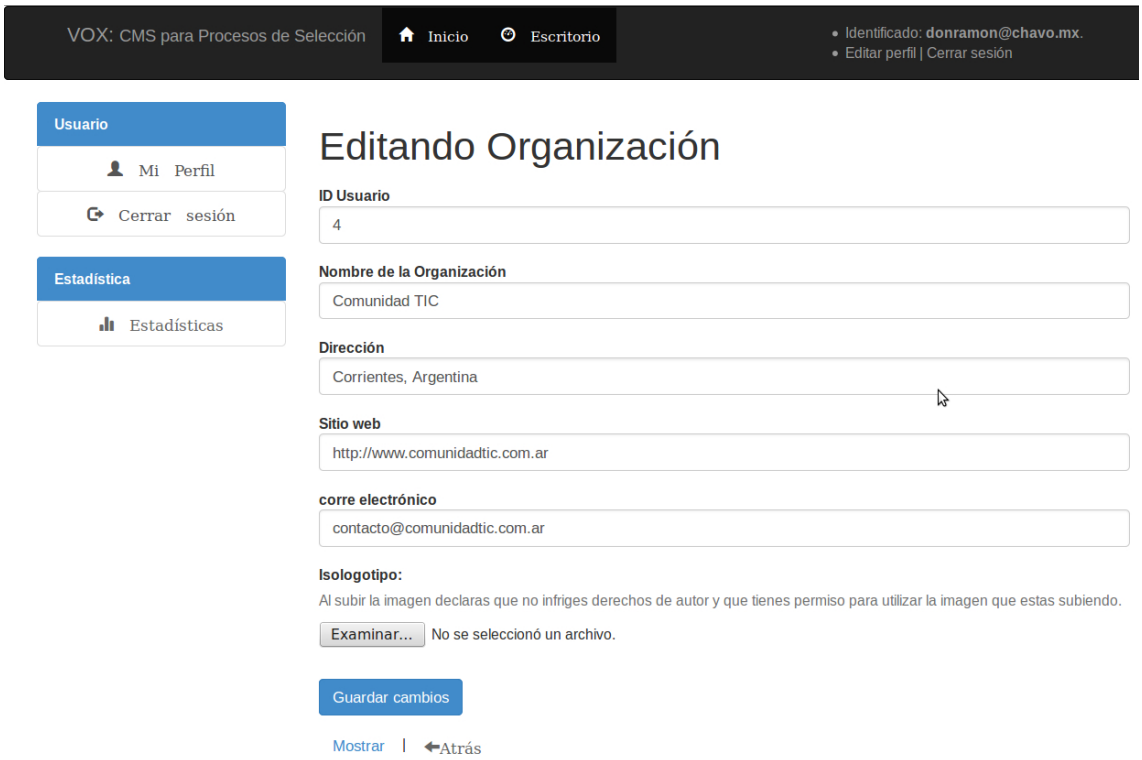


Figura 11: Interfaz alta de entidad organizadora después de integrar con Bootstrap

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 figura 6, esto permite usar la aplicación desde computadoras con diferentes tamaños de pantalla o desde los llamados teléfonos celulares inteligentes. 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) 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. Al estar integrado en la aplicación se evitan los tiempos extra de procesamiento y comunicación. Cabe aclarar que SQLite es un proyecto de código abierto de dominio publico.
[12][13]

En Ruby y en Rails, su utilización se hace mediante una gema, en nuestro 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 SQL estándar, en nuestro 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.

3.9 MySQL Workbench v6.0 – Modelador de datos.

MySQL Worckbench 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. MySQL Workbench esta disponible para plataformas Windows, Linux y Mac OS X. Es una aplicación de código abierto y uso libre con licencia GPL.
[14]
Se utilizo 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.

3.10 Sublime Text 2/3 – Editor de texto.

Se utilizo este editor de texto especializado para la edición de código fuente, en su versión 2.0.2, por su facilidad de uso, variedad de lenguajes soportado y por ser ampliamente utilizado en la documentación consultada, referente al ecosistema tecnológico utilizado.
Cabe aclarar que si bien es una herramienta de código cerrado, su modo de evaluación no tiene tiempo de caducidad, ni funcionalidades reducidas.
Sublime Text es un editor de código, ligero y minimalista. El código se presenta en pestañas, con coloreado de sintaxis para la mayoría de lenguajes y diecinueve elegantes esquemas de color para el fondo.
Dos funciones de Sublime Text ayudan a navegar por el código: el minimapa y QuickPanel. El primero, situado en el lado izquierdo, permite navegar por el código visualmente. El segundo, que se invoca con el atajo Control+Mayús.+O, da acceso a un navegador de ficheros con filtro rápido.
Otras características de Sublime Text le aúpan a un lugar destacado, como el sistema de plugins, la inclusión de *snippets* de código, la grabación de macros, la vista a pantalla completa y la disposición en múltiples paneles, ideal para quien usa varios monitores.

Ventajas

- Minimapa lateral y QuickPanel
- Macros y sistema de plugins
- Coloreado de sintaxis
- Editor en pantalla completa
- 19 esquemas de colores

Desventajas

- Las opciones se editan a mano
- Ayuda muy escasa
- Código cerrado.

3.11 Git / GitHub. - Control de versiones de archivos y código.

Un sistema de control de versiones (Version Control System o VCS en inglés) es un sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que sea posible recuperar versiones especificas de estos archivos.

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. Algunos de los objetivos del nuevo sistema fueron los siguientes:

- Velocidad
- Diseño sencillo
- Fuerte apoyo al desarrollo no lineal (miles de ramas paralelas)
- Completamente distribuido
- Capaz de manejar grandes proyectos (como el núcleo de Linux) de manera eficiente (velocidad y tamaño de los datos)

“Git es tremendamente rápido, muy eficiente con grandes proyectos, y tiene un increíble sistema de ramificación (branching) para desarrollo no lineal.” [15.ProGit.Chacon]

Por las cualidades citadas se utilizo Git para hacer el versionado local de archivos del proyecto, en la maquina de desarrollo.

GitHub.

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 github.com/matiasmasca/vox. No existe una versión auténtica de ningún proyecto, lo que permite a cualquiera de ellos ser movido fácilmente de un usuario a otro en el caso de que el primer autor lo abandone. GitHub es también una compañía comercial, que cobra por las cuentas que tienen repositorios privados. Pero, para albergar proyectos públicos de código abierto, cualquiera puede crear una cuenta gratuita. [ProGit.Chacon]

Se vinculo el repositorio local Git con una cuenta de GitHub, 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.com, que se describe más adelante. Se utilizo este servicio en linea para evitar tener que montar un servidor Git, tarea que escapa a los objetivos del trabajo.

Para el archivo Readme, del repositorio, se tuvo que usar un tipo especial de formato llamado Markdown.

Se vinculo el repositorio en GitHub con:

- **Coveralls.io** (https://coveralls.io/) [17] es un servicio para medir la cobertura de código, se describe más adelante.
- **Travis** (https://travis-ci.org/)[18.Travis] es un servicio de integración continua y despliegue, para realizar la practica ágil de “integración continua”. Se describe junto con las herramientas de despliegue en producción.

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

3.12 Google Docs y Libre Office – Ofimática.

Para la redacción de los documentos relacionados con el proyecto se utilizo Google Docs, que es una versión en linea de LibreOffice; también se lo utilizo para pasar en limpio las interfaces en baja fidelidad (mockup). 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 utilizo Writer, parte del paquete LibreOffice, en su versión 4.2.

3.13 Mozilla FireFox. - Navegador Web.

Es un navegador web de código abierto, creado y mantenido por la fundación Mozilla.
Se lo utilizo, en su versión 32, para hacer las búsquedas bibliográficas, descargar los programas utilizados y su documentación; así como 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) [22] es una herramienta de comunicación y colaboración pensada para soportar el proceso de BDD (behavior-driven desing); 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. [23]
Escribiendo estas pruebas antes que el código, se pueden eliminar muchos mal entendidos mucho antes de que estos lleguen al código fuente.
El lenguaje que entiende Cucumber se llama “Gherkin” (pepinillo en español) que es un DSL orientado a ser leído por la gente del negocio “BusinessReadableDSL” y disponible en 40 idiomas; representa la forma en que deben ser escritos estos textos.
{{Una descripción más completa sobre Gherin podría ir en un anexo. https://github.com/cucumber/cucumber/wiki/Gherkin}}

Las pruebas de aceptación escritas en este estilo se conviertes en más que solo pruebas, ellas son “Especificaciones Ejecutables”. [23]

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 esta decirnos que tan precisas son. En la practica 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.

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 requisitos, 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.

Como funciona.

Cucumber es una herramienta de linea de comando, cuando se ejecuta con el comando "cucumber", esto las especificaciones escritas en archivos de texto plano llamados "features", con extensión .feature, los examina en busca de "escenarios" a probar, y ejecuta estos escenarios contra tu 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".
Junto con los archivos .feature, se le da a cucumber un conjunto de "definiciones de pasos" (step definitions) que mapean cada oración escrita en lenguaje natural con un paso (step) escrito en código Ruby, que describe la acción que ese paso realizar.

Las definiciones de los pasos se las registra en archivos de texto plano cuyos nombres terminan en "_steps.rb". Estos pasos son algunas lineas de código Ruby que interactúan con el código de la aplicación. Normalmente se vincula con alguna librería de automatización de procesos para que interactué con el sistema real; en nuestro caso la librería para automatización utilizada fue Capybara. Se ubica estos archivos en la carpeta "step_definitions" en la ruta ../feature/step_definitions

Si el código Ruby del paso (step) se ejecuta sin errores, Cucumber procede a el 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 media que se ejecutan los escenarios, Cucumber imprime en la pantalla cuales pasos pasaron y cuales no, indicando el numero de linea donde no esta pasando.

Utilizando una herramienta para colorear la consola, se le puede asignar Verde, Amarillo, Azul y Rojo. Cuando todas las pruebas pasan la interfaz queda de un color verde y de allí su curioso nombre.

Esta jerarquía, desde los archivos features hasta la librería de automatización, esta ilustrada en la siguiente figura. (Fig. 1)

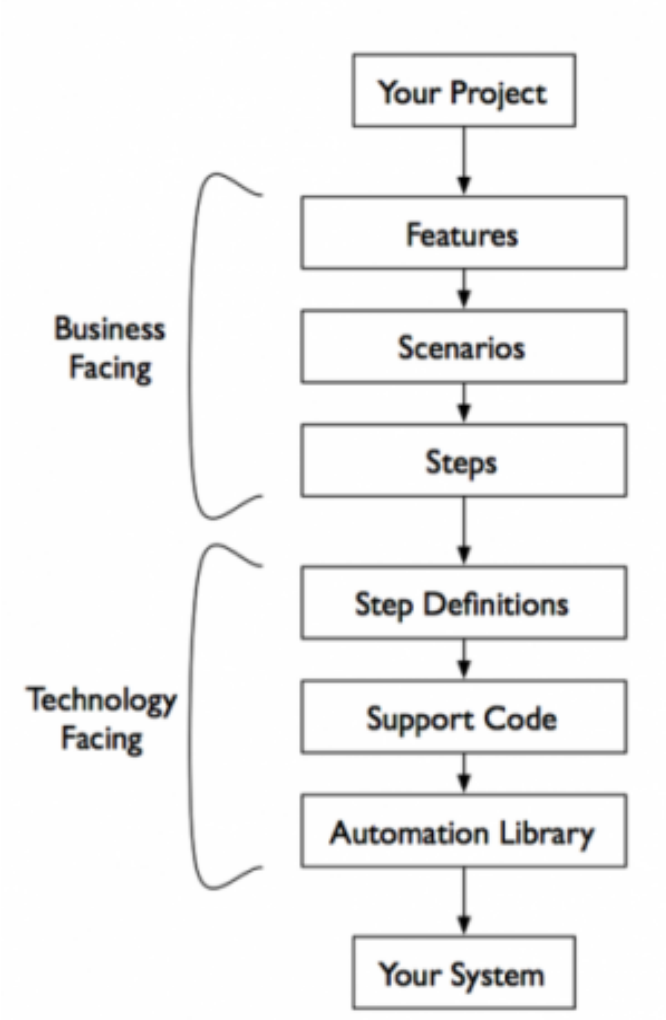


Figura 1: Cucumber testing stack. [23]

{{ Trafico del libro. Traducir [23] }}

Hay muchas otras ventajas que hacen de Cucumber una excelente elección: tu puedes escribir una especificación en mas de 40 lenguas habladas. Tu puedes usar etiquetas (TAG) para organizar y agrupar los escenarios, y se puede integrar fácilmente las librerías de Ruby.

Se utilizo Cucumber como una herramienta de linea de comandos, posicionándose en el directorio del proyecto y escribiendo el comando: “cucumber” para ejecutar todas las pruebas (archivos con extensión .feature) que se encuentren en la carpeta /features y subcarpetas.

 # Comentarios experiencia con Cucumber. {{donde integrar estos}}

- 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.

- Pude detectar que faltaban algunas features, al avanzar en el proceso. Por ejemplo las feature de "mostrar categoría" para el Admin. lo detecte cuando estaba haciéndolo para el caso particular de “mostrar categoría” para el usuario Organizador.

- 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 esos estados.

3.15 Capybara – Librería de automatización.

Capybara [22] es una herramienta, 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:
- No es necesaria una configuración en aplicaciones Rails o Rack, funciona por defecto.
- 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 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.

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 maquinas virtuales, pero esta limitado para la ejecución de JavaScript y tiene que ejecutarse en el mismo lugar donde esta la aplicación.

Su DSL, provee selectores para: Navegación, clic en botones y enlaces, interactuar con formularios, hacer consultas y búsquedas en el código de la pagina, trabajar con ventanas, ejecutar scripts, responder a mensajes modales, macheo y depuración. Ayudándonos en la lectura de las propiedades del DOM.

Su DSL, helpers y selectores, ayudo en la productividad de la creación y ejecución de las pruebas de aceptación. Para su utilización en el proyecto hubo que agregar las gemas “capybara” y “cucumber-rails”

Por ejemplo “save_and_open_page” abre en un navegador web el estado actual del DOM lo que ayuda a entender el proceso y como se ejecutan las pruebas.

3.16 RSpec - Pruebas Unitarias.

Rspec (http://www.rspec.info) [1] es una herramienta para aplicar la practica ágil de desarrollo dirigido por comportamiento, más conocida como BDD (del ingles Behaviour-Driven Development) para el lenguaje Ruby. Esta diseñado para hacer de la programación dirigida por pruebas (Test-Driven Development) una experiencia agradable y productiva. Se caracteriza por tener:
- Una herramienta de linea de comandos, muy completa. El comando: rspec.
- Descripciones textuales de ejemplos y grupos. (rspec-core)
- Reportes flexibles y personalizables.
- Lenguaje de expectativas extensible. (rspec-expectations)
- un framework para dobles de prueba (mocking/stubbing framework) integrado. (rspec-mocks)

RSpec provee un lenguaje de dominio especifico (DSL) para especificar el comportamiento de los objetos. Adaptando la metáfora de describir el comportamiento de la manera en que lo haríamos si estamos hablando con el cliente o con otro desarrollador.

[23. TheRSpec.Book pag. 25] Se utiliza RSpec para para escribir ejemplos ejecutables del comportamiento esperado de un pequeño fragmento de código en un contexto controlado. Así es como lucen las pruebas:

```
describe VoterList do
  context "when first created" do
    it "is empty" do
      voter_list = VoterList.new
      voter_list.should be_empty
    end
  end
end
```

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. Esta escrito en Ruby y su código fuente esta 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.

Se utilizo 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/), esta herramienta de linea 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 lenguaje de dominio especifico (en ingles *domain-specific language o DSL*) 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 utilizo 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 inic rails”).
En algunas oportunidades, mientras se encontraba activo el monitoreo se detectaba un incremento notable en el uso de recursos de la maquina que provocaban un recalentamiento

del hardware o una enlentecimiento general, entonces se optaba por no usarla cuando la maquina de desarrollo era una notebook. Sin embargo la experiencia de recibir una retroalimentación, casi, inmediata sobre el código recién modificado, ayudaba a seguir con mayor velocidad al técnica TDD o BDD.
Para ejecutarlo, una vez instalado, basta abrir una con solo, 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/) [17] es un servicio para medir la cobertura de código que realizan las pruebas, que posee una versión gratuita para proyectos de código libre. Es diagnostico en términos de lenguaje de programación que evalúa o de servidor de integración continua que se utiliza. Este servicio hace un seguimiento de la cobertura del código y lleva estadísticas sobre los cambios, mostrando en una interfaz web sencilla; el el caso de este proyecto 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 necesito crear una cuenta en el servicio, vincularla con el repositorio en GitHub desde el sitio web del servicio; esta vinculación es el único requisito que tiene el servicio. Y luego agregar la gema “coveralls” al archivo /Gemfile. Finalmente agregar dos lineas de código en la configuración de las pruebas, tanto para Cucumber como para RSpec:
1. require 'coveralls'
2. Coveralls.wear!
La integración de este servicio permite saber el grado de cobertura de las pruebas lo que permitió mantenernos en unos margenes aceptables.
De las herramientas del entorno de pruebas, Coveralls es la única que se uso exclusivamente cuando la aplicación ya estaba desplegada en el servidor.

Entorno de Producción

3.19 Travis – Integración Continua.

Travis-CI. (https://travis-ci.org/)[18.Travis] es un servicio de integración continua y despliegue, para realizar la practica ágil de “integración continua”. Esta integrado con GitHub y ofrece soporte para más de 15 conocidos lenguajes de programación; además permite ejecutar las pruebas sobre diferentes versiones de una tecnología y encriptar archivos que

tengan información sensible. Se lo puede configurar para que haga el despliegue ante un “pull request” de cualquier rama, o sólo cuando sea sobre la rama “master” del repositorio. 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 un repositorio en GitHub; luego activar el WebHook dando permisos de lectura a Travis mediante GitHub. 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
[18.Travis]
Para obtener los detalles técnicos de como realizar la integración continua para proyectos escritos en Ruby se puede consultar la guía de Travis-Ci para Ruby en http://docs.travis-ci.com/user/languages/ruby/

La utilización de este servicio permitió ejecutar las pruebas en un ambiente controlado y repetible. Con cada ejecución 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 Heroku.com [Ref. A Heroku.com], 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 online en https://travis-ci.org/matiasmasca/vox , también se puede acceder a un historial de integraciones (build) que se realizaron desde la plataforma. {{Podría ir un LOG completo como Anexo}}. En casos de fallo, suspende el deploy y envía un email informando que el proceso ha fallado.

3.20 Heroku – Servidor de aplicaciones en producción.

Heroku (www.heroku.com) [26.SitioHeroku] es un plataforma de aplicaciones en la nube, servicio de plataforma como servicio (PaaS), que soporta varios lenguajes (Ruby, PHP, Node.js, Python, Java, Cloujer, Scala) y permite realizar el despliegue de una aplicación bajo demanda abstrayéndose sobre la plataforma que esta 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. Para Rails utiliza un servidor web llamado Unicorn**.**

Sobre las Plataformas como Servicio o PaaS.

PaaS (en inglés *platform as a service*) es un modelo de servicio en la nube que provee herramientas para los procesos de construcción de aplicaciones en la nube; existe una estrecha analogía de entenderlo como una abstracción del sistema operativo y el middelware del entorno en la nube. PaaS provee a los desarrolladores la plataforma que estará por debajo para desarrollar su aplicación. Se encargan de dar soporte a un lenguaje o tecnología especifica y la pila de servicios que necesitan usar los desarrolladores. Algunos proveedores también habilitan la posibilidad de escalar los recursos bajo demanda, tanto de computación como de almacenamiento, automáticamente liberando al administrador de la tarea de asignarlos manualmente.

En PaaS, el consumidor del servicio controla el despliegue y la configuración. El proveedor de la plataforma se encarga de proveer los servidores, la red y las necesidades computacionales que pueda tener la aplicación del consumidor.

El modelo PaaS permite tener arquitecturas multiusuario, para que multiples usuarios pueden usar la aplicación web de manera segura, estable, concurrente y aprueba de fallos. Servicios más sofisticados pueden tambien integrar entornos de desarrollo de aplicaciones, con editores colaborativos, control de versiones y despliegue. 27.Heroku.book pag. 8]

Heroku, Google App Engine, Engine Yard, AWS Elastic Beanstalk, Cloud Foundry, OpenShift, Microsoft Azure, Bluemix son algunos ejemplos de plataformas PaaS.

Introducción a Heroku.

Heroku es un proveedor reconocido en el negocio de software en la nube, ha prabado ser una solución tanto para pequeños como para grandes negocios por igual. Con la mejora continua y la filosofia de "conveniencia" sobre "configuración", se ha convertido en una plataforma reconocida, ha sido usada por más de 40.000 sitios web hasta la fecha. La filosofia de Heroku es dejar que los desarrolladores se enfoquen unicamente en escribir sus aplicaciones web y se olviden acerca de los servidores. Heroku se hará cargo, automaticamente y bajo demanda, de construir, desplegar, ejecutar y escalar la aplicación por ellos.

Heroku es una plataforma poliglota que provee soporte para Ruby, Ruby on Rails, Java, Node.js, Clojure, Scala, Python, y PHP. La arquitecura de complementos (add-on) permite a los desarrolladores customizar el servicio con el uso de más de 100 paquetes de servicios de terceros que puede agregar segun sus necesidades. Tiene la flexibilidad para permitr elegir entre un cuenta basica gratuita y planes pagos que se ajustan a los requerimientos de cada sitio. Agregando complementos para potenciar las aplicaciones.

Heroku tambien provee flexibilidad para administrar la aplicación una vez que este desplegada en su plataforma. El desarrollador puede admnistrarla utilizando una herramienta de linea de comandos desde su maquina o desde un panel de control que corre en su infraestructura.
Heroku es altamente escalable, este escalamiento es transparente para el usuario, puede soportar picos de trafico y servir hasta 1000 peticiones sostenidas por segundo.
Heroku tiene integrado Git, lo que permite un flujo de trabajo concentrado haciendo facil compartir codigo, colaborar con otro desarrollador y hacer despliegues frecuentes, reduciendo el tiempo que tarda la aplicación en llegar a los usuarios.
[27.Heroku.book pag. 12]
Se puede seguir la siguiente guía paso a paso, para utilizar el servicio:

https://devcenter.heroku.com/start

La utilización de este servicio, en su versión gratuita, permitió desplegar la aplicación en un entorno controlado y repetible, de forma sencilla, rápida y sin costo económico. Además de permitió la vinculación con el servidor de integración continua, Travis [], sin mayor esfuerzo; esta combinación hizo posible tener una versión del sistema publicada desde el primero momento.

3.21 PostgreSQL - Base de Datos Producción.

[revisar]

PostgreSQL es un sistema de gestión de bases de datos objeto-relacional (en ingles object-relational database management system o ORDBMS), de código libre, pasado en un motor anterior llamado POSTGRES. PostgreSQL utiliza un modelo cliente/servidor y usa *multiprocesos* en vez de *multihilos* para garantizar la estabilidad del sistema. Un fallo en uno de los procesos no afectará el resto y el sistema continuará funcionando.
La última serie de producción es la 9.3. Sus características técnicas la hacen una de las bases de datos más potentes y robustas del mercado. Su desarrollo comenzó hace más de 16 años, y durante este tiempo, *estabilidad, potencia, robustez, facilidad de administración e implementación de estándares* han sido las características que más se han tenido en cuenta durante su desarrollo. PostgreSQL funciona muy bien con grandes cantidades de datos y una alta concurrencia de usuarios accediendo a la vez a el sistema. [15. PostgreSQL]
Si bien la utilización de este motor no fue una decisión tomada, es algo que estaba disponible en el servidor de producción, no encontramos 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 limite que permite hasta 10.000 registros.

Comentarios y discusiones.

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 capitulo y la cantidad de ellas.

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 esta 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.

Cabe destacar que se trato de utilizar herramientas libres o en su defecto de código abierto.

Capitulo 4. Resultados

NOTAS APUNTES CATEDRA TFA.

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

[- Puede ser denominado como Desarrollo.

- Describe el trabajo realizado por el alumno.

- Puede contener varias secciones o abarcar varios capítulos.

- Incluir tablas, figuras, los resultados de procesamientos de los datos, descripción del desarrollo y/o implementación, problema(s) resuelto(s), conocimiento(s) aportados, solución(es) brindada(s).]

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.

]

Capitulo 5. Conclusiones y futuros trabajos

NOTAS APUNTES CATEDRA TFA.

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

[- 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 capitulo.

- La redacción debe ser clara, concreta, directa y enfática.]

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

{{

El objetivo general de este TFA es profundizar el estudio de los temas vinculados con la calidad del software, en particular, sobre las pruebas del software utilizando “Desarrollo guiado por pruebas” o TDD (*Test-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 comprobar las ventajas que ofrece.

Objetivos Específicos:

- Realizar una revisión bibliográfica sobre el estado del arte, los conceptos y las técnicas que conforman el desarrollo guiado por las pruebas (TDD).
- Comprobar las ventajas de este enfoque mediante la aplicación de sus conceptos y técnicas en el desarrollo de una aplicación informática particular.
- Construir un software multiplataforma que permita a una institución organizar premios y certámenes, gestionando a través de este todos los procesos involucrados; integrando los conocimientos sobre el desarrollo dirigido por pruebas para la resolución del problema.

}}

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

Freeman y Pryce, podemos 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.
- Facilita la detección de errores mientras el contexto está fresco en nuestras mentes.
- Nos dice cuándo debemos 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.

// **Notas mentales**. Conclusiones **Preliminares**.

- Hay que aprender a leer los errores e identificar de que 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.

- Es difícil pensar primero en la prueba, e imaginarse como van a ser las cosas en una nueva tecnología o herramienta. [C. Ble en su libro habla de hacer Spikes pag. 60]

- Es como tejer una red de seguridad, nodo por nodo, pero al final te queda una red donde uno se siente muy seguro modificando el código, saltando al vacío.

- 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 aplico 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 siguió un enfoque de **Afuera hacia Adentro** (empezando por las pantallas, el comportamiento).

- Es mejor refactorizar un código que tiene un comportamiento esperado, a un que no sabemos si tiene el comportamiento esperado pero que funciona bien.

- Al no usar el **framework** se percibe el incremento de trabajo en tiempo y esfuerzo.

- El **framework** libera de pensar, y consumir energía, en algunas decisiones triviales o que están fuera del alcance del desarrollador promedio, e incluso que están fuera de los alcances económicos de un proyecto normal. La comunicación con la BD, la comunicación entre capas, la seguridad, los helpers, validaciones, etc. etc.

- Si se utiliza el **framework**, creando el scaffold, al momento de hacer las pruebas en unitarias, se enfoca el esfuerzo en los cambios que se realicen sobre el funcionamiento habitual. Se entiende que el framework hace bien su trabajo y no es necesario testearlo, es un esfuerzo extra que no se justifica.

- 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.

- Las herramientas te ayudan a encontrar exactamente el lugar donde esta fallando el código o la prueba.

- El proceso va guiando el trabajo del programador, de modo que se tiene la respuesta a "Y ahora que sigue?"

- Es cuestionable la sobre carga mental de tener que programar el comportamiento y las pruebas antes que el código.

- Hay una carga extra, de tener que programar acciones de forma distinta a las del código de producción. Por ejemplo para probar la subida de un archivo. El framework proporciona herramientas para hacer de esta tarea sencilla, parametrizando una llamada. Pero la prueba desde Cucumber/Capybara y luego RSpec, para probar este comportamiento resultado complicada y un consumo de tiempo excesivo.

- Con Cucumber, 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. Se ahorra tiempo y código repetitivo.

- Las pruebas por defecto en Rspec, que genera Rails, son realmente frágiles.

- La fragilidad de las pruebas se hace notar a medida que uno avanza con la técnica, e incluye nuevas funcionalidades. Por momentos se nota ese esfuerzo doble que genera un malestar entre los desarrolladores; no se ve el avance en funcionalidades para el usuario.

- Agregar la "**Integración Continua**" (CI en Ingles) una vez se tenia un conjunto de pruebas fue una tarea bastante sencilla, sólo hubo que dar de alta una cuenta en el servicio travis-ci.org y configurar un archivo "travis.yml". Luego se configuro una cuenta en el servidor de aplicaciones Heroku.com y se agrego algunos datos al archivo "travis.yml". Llevo 2 tardes.

- Si bien a primera vista se aprecia una sobre carga de trabajo, cuando se repite el tipo de pruebas en Cucumber y en RSpec, que evalúan el mismo comportamiento. Pero en el caso de TDD (RSpec) también se deberían evaluar con pruebas otro tipo de interacciones y alternativas que no tienen que ver con el Comportamiento Observado por el usuario; como ser interacciones entre clases, interfaces entre capas y utilización de sistemas de terceros.

- **Refactorizando**, realmente ayuda a encontrar errores más rápido. Ejemplo: estaba refactorizando un mensaje de error en la vista y cree un helper; cuando estaba aplicando el helper a Organizadores, escribí mal el nombre del modelo @organizer y al correr las pruebas aparecieron los errores y al revisar en exactamente ese punto se notaba 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 paso de un mensaje especifico a uno general.

- **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

- Con la existencias de estas herramientas, y su facilidad de integración en el flujo de trabajo, se puede decir que la existencias de pruebas creadas por el mismo Desarrollador se vuelven una condición exigible. Por lo tanto su enseñanza formal debería empezar a

considerarse también; pensando en la forma de enseñar TDD:
http://blog.testdouble.com/posts/2014-01-25-the-failures-of-intro-to-tdd.html

- Si se abandona la técnica, por un momento y el programador agrega funcionalidades sin hacer antes las pruebas entonces esta documentación viva sufre el mismo problema de des-actualización que la documentación tradicional.

- Es muy fácil salirse de la técnica, de las pruebas primero, cuando la funcionalidad a agregar parece pequeña. Sobre todo porque uno piensa que gana en velocidad (producción); lo cual en el corto plazo (lo inmediato) es cierto, pero cuando haya cambios más adelante aparecerán los problemas.

- 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.

Rails y BDD/TDD.

- En el caso del framework, es condición necesaria conocer el MVC y su funcionamiento, para entender como deberían funcionar las cosas. Cuando uno entiende como fusionan las cosas o como debería funcionar es más fácil crear las pruebas primero antes que el código, porque uno sabe a donde quiere llegar, donde se van a estar presentando los errores y el código que uno le gustaría tener (concepto: "Code we wish we had" o CWWWH); uno sabe en que pasos intermedios uno puede ir definiendo cosas que aun 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.

- El paso de refactor, de la práctica, es difícil de realizar para un usuario novicio, ya que no se sabe donde están los problemas. Con que funcione, pasando las pruebas de aceptación, se pasa a la siguiente característica (feature) desperdiciando parte del esfuerzo.

Sobre Errores.

- 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 integro 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 falla porque no puede leer estas variables nos veremos a tomar caminos lógicos que no contemplen esas variables o a poner excesivas condiciones IF.

// 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 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 interprete 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, volvimos a tener el problema de que la plataforma tenia 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?

#Falsos Positivos.

- Falsos positivos desde BDD, uno no sabe si realmente esta probando bien, puede tener falso positivos. Por ejemplo tenia que hacer Click 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 tenia que probar.

Por ejemplo: en la features "listado_entidades_organizadoras.feature" tenia una tabla con 1 sola entrada en lugar de 2 entradas, entonces todo estaba en verde, pero en realidad no estaba probando que tenia 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

- 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, 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 ble, 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 como hacer las pruebas en Cucumber y en RSpec. Lo cual consumía su tiempo extra.

BDD vs. TDD.

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

- Como regla, debe haber al menos 1 prueba de integración de alto nivel para cada acción de los controladores.

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

- Las pruebas de comportamiento pueden pasar incluso cuando un camino lógico sea incorrecto, por ejemplo un IF que da falso en situaciones que se esperaba de positivo. Entonces si la prueba de Integración, solo comprueba que la información se muestre y no prueba todos los caminos, el error puede seguir estando presente sin ser notado. En cambio la prueba de más bajo nivel debería de probar necesariamente esa bifurcación y comprobar ambos caminos en varias pruebas.

- BDD usa ejemplos en las conversaciones para ilustrar comportamiento.

- hablar el lenguaje del cliente ayuda a mantener alta la abstracción, sin caer en detalles de implementación.

Cucumber vs. RSpec.

- Probar el comportamiento de la vista (Interfaz), 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.

Cucumber.

- A medida que se avanza en el proyecto, se van sumando features y sus steps, se puede apreciar como la reutilización de pasos se vuelve útil y necesaria; además acelera notablemente el proceso de crear los features y sus procesos.

- Pude detectar que faltaban algunas features, al avanzar en el proceso. Por ejemplo las feature de "mostrar categoría" para el Admin. lo detecte cuando estaba haciéndolo para el caso particular del usuario Organizador.

- 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.

Integración Continua.

- A mayor impacto del cambio, mayor defensa proveen las pruebas. Al integrar la gema Devise, impacto en varias partes del sistema, provocando muchos errores. Estos errores nunca fueron vistos por los usuarios finales del sistema porque Travis-ci, no hacia el deploy en Heroku ya que fallaban las pruebas.

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.

5.2 Futuros trabajos.

NOTAS APUNTES CATEDRA TFA.

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

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

- Utilizar herramientas para revisar el código y refactorizar, como CodeClimate y CodeReview.

- Agregar funcionalidades, agregar lo relativo a los Certámenes.

- Agregar un sistema de plantillas, para los eventos.

- Mejorar el diseño gráfico.

- Profundizar sobre la practica de Refactoring y aplicar diferentes tipos de ella a este trabajo.

Referencias

1. C. Fontela. “Estado del arte y tendencias en Test-Driven Development”, Facultad de Informática, Universidad Nacional de La Plata, La Plata, Argentina, 2011. cap. 1 y 2, pp. 07-22
2. C. Blé Jurado. “Diseño Ágil con TDD”, Primera Edición. Madrid, España: iExpertos, 2010, cap. 1, pp. 33-52
3. R.S. Pressman.“Estrategias de prueba de software” en “Ingeniería del software. Un Enfoque Práctico”, Sexta Edición. Madrid, España: Mc Graw Hill, 1998, cap. 13, sec. 2, pp. 384
4. I.Sommerville, ”Verificación y Validación” en “Ingeniería del Software” Séptima Edición. Madrid, España: Pearson Educación S.A.,2005, cap. 22, sec. 5, pp. 470
   {{ editar a formato IEEE }}
5. K. Beck, “Extreme Programming Explained: Embrace Change”, Boston: Addison-Wesley Professional, 1999. cap 7 pp. 30 cap 8 pp. 35
6. K. Beck, “Test Driven Development: By Example”, 1st. de., Boston: Addison-Wesley Professional, 2002. cap. 31-32 cap. 0. (prefacio).
7. D. North, “Introducing BDD” [online], Dan North, 2006. Disponible en: http://dannorth.net/introducing-bdd/
8. M. Gärtner, “ATDD by Example: A Practical Guide to Acceptance Test-Driven Development” Boston: Addison-Wesley Professional, 2012. cap. 9 pp 131
9. M. Fowler, “Continuous Integration” [Online], Martin Fowler, 2006. Disponible en: http://martinfowler.com/articles/continuousIntegration.html
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. 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

12. Ruby, programing language. Website, Ultima visita 23/09/2014. https://www.ruby-lang.org/es/.
13. ECMAScript, Web Site, Ultima visita 23/09/2014. http://www.ecmascript.org/docs.php
14. Ruby On Rails, framework, Website, Ultima vista 23/09/2014 http://rubyonrails.org/
15. D. A. Patterson and A. Fox, *Engineering Software as a Service: An Agile Approach Using Cloud Computing*, 1st ed., Strawberry Canyon LLC, 2014.

16. Boostrap, framework front-end. Website, Ultima visita 24/09/2014. http://getbootstrap.com/ .
17. J. Spurlock, “Bootstrap”, 1st de., O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472. 2013.
18. SQLite, Web Site, Ultima visita 25/09/2014 http://www.sqlite.org/about.html
19. M. Owens, “The Definitive Guide to SQLite”, Apress, Berkeley, CA, 2010.
20. MySQL Worckbench, Web Site, Ultima visita 25/09/2014 http://www.mysql.com/products/workbench/
21. Postgresql. Website, Ultima visita 25/09/2014. http://www.postgresql.org/.
22. S. Chacon, “Pro Git”,1st. de., Apress, 2009
23. Coveralls.io, Web Site, Ultima visita: 26/09/2014. https://coveralls.io/.
24. Travis, Web Site, Ultima visita: 27/09/2014. https://travis-ci.org/.
25. PivotalTracker, Web Site, Última visita: 28/09/2014. http://www.pivotaltracker.com/.
26. D. M. Brown, “*Communicating Design: Developing Web Site Documentation for Design and Planning”, Second Edition*. New Riders. 2011
27. Cucumber, Web Site, Última visita: 30/09/2014. http://cukes.info/.
28. M. Wynne and A. A. Hellesøy, *“The Cucumber Book: Behaviour-Driven Development for Testers and Developers”*, 1st ed., Pragmatic Programmers, LLC., 2012.
29. D. Chelimsky, “The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends”, 1st. ed., The Pragmatic Programmers, LLC. , 2010.
30. Capybara, Web Site, Última visita: 01/10/2014. https://github.com/jnicklas/capybara.
31. Rspec, Web Site, Última visita: 01/10/2014. http://rspec.info/
32. Heroku (2014, Oct 10). Sitio web [Online]. Available: https://www.heroku.com/about
33. A. Hanjura, "Heroku Cloud Application Development: A comprehensive guide to help you build, deploy, and troubleshoot cloud applications seamlessly using Heroku", 1ra. Edición, Packt Publishing Ltd., Birmingham, UK , 2014
34.

Libro - Formato básico:
[1] J. K. Author, “Title of chapter in the book,” in Title of His Published Book
, xth ed. City of Publisher, Country if not USA: Abbrev. of Publisher, year, ch. x, sec. x, pp. xxx–xxx.-

Recurso Online:
E. H. Miller, “A note on reflector arrays,” [online] *IEEE Trans. Antennas Propag.*, vol. 53, pp. 475, 2005. Disponible en: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1549967&tag=1

[1] J. Jones. (1991, May 10). Networks (2nd ed.) [Online]. Available: http://www.atm.com

[3] Dan North, “What´s in a Story”, http://blog.dannorth.net/whats-in-a-story/, como estaba en junio de 2011.

Bibliografía Recomendada.

Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. ***Refactoring: Improving the Design of Existing Code****. Addison-Wesley, Reading*, MA, 1999.

Gojko Adzic. ***Specification by Example***. Manning Publications Co., Greenwich, CT, 2011.

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

David Chelimsky, Dave Astels, Zach Dennis, Aslak Hellesøy, Bryan Helmkamp, and Dan North. ***The RSpec Book***. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2009.

Matthew Robbins, ***Application Testing with Capybara****: onfidently implement automated tests for web applications using Capybara*.

Outside-in Software Development: A Practical Approach to Building Successful Stakeholder-based Products *by Carl Kessler), John Sweitzer (IBM), 2007.* ISBN: 978-0-13-157551-6

“**Continuous Integration: Improving Software Quality and Reducing Risk**” de Paul Duvall, Steve Matyas, and Andrew Glover. Addison-Wesley Professional; 1 edition (July 9, 2007)

“**Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation**” de Jez Humble and David Farley, Addison-Wesley Professional; 1 edition (August 6, 2010)

Bibliografía para el Ruby-Path.

Anexos

NOTAS APUNTES CATEDRA TFA.

[Por ejemplo si se incluye un detallado trabajo del análisis del sistema]

[Presentan documentación de relevancia para la justificación y comprensión del TFA para quienes lo leerán.

- No son obligatorios.
- Se enumeran de forma progresiva.
- Deben estar referenciados en el cuerpo del trabajo.

No incluir anexos como cuadros,figuras, gráficos que no son analizados en el informe.

- Podrá incluir modelos de las planillas de registro de datos y del plan de tabulación, técnicas cualitativas (encuestas, entrevistas estructuradas ,semiestructuradas), esquemas, normas, mapas, planos, tablas gráficos, fotografías, copia de documentos analizados (leyes, normas, discursos, planes, etc.), u otros instrumentos empleados para la recolección y procesamiento de los datos.

]

VOCABULARIO Y ACRONIMOS.

TDD: Test-Driven Development

BDD:

Prueba de Aceptación

Prueba de Integración:

Prueba Unitaria:

Historia de Usuario:

Refactorización:

Bad Smell: En la jerga de los desarrolladores, se dice que algo en el diseño “huele mal” cuando ciertas cosas en el código violan principios de diseño conocidos, de allí que a estas situaciones se las llame “olores” o “malos olores” (“smells” o “bad smells”, en inglés). Un mal

olor no es necesariamente un problema de diseño que amerita una refactorización, aunque sí es un poderoso indicio. [11]

Pruebas automatizadas de programador: Se llama pruebas automatizadas de programador [Elssamadisy 2007] a las pruebas de software que un programador escribe en forma de código y que determinan la calidad de lo que él mismo (o, en algunos casos, otro programador) desarrolló. Son “automatizadas” en el sentido de que, una vez escritas, se las puede correr una y otra vez sin un costo muy alto, solamente haciendo trabajar a la computadora. [11]

Definiciones de la Agile Alliance
http://guide.agilealliance.org/guide/bdd.html

ANEXO 1: Como instalar el entorno de la aplicación.

Pasos seguidos en el sistema operativo Linux Mint (Debian), para instalar el entorno de desarrollo de la aplicación y poder probarla localmente. Estos pasos se podrían convertir en un script bash para automatizar el proceso.

*** Instalar Curl.**

"sudo sudo apt-get install curl"

- para ejecutar un script que instala RVM, clonando desde su repositorio Git.

*** Instalar RVM:**

- Ejecutando el script: \curl -sSL https://get.rvm.io | bash -s stable
- Dependencias: Bash, Curl y Git

Probar que instalo bien RVM:

- source .bash_profile
- type rvm | head -1
- Sale algo así como: rvm: es una función
- rvm notes para ver más info. y que dependencias pueden faltar.
- "rvm requeriments" instala las dependencias que puedan faltar para ruby.

*** Instalar dependencias de Ruby.**

- Dependiendo de la versión del interprete, en este caso usaremos MRI.
- Dependencias: sudo aptitude install build-essential bison openssl libreadline6 libreadline6-dev curl git-core zlib1g zlib1g-dev libssl-dev libyaml-dev libsqlite3-0 libsqlite3-dev sqlite3 libxml2-dev libxslt-dev autoconf libc6-dev ncurses-dev

*** Instalar Ruby en RVM**

- "rvm list known" para ver que versiones de Ruby conoce RVM.
- "rvm install ruby-2.1-head" para utilizar la versión más reciente de la rama 2.1.XXX
- "rvm use ruby-2.1-head"

#Para vox: rvm install ruby-1.9.3-p547

*** Instalar Rails en RVM.**

gem install rails

*** Crear mi Gemset.**

rvm gemset create 'vox_project'
rvm 2.1.1@vox_project
gem install rails -v 4.1.1

rvm 1.9.3@vox_project --create

*** Instalar o crear mi Rails App.**

*** Clonar repo. GitHub**

posicionarse en directorio, o home y luego:
git clone https://github.com/matiasmasca/vox.git

cd vox

bundle install --without production

Configurar la base de datos.

- Realiza las migraciones, y luego carga los seeds.

Rake db:setup

- Prepara la base de datos que se utiliza en las pruebas.

rake: db:test:prepare

Opcionalmente: git remote add --track master upstream

$ git remote add --track master upstream git://github.com/matiasmasca/vox.git

```
$ git fetch upstream
$ git merge upstream/master
```

- Para poder sincronizar con el Master mediante Pull.

Ejecutar la aplicación:

rails s

luego ingresar a la url: http://0.0.0.0:3000/

Ejecutar pruebas:

cucumber

rspec

==========

Editor de texto Sublime Text 3

- Agregar el repositorio no oficial de Sublime text a nuestros repositorios:

sudo add-apt-repository ppa:webupd8team/sublime-text-3

- actualizar:

sudo apt-get update

- Instalar Sublime text 3:

sudo apt-get install sublime-text-installer

Anexo 2: Cómo configuro Travis para mi aplicación rails hosteada en GitHub

Vamos a https://travisci.org/ y tocamos en ‘Sign in with github’ y seleccionar "allow access" en la página de GitHub.

Luego en la página https://travisci.org/profile, donde se nos mostrará cada uno de nuestros repositorios en Github, junto con un botón On/Off a su lado.

Cambiar el botón de estado OFF a ON si es necesario del repositorio a integrar.

Luego hacer clic en configuración lo cual redireccionará a una página de GitHub (https://github.com/<nombre_usuario>/<nombre_del_proyecto>/settings/hooks).

Buscar en la lista de Available Service Hooks la entrada ‘travis’ y hacer clic. Esto nos mostrará un formulario. Llenar los campos:

user: nombre de usuario de GitHub

token: lo pueden obtener https://travisci.org/profile/<nombre del repositorio>/profile

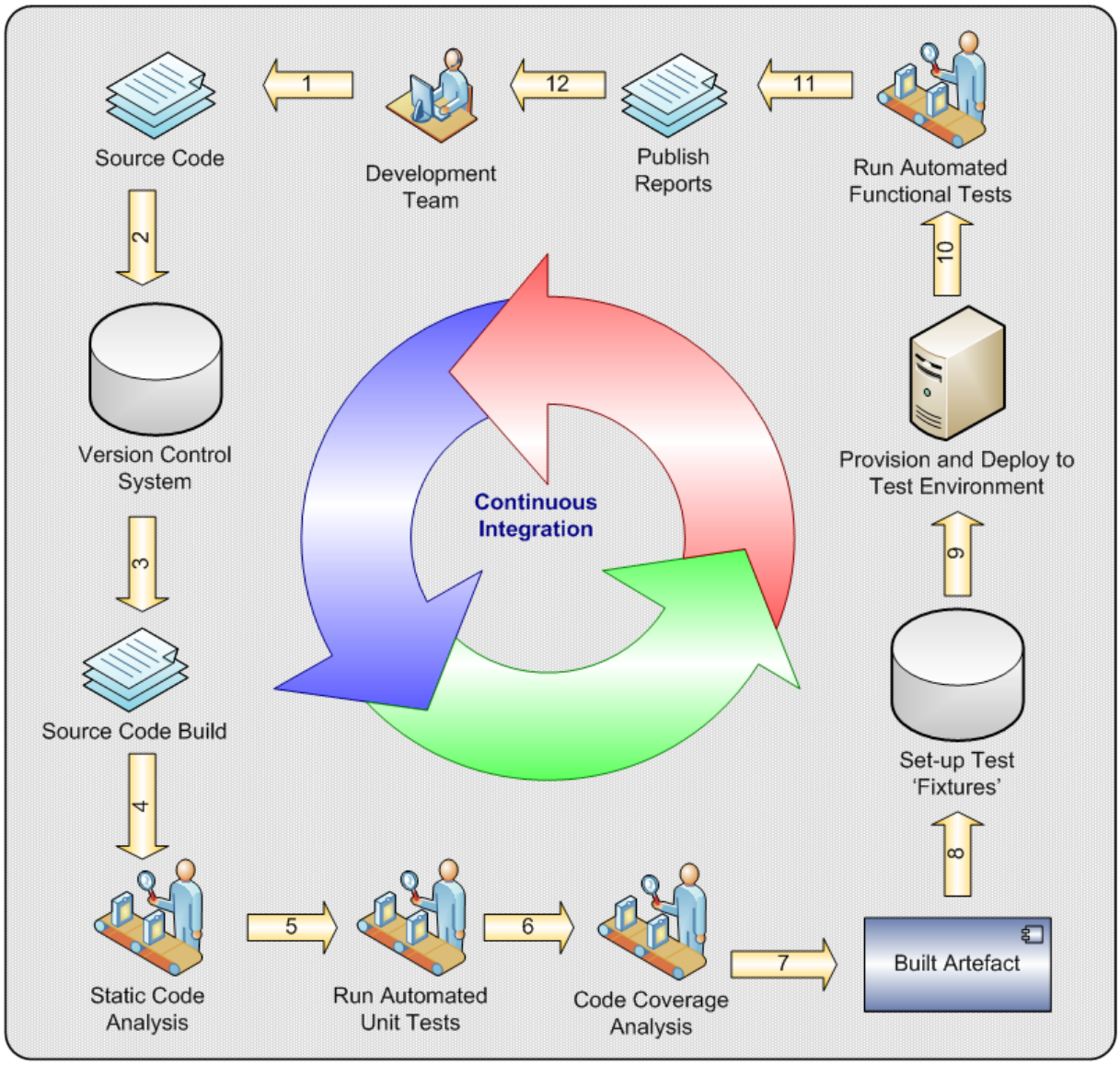
Nos aseguramos de que la casilla active este chequeada.

Clic en Update settings

Clic Test Hook

Para aplicaciones Rails, Travis-Ci conoce cómo son los distintos steps standards. Pero lo mejor es configurar un archivo .travis.yml según se indica para proyectos ruby en http://docs.travis-ci.com/user/languages/ruby/

Anexo 3: Integración Continua.



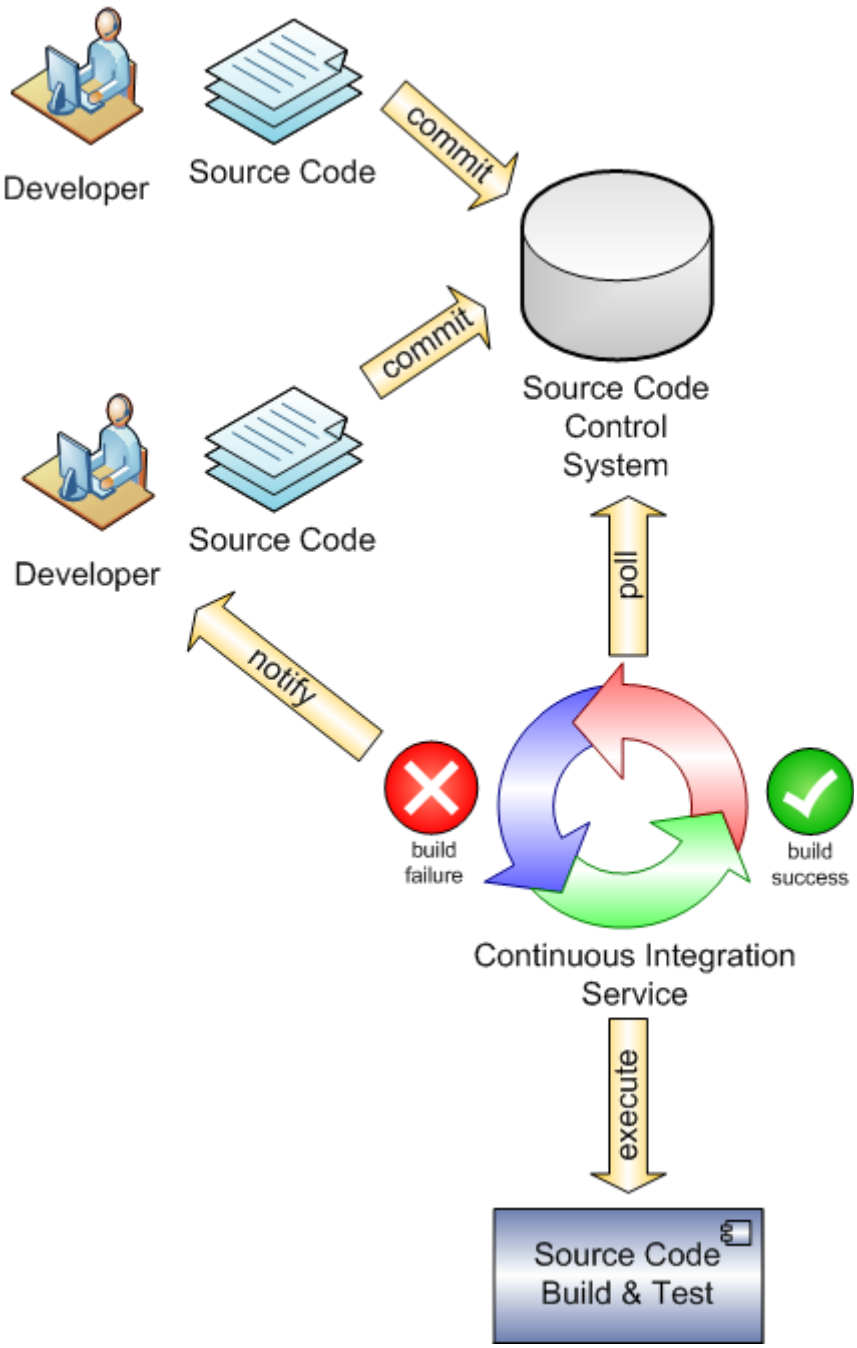
Resumen del proceso:

1. Developers work to transform the requirements or stories into source code using the programming language of choice.

2. They periodically check-in (commit) their work into a version control system (VCS)
3. The CI server is polling the VCS for changes. It initiates the build process when it encounters a change. The build is executed using a dedicated tool for the job such as

Maven, Ant or Rake etc. Depending upon the language used, the source code may need to be compiled.

4. Static analysis is performed on the source code, to ensure compliance with coding standards and to avoid common causes of bugs.
5. Automated unit tests are executed.
6. The percentage of the production code exercised by the unit tests is measured using a coverage analysis tool.
7. A binary artefact package is created. At this point we might want to assist derivation and provenence by including some additional metadata with the artefact e.g. a build timestamp, or the source code repository revision that was used to produce it.
8. Prepare for functional testing by setting up the test fixtures. For example, create the development database schema and populate it with some data.
9. Prepare for functional testing by provisioning a test environment and deploying the built artefact.
10. Functional tests are executed. Post-execution, tear down any fixtures or environment established in 8 and 9.
11. Generate reports to display the relevant metrics for the build. E.g. How many tests passed? What is the number and severity of coding standard violations?
12. The process is continuous of course! So rinse…and repeat….


Developer
Source Code
commit
Source Code
Control
System
commit
Developer
Source Code
poll
notify
build
failure
build
success
Continuous Integration
Service
execute
Source Code
Build & Test

Fuente:

http://mikeciblogs.wordpress.com/2010/03/30/continuous-integration-for-agile-project-managers-part-1/

http://mikeciblogs.wordpress.com/2010/04/27/continuous-integration-for-agile-project-managers-part-2/

http://mikeciblogs.wordpress.com/2010/06/15/continuous-integration-for-agile-project-managers-part-3/