UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

ESTUDIO COMPARATIVO DE METODOLOGÍAS DE DESARROLLO DE SOFTWARE ORIENTADAS A LA CALIDAD INTRÍNSECA

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

ERWIN GONZALO ÁLVAREZ CONTRERAS

PROFESOR GUÍA:
AGUSTÍN VILLENA MOYA

MIEMBROS DE LA COMISIÓN:
LUIS MATEU BRULÉ
TOMÁS BARROS ARANCIBIA

SANTIAGO DE CHILE
2014

Resumen

En el mundo del desarrollo de software, existe una necesidad constante de construir sistemas capaces de enfrentar los desafíos de un ambiente siempre cambiante. Uno de los movimientos que acepta el riesgo como parte de su proceso son las metodologías ágiles de desarrollo de software, y dentro de ellas, las metodologías de desarrollo guiado por pruebas (TDD).

El objetivo del Trabajo de Titulo es analizar estas metodologías, en particular el desarrollo guiado por pruebas clásico y el desarrollo guiado por comportamiento, con el fin de entender las fortalezas y debilidades de cada una de ellas. De ésta forma, a partir de este análisis, se espera dar una guía a aquellas personas que se interesan por la calidad, pero se ven abrumadas por la cantidad de estilos o variantes de TDD.

Con el fin de entender a fondo las metodologías, se realizaron experimentos técnicos, los cuales fueron analizados considerando diferentes criterios, entre ellos la necesidad de un diseño a priori, valor entregado al cliente y la curva de aprendizaje. Sin embargo, al momento de la comparación, estos criterios mostraron ser insuficientes pues resultaron ser cualitativos y muy subjetivos. Posteriormente, se intentó un análisis más abstracto, enfocado en el proceso mismo de desarrollo.

A partir de los resultados obtenidos, fue posible comprender que las técnicas presentadas no eran excluyentes, y que cada una de ellas puede entregar recursos interesantes tanto al equipo de desarrollo como al cliente de la aplicación o sistema, fortaleciendo la comunicación entre los actores del desarrollo de software o bien dándole la capacidad de responder de forma confiable al cambio.

Tabla de contenido

Introducción

En el mundo del desarrollo de Software, suelen existir los sistemas *legados*. Estos sistemas, o el código que los componen, se denominan así porque contienen componentes antiguos, los cuales se siguen usando porque "*funcionan"*. A su vez, no poseen un comportamiento totalmente documentado, y realizar estimaciones acerca de cuánto puede tomar implementar un nuevo requerimiento en ellos puede resultar muy difícil. Todo lo anterior los convierte en difíciles de cambiar.

Una definición actual, y tal vez más acertada, indica que código legado se diferencia del no-legado, no en su antigüedad, sino en que el primero sufre una ausencia de pruebas automatizadas (1) (2). Esto debido a que realizar un cambio en un sistema con pruebas automatizadas disminuye considerablemente la probabilidad de introducir un error en el mismo. Las pruebas a su vez, sirven de documentación para el sistema.

El concepto de "*Build Quality In"* (2) busca crear calidad en cada paso del desarrollo de software, y no sólo en el final del ciclo. Para ello, considera la creación de unidades de pruebas automatizadas, y por sobre todo, el cambio en la forma en que se afronta la construcción del producto.

Éste enfoque fue planteado, o "redescubierto" (3), por Kent Beck (4) en 1999, con el nombre de *Desarrollo Orientado a Pruebas* (o Test Driven Development: TDD) y desde entonces han surgido enfoques derivados de él, como TDD londinense y Desarrollo Guiado por Comportamiento (o *Behaviour Driven Development*: BDD).

En el transcurso de éste trabajo de Título, se espera estudiar éstas metodologías, para realizar un análisis comparativo de valor entregado versus los costos de implementación y así orientar a los programadores en cuál es el enfoque más adecuado a una situación específica.

Motivación

El desarrollo de software difiere de la práctica común de las ingenierías, en que los productos o artefactos generados por éste, en la mayoría de los casos, deben estar preparados para el cambio en sus especificaciones, o bien deben ser capaces de integrar nuevas funcionalidades de forma posterior a su producción. Bajo este contexto, es necesario asegurar la adaptabilidad de los sistemas, para así disminuir los costos de las soluciones, tanto humanos como financieros, a mediano y largo plazo.

La calidad intrínseca o interna, quiere decir que el software se construye desde el comienzo con evidencias de que cumple con ciertos criterios de calidad (fácil de entender, fácil de cambiar etc.). Esta característica es igual de importante que la anterior, pero es mucho más difícil darle visibilidad. La calidad intrínseca es la que nos ayuda a enfrentarnos a los cambios continuos y no previstos, pues permite modificar el comportamiento de un sistema de forma segura y previsible, ya que minimiza el riesgo de rehacer grandes partes del sistema ante un requerimiento nuevo. (5) Es por ello que todo diseño debe primero responder a la pregunta ¿Cómo probar la correctitud de lo que se espera crear?

Cuando un sistema de software no es capaz de adaptarse a nuevos requerimientos, o la implementación de ellos genera nuevos errores no controlados, la confianza dentro del equipo de desarrollo a cargo empieza a decaer. Así también, la confianza del cliente (o a quién se le provee la solución) también merma con respecto al equipo. Los costos de desarrollo se elevan y las fechas de entrega, previamente acordadas, no logran ser alcanzadas. Dentro de este escenario, es común afrontar el

problema integrando nuevas personas al equipo o invirtiendo más tiempo y dinero en el desarrollo, recursos no considerados inicialmente.

Entendiendo que dentro de las soluciones de software, cada línea de código puede romper o bien cambiar el comportamiento de manera inesperada, nace la necesidad de utilizar metodologías o procesos que aseguren la existencia de calidad intrínseca o “a priori”, (concepto denominado “Build Quality In”*)* desde el nivel más bajo del proceso de desarrollo.

A raíz de lo anterior, han nacido metodologías que se centran en la creación de pruebas automatizadas, las que en lugar de verificar la correctitud del sistema al final, impulsan el pensar en ella desde un inicio y de forma constante. Entre ellas, suele destacar el *Desarrollo de Software Orientado a Pruebas*, desde la cual han nacido diversas variantes, que difieren entre ellas en la forma de enfrentar el problema, así como los costos tanto humanos como financieros para implementar cada una de ellas.

Se plantea entonces la pregunta: ¿Cuál es la razón de ser de cada una de estas variantes del TDD? ¿Qué ventajas comparativas tienen unos respectos de las otras? ¿En qué situaciones es recomendable usar cada una de ellas?

Esto es lo que exploraremos en este Trabajo de Título.

Antecedentes

Metodología clásica de desarrollo *waterfall* y la calidad

Uno de los enfoques tradicionales más utilizados para enfrentar el desarrollo de software se denomina *waterfall* (6), y presenta una serie de procesos consecutivos en los cuales se separan los componentes del desarrollo en grandes etapas secuenciales, donde cada una de ellas se preocupa de alguna parte del desarrollo. Este enfoque se orienta a la calidad externa del software - que tan bien el sistema resuelve las necesidades de sus clientes y usuarios (si es funcional, confiable, seguro, nivel de disponibilidad, tiempo de respuesta, etc.). Este concepto de calidad es fácil de comprobar y entender, pues con regularidad es parte de un contrato o acuerdo pre definido al momento de decidir la construcción del software.

En general, se ejecutan las etapas de toma de requerimientos, análisis, diseño, implementación, pruebas y finalmente, mantenimiento (Ilustración 1). En el apartado de *pruebas*, se realiza la verificación de los requerimientos y el control de calidad del producto (o “Quality Assurance”, QA).

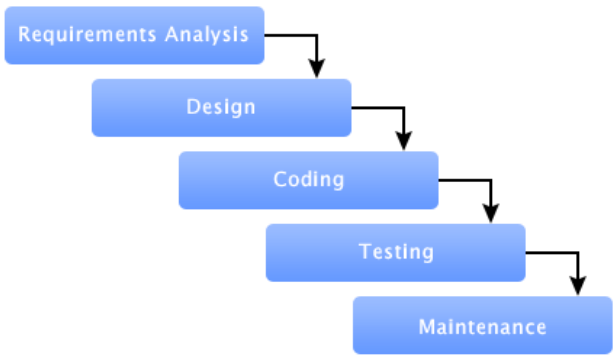

Ilustración 1: Modelo de cascada o "waterfall"

El principal problema de éste enfoque es que es una visión sumamente simplificada del concepto original, y, como su mismo autor explica en el artículo del cual se extrae el concepto, sufre de un riesgo muy alto e “invita al fracaso” (6).

Los riesgos inherentes a la naturaleza del desarrollo de software

El riesgo es uno de los principales problemas de la disciplina del desarrollo de software, entre ellos, podemos distinguir los siguientes (4):

- Retraso en la entrega: El día de la fecha comprometida llega, y el software aún no está completo ni es usable
- Proyecto cancelado: Debido a números retrasos, el proyecto se cancela y jamás es puesto en producción.
- El sistema se “oxida”: Luego de muchas fechas no cumplidas, es al fin posible poner el producto en producción, sin embargo, luego de un tiempo, el costo de adaptarlo a nuevas condiciones de negocio hace necesario reemplazarlo por un nuevo sistema.
- Tasa de fallos: La tasa de fallos del software una vez entregado es tan alta, que finalmente no es utilizado.
- Problema mal entendido: El software es puesto en producción, pero no soluciona el problema para el cual fue creado.
- Necesidades de negocio cambiantes: El software soluciona el problema por el cual fue creado, sin embargo éste hace mucho fue relegado a segundo plano por necesidades mucho mas importantes

- Características innecesarias: El software posee muchas características ingeniosas y entretenidas de desarrollar, pero no le reportan valor real al cliente.

- Abandono: En ocasiones, los desarrolladores que iniciaron el proyecto, al ver que nunca se le entrega valor al cliente, se frustran cada vez más, y finalmente deciden abandonarlo.

El modelo clásico de software intenta enfrentar éstos escenarios en cada una de sus etapas, en especial en las etapas de análisis y testing. En la etapa de análisis, el encargado de tomar los requerimientos intenta trasladar lo que él considera es el problema que hay que solucionar (a partir de reuniones o conversaciones con el cliente), en requerimientos de software, los que en su conjunto conformarán las directrices que permitirán generar una solución acorde al problema planteado originalmente.

Posteriormente, en etapa de testing se revisa y evalúa el software para ser llevado a un ambiente de producción. En la mayoría de los casos esta es la única instancia en la cual el producto es probado por sus posibles usuarios, dado que éste tipo de entregas se realizan con baja periodicidad, debido al extenso ciclo de desarrollo.

Como consecuencia de lo anterior, el aprendizaje del equipo de desarrollo acerca del negocio y del problema que se está intentando solucionar queda relegado a las instancias finales contempladas en la metodología, lo que hace muy caro (tanto en dinero como en tiempo), el corregir o adaptar el software.

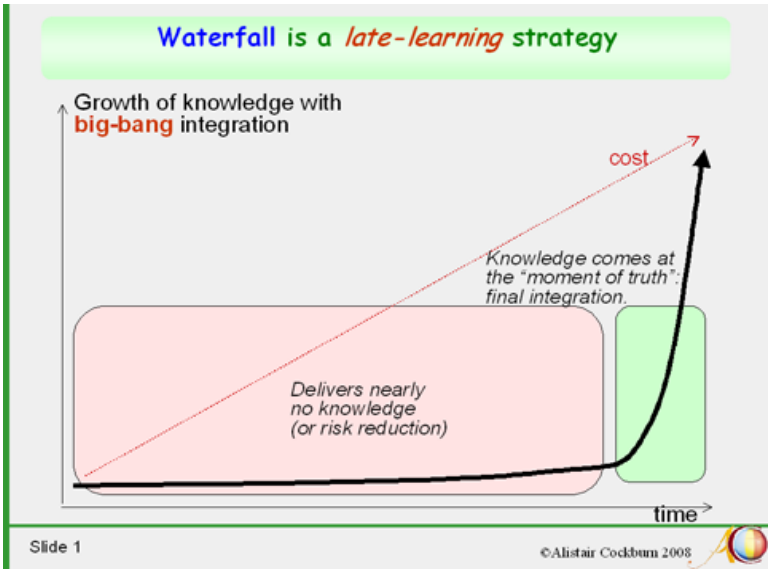

Ilustración 2: *Waterfall* es una metodología con creación de conocimiento tardío

En el gráfico anterior (Ilustración 2), se observa el flujo de conocimiento tardío descrito para *waterfall*. Durante gran parte del ciclo de desarrollo, el riesgo del proyecto no se reduce, pues se generan muchas hipótesis acerca del problema y del negocio que no son validadas, aumentando la incertidumbre en lugar de generar conocimiento. Éste último llega cerca del final del ciclo, en la etapa denominada *big-bang integration* (o integración big-bang), pues es en éste momento dónde todas los supuestos son puestos a prueba. Finalmente, cuando se logra la integración del sistema, si es que se logra, es cuando se genera el conocimiento (usualmente, éste proceso de conocimiento tardío suele ser muy agotador, tanto para el equipo de desarrollo como para el cliente) (7).

Extreme Programming (XP)

Es necesario aceptar que el riesgo es parte fundamental del desarrollo de software, y buscar una metodología que intente minimizarlo de forma continua, y que lo acepte como un problema que debe ser resuelto lo más pronto posible.

Extreme Programming (o XP) es una metodología ágil de desarrollo de software, que intenta ser flexible, liviana, eficiente, predecible, poco riesgosa y entretenida (4). Se enfoca principalmente en reducir el riesgo obteniendo retroalimentación acerca del producto lo más rápido y temprano posible, por medio de ciclos cortos de desarrollo y entrega continua de valor.

Para ello, integra al cliente con el equipo de trabajo, involucrándolo desde un comienzo en el proyecto y no sólo al inicio y final. Esto asegura que cualquier cambio en las prioridades o en el problema de negocio que se intenta resolver, sea comunicado rápidamente al equipo de desarrollo, evitando implementar funcionalidades innecesarias y reduciendo el desperdicio.

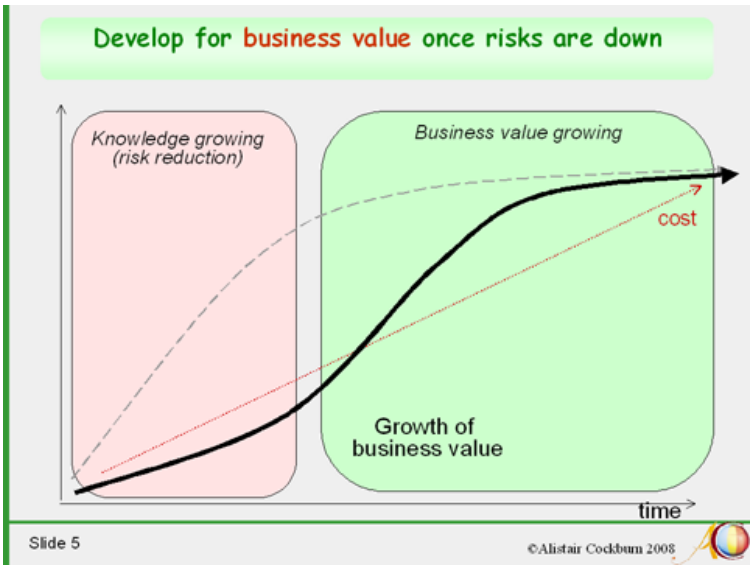

Ilustración 3: Generar valor una vez que se han reducido los riesgos

Lo anterior da como resultado que el conocimiento dentro del equipo se incremente fuertemente desde un inicio, eliminando incertidumbre y reduciendo el riesgo (Ilustración 3). Una vez que la curva de aprendizaje empieza a decaer, es el momento de tomar aquellas características que más agreguen valor al negocio y comenzar a implementarlas. De ésta manera, se cambia el foco desde el aprendizaje del equipo hacia el valor entregado al cliente.

Ésta visión de abrazar el cambio, tiene su costo en que el software que está siendo creado no sólo debe ser fabricado de forma incremental, sino que debe ser lo más flexible posible para ser capaz de responder ante requerimientos inesperados o cambios críticos de prioridades. Es por ello que uno de los principios fundamentales de XP junto con rápida retroalimentación, simplicidad y cambio incremental, es trabajo de calidad. ¿Pero cómo lograr calidad en un software que cambia continua e incrementalmente? La respuesta que da la metodología, son los test automatizados.

Desarrollo de Software Guiado por Pruebas (TDD: Test Driven Development)

La necesidad de comprobar que un componente se comporta de la manera esperada siempre ha existido en la ingeniería de software. Para ello, la comprobación manual no es suficiente, por lo que definimos como prueba o *test* automatizado, a un trozo de código fuente cuya finalidad es que:

- Dado un estado inicial (contexto de ejecución acotado) en cierto contexto pre-definido,

- cuando se realiza un cambio en el módulo bajo prueba (o SUT, Subject Under Test),
- se observe que el SUT se **comporta** (cambia su estado o entrega un resultado) de la forma esperada.

Éste tipo de pruebas son muy antiguas dentro de la ingeniería de software, y se habían utilizado con menor o mayor éxito mucho antes de la definición de XP. La diferencia es que XP no sólo impulsa el uso de pruebas automatizadas para demonstrar la correctitud del código generado, sino que lo engloba dentro de una disciplina en la cual se deben definir las pruebas antes de la implementación que los hacen válidas, denominada *Test driven development* (TDD) o *Desarrollo de software guiado por pruebas*.

Esto genera varios beneficios, principalmente que las pruebas muestran la intención del software (proveen una especificación acerca de cómo se debe comportar un módulo o componente). Además, las pruebas dirigen el desarrollo: se motiva al desarrollador a programar la solución más sencilla que pueda funcionar (8) lo que a su vez genera menos desperdicio, así también, el código generado tiende a ser más modular (y por tanto reutilizable), puesto que de ésta forma es más fácil probarlo de forma aislada.

Adicionalmente, tener una batería de pruebas ayuda a la confianza dentro del equipo de desarrollo, pues los defectos son capturados de forma inmediata (al no pasar las pruebas) y permite desarrollar sabiendo que nada de lo hecho ha introducido un error (9).

Es importante recalcar, que el desarrollo orientado a pruebas es mucho más que la creación de tests unitarios. Comúnmente, se considera que es una metodología acerca del diseño, no sólo sobre pruebas (10) (5) (11). Cambia el foco del desarrollo, en lugar de pensar que es lo que hace un módulo, se analiza cómo será utilizado y bajo que contextos.

A menudo, incluso por su autor, se considera que TDD fue más un redescubrimiento que una creación de la metodología XP (3), puesto que ya se había planteado anteriormente la necesidad de que las pruebas y el código que validan debían crecer juntos y la importancia de generar pruebas antes de la implementación (12) (13). Aun así, es en ella en que la disciplina se concretiza y se define como sigue:

- Escribir una prueba única que describa algún aspecto del programa o sistema.
- Ejecutar la prueba. Ésta debe fallar, pues aun no se implementa el comportamiento descrito.
- Escribir el código mínimo necesario para hacer que la prueba se ejecute exitosamente.
- Refactorizar (simplificar y clarificar) el código, tanto el de la solución como el de la prueba. Ésta etapa es crítica si se desea que el desarrollo orientado a pruebas produzca código de calidad. Al final de la refactorización, la suite de pruebas debe seguir en verde.
- Repetir, acumulando pruebas durante todo el proceso el desarrollo.

Éste flujo se denomina *red green refactor* (9), y tiene relación con la forma en que los IDE, generalmente, muestran el estado de las suites de pruebas. Cuando algún test

falla se muestra en rojo y cuando todos se ejecutan sin errores, en verde. Así también, terminados ambos pasos, es necesario refactorizar el código, cerrando el ciclo (Ilustración 4).

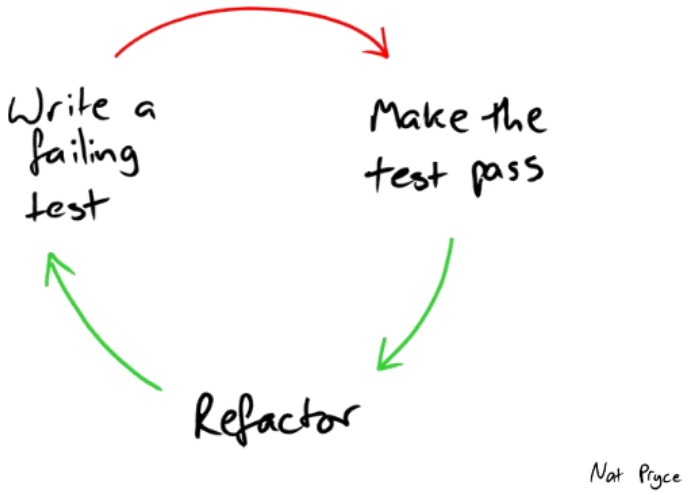

Ilustración 4: Proceso de TDD (5)

Este ciclo debe ser realizado de forma rápida, idealmente no debe tomar más de algunos segundos por cada etapa. Al agregar una característica, es posible que se creen docenas de pruebas, implementándola y refinándola en pequeños pasos, hasta que ya no quede nada por agregar o nada que eliminar por medio de refactorización. Esta forma de desarrollar provee muchos beneficios, entre los que se encuentran (5) (4) (10) (14):

- Separar interfaz con implementación: Al escribir primero el test, el desarrollador se enfoca en el "¿Qué?" en lugar del "¿Cómo?". Adicionalmente, al implementar, TDD recomienda escribir la cantidad mínima de código que haga el test pasar, minimizando el riesgo de crear desperdicio.

- Modularidad: TDD crea la necesidad de aislar cada componente del sistema, de forma que sea posible probar su comportamiento sin depender de su entorno de ejecución.

- Simplicidad: Al forzar escribir el código mínimo para pasar a la etapa *green*, es posible que el diseño generado sea simple, pues se encarga únicamente de generar valor y no en implementar características que *quizás* se necesiten a futuro.

- Documentación: Cada test implementado, indica claramente el "¿Qué?" que se intenta solucionar, lo que toma la forma de una documentación actualizada del diseño y del código fuente. Adicionalmente, si otro desarrollador quisiera entender cómo utilizar un componente, o bien que es lo que hace, puede ir directamente a las pruebas escritas para él.

- Reducir incidencia de errores: Estudios muestran (15), que TDD disminuye la ocurrencia de errores. Y si se utiliza correctamente, ayuda a crear una barrera para evitar introducir errores al modificar el sistema. Esto último ocurre debido a que los test aseguran que los componentes, por separado, sigan cumpliendo el mismo propósito para el cual fueron creados.

Premisa de la prioridad de las transformaciones (16)

Uno de los conceptos importantes acerca del proceso de TDD, es el de las *transformaciones*. Una *transformación* es una operación simple, que al ser aplicada en el código, modifica el comportamiento del sistema. Es por definición la antítesis de la refactorización.

Ésta definición es importante ya que para lograr que cada uno de los test pase desde rojo a verde, es necesario realizar algún tipo de transformación en el código. Estas transformaciones siempre deben ir desde lo específico a lo genérico, por ejemplo, reemplazando constantes con variables, enteros con arreglo o listas, o expresiones con funciones. Cabe preguntarse entonces, cuales son aquellas transformaciones que

permiten llegar a un algoritmo que solucione el problema de forma rápida, confiable y sin mayores sobresaltos.

La premisa de la prioridad de las transformaciones, señala que se pueden definir, de mayor a menor prioridad, las siguientes transformaciones (pueden existir más):

{}	→	nil
Nil	→	constante
constante	→	constante más compleja
constante	→	variable o argumento
declaración	→	declaraciones
código sin condiciones (*if*)	→	condiciones
escalar	→	arreglo o lista
arreglo	→	contendor u objeto
declaración	→	recursión
condición (*if*)	→	iteración
expresión	→	función
variable	→	asignación

En los primeros lugares de la lista, se encuentran aquellas transformaciones que revisten bajo riesgo y son más simples, mientras en los últimos lugares estarán las más arriesgadas y que involucrarán mayor cantidad de modificaciones al sistema para poder ser aplicadas. La premisa asegura que siempre que se quiera hacer pasar un test, se debe intentar ocupar las que están más arriba en la lista y si se sigue esta regla, la posibilidad de construir algoritmos erróneos que requieran grandes transformaciones es más baja, y por tanto se obtendrá un código más preparado para el cambio.

Todo lo anterior refuerza aún más la idea que los test deben ser incrementos pequeños de funcionalidad, pues si es así, será más fácil implementar transformaciones de alta prioridad para ellos.

Como se implementa TDD

Kent Beck construyó el primer framework de pruebas automatizadas (17), escrito en SmallTalk, en dónde se definió la filosofía y la estructura que posteriormente especificarían a toda una familia de frameworks, denominada xUnit. Más adelante, y con la ayuda de Erich Gamma (18), una versión en Java fue implementada (jUnit), ayudando a la masificación de la idea e instaurando el estándar de verde y rojo para los casos de pruebas correctas y fallidas respectivamente.

Actualmente, existen implementaciones del diseño original en virtualmente casi todos los lenguajes de programación, tales como Test::Unit (Ruby), unittest (Python), CppUnit (C++), entre otras.

Estas librerías poseen un diseño común, ya sea los definidos estrictamente por la arquitectura xUnit, o por variaciones de ésta. En general, se pueden definir los siguientes componentes (17):

- Fixture: Son las condiciones o estados en los cuales se necesita que esté el sistema para ejecutar las pruebas. Se debe asegurar no sólo que exista este contexto previamente, sino que posteriormente se revierta para no influir en el resto de las pruebas.

- TestCase: Define una clase desde la cual todos los test unitarios heredarán. Dentro de estas subclases se implementan las pruebas unitarias.

- SetUp y TearDown: Métodos de TestCase, que pueden ser sobrescritos para definir estados específicos para el módulo bajo prueba. El método SetUp se ejecutará antes de cada test unitario y TearDown de forma posterior a cada ejecución.

- Tests: Son las pruebas que componen una subclase de TestCase y se definen como sus métodos de instancia. Se suelen definir tres etapas:

    - Arrange: Se definen variables o componentes aún más específicos para la prueba a ejecutarse.

    - Act: Se realiza la acción que está bajo prueba.

    - Assert: Se comprueba que el resultado generado por la acción entrega el resultado esperado.

- Test Suite: Es un set de pruebas que comparten un mismo *fixture*. El fin de agruparlas es ejecutarlas juntas, pero siempre considerando que el orden en que lo hagan no debe influir en los resultados.

Variantes de TDD

Dada la definición anterior, existen varias formas de afrontar el concepto de TDD, algunas de ellas más técnicas y otras orientadas al ámbito de negocio. Entre ellas, se pueden consideran las siguientes como las más importantes.

TDD Clásico

Llamado también “Chicago school TDD” (o de la escuela de Chicago), en ocasiones denominada también escuela de Detroit, enfrenta el problema de la

realización de pruebas automatizadas con una visión “bottom up” (9). Esto quiere decir que se realizan pruebas incrementalmente desde las funcionalidades más pequeñas del sistema hasta alcanzar las capas superiores de la arquitectura. Se denomina “clásico” pues fue el que originalmente se impuso como metodología de TDD.

Desarrollo Guiado por Comportamiento (BDD: Behaviour Driven Development)

El desarrollo guiado por comportamiento, nace de la premisa que los términos utilizados por TDD no son capaces de comunicar correctamente su propósito, y que a su vez no ayudan a separar el “¿Qué?” del “¿Cómo?” al momento de implementar o diseñar el sistema. Para suplir esta falencia, BDD construye un vocabulario nuevo, donde se hace énfasis en el comportamiento del SUT, por medio de intercambiar palabras técnicas como *assert,* por conceptos más naturales del lenguaje, como *should* (o *debería*, en español) (19). Esto último hace posible elevar la metodologías a un plano más de negocio, menos técnico y accesible al cliente. Requiere no probar funcionalidades directamente, sino el comportamiento del software dados ciertos escenarios, o historias de usuario.

Adopción de la práctica de TDD

En general, implementar cualquiera de los procesos de desarrollo definidos anteriormente resulta costoso inicialmente, principalmente porque es necesario un cambio en la manera de entender y desarrollar software. Se vuelve interesante entonces, considerar las metodologías de software orientadas a pruebas automatizadas, como una forma de implementar un mejora temprana y continua del producto o solución, evitando siempre el desperdicio.

A su vez, es interesante entender y aplicar las herramientas adecuadas para las particularidades de cada sistema, sin ocupar procesos que se transforman en un problema y una carga para el equipo de desarrollo. Dada la alta barrera de entrada a ellos, éste trabajo busca realizar un análisis comparativo de las dos metodologías presentadas (TDD y BDD), enfocándose en el valor que pueden ser capaces de aportar y el costo asociado a ese valor.

Objetivos

Objetivo general

Para este trabajo de título, el objetivo es desarrollar un marco comparativo para dos metodologías de desarrollo orientado a pruebas, que permita escoger la metodología adecuada según el producto que se desee crear.

Objetivos específicos

Los objetivos específicos del Trabajo de Título son los siguientes:

- Estudiar los procesos de desarrollo que incorporen calidad intrínseca, en particular:
    - Test Driven Development, escuela clásica
    - Behaviour Driven Development
- Aplicar técnicamente los métodos en cuestión, realizando experimentos simples para verificar sus propiedades más importantes.
- Reconocer el flujo de trabajo de cada método y analizarlos, buscando patrones comunes y diferencias.

Plan de Trabajo

Durante el transcurso de éste Trabajo de Titulo, se espera que el alumno logre lo siguiente:

- Conocer la bibliografía necesaria, así como los exponentes más relevantes de las metodologías.
- Aplicar los conocimientos adquiridos técnicamente, ya sea mediante ejemplos simples, tutoriales o experiencias más elaboradas. Se debe elegir un lenguaje de programación y los frameworks de pruebas a utilizar. El lenguaje en sí es irrelevante, pero se priorizará aquel que entregue las herramientas necesarias para realizar de mejor forma la experimentación.
- Codificar los patrones de cada variante, en particular su flujo de trabajo, y analizar las diferencias encontradas entre cada una. Finalmente, entregar artefactos que permitan el acceso de la comunidad a lo aprendido (tanto al uso de cada una de las metodologías como a las conclusiones del análisis comparativo). Éstos puede ser en forma de artículos, tutoriales, guías, “screencasts”, “Coding Dojos”, etc.
- Validar y discutir lo aprendido con los integrantes de la comunidad de desarrollo de software local, con el fin de nutrir la experiencia personal con la de la comunidad. Esto debe continuar durante todo el resto del proceso.

Descripción de la solución

Herramientas seleccionadas

El lenguaje de programación escogido para realizar la experimentación, es Ruby. Se consideró inicialmente lenguajes como Java o Python. Sin embargo, se escogió Ruby por lo siguiente:

- Herramientas: El lenguaje posee herramientas avanzadas y simples para realizar pruebas. Se utilizará *Unit::Test* para las pruebas unitarias, la gema *mocha* para el uso de mocks y *Rspec* como framework de BDD.

- Expresividad y simpleza: La sintaxis de Ruby es clara, concisa y simple de leer, facilitando el entender los ejemplos. Incluso su creador ha mencionado que una de sus metas es que sea un *lenguaje natural*.

- Comunidad: Ruby posee una comunidad bastante activa en Chile, y suele ser un lenguaje utilizado por desarrolladores jóvenes y startups. Debido a lo difícil que suele ser implementar TDD en desarrolladores con más experiencia, dado el cambio de paradigma que deben enfrentar, es interesante mostrar los resultados de la investigación a un grupo más abierto a adoptar éste tipo de metodologías.

Metodología para el estudio comparativo

Para comparar objetos de forma efectiva, es necesario precisar una serie de características que permitan definirlos de manera única, teniendo en mente el contexto y el objetivo final de esta comparación.

Los objetos se pueden definir mediante un conjunto de pares clave-valor, luego para compararlos es necesario realizar la intersección de sus conjuntos de claves. Este nuevo conjunto se denomina criterios de comparación.

Por ejemplo, si se quiere comparar 3 departamentos en arriendo, considerando que se busca encontrar la mejor opción por el menor precio y que además se quiere ahorrar lo más posible, se tiene la siguiente tabla comparativa:

Criterio	Departamento 1	Departamento 2	Departamento 3
Ubicación	Blanco Encalada 1723	Eliodoro Yáñez 808	Ricardo Lyon 2874
Distancia al metro	400 metros	108 metros	3.3 kilómetros
Metros totales	74 m²	80 m²	67 m²
Piso	13	5	8
Dormitorios	3	3	2
Baños	2	2	2
Orientación	sur	nor poniente	sur poniente
Gastos comunes	$80.000	$50.000	$80.000
Arriendo	$350.000	$380.000	$380.000

Dado el contexto y el objetivo de la comparación, la mejor elección es el Departamento 2, ya que es el que se encuentra más cercano al metro (por ende se puede ahorrar en combustible), tiene una orientación que permite ahorrar en calefacción y tiene bajos gastos comunes.

En este trabajo de título, no se definirá la mejor metodología, sino que las fortalezas y debilidades de cada una para así tomar una decisión informada al momento de elegir una de ellas para realizar un proyecto.

Criterios de comparación

A continuación se describen los criterios de comparación que se utilizarán en este trabajo investigativo.

- Madurez de las herramientas: ¿Qué grado de madurez tienen las herramientas en los distintos lenguajes de programación?
- Dependencia de un diseño a priori; ¿Qué tan necesario es tener un diseño de la solución antes de implementarla siguiendo la metodología?
- Fragilidad de los test: ¿Qué tan frecuente es el caso de que luego de implementar algo, los test comiencen a fallar al usar la metodología?
- Valor directo al cliente: ¿Cómo la metodología ayuda a que el cliente obtenga un mejor producto y como ese valor es claro para él?
- Curva de aprendizaje: ¿Qué tan difícil es comenzar a practicar de forma correcta la metodología?
- Costo de mantención: ¿Qué tan costoso, tanto en tiempo y recursos, es mantener la solución implementada con la metodología?

Implementación de la solución

TDD Clásico

El desarrollo guiado por pruebas clásico define una perspectiva *bottom up*, en la cual los componentes se desarrollan desde los más sencillos, hacia los más complejos. En general, si bien permite una etapa reducida de análisis y diseño, se preferirá una arquitectura que evolucione constantemente y que emerja a través del aprendizaje creado por cada nuevo *test*.

Bowling Kata

Se define una *Kata* (20), como un pequeño y entretenido ejemplo que no debería tomar más de una hora o dos para ser resuelto. El objetivo es repetir el ejercicio una y otra vez, y en cada iteración, encontrar una mejor manera de solucionarlo. No es acerca del código que se genera, sino acerca del proceso realizado para obtenerlo.

Un ejemplo clásico es el ejercicio de desarrollar una solución de software capaz de calcular el puntaje de una partida de bolos o *bowling* (10). El bowling es un deporte que consiste en derribar un conjunto de 10 bolos o pinos, lanzando contra ellos una bola. Una partida típica consiste en 10 juegos o *tiradas*, donde cada juego consta de a lo más dos lanzamientos. El puntaje de cada lanzamiento es el número de pinos derribados. Existen dos casos especiales:

- *Strike* o *chuza:* consistente en derribar los 10 pinos en el primer lanzamiento de una *tirada*. El puntaje en este caso es 10, más la suma de los siguientes dos

lanzamientos. En el caso que esto ocurra en la última *tirada*, se permiten dos lanzamientos adicionales para calcular su puntaje.

- *Spare o media chuza*: Ocurre cuando se derriban los 10 pinos usando los dos lanzamientos de una *tirada*. Para éste caso, el puntaje es de 10 más lo obtenido en el siguiente lanzamiento.

Implementación

La implementación se hará utilizando Ruby 2.0 y el framework de pruebas será Unit::Test, el código final se puede revisar en los anexos (pág. 95). Además, por simplicidad, no se consideran entradas erróneas.

Al iniciar el desarrollo, una solución posible se puede presentar utilizando el siguiente diagrama (Ilustración 5).

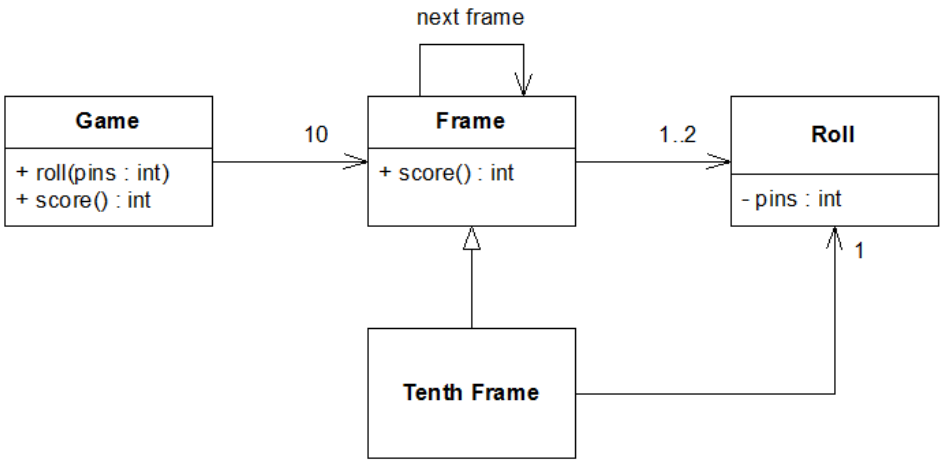

Ilustración 5: Diseño para Bowling Kata

El diseño indica que se creará una clase *Game*, encargada de entregar el puntaje y de crear los lanzamientos. Para ello, se almacenara cada *tirada* en *Frame*. Esta clase guarda su puntaje y además una referencia hacia la siguiente *Frame*. Adicionalmente, tiene una subclase *Tenth Frame*, para manejar el caso especial de la última *tirada*.

En el desarrollo guiado por pruebas, el diseño es importante, pero se considera una guía y no algo escrito en piedra. El diseño en TDD, se espera que emerja de las pruebas y no de una sesión de diseño.

En la figura del ciclo de TDD clásico (Ilustración 4), el primer paso para escribir código de producción es escribir una prueba que falle. Es por esto que antes de crear directamente la clase se debe implementar un test sencillo, tal que para hacerlo pasar, debo escribir el código que necesito. En este caso, este paso es trivial:

```
# test/bowling_game_test.rb
require "test/unit"
...
def test_bowling_game_object
  BowlingGame.new
end
```

Para pasar a la etapa *green*, la solución también es trivial, pues se implementa lo mínimo que haga al test válido.

```
# lib/bowling_game.rb
class BowlingGame

end
```

Si bien este tipo de pruebas parecen triviales, es importante implementarlas cuando se inicia el aprendizaje de la metodología, pues ayudan a acostumbrarse a la

mecánica. De ésta forma también podemos agregar el método *roll*, el cual genera una *tirada*:

```
# test/bowling_game_test.rb
...

def test_bowling_game_has_method_roll
  assert BowlingGame.new.respond_to?(:roll)
end
```

Y la prueba que lo hace *pasar*:

```
# lib/bowling_game.rb
class BowlingGame
  def roll

  end
end
```

En este punto, es interesante observar la clase test, pues puede ser necesario refactorizar. La refactorización es un proceso continuo y constante, y es vital si se quiere lograr un software mantenible y autodocumentado.

```
# test/bowling_game_test.rb
class TestBowlingGame < Test::Unit::TestCase

  def test_bowling_game_object
    BowlingGame.new
  end

  def test_bowling_game_has_method_roll
    assert BowlingGame.new.respond_to?(:roll)
  end
end
```

Para este caso, existe una duplicación en la instanciación de la clase *BowlingGame*, que se puede remover fácilmente, moviéndola al método *setup* de

la clase test. Esto último provoca que el primer test que se escribió sea redundante, y se puede eliminar. La prueba queda de la forma:

```
# test/bowling_game_test.rb
class TestBowlingGame < Test::Unit::TestCase

  def setup
    @game = BowlingGame.new
  end

  def test_bowling_game_has_method_roll
    assert @game.respond_to?(:roll)
  end
end
```

Los test deben ser los más expresivos posibles, pues ayudan a la documentación del código fuente.

En este punto, el desarrollo guiado por pruebas genera la pregunta ¿Cuál es el próximo test que genera valor? Siguiendo con los métodos de instancia para *BowlingGame*, sería interesante implementar *score*, lo cual origina otro problema, y es que *score* posee casos especiales, tales como chuza o media chuza.

La filosofía de TDD, indica que se debe implementar lo más simple que haga al test pasar, en el caso de un juego de bolos, podemos considerar que el juego más simple, es aquel con puntaje final cero, pues todos los lanzamientos fueron errados.

```
# test/bowling_game_test.rb
...
def test_game_with_all_rolls_missed
  20.times { @game.roll 0}
  assert_equal(0, @game.score)
end
```

Podemos hacer pasar el test, con una implementación trivial.

```
# lib/bowling_game.rb
...
def score
  0
end
```

¿Qué valor entrega ésta solución? La implementación de score es burda, pues simplemente retorna cero. El valor en éste caso no está en ella, sino en los test que la guiaron. Esto es importante, porque define un invariante: “Cuando todos los lanzamientos son errados, entonces el puntaje final es cero”. Las pruebas aseguran que los cambios posteriores que ocurran en el método *score* seguirán cumpliendo esta condición.

Se puede definir otro caso simple antes de enfrentar los casos especiales como chuza o media chuza. Estas pruebas ayudan a fortalecer la confianza en la suite de test al momento de implementar casos más complicados. En particular, esta prueba señala un juego donde en todos los lanzamientos se derriba sólo un pino, para un puntaje final de 20.

```
# test/bowling_game_test.rb
...
def test_game_with_all_ones
  20.times { @game.roll 1}
  assert_equal(20, @game.score)
end
```

Es evidente que existe duplicación en las últimas dos pruebas, pero en éste momento no es relevante. El foco cuando se está en red es hacer los test pasar. Para ello se puede agregar una variable de instancia que almacene el score del juego y entregarla en el método score.

```
# lib/bowling_game.rb
class BowlingGame

  def initialize
    @score = 0
  end

  def roll(pins)
    @score += pins
  end
```

```
  def score
    @score
  end
end
```

En el paso de refactorización, se elimina la duplicación de los test, extrayendo un método para los lanzamientos consecutivos. Adicionalmente, se puede eliminar el test que verifica la existencia el método roll.

```
# test/bowling_game_test.rb
...
def test_game_with_all_rolls_missed
  roll(20, 0)
  assert_equal(0, @game.score)
end

def test_game_with_all_ones
  roll(20, 1)
  assert_equal(20, @game.score)
end

private

def roll(n, pins)
  n.times { @game.roll pins}
end
```

Dada la implementación de *score*, un juego con sólo dos, tres o cuatro aciertos en cada uno, no va a producir errores, por tanto escribir estos test no genera valor. Sin embargo, para el caso de 5 aciertos, no debería ser así, pues este caso produce medias chuzas. Para simplificar esta prueba, considerar una única media chuza, seguida sólo por un tiro con 3 puntos y luego intentos fallidos sin puntaje.

```
# test/bowling_game_test.rb
...
def test_one_spare
  @game.roll(5)
  @game.roll(5)
  @game.roll(3)
  roll(17, 0)
  assert_equal(16, @game.score)
end
```

¿Cómo hacer *pasar* este test? Si se observa el diseño de la aplicación hasta el momento, habría que implementar algún tipo de variable encargada de guardar el intento anterior y hacer la verificación dentro del método *roll*, ésta sería la solución más simple que haga pasar el test.

Lo anterior refleja un problema de diseño, denominado *misplaced responsability* (11) o *responsabilidad fuera de lugar*. Si otro desarrollador quisiera saber cómo se calcula el puntaje, intuitivamente iría al método *score*. Para su sorpresa, no es allí donde está el algoritmo, sino en *roll*. Es necesario entonces refactorizar. Sin embargo, en este momento los test están en *red*. Se recomienda para este tipo de casos comentar el test que falla y trabajar con aquellos que se sabe son correctos. Una vez hecho esto, hay que mover la responsabilidad del cálculo de puntaje a la función *score*, de tal forma que el nuevo diseño aprueba todos los test.

```ruby
# lib/bowling_game.rb
class BowlingGame

  def initialize
    @rolls = []
  end

  def roll(pins)
    @rolls << pins
  end

  def score
    @rolls.reduce(:+)
  end
end
```

Con esta implementación, los test pasan y se mueve la responsabilidad del cálculo de puntaje donde corresponde. Sin embargo, aún tiene un problema y es que para comprobar la existencia de medias chuzas, se debe verificar que la suma de los

intentos de un lanzamiento sea igual a 10, cosa que la implementación actual de *score* no permite.

Se vuelve a comentar el test que falla y a refactorizar, con el fin de agregar el concepto de *frame* a la implementación.

```
# lib/bowling_game.rb
...
def score
  frames = (0..19).step(2)
  frames.reduce(0) do |score, i|
    score + @rolls[i] + @rolls[i+1]
  end
end
```

Ahora, se descomenta el test_one_spare (vuelve a pasar a *red)* y nos volvemos a enfocar en verificar que la suma de una tirada sea 10. En tal caso, se le suma el siguiente lanzamiento al score.

```
# lib/bowling_game.rb
...
def score
  frames = (0..19).step(2)

  frames.reduce(0) do |score, i|
    if @rolls[i] + @rolls[i+1] == 10
      score + @rolls[i] + @rolls[i+1] + @rolls[i+2]
    else
      score + @rolls[i] + @rolls[i+1]
    end
  end
end
```

Una vez que los test pasan es momento de refactorizar. La implementación posee duplicación evidente, además de ser confusa. Una mejor versión puede ser:

```
# lib/bowling_game.rb
...
def score

  firsts_in_frames.reduce(0) do |score, i|
```

```
    if is_spare(i)
      score += @rolls[i+2]
    end
    score + frame_score(i)
  end

end

private

def is_spare(first_in_frame)
  @rolls[first_in_frame] + @rolls[first_in_frame +1] == 10
end

def frame_score(first_in_frame)
  @rolls[first_in_frame] + @rolls[first_in_frame+1]
end
```

Adicionalmente, se pueden refactorizar los test.

```
# test/bowling_game_test.rb
...
def test_one_spare
  roll_spare
  @game.roll(3)
  roll(17, 0)
  assert_equal(16, @game.score)
end

private

def roll_spare
  @game.roll(5)
  @game.roll(5)
end
```

La otra regla especial de bowling, es la chuza o strike. Esto ocurre cuando en el primer intento del lanzamiento, se derriban los 10 pinos. Esto da 10 puntos, además de los puntos obtenidos en el siguiente lanzamiento. El test más simple que muestra este comportamiento es:

```
# test/bowling_game_test.rb
...
def test_one_strike
  @game.roll(10)
  @game.roll(4)
  @game.roll(3)
  roll(16, 0)
```

```
  assert_equal(24, @game.score)
end
...
```

Esto muestra un problema en la implementación. Ésta asume que hay dos intentos exactos por lanzamiento, lo cual no siempre cierto, en particular en el caso en que se logra una chuza o *strike*. Cuando esto ocurre, la siguiente *tirada* comienza inmediatamente, por tanto el primer lanzamiento de cada *tirada* varía según el resultado que se obtuvo en las anteriores, lo que lleva a deducir que una implementación con estos valores fijos no es correcta. Una implementación que aprueba todos los test y que corrige el problema es como sigue:

```
# lib/bowling_game.rb
...
def score

  score = 0
  first_in_frame = 0
  10.times do |frame|

    if is_strike(first_in_frame)
      score += score_for_strike(first_in_frame)
      first_in_frame += 1
    else
      if is_spare(first_in_frame)
        score += score_for_spare(first_in_frame)
      else
        score += frame_score(first_in_frame)
      end
      first_in_frame += 2
    end
  end

  score
end
...
```

Podemos ahora, agregar un nuevo test para el caso del juego perfecto, es decir, 10 lanzamientos perfectos. En el caso de la última *tirada*, se consideran dos

lanzamientos extras, en los cuales también se deben derribar todos los pinos, consiguiendo 30 puntos. El puntaje final para este caso es 300 puntos.

```
# test/bowling_game_test.rb
...
def perfect_game
  roll(12,10)
  assert_equal(300, @game.score)
end
...
```

Sorprendentemente, este test pasa, y la suite sigue en *green*. ¿Por qué? La respuesta se puede ver en el código de *score*. El código dentro del bloque de *times* es exactamente las reglas de *bowling*. De hecho, éste algoritmo es el que calcula correctamente el puntaje y la solución ha terminado.

Gracias al desarrollo guiado por pruebas, se ha llegado a una solución muchísimo más sencilla que el diseño tentativo original (Ilustración 5) y se han reemplazado cuatro clases por un algoritmo simple. Aún así, es una solución válida para el problema.

Finalmente si se quisiera implementar el diseño original, gracias a los tests se podría refactorizar el código con la seguridad que la solución se sigue comportando de la forma esperada.

Game of life Kata

El juego de la vida (o *Game of life*) es un autómata celular de *cero jugadores*. Esto quiere decir que su evolución no está determinada por variables externas, sino sólo por su estado inicial, y una serie de reglas que sistematizan cada iteración. El universo o tablero es una grilla de dos dimensiones que se extiende indefinidamente, y

cada par de coordenadas (x, y) define una *célula*. Cada célula puede estar muerta o viva, y cada una de ellas posee 8 células vecinas.

El estado de la malla cambia en turnos discretos, y en cada turno se actualizan simultáneamente cada una de las células, respetando las siguientes reglas:

1. Cualquier célula viva con dos o tres vecinas vivas *sobrevive* la siguiente iteración.
2. Cualquier célula muerta con tres vecinas vivas, *nace* en la siguiente iteración, causado por reproducción.
3. Cualquier célula viva con menos de dos vecinas vivas *muere*, causado por baja población.
4. Cualquier célula viva con más de tres vecinas vivas *muere*, causado por sobrepoblación.

El patrón inicial se denomina *semilla*, y las reglas son aplicadas sobre ella continuamente, generación por generación, definiendo los estados del sistema.

La kata o ejercicio consiste en la implementación del juego, usando TDD.

Implementación

Primero que todo, se definirá el universo de las células como un sistema de coordenadas cartesiano, en el cual los valores de X se definen horizontalmente, aumentando de izquierda a derecha y los valores de Y definidos de forma vertical, aumentando desde abajo hacia arriba.

En ocasiones como ésta, en las cuales puede ser difícil analizar el resultado final del sistema rápidamente, es útil pensar en ejemplos para facilitar su implementación,

utilizando patrones conocidos con resultados esperados. Uno de los casos canónicos en el juego de la vida es el denominado *blinker* (Ilustración 6), en el cual el sistema oscila entre dos estados indefinidamente.

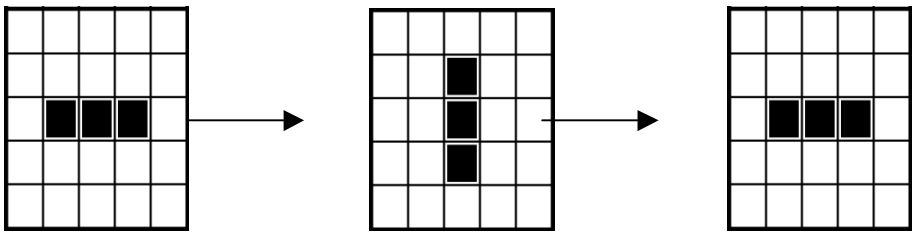

Ilustración 6: Tres Iteraciones de Blinker: El sistema oscila entre el estado 1 y 2 de forma indefinida.

Éste ejemplo es interesante pues considera la aplicación de varias reglas simultáneamente. Si bien no se va a implementar inmediatamente, escribir el test ayuda a entender *hacia donde se va*, evitando *analysis paralysis* (o parálisis del desarrollo debido a un proceso de análisis demasiado extenso).

```
# test/game_of_life_test.rb
...
def test_blinker

  step_one = [
    [1,0],
    [1,1],
    [1,2]
  ]

  step_two = [
    [0,1],
    [1,1],
    [2,1]
  ]

  game = GameOfLife.new(step_one)

  assert_equal(step_two.sort, game.next.sort)
  assert_equal(step_one.sort, game.next.sort)
end
```

Una vez escrito, simplemente se comenta, pues es no va a *pasar* inmediatamente, y la suite de test debe estar en *green* para poder avanzar. Al crear el test se han definido implícitamente algunas convenciones, como por ejemplo, que las células utilizarán un plano cartesiano para dar su posición, y que sólo se especifican aquellas que están *vivas*.

El caso más sencillo con el cual se puede comenzar, es aquél en que luego de una iteración, el juego de la vida que se inició sin ninguna célula viva mantiene su estado indefinidamente.

```
 # lib/game_of_life.rb

...

def test_blank_board
  seed = []
  game = GameOfLife.new(seed)
  assert_equal([], game.next)
end
```

A medida que se empieza a interiorizar la metodología en la forma de desarrollar, se pueden elegir dar pasos más grandes para cada test. Por ejemplo, para validar éste test, no basta con crear la clase, sino también el método de instancia *next*, lo cual es bastante sencillo.

```
class GameOfLife

  attr_reader :alive_cells

  def initialize(seed=[])
    @alive_cells = seed
  end

  def next
    @alive_cells
  end
end
```

El tamaño del *paso* no es una restricción, sino que sólo guarda relación con la experiencia del practicante de TDD.

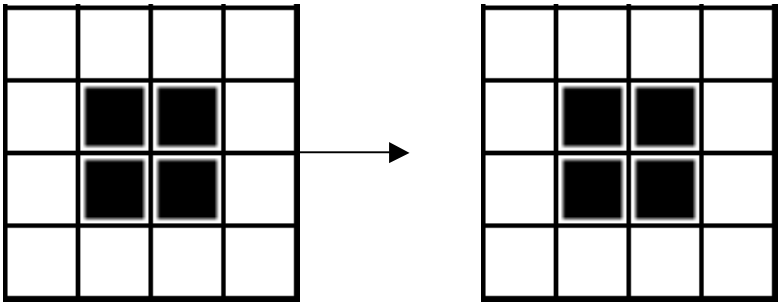

Ilustración 7: El patrón *block* se mantiene constante durante todas las iteraciones.

Nuevamente con la suite de tests en *green*, se debe pensar en una prueba que falle. Esto es importante, ya que se podría probar por ejemplo, que al utilizar como semilla cuatro células colindantes, el resultado de todas las iteraciones siguientes es constante (patrón conocido como *block:* Ilustración 7). Sin embargo, es claro que éste test no llevará a un estado *red*, por tanto no sirve *en éste momento*. En general, se necesita que cada nuevo caso aporte conocimiento, ya sea de la naturaleza del problema o acerca de la correctitud de la implementación. Así también, se necesita que sea rápido de implementar, pues se están tomando pasos incrementales que generen valor.

Un caso que si bien es más simple, pero si genera aprendizaje, es calcular la regla de sobrevivencia, es decir, que deben sobrevivir en cada iteración aquellas células que tienen entre 2 y 3 vecinas vivas. Para ello, se necesitará una función que devuelva los vecinos de una célula. Se puede construir una prueba como la siguiente:

```ruby
# test/game_of_life_test.rb

  def test_neighbours_in_origin
    origin = [0,0]
    game = GameOfLife.new([origin])
    neighbours = game.neighbours(origin)

    assert_equal(8, neighbours.size)
    assert(!neighbours.include?([0,0]))
    assert(neighbours.include?([1,0]))
    assert(neighbours.include?([-1,-1]))
  end
```

La prueba asegura que, para el caso de una célula en el origen, se devuelven correctamente sus células adyacentes. En éste caso no se prueba cada uno de los elementos devueltos, sino aquellas propiedades que aseguran un resultado correcto, como por ejemplo que no se incluya a sí mismo, o que la cantidad de sus vecinos sea 8. Una implementación que hace pasar el test es la siguiente:

```ruby
# lib/game_of_life.rb

  def neighbours(cell)
    delta = [
      [-1, 1], [ 0, 1], [ 1, 1],
      [-1, 0],          [ 1, 0],
      [-1,-1], [ 0,-1], [ 1,-1]
    ]

    neighbours = []
    delta.each do |x,y|
        neighbours << [x + cell[0], y + cell[1]]
    end

    neighbours
  end
```

Uno de los pasos vitales en el desarrollo orientado a pruebas, es que una vez que la suite este en *verde,* se debe refactorizar el código generado, lo que quiere decir que debe mejorarse para su mantenimiento y para que lectores futuros no tengan dificultad para acceder a él. El método *neighbours* parece un excelente candidato para

ello, ya que su lógica es confusa, y la inclusión de una variable temporal a modo de *carrier* tampoco ayuda a dar claridad. Quizás una mejor implementación, sería como sigue:

```
# lib/game_of_life.rb

def neighbours(cell)

    x, y = cell

    delta = [
      [-1, 1], [ 0, 1], [ 1, 1],
      [-1, 0],          [ 1, 0],
      [-1,-1], [ 0,-1], [ 1,-1]
    ]

    delta.map { |dx, dy| [x + dx, y + dy] }

 end
```

De ésta forma se explicita la naturaleza cartesiana de las coordenadas de la célula y se evita la creación de variables temporales de agregación. Otro problema de diseño que se puede considerar, es que el método no depende de la instancia, por tanto sería erróneo definirlo dentro de ella. Sin embargo se dejará dentro de la clase *GameOfLife* por fines prácticos.

Con *neighbours* funcionando de la manera esperada, se puede definir un método que entregue aquellas células adyacentes a de una célula dada para aplicar a la regla de sobrevivencia sobre ella.

```
# test/game_of_life_test.rb

  def test_alive_neighbours_with_two_alive
    game = GameOfLife.new([
      [0,0],
      [1,0],
      [1,1]
      ])
    assert_equal([[1,0],[1,1]], game.alive_neighbours([0,0]))
  end
```

El test indica que dada la semilla con las células [0, 0], [1, 0] y [1, 1], las células vivas adyacentes al origen serán [1, 0] y [1, 1]. Gracias al método *neighbours*, la implementación de esto es trivial, pues sólo debe retornar la intersección entre los vecinos de la célula y las vivas dentro del sistema.

```
# lib/game_of_life.rb

  def alive_neighbours(cell)
    neighbours(cell) & @alive_cells
  end
```

Si se vuelve a analizar el patrón *block (*Ilustración 7*)*, es posible notar que la única regla que se aplica en ese caso es la aquella que calcula las células sobrevivientes y que acaba de ser implementada. En éste momento sí es interesante el caso, pues ayudará a implementar la función *next*, la cual calcula la siguiente iteración del sistema.

```
# test/game_of_life_test.rb

  def test_survivors

    # .**.
    # .**.

    seed = [
      [0,0],
      [0,1],
      [1,0],
      [1,1]
    ]

    game = GameOfLife.new(seed)

    assert_equal(seed, game.next)
    assert_equal(seed, game.next)
  end
```

Para lograr que éste test pase, se debe aplicar la regla de sobrevivientes sobre cada una de las células del sistema.

```ruby
# lib/game_of_life.rb

def next

  survivors = []

  @alive_cells.each do |cell|
    if alive_neighbours(cell).count.between?(2,3)
      survivors << cell
    end
  end

  survivors
end
```

Como cada vez que la suite de pruebas vuelve a estar en verde, se refactoriza. A primera vista se ve que la función no expresa claramente su intención, además de que explícitamente mantiene una variable *carrier* de los sobrevivientes. Quizás una implementación más concisa y clara puede ser como sigue:

```ruby
# lib/game_of_life.rb

def next
  @alive_cells.select { |cell| alive_neighbours(cell).count.between?(2,3) }
end
```

Al ejecutar nuevamente los test, se verifica que el comportamiento es correcto. Un análisis simple de lo que se acaba de hacer da cuenta de algo muy importante: Las tres primeras reglas ya están contenidas en ésta implementación del juego de la vida. Por lo tanto, la única que queda por considerar es cuando una célula muerta “nace” debido a reproducción.

Lo anterior hace dudar de la decisión inicial de diseño de sólo almacenar las células vivas, pues ahora es necesario iterar sobre células muertas y verificar si tienen 3 vecinas vivas. Sin embargo, para ésta regla sólo es necesario obtener las vecinas muertas de las almacenadas en el sistema, pues ellas son las únicas candidatas

válidas para “nacer” (tienen por lo menos 1 célula viva a su alrededor). Se escribe entonces la siguiente prueba, con el fin de implementar un método que entregue las células muertas alrededor de una posición en el plano:

```
# test/game_of_life_test.rb

  def test_dead_neighbours
    seed = [
      [-1, 0],
      [ 1, 1],
      [-1,-1]
    ]

    game = GameOfLife.new(seed)

    expected_dead_neighbours = [
      [-1,  1],
      [ 0,  1],
      [ 1,  0],
      [ 0, -1],
      [ 1, -1],
    ]

    assert_equal(expected_dead_neighbours, game.dead_neighbours([0,0]))

  end
```

Dadas las herramientas construidas en los pasos anteriores, ésta implementación resulta trivial, pues basta con tomar el conjunto de las vecinas de la célula de entrada y restarlo con el conjunto de células vivas.

```
  # lib/game_of_life.rb

  def dead_neighbours(cell)
    neighbours(cell) - @alive_cells
  end
```

A partir de lo anterior, se puede pensar en crear un método que devuelva todas las células que deben nacer en la siguiente iteración.

```ruby
# test/game_of_life_test.rb

  def test_births

    # .*
    # **

    game = GameOfLife.new([
      [0,0],
      [1,0],
      [1,1]
      ])

    assert_equal([[0,1]], game.births)
  end
```

Para llevar la prueba a *green*, el algoritmo debe iterar sobre las células muertas y entregar aquellas que poseen exactamente 3 vecinas vivas. Esto se puede hacer fácilmente utilizando la función *alive_neighbours*, implementada con anterioridad.

```ruby
  # lib/game_of_life.rb

  def births
    @alive_cells.flat_map do |cell|
      dead_neighbours(cell).select do |neighbour|
        alive_neighbours(neighbour).count == 3
      end
    end.uniq
  end
```

Han sido implementadas todas las reglas del juego de la vida. Es el momento de enfrentarse al test planteado al inicio, al patrón *blinker:*

```ruby
# test/game_of_life_test.rb
...
def test_blinker

  step_one = [
    [1,0],
    [1,1],
    [1,2]
  ]

  step_two = [
    [0,1],
    [1,1],
```

```
    [2,1]
  ]

  game = GameOfLife.new(step_one)

  assert_equal(step_two.sort, game.next.sort)
  assert_equal(step_one.sort, game.next.sort)
end
```

El test obviamente falla, pues la implementación actual de *next* sólo considera los sobrevivientes. Debiese considerar la suma del conjunto de los sobrevivientes con las células nacidas en la iteración. Para ello, primero se comentará la prueba y se refactorizará el código, con el fin de mover la responsabilidad del cálculo de los sobrevivientes a un método aparte, logrando lo siguiente:

```
  # lib/game_of_life.rb

  def next
    survivors
  end

  def survivors
    @alive_cells.select { |cell| alive_neighbours(cell).count.between?(2,3) }
  end
```

Comprobado que la suite de pruebas siga en *green*, se descomenta nuevamente el test para *blinker*, y la implementación de *next* resulta trivial:

```
  # lib/game_of_life.rb

  def next
    @alive_cells = survivors + births
  end
```

Finalmente, con ésta implementación, se da fin a la kata. El código de la implementación está en los anexos.

Hay aprendizajes importantes en éste ejercicio. Uno de ellos, es que en ocasiones, es interesante considerar un ejemplo complejo, que si bien no va a ser implementado inmediatamente, sirve para mantener el foco en la solución global del problema y a su vez puede ser utilizado para definir un punto en el cual el código es capaz de ejecutar de forma esperada y correcta. Otro punto importante, es que nuevamente es evidente la importancia del paso de refactorizar, pues no basta tener el código con una buena suite de pruebas, sino que para considerar que un componente de software posee calidad, debe también ser mantenible, es decir, debe expresar claramente su propósito a aquellos que en un futuro quisieran modificarlo o extenderlo. Éste punto es particularmente importante en los tests, pues éstos deben ser los más claros posibles, pues son ellos los que definen el comportamiento de cada componente.

Análisis

Considerando lo mencionado en el capítulo *Metodología para el estudio comparativo*, se puede analizar la experiencia del desarrollo orientado a pruebas clásico utilizando los siguientes criterios:

Madurez de herramientas

Dada la popularidad obtenida por la suite de pruebas automatizadas jUnit, nació a partir de ella una familia de frameworks de testing denominada xUnit, las que abarcan gran cantidad de herramientas en distintos lenguajes de programación. Siendo la "escuela" inicial de pruebas automatizadas, es la que presenta herramientas más maduras y simples de usar.

Dependencia de un diseño a priori

El desarrollo guiado por pruebas no impone un proceso de diseño ni análisis. Se espera que el diseño emerja desde los test, pues la creación de pruebas modulares ayuda a entender las particularidades del sistema y a visualizar los patrones o puntos comunes del mismo. Una vez visualizados, se espera que resulte mucho más fácil generar diseños o arquitecturas. En general, el desarrollo guiado por pruebas adhiere a la idea de que el diseño de sistemas complejos de Software no es un proceso racional, postulado por David L. Parnas (21), y que el mejor momento para generar diagramas y documentación es cuando el sistema ya está construido.

Sin perjuicio de lo anterior, se acepta crear diseños o arquitecturas a modo de *guía* durante el desarrollo de software, pero no como un *requerimiento*.

Fragilidad de los tests

A partir de las experiencias anteriores y la bibliografía, se puede concluir que la fragilidad de las pruebas está directamente relacionada con lo que éstas intentan mostrar. A medida que se acercan a la implementación se vuelven frágiles, mientras que al acercarse al comportamiento del SUT, se vuelven más robustas. Es por ello que en general, se recomienda implementar aquellos test que describan el comportamiento del sistema, evitando, por ejemplo, probar funciones privadas, las cuáles comúnmente guardan relación con el *"como"* y no con el "*que"*.

En el caso particular del desarrollo guiado por pruebas clásico, el flujo green-red-refactor con la visión de "abajo hacia arriba" procura pensar en el comportamiento del sistema paso a paso, y una vez que el comportamiento es el correcto, en el ciclo de *refactor* se crearán todas aquellas funciones, módulos o clases que ayuden a hacer el

código más mantenible, siendo éstas no necesariamente testeadas, pues son detalles de implementación que no modifican el comportamiento inicialmente esperado.

Valor directo al cliente

El desarrollo guiado por pruebas es una metodología de desarrollo, y no genera valor explícitamente al cliente. Si lo hace implícitamente, reduciendo costos de mantenimiento y el tiempo que le toma al equipo de desarrollo implementar nuevas funcionalidades o bien adaptarse a los cambios constantes en los requerimientos del cliente o los del negocio. Adicionalmente, ayuda a los desarrolladores a tener confianza en su propio trabajo y en de sus pares.

Aun cuando lo anterior es positivo para el equipo de desarrollo, resulta difícil transmitir el beneficio a un cliente no técnico, ya que no necesariamente pertenece a su campo de experticia, ni tampoco le puede interesar el “cómo” se enfrenta el problema, sino que se enfoca principalmente que la solución satisfaga sus criterios de aceptación.

Curva de aprendizaje

Se puede analizar la dificultad del proceso de aprendizaje, considerando principalmente la forma en que un desarrollador de Software suele realizar su trabajo. El proceso del desarrollo de pruebas es simple a primera vista, pero detrás de él yace una forma diametralmente distinta de enfrentarse a un problema.

Cuando se enfrenta un nuevo desafío, la metodología indica que el enfoque debe centrarse en qué es lo que se intenta solucionar. Es importante entonces entender cuál es el comportamiento que debe tener el SUT, pues éste se verá reflejado en la prueba que será escrita. La prueba no refleja la implementación ni la arquitectura, refleja el comportamiento.

Lo anterior, suele ser un problema en aquellos ambientes de desarrollo en los cuales los análisis de arquitectura se crean antes de la etapa de implementación y no durante la misma, debido a que no es posible preguntarse la razón de la existencia de cada componente, pues al hacerlo se enfrenta una decisión ya aprobada, enmarcada en un diseño más grande.

Sin embargo, en aquellos ambientes en los cuales se apliquen metodologías ágiles de desarrollo de Software, se considere el diseño emergente como una solución válida y donde los decisiones de arquitectura se tomen junto con la implementación, la curva de aprendizaje no debería ser tan elevada, pues la única dificultad que se enfrenta son las preconcepciones acerca del desarrollo que se puedan tener desde la academia. En ésta se suelen priorizar entregas cortas enfocadas en resultados a corto plazo y no en calidad, ni facilidad de mantención, por lo que puede resultar difícil mostrar la importancia de pruebas automatizadas cuando no se ha enfrentado una base de código difícil de mantener y de un tamaño importante, o bien soluciones poco prolijas con diseños deficientes. Aun así, éste tipo de problemas y trabas son generadas por el desarrollador y no por la estructura en la cual se desenvuelve.

Costo de mantención

El costo de mantención de una solución creada usando el desarrollo de guiado por pruebas clásico está directamente relacionado con la fragilidad de las pruebas implementadas.

Independiente de la calidad de los test, una buena suite de pruebas con un porcentaje de cobertura elevado, ayuda a que el sistema sea capaz de adaptarse a nuevas condiciones de negocio (o bien solucionar nuevos problemas) sin introducir

errores o conductas indeseadas. Esto debido a que cada prueba verifica un comportamiento del sistema y cada vez que éste se modifica, si alguna de ellas falla, se simplifica bastante el encontrar la causa y el método, clase o módulo que la está produciendo.

Adicionalmente las pruebas funcionan a modo de documentación, pues ellas muestran cada acción ejercida sobre el sistema, dado cierto contexto y entrada, y la salida esperada para ella. La ventaja de éste tipo de documentación por sobre la clásica es que ésta siempre está actualizada y puede ser ejecutada por quien desee entender el sistema.

Todo lo anterior, ayuda en gran manera a disminuir los costos de mantención del Software, reduciendo los tiempos de corrección de errores y disminuyendo el tiempo de implementación de nuevas características.

Behaviour driven development (BDD)

El desarrollo guiado por comportamiento, se distancia de la práctica anterior en dos aspectos importantes. Uno es las palabras utilizadas y el lenguaje en que se escriben las pruebas, tratando de eliminar la mayor cantidad de términos técnicos como *assert* o *testCase*, por lenguaje natural como *should* o *describe.* La plantilla original para la creación de pruebas (19) es como sigue:

- **Dado** un contexto inicial
- **Cuando** ocurre un evento
- **Entonces** nos aseguramos que la salida es lo que se esperaba

Lo anterior busca no sólo dar claridad a los test, sino reforzar la idea y la concepción de ellas como documentación. Existen frameworks que en lugar de seguir las convenciones xUnit para testing, utilizan éste tipo de patrones, en los cuales se refuerza la expresividad de las pruebas (como Rspec).

Lo segundo, es la integración muchísimo más cercana con el cliente, proponiendo incluso que sea él el que establezca las pruebas que debería satisfacer la solución, denominadas pruebas de aceptación. Dado que en general, el cliente no posee los conocimientos técnicos para escribirlas en el sentido tradicional, se escriben pruebas de aceptación en lenguaje natural, para luego ser implementadas por el equipo de desarrollo. Estos test en lenguaje natural se almacenan dentro del código fuente y deben poder ser ejecutados automáticamente.

Finalmente, pone el énfasis en siempre probar comportamientos de módulos y nunca su implementación. Es decir, no es necesario saber el “Cómo” de la

implementación, sino que lo que realmente es probado (y lo que normalmente tiene más valor para el usuario final) es el “Qué”, el comportamiento del sistema.

En resumen, los dos aspectos anteriores se dividen en pruebas para el comportamiento del sistema como un todo, escritas por el usuario en lenguaje natural y un segundo acercamiento más a nivel de pruebas unitarias tipo xUnit, teniendo el mismo enfoque que éstas pero cambiando el lenguaje de los conceptos.

Rspec

Rspec es un framework de testing para Ruby, diseñado con BDD en mente. Esto quiere decir que pone énfasis en el lenguaje en que se escriben las pruebas, con el fin de que al ser ejecutadas, generen documentación fácil de comprender o bien a modo de especificaciones.

Este lenguaje se enfoca en el comportamiento del SUT (s*ubject under test*). Utilizando palabras como *describe* e *it*, se pueden expresar los casos de prueba como oraciones en idioma natural:

```
"Describe an order."
"It sums the prices of its line items."
```

Para éste caso, se desea comprobar que dada un Orden, su valor es la suma de los ítems que contiene. Lo anterior puede ser expresado en Rspec como sigue:

```
describe Order do

  it "sums the prices of its line items" do

    order = Order.new
    order.add_entry(LineItem.new(item: Item.new(price: Money.new(1.11, :USD))))
    order.add_entry(LineItem.new(item: Item.new(price: Money.new(2.22, :USD),
                                                quantity: 2)))
```

```
    expect(order.total).to eq(Money.new(5.55, :USD))
  end
end
```

El método *describe* define un *ExampleGroup*, el cual genera un *scope* dentro del cual se escribe las pruebas utilizando el método *it*. En este caso, se crea una nueva orden, se agregan 3 elementos a ella, y luego se verifica que el total de la orden sea el mismo que la suma de todos sus artículos (22).

Cucumber

Cucumber es una herramienta para BDD, que ejecuta casos de prueba escritos en texto plano, generalmente representando historias de usuario. Debido a su naturaleza no técnica, pueden ser escritos por expertos en el dominio de la aplicación, analistas de negocio o cualquier otra parte interesada. De ésta forma, se fomenta que los test sean escritos *outside-in*, es decir, de lo más general a lo más específico (23).

A modo de ejemplo, considerar el siguiente test:

```
Feature: Search courses
  Courses should be searchable by topic
  Search results should provide the course code

  Scenario: Search by topic
    Given there are 240 courses which do not have the topic "biology"
    And there are 2 courses A001, B205 that each have "biology" as one of the topics
    When I search for "biology"
    Then I should see the following courses:
      | Course code |
      | A001        |
      | B205        |
```

La tienda de videojuegos

Una tienda de video juegos independiente, desea crear un sistema de venta y manejo de juegos en línea. Para ellos desea crear un sitio web.

Implementación

Uno de los pilares fundamentales de BDD es la comunicación, por lo que lo primero es preguntar al cliente, cuales son las funcionalidades que necesita y a partir de ello, crear historias de usuario asociadas, las cuales podrán ser implementadas dentro del proyecto utilizando lenguaje natural.

Una funcionalidad básica del sistema es la capacidad de gestionar los juegos, para que posteriormente sean presentados en una lista. Esto puede ser escrito en una historia de usuario:

- **Como** administrador de la tienda
- **Deseo** manejar la lista de juegos
- **Para** mostrar mis productos en línea

A partir de ella, se pueden definir distintos criterios de aceptación, como agregar, editar, eliminar y listar juegos. Uno de ellos puede ser:

- **Dado** que estoy en la vista de agregar juego
- **Cuando** lleno su descripción y lo creo
- **Entonces** puedo ir a la lista de juegos y ver el recién ingresado

Terminada las definiciones junto con el cliente, es necesario implementarlas en código, de ésta forma se mantienen como pruebas ejecutables que deben estar en *green*. Para ello, se utilizará Cucumber (24). Adicionalmente, la solución será escrita en Ruby on Rails y usando Rspec como framework de pruebas. Mongodb se utilizará para la persistencia de datos.

En Cucumber, el criterio de aceptación puede ser escrito como sigue:

```
# user_manages_games.feature

Feature: User manages games

Scenario: User adds a game to the list
  Given A new game with name The Witcher 3 and description Open world action RPG
  When I go to the add game page
  And I fill in the game's name
  And I fill in the game's description
  When I press 'Create'
  Then I'm redirected to games list page
  And I see The Witcher 3
```

Lo anterior es un *feature* y cada una de las oraciones que empiezan con una preposición o un adverbio anglosajón (*Given*, *When*, *Then*, *And*, *But*) se denomina *steps* o *pasos* y deben ser definidas para que Cucumber pueda ejecutarlas.

El primero de los steps, se define como sigue, utilizando expresiones regulares para atrapar las variables):

```
# add_game_to_store_steps.rb
Given(/^A new game with name (.*) and description (.*)$/) do |name, description|
  @name = name
  @description = description
end
```

Éste paso sólo consiste en crear un escenario previo, por tanto se pasa al siguiente inmediatamente:

```
# add_game_to_store_steps.rb
When(/^I go to the add game page$/) do
  visit new_game_path
end
```

Para implementar éste paso es necesario crear el controlador, la vista y agregar la ruta en Rails. Se usa entonces un test de Rspec para comprobar su correctitud.

```
# games_controller_spec.rb
describe GamesController do
  describe '#new' do
    it 'is successful' do
      get :new
      expect(response).to be_success
    end
  end
```

Ésta prueba verifica que al hacer un request a la ruta para crear un nuevo juego, ésta retorna un status 200 o *success*. Para que esté en verde, primero se agrega la ruta:

```
# routes.rb
VaporStore::Application.routes.draw do

  resources :games, only: [:new]
end
```

Luego, se crea la acción del controlador que corresponde

```
# games_controller.rb
class GamesController < ApplicationController
  def new
  end
end
```

Para fines del ejemplo, las vistas tendrán el contenido mínimo para aprobar los test, en este caso, se crea la vista vacía (basta con crear el archivo). Luego de eso, tanto como el test de Rspec como el de Cucumber pasan a verde. Dado lo anterior, se puede pasar al siguiente step:

```
# add_game_to_store_steps.rb

And(/^I fill in the game's name$/) do
  fill_in :name, with: @name
end

And(/^I fill in the game's description$/) do
  fill_in :description, with: @description
end
```

Por simplicidad, se agregan ambos pasos pues hace referencia a un mismo formulario. Rails provee helpers para poder implementar formularios de modelos de dominio. Lo primero que hay que hacer entonces, es crear el modelo de Game. Es posible crear pruebas para verificar la presencia de cada uno de los campos que son necesarios en el modelo, pero ya que no posee en si ningún tipo de comportamiento, se implementará directamente:

```
# game.rb
class Game
  include Mongoid::Document

  attr_accessible :name, :description

  field :name, type: String
  field :description, type: String
end
```

Ahora se puede crear el formulario en la vista de añadir juego:

```
<!-- new.html.erb -->

<%= form_for @game do |f| %>
    <%= f.label :name %>
    <%= f.text_field :name, id: 'name' %>
```

```
    <%= f.label :description %>
    <%= f.text_field :description, id: 'description' %>

    <%= f.submit 'Create' %>
<% end %>
```

Lo cual falla, pues la variable @game no se está enviando desde el controlador. Se agrega una prueba para verificar lo anterior.

```
# games_controller_spec.rb
describe GamesController do

  describe '#new' do
    it 'creates a new game object' do
      get :new
      expect(assigns(:game)).not_to be_nil
    end
  end
end
```

La implementación es simple:

```
# games_controller.rb
class GamesController < ApplicationController

  def new
    @game = Game.new
  end
end
```

Ahora tanto la prueba de Rspec como la de Cucumber están en *green*. Es momento de pasar al siguiente *step*.

```
# add_game_to_store_steps.rb
When(/^I press 'Create'$/) do
  click_button 'Create'
end
```

Lo cual entrega un error, pues la acción de crear no está implementada. Para ello, se agrega una prueba de Rspec

```
# games_controller_spec.rb

describe GamesController do

  describe '#create' do
```

```
    before { post :create, game: {name: 'Starcraft', description: 'Military science
fiction real-time strategy'} }

    it 'creates a new game' do
      expect(assigns(:game)).not_to be_nil
      expect(assigns(:game).name).to eq 'Starcraft'
    end
  end
end
```

```
# routes.rb
VaporStore::Application.routes.draw do
  resources :games, only: [:new, :create]
end
```

Con ella se verifica que existe la acción *create* y además que se instancia correctamente la variable @game con los datos entregados por la vista

```
# games_controller.rb
class GamesController < ApplicationController

  def create
    @game = Game.new(params[:game])
    @game.save
  end
end
```

Siguiendo con los steps de Cucumber:

```
# add_game_to_store_steps.rb
Then(/^I'm redirected to games list page$/) do
  expect(current_path).to eq games_path
end
```

Primero se verifica que un request a la vista del listado de juegos, es resuelto correctamente (es necesario crear una vista vacía para la acción *index*).

```
# games_controller_spec.rb
describe GamesController do
  describe '#index' do
    it 'is successful' do
      get :index
      expect(response).to be_success
    end
  end
end
```

```ruby
# games_controller.rb
class GamesController < ApplicationController
  def index
  end
end
```

```ruby
# routes.rb
VaporStore::Application.routes.draw do
  resources :games, only: [:new, :create, :index]
end
```

Adicionalmente, es necesario modificar la implementación de *create*, para que re direccione a la lista de juegos si es que guarda correctamente.

```ruby
# games_controller.rb
class GamesController < ApplicationController
  def create
    @game = Game.new(params[:game])
    redirect_to games_path if @game.save
  end
end
```

Finalmente, queda por implementar el último step del *feature*.

```ruby
# add_game_to_store_steps.rb
And(/^I see The Witcher 3$/) do
  expect(page).to have_content 'The Witcher 3'
end
```

Para ello definimos un nuevo test de Rspec que verifique que los juegos se envían correctamente a la vista

```ruby
# games_controller_spec.rb
describe GamesController do

  describe '#index' do
    context 'there is at least a game in the store' do
      before { Game.create(name: 'Deus Ex: Human Revolution', description:
'Cyberpunk-themed action role-playing video game') }

      it 'sends games to the view' do
        get :index
        expect(assigns(:games)).not_to be_nil
        expect(assigns(:games).size).to be 1
      end
```

```
    end
  end
end
```

Para ello se envía la información correctamente en el controlador

```
# games_controller.rb
class GamesController < ApplicationController
  def index
    @games = Game.all
  end
end
```

Y luego se despliega la información correctamente en la vista, una versión simplificada que pasa el test es:

```
<!-- index.html.erb -->
<div>
  <ul>
    <% @games.each do |game| %>
        <li>
            <h3>
              <%= game.name %>
            </h3>
            <p>
              <%= game.description %>
            </p>
        </li>
    <% end  %>
  </ul>
</div>
```

Con toda la implementación anterior, se puede ejecutar el criterio de aceptación escrito con Cucumber y éste pasará correctamente.

Otro escenario interesante, es cuando se intenta agregar un juego en el sistema, pero ya existe uno con el mismo nombre. Lo anterior se define como:

```
# user_manages_games.feature

Feature: User manages games
  Given This games exists:
    | name          | description |
    | The Witcher 3 | Open world action RPG |
  When I go to the add game page
  And I fill in name with The Witcher 3
  And I fill in description with Open world action RPG
```

```
  When I press 'Create'
  Then I see and error message
  When I go to the games list page
  Then The game is listed only once
```

El primer paso involucra simplemente crear el juego en el sistema.

```
# add_game_to_store_steps.rb
Given(/^This games exists:$/) do |table|
  Game.create table.hashes
end
```

Aquellos pasos que fueron implementados en el escenario anterior se pueden reutilizar, por tanto hay que implementar:

```
# add_game_to_store_steps.rb
And(/^I fill in (.*) with (.*)$/) do |field, value|
  fill_in field, with: value
end
```

Éste pasa a green inmediatamente, debido a lo escrito en el *feature* anterior. Continuando, es necesario testear que se muestra un mensaje de error cuando el juego que se intenta crear ya existe, por tanto el *step* se puede definir como:

```
# add_game_to_store_steps.rb
Then(/^I see and error message$/) do
  expect(page).to have_selector '.alert', text: 'The game already exists'
end
```

Para implementarlo, se debe prevenir la creación del objeto en la base de datos y adicionalmente, mostrar el mensaje de alerta en pantalla. Para lo primero, es necesario agregar una validación al modelo Game para que el nombre sea único. Se agrega entonces, el test que lo verifica.

```
describe Game do
  describe 'name' do

    let(:game) {{ name: 'Grand Theft Auto IV',
                  description: 'Open world action-adventure video game' }}

    before { Game.create(game) }
```

```
    it 'has to be unique' do
      expect{Game.create(game)}.not_to change(Game, :count)
    end
  end
end
```

Ahora se agrega la validación al modelo que hace pasar este test

```
# game.rb
class Game
  include Mongoid::Document

  attr_accessible :name, :description

  field :name, type: String
  field :description, type: String

  validates_uniqueness_of :name
end
```

Es necesario que cuando el método save falle, se regrese a la vista de creación y muestre el mensaje de error. En Rails, esto resulta simple, pues cuenta con el helper flash para realizar esta tarea.

```
# games_controller.rb
class GamesController < ApplicationController
  def create
    @game = Game.new(params[:game])
    if @game.save
      redirect_to games_path
    else
      flash[:alert] = 'The game already exists'
      redirect_to new_game_path
    end
  end
end
```

Ahora se agrega el mensaje al layout de la página

```
<!-- layouts/application.html.erb -->
<body>
<% flash.each do |name, msg| %>
    <%= content_tag :div, msg, class: name %>
<% end %>

<%= yield %>

</body>
```

Con lo anterior, el step de Cucumber pasa a verde. El siguiente paso es trivial

```
# add_game_to_store_steps.rb
When(/^I go to the games list page$/) do
  visit games_path
end
```

Finalmente, es necesario verificar que sólo existe un juego, por tanto la prueba consiste en que aparecerá sólo una vez en el listado. La implementación de esa lógica en el dominio de la aplicación ya existe, pues no se pueden crear dos objetos con el mismo nombre, por lo tanto basta con verificarlo solo a nivel de interfaz

```
Then(/^The game is listed only once$/) do
  expect(page).to have_content 'The Witcher 3', count: 1
end
```

Lo cual devuelve la suite de test nuevamente a *green*. Con ello se termina de implementar ambos criterios de aceptación utilizando BDD.

Análisis

Madurez de las herramientas

Aun cuando BDD es una metodología reciente, existen frameworks como Cucumber en la mayoría de los lenguajes de programación más utilizados, como Specflow (25) para .net, JBehave (26) para Java o Lettuce para Python, entre otros.

En el caso de Rspec, éste tipo de herramientas de BDD a nivel más técnico, no son tan populares como Cucumber, y en general, resulta un poco más difícil encontrar su símil en otros lenguajes.

Dependencia de un diseño a priori

Si bien no es necesario tener una arquitectura definida al momento de implementar, si se requiere tener una idea o visión acerca de los componentes centrales del sistema, pues el cliente debe tener una idea de cómo interactuar con el sistema para construir los criterios de aceptación. Una vez que los criterios de aceptación estén escritos, la manera en que se ejecutan es independiente del cliente y es la responsabilidad del equipo de desarrollo el implementar cada una de las funcionalidades, módulo o componentes necesarios como crean conveniente o utilizando las tecnologías que consideren apropiadas.

Fragilidad de los test

Una de las críticas comunes a BDD, es la fragilidad de los test. Esto debido a que dado un criterio de aceptación, éste asume que la interfaz o los objetos del sistema tienen una cierta estructura, la cual al iniciar el desarrollo, no es clara ni definitiva. En el ejemplo, si cambia la forma de ingresar un juego, o bien se desea agregar información adicional, el criterio de aceptación debe actualizarse y por tanto el cliente debe describir la nueva forma de ingresar un juego.

Lo anterior lleva a una pregunta interesante ¿Qué es lo que está siendo definido por el criterio de aceptación? ¿Es la forma de ingresarlo, o la acción de hacerlo? Si se quiere probar que la creación de juegos es válida, entonces detalles acerca de los campos y estructura del objeto juego son demasiado específicos, lo que claramente provocará que sea necesario cambiar esas pruebas si lo anterior cambia. Los criterios de aceptación son a nivel de negocio y de usuario, nunca a nivel de implementación,

por lo que deben cambiar cuando las reglas del negocio cambien y no cuando la arquitectura del sistema lo haga.

Si se sigue la regla anterior, no deberían presentar problemas de fragilidad importantes.

Valor directo al cliente

Uno de los puntos fuertes de la metodología, es el valor que le entrega al cliente. Incluso, nace de la idea de transparentar la parte del problema que se está solucionando con cada funcionalidad.

Comúnmente en los equipos de desarrollo, existe el problema de ser capaz de entender el negocio en el cual la solución va a existir y cómo esas reglas de negocio afectan el desarrollo y las herramientas utilizadas en él. Además, suele ser complejo familiarizarse con las necesidades (performance, usabilidad, manejo de datos) específicas para un cliente. BDD intenta solucionar éste problema explicitando junto al cliente cuáles son las funcionalidades que, a su parecer, deben estar en la solución, y además, cuales son los criterios que definen una implementación satisfactoria de ellas. De ésta forma el cliente sabe qué esperar del equipo y a su vez el equipo se alineará con el objetivo de negocio más fácilmente, al tener metas concretas y explícitas.

Curva de aprendizaje

En el caso de BDD, como en todo proceso de aprendizaje, qué tan difícil sea adoptar la metodología dependerá de la experiencia de los desarrolladores y que tan simple les sea integrar el flujo green-red-refactor dentro del proceso de trabajo. Sin embargo, BDD tiene la particularidad que involucra actores del negocio dentro del

proceso de desarrollo. Esto puede ocasionar problemas, pues en *waterfall*, el cliente es parte del inicio del ciclo (análisis) y no en la implementación.

Una de las partes importantes de la metodología que es necesario aprender, es aislar los requerimientos funcionales y no mezclarnos con los técnicos. En ocasiones, el no lograr niveles importantes de abstracción, puede provocar que las pruebas, al crecer en complejidad y tamaño, se comporten como una carga y no como un apoyo a la gestión y desarrollo. Por ejemplo, si no se logra capturar la esencia del negocio con herramientas tradicionales como Cucumber o Rspec, es interesante explorar otras opciones, como lenguajes específicos de dominio (DSL), con el fin de mantener las pruebas en el mundo al que pertenecen y no mezclar lo técnico con el negocio. Sin embargo, el beneficio viene con el costo de implementar tales DSLs, los cuales sin duda dificultarían el acceso a la metodología.

Adicionalmente, la creación de las pruebas junto con el cliente y la búsqueda de un lenguaje ubicuo requieren de un componente de comunicación muy importante. Éste componente se basa en la confianza (explicitar problemas o restricciones del equipo de forma transparente), el debate (expresar ideas y argumentos), involucrarse con el problema a solucionar y la capacidad de abrir el proceso de desarrollo al cliente (mostrando el avance del mismo paso a paso).

Costo de mantención

Uno de los mayores problemas es el costo de ejecución y mantención de las pruebas de más alto nivel. Por un lado, las pruebas del tipo Cucumber, en las cuales el sistema se analiza desde una perspectiva del usuario, tienden a tener una baja velocidad de ejecución, pues deben cargar todo el ambiente de la aplicación antes de

crear el *fixture* para cada caso de prueba. Cuando las pruebas alcanzan una cantidad considerable, esto se vuelve un problema, ya que un desarrollador no puede esperar minutos para saber si los criterios de aceptación de cumplen. Existen soluciones para ésta clase de problemas, como por ejemplo, ejecutar las pruebas de Cucumber en un servidor de integración continua, de tal modo que cada desarrollador se preocupa de ejecutar test más pequeños y específicos (como los de Rspec) y el servidor de integración le indicará si el criterio ha sido aprobado. Lo anterior adiciona costos de mantenimiento al proceso de desarrollo que dependerán del tamaño de la solución y de la suite de test.

Adicionalmente, uno de los problemas de mantener éste tipo de soluciones es el tiempo. Con regularidad, aquellos que se espera ayuden en la creación de criterios no poseen el tiempo para crearlas. Las reuniones en que se definen suelen ser extensas, lo cual es un error común cuando se inicia en la metodología, pues no se controlan los tiempos ni el flujo de la conversación adecuadamente. Es necesario entender que el definir que lo que el usuario *realmente* quiere, es uno de los desafíos más importantes y difíciles del desarrollo de software y no podrá ser realizado en una reunión maratónica. El nivel de participación del cliente con el proyecto no solo debe ser alto, sino también constante, y los criterios de aceptación deben ser definidos y revisados de forma periódica, lo cual conlleva un gasto en tiempo que tanto el equipo de desarrollo como el cliente deben saber afrontar.

Comparación de las metodologías

En ésta sección se usarán los criterios de comparación definidos en *Descripción de la solución,* con el fin de entender las ventajas y desventajas de cada una de las metodologías orientadas a la calidad presentadas. Posteriormente, se explicará los problemas encontrados con éste enfoque y la manera en que se enfrentaron.

Comparación inicialmente propuesta

Madurez de las herramientas

Si bien, en la experiencia del autor de este Trabajo de Título, gran parte de la industria chilena aún carece de la necesidad o la motivación de implementar éste tipo de metodologías, el desarrollo guiado por pruebas ha recibido un apoyo en los círculos ligados a las metodologías ágiles en el resto del mundo.

Es por esto, que en el caso de la madurez de las herramientas, aún cuando BDD es un concepto medianamente nuevo (fue presentado en el 2009), ya existe una gran cantidad de herramientas disponibles para implementar cada una de ellas y su uso está bien documentado. Lamentablemente, la cantidad de este tipo de material en español es sumamente escasa, por lo que si no se es capaz de entender a lo menos documentos técnicos en inglés, puede presentar un problema.

Sin embargo, dado que en la disciplina de Ciencias de la Computación, la mayoría del material técnico ya se encuentra en éste idioma, este escenario es poco probable. Aún así la creación de material en un lenguaje nativo puede ayudar significativamente a la adopción de estas metodologías.

Dependencia de un diseño a priori

Ambas metodologías nacen en un contexto de agilidad, por lo que por definición, tratan de no seguir un diseño o una arquitectura de software rígida. Por el contrario, intentan definirla con la mayor cantidad de información posible, construyéndola paso a paso. El desarrollo guiado por pruebas clásico sigue esta idea fielmente. En el caso de el desarrollo guiado por comportamiento, aún cuando también construye el diseño incrementalmente, necesita de algunas definiciones acerca de la solución para que el cliente sea capaz de visualizarlo y expresar de mejor manera sus necesidades.

Considerando lo anterior, es claro que si bien no requiere de un diseño a priori, para el caso del desarrollo guiado por comportamiento, realizar inicialmente algunas definiciones respecto al sistema puede ayudar al usuario a concretar mejor lo que realmente necesita, y por lo tanto a dar más información para el estado inicial a la metodología.

Fragilidad de los tests

La fragilidad de las pruebas, para todos los casos, guarda relación con la ejecución más que con la teoría. Siempre que se intente probar un detalle específico, ya sea en a nivel de implementación o de interfaz, se producirán pruebas que deben ser modificadas constantemente, lo cual resulta en un aumento en los tiempos de desarrollo. Éste en un problema común cuando se empieza a utilizar la metodología, pues el impulso inicial es probar absolutamente todo.

Indudablemente en ambos casos, mientras se aprende el método, se enfrentarán pruebas con una alta fragilidad, pero la ocurrencia de ellas debería disminuir con la práctica.

Aún así es claro que para el caso de BDD, al encontrarse en ocasiones demasiado cerca del nivel de negocio, las pruebas pueden ser más frágiles. Sin embargo, desde el punto de vista del autor, esto no muestra un problema con la metodología, sino el reflejo de que el ámbito del negocio de la solución y las necesidades que ésta debe suplir son siempre cambiantes.

Valor directo al cliente

En éste criterio es en el cual se encuentran las mayores diferencias entre ambas metodologías.

Para el caso del desarrollo guiado por pruebas clásico, el valor entregado al cliente no es directo y la metodología tampoco lo propone. Se define a sí misma como una metodología de desarrollo y es implementada por el desarrollador, dentro del equipo de trabajo. Para el cliente, *product owner*, o cualquier entidad que reciba la solución, no habrá diferencia explícita en su funcionamiento si se realizó utilizando TDD o no.

Por otra parte, el desarrollo guiado por comportamiento integra al cliente dentro del equipo de desarrollo, llegando incluso a darle la responsabilidad de redactar los criterios de aceptación bajo los cuales la solución es válida. Esto es importante, ya que no sólo se transparenta el avance entre equipo y cliente (ya que se puede saber inmediatamente cuales de los criterios ya son aprobados) sino que ayuda al equipo de desarrollo a visualizar claramente metas de mediano y largo plazo para el producto que se está construyendo. Lo anterior no sólo genera confianza entre ambas partes, sino que las ayuda a estar alineadas con un objetivo común.

Curva de aprendizaje

La dificultad de aprender las metodologías presentadas radica principalmente en entender que es un paradigma no común de desarrollo, y como todo cambio paradigmático requiere un salto de fe importante.

Sin embargo, es importante señalar que ambas metodologías, si bien no son complejas a simple vista, requieren de un período importante de práctica y por sobre todo, exploración. Adicionalmente, en el caso del desarrollo guiado por comportamiento, requieren que el interesado fortalezca habilidades que dado el enfoque tradicional de desarrollo, quizás no esté acostumbrado a utilizar con exhaustividad, como son las habilidades de comunicación.

Considerando lo anterior, el aprendizaje de TDD clásico puede considerarse como el más sencillo de los dos, aun cuando involucra una dificultad técnica importante. BDD generalmente requiere del manejo de más herramientas y el fortalecimiento de las habilidades de comunicación. Adicionalmente, uno de sus requerimientos claves es la interacción entre los distintos actores en entorno a la solución, lo cual no siempre es posible.

Costo de mantención

En ambos casos, el costo de mantención de la suite de pruebas está directamente relacionado con la calidad de los test en ellas. Si bien se encontró que los test a nivel de usuario pueden ser más frágiles que aquello a nivel de implementación, todo se reduce a que tan bien estén definidos y no se pudo comprobar una fragilidad inherente en éstos casos.

Lo que si puede presentar algún tipo de problema, es en los tiempos de las suites de pruebas. Tal como se mencionó al momento de analizar BDD, los test a nivel de usuario tienden a tomar más tiempo en ejecutarse (esto es particularmente evidente para el caso de aplicaciones web). Los tiempos de ejecución de las pruebas es un factor muy importante, puesto que si toman demasiado tiempo, dejarán de ser ejecutadas, perdiéndose el propósito de las mismas.

Debido a lo anterior, se suelen ejecutar fuera del entorno de cada desarrollador, delegando la responsabilidad a los servidores de integración, y a juicio del autor de éste Trabajo de Título, esto puede presentarse más tempranamente en un ambiente con BDD que uno con TDD (aun cuando suceda eventualmente en ambos casos).

Comparación no cualitativa

Uno de los problemas encontrados durante el desarrollo de éste Trabajo de Título, es que en el enfoque anterior de comparación los parámetros consideran resultaron altamente cualitativos, lo que si bien sirve a modo de referencia, lo subjetivo de los criterios hace que el rango de interpretación sea demasiado amplio.

Aun cuando es un ejercicio interesante y una guía, no existe un análisis profundo acerca del proceso de cada metodología y de cómo, a partir de un problema, éstas generan una solución utilizando su propio flujo de trabajo.

Es por lo anterior que se realiza un segundo análisis, pero esta vez con énfasis en los procesos.

Test Driven Development clásico

Puesto que lo que se busca analizar es una metodología de trabajo, no es suficiente sólo considerar el resultado de ella. Por ello, se analizará el flujo de trabajo de cada uno de los ejemplos.

En el primer caso (*bowling kata*), se tiene la siguiente tabla (Tabla 1):

Paso	Prueba	Implementación	Refactorización
1	Crear objeto		
2	Método roll	Implementación vacía	
3	Todos los tiros fallidos	Retornar 0	
4	Botar un pino en cada tiro	Sumar un pin por cada tiro	
5	Lograr un spare	Agregar if al calculo de score que refleje la regla del spare	Mover puntaje a score
6			Agregar concepto de frame
7			Método privado is_spare
8			Método roll_spare en suite de tests
9	Conseguir strike	Agregar if al calculo de score que refleje la regla del strike	Cambiar implementacion de frames
10	Juego perfecto		

Tabla 1: Flujo de implementación de bowling kata

La tabla muestra un análisis simple acerca de los pasos realizados. Tomando como punto de partida pruebas sencillas, e implementado las funcionalidades poco a

poco. Se puede observar además que, como en todo sistema de software, una vez que se alcanza cierto nivel de complejidad realizar refactorizaciones al mismo se vuelve una necesidad. Es interesante considerar que éstas se realizaron con la seguridad que el comportamiento del sistema no se vería afectado, gracias a las pruebas automatizadas generadas en los pasos anteriores.

El segundo ejemplo sin embargo, resulta mucho más interesante. El análisis del proceso puede ser visto a continuación (Tabla 2):

Paso	Prueba	Implementación	Refactorización
1	Tablero en blanco	Crear objeto y definir variables de instancia	
2	Caso particular regla de sobrevivencia para origen	Calculo de vecinos utilizando un delta	Explicitar naturaleza cartesiana de las coordenadas
3	Regla de sobrevivencia para 2 vecinos vivos	Intersectar conjunto de vivos con los vecinos	
4	Calcular siguiente generación	Obtener sobrevivientes del sistema	Esclarecer código
5	Calcular vecinos muertos	Implementación trivial (vecinas del sistema, menos vivas)	
6	Calcular células que nacen	Iterar sobre vecinas muertas y aplicar regla de nacimiento	Crear método auxiliar survivors
7	Test blinker	Calcular nacimientos del sistema	
8		Aplicar todas las reglas en *next*	

Tabla 2: Flujo implementación game of life kata

Aún cuando es difícil reflejarlo en la tabla, la importancia de éste ejemplo está dada por el uso de *pruebas guía*. Cuando un sistema es suficientemente complejo, es necesario construir componentes modulares, los cuales se comunicarán entre si para lograr una solución satisfactoria. En éste tipo de casos, es difícil lograr mantener el foco del desarrollo y del problema que se intenta solucionar a más alto nivel.

Una solución simple para lo anterior, es la creación de pruebas guías que indiquen la dirección que se esta siguiendo. Para el caso de *game of life kata*, considerar la siguiente (Ilustración 8) figura:

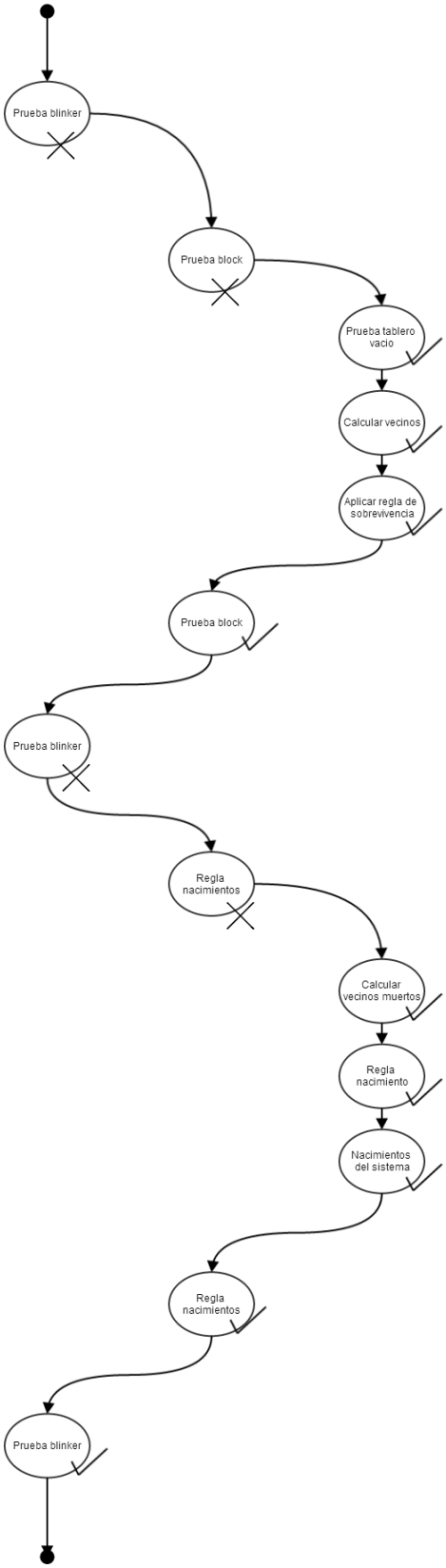

Ilustración 8: Uso de test guías en TDD clásico.

Como primer acercamiento a solucionar el problema del juego de la vida, se consideró un caso complejo, denominado *blinker (*Ilustración 6*),* el cual utiliza todas las reglas que deben ser implementadas. Sin embargo, fue claro que este ejemplo es demasiado complejo para enfrentarlo inmediatamente, por lo que en su lugar, se utilizó un caso que considerará sólo la regla de sobrevivencia, denominado *block* (Ilustración 7). Ése caso también puede ser complejo enfrentarlo directamente, pero ésta vez los pasos para que el sistema cumpliera con el comportamiento esperado resultaban mucho más claros. Es por ello que luego de implementar algunos métodos auxiliares, el test guía paso a *green*.

Luego del avance anterior, se probó nuevamente el patrón blinker, sin embargo, aún no se tenía un camino claro para ése comportamiento. Por ello, el nuevo test guía fue la implementación de la regla de nacimientos. Creadas las herramientas necesarias, este test también paso a verde. Dado que ésta era la última regla del sistema y utilizando todos los artefactos creados, se consideró nuevamente el patrón blinker y al pasar, el ejercicio se dio por terminado.

Se puede abstraer, a partir de ambos ejemplos, el siguiente diagrama para el flujo de trabajo de TDD (Ilustración 9):

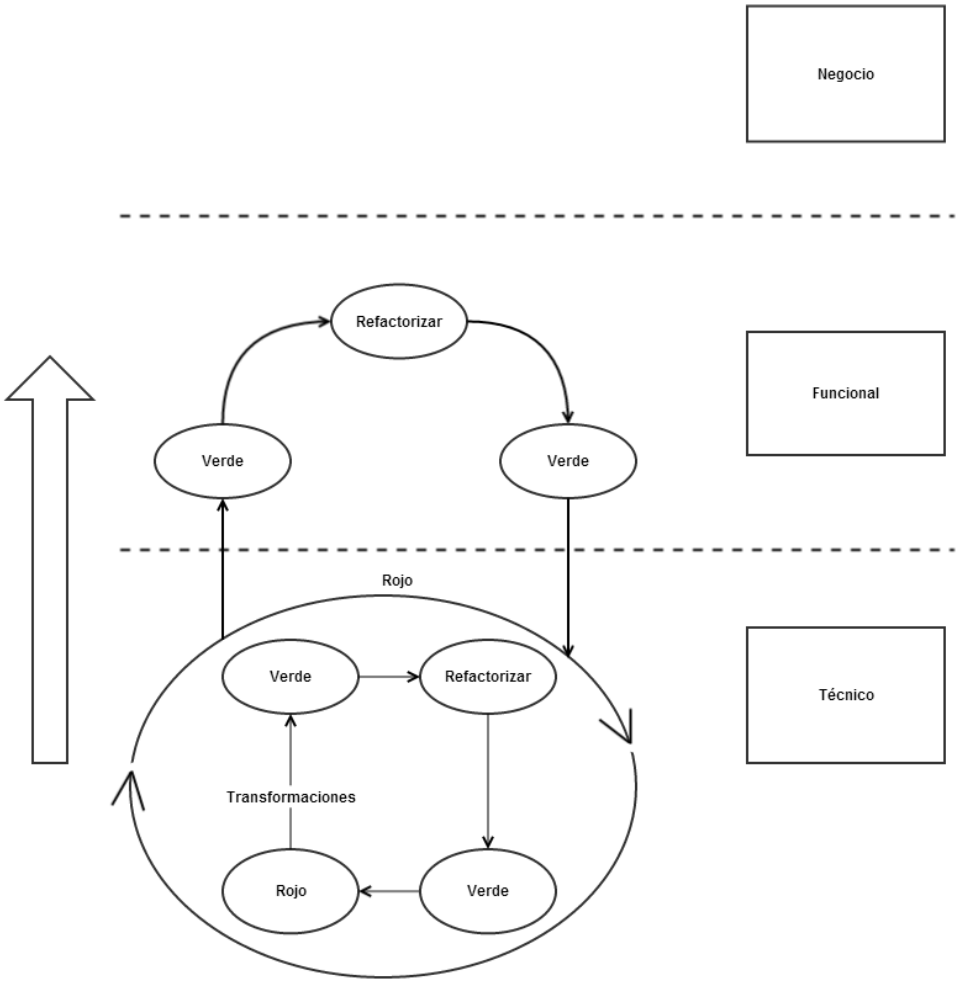

Ilustración 9: Flujo de desarrollo de TDD

Behaviour Driven Development

La metodología de Behaviour Driven Development hace énfasis la transparencia desde el equipo de desarrollo hacia el cliente. Por ello el primer paso, antes de la implementación o el diseño del sistema es escribir un criterio de aceptación que ejemplifique que es lo que realmente se espera de él. En el caso del ejemplo, lo primero fue definir que debería pasar cuando se agrega un juego al sistema, verificando de forma básica su correctitud. Esto se escribe en Cucumber, ya que permite expresarlo en

lenguaje natural, el cual puede ser entendido y escrito por el cliente. De ésta forma, queda registrado en el código del proyecto y no sólo es una guía para el desarrollo de la aplicación, sino que además es una documentación *viva*, lo cual quiere decir que es ejecutable y verificable, por tanto siempre está al día y actualizada.

Los criterios escritos con Cucumber se denominan *features,* y están compuestos de distintos *scenarios.* Cada escenario representa un flujo dentro de la aplicación, desde el punto de vista del usuario, que especifica un resultado. A su vez, cada uno de ellos se describe como una serie de *steps*.

Una vez formulados, el equipo de desarrollo implementa cada uno de los steps de forma secuencial, hasta que el criterio pase a *green*. Cada una de las funcionalidades necesarias para aprobar un test se implementan también usando TDD, pero nuevamente con un enfoque especial en el lenguaje. Si bien a éste nivel se trata de un test técnico y que no se presenta al cliente, idealmente éstos deben ser más descriptivos que aquellos basados en xUnit tradicional (por ejemplo, intercambiando palabras claves, como *assert* por *expect*). En éste caso se utilizó Rspec, una librería de testing especialmente creada para ello.

Finalmente, cuando se han escrito y aprobado todos los test que hacen posible el *step*, se pasa al siguiente, proceso que es claramente iterativo y cuyo diagrama se puede visualizar en la siguiente figura (Ilustración 10):

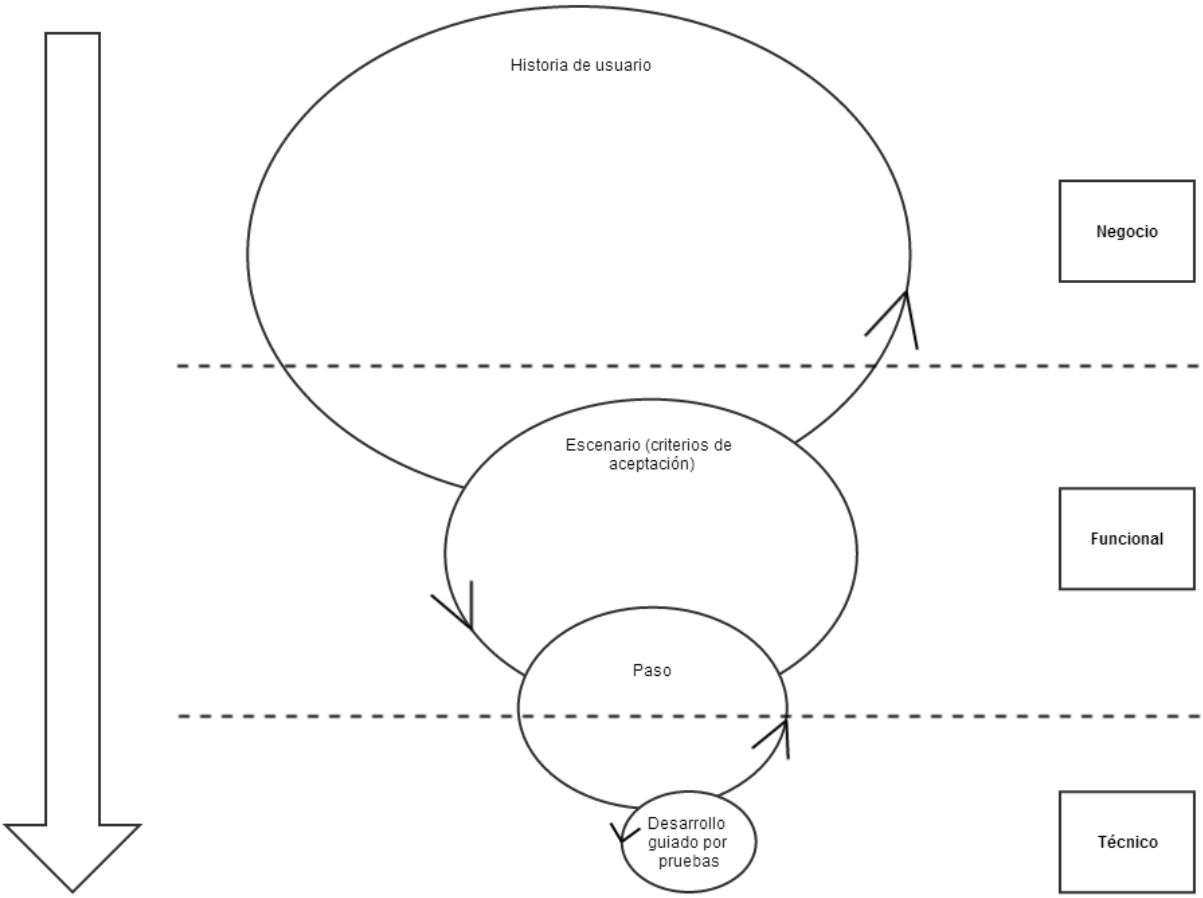

Ilustración 10: Flujo de desarrollo de BDD

Una de las ventajas que se puede apreciar inmediatamente en este flujo de trabajo, es que el desarrollo de software tiende a estar explícitamente alineado con el negocio o problema que se intenta solucionar. No se construyen componentes modulares buscando crear componentes más grandes, sino que se definen metas cortas que deben ser implementadas rápidamente, y cada una de estas metas está definida para lograr un comportamiento especificado por el cliente.

En el caso del ejercicio realizado como ejemplo, su flujo puede ser representado por la siguiente tabla (Tabla 3):

Paso	Prueba	Implementación	Refactorización
1	Navegar hacia la página de creación de juegos	Agregar ruta	
2		Crear controlador y acción asociada	
3		Agregar vista vacía	
4	Llenar campos en el formulario de ingreso	Crear modelo Game	
5		Crear formulario en la vista	Utilizar formulario de Rails
6	Apretar el boton de crear	Crear acción de creación en el controlador	
7	La aplicación redirige al home	Crear acción index y vista vacía	
8		Agregar index como ruta	
6		Agregar redirección al crear nuevo juego	
7	El juego agregado se presenta en pantalla	Enviar todos los juegos a la vista	
		Llenar la vista de index con los juegos	

Tabla 3: Flujo implementación tienda de videojuegos

Si bien la tabla solo muestra el primer escenario, es claro que para éste flujo de desarrollo, los test son mucho más orientados hacia el negocio de la aplicación, así como también requieren pasos un tanto más complejos (en comparación a la Tabla 2)

para pasar a *verde.* Es por ello que se escriben pruebas pequeñas entre cada uno de estos pasos, utilizando Rspec.

Comparación de flujos de trabajo

A partir los análisis anteriores, se puede tener una idea más clara acerca de los procesos involucrados en cada una de las metodologías.

Lo más evidente es entender que las técnicas tienen enfoques distintos, explicitados incluso en sus definiciones. BDD tiene un enfoque *outside-in*, lo que quiere decir que intenta ver el problema desde la perspectiva del cliente, y TDD tiene un enfoque *bottom-up*, es decir, parte buscando la solución al problema desde una perspectiva técnica, construyendo componentes modulares y agrupándolos.

Esto explica por qué BDD tiene un componente en el negocio de la solución, componente que simplemente no existe en TDD clásico. Es por ello que fuerza un lenguaje natural o ubicuo, con el cual tanto clientes como desarrolladores pueden intercambiar ideas con la seguridad que poseen las mismas definiciones conceptuales. Esto ayuda por un lado al equipo, pues les define metas escritas en lenguaje natural por el cliente, alineándolos con las necesidades del negocio. Por el lado del cliente, BDD ayuda a manejar las expectativas de la solución, ya que cada historia genera criterios en forma de pruebas, las cuales deben estar en *green* para considerarse terminadas. Estos criterios quedan estipulados en el código, por lo que cuando el equipo considera una historia terminada, no deberían encontrarse con mayores sobresaltos al entregarla al cliente.

Todo lo anterior tiene un costo en tiempo importante, pues el cliente debe ser parte de las reuniones de planificación y debe crear los criterios de aceptación. En

general, el tiempo es un bien costoso y en ocasiones es difícil encontrar la forma de lograr el nivel de compromiso que BDD requiere. En el caso de TDD clásico, este gasto de tiempo no está incluido en el flujo de trabajo, por lo que le es indiferente. Por lo demás, se define como una metodología de desarrollo puro, por lo que quizás jamás fue considerada como una responsabilidad del proceso.

Otro punto importante de inflexión entre las metodologías, es la forma en que enfrentan el diseño y la arquitectura de software. Dada la experiencia en este Trabajo de Título, el flujo de trabajo de TDD es una exploración constante, pues no parte desde un diseño, sino que se espera que éste emerja desde las pruebas. Es por ello que en general, se intenta que los pasos hacia la solución sean pequeños, pues al ser una exploración, cualquier error en la ruta podrá ser solucionado rápidamente. TDD ocupa herramientas como la premisa de la prioridad de las transformaciones o bien recursos más clásicos como los patrones de diseño, con el fin evitar errores importantes de arquitectura. Lo anterior provoca que sea una metodología muy técnica.

Por otro lado, BDD tampoco requiere explícitamente de un diseño, sin embargo, puesto que tiene una visión desde el cliente, la primera historia de usuario, así como sus criterios, forzarán a que el equipo de desarrollo defina características importantes del sistema, como sus componentes principales. Es necesario construir un esqueleto mínimo que soporte la solución de forma que pueda ser mostrada al cliente y a su vez apruebe sus criterios. Esto puede producir que las pruebas generadas en las primeras iteraciones de desarrollo, o bien aquellas que dependan de componentes de la interfaz, sean frágiles, y deban ser corregidas y modificadas continuamente.

TDD entonces, producirá componentes modulares que interactuarán entre sí para elaborar funcionalidades más grandes, mientras que BDD intentará ante todo construir funcionalidades, y luego generará los componentes que sean necesarios para ello.

La documentación producida por las metodologías también varía. Esta variación viene dada más por el lenguaje que por el contenido de la misma. Sin embargo, para el caso de BDD, se produce documentación para los criterios de aceptación que puede simplificar la comunicación con el cliente, mientras que por otro lado la documentación generada por TDD clásico puede ser técnicamente más exhaustiva.

Finalmente, el tiempo de ejecución de las pruebas es otro punto a considerar, ya que en sistemas más complejos, las pruebas para BDD pueden tomar más tiempo en ejecutarse, ya que prueban funcionalidades. TDD clásico también prueba funcionalidades, en forma de test de integración, pero éstas suelen ser más acotadas, pues el grueso de las pruebas está en las pruebas unitarias, y se debe evitar la duplicación donde sea posible. Además las pruebas de comportamiento van desde la interfaz del usuario, hacia dentro del sistema.

Conclusiones

Durante el desarrollo de éste Trabajo de Título, se estudiaron distintas metodologías de desarrollo de software orientadas a la calidad.

El primer paso, fue estudiar y analizar la metodología clásica de Test Driven Development. Para ello se usó bibliografía, apoyada por la experimentación y uso de Katas para conocer el proceso a fondo y ser capaz de analizarlo.

Posteriormente, fue el turno de Behaviour Driven Development. Para este caso, se usó a modo de ejemplo el esqueleto de una aplicación web, con el fin de analizarla emulando un cliente, y las historias de usuario creadas por él.

Todo lo anterior busca dar al lector de este Trabajo de Título las herramientas suficientes para que pueda definir cuál de las metodologías se adaptaban de mejor manera a su flujo de trabajo y qué es lo que podía esperar si las integraban a él. Tal como se especificó al inicio del trabajo, éste tenía por fin responder principalmente tres preguntas, las cuales, a partir de lo aprendido, serán contestadas a continuación.

El primer desafío es comprender la razón por la cual estas metodologías se idearon inicialmente. En el caso de Test Driven Development clásico, nace de la necesidad de enfrentar principalmente los siguientes problemas:

- Alcance variable: Es fácil dejarse llevar cuando se implementa una funcionalidad, o bien escribir código que se cree será utilizado “en el futuro”.

- Componentes acoplados: Si una prueba resulta demasiado compleja de implementar, indica componentes con demasiadas responsabilidades o demasiado acoplados.

- Confianza: No es sencillo mantener la confianza dentro del equipo de desarrollo, si el código que genera un integrante no funciona. Al crear test automatizados se declara explícitamente la intención y la correctitud.

Adicionalmente, el hecho de mantener pruebas automatizadas evita que el código se deteriore, ya que se puede mejorar la implementación del sistema sin modificar su comportamiento.

Behaviour Driven Development, nace de los problemas que se enfrenta usando TDD de forma clásica. Entre éstos problemas, están lo difícil que resulta para los que lo practican por primera vez, el entender el flujo de trabajo o bien qué es lo que se está testeando. Aún más, TDD es una práctica de desarrollo que requiere compromiso y tiempo y es difícil mostrar al cliente el valor que se agrega al producto final. BDD combate esto integrándolo al equipo y motivándolo a explicitar sus criterios de aceptación, de forma tal que éstos quedan en el código y pueden ser ejecutados. Además, intenta cambiar el lenguaje técnico de TDD por uno natural dando la posibilidad que tanto el equipo de desarrollo como el cliente pueden comunicarse.

Segundo, ¿Qué ventajas comparativas presenta cada una? Las ventajas de cada una de las metodologías fueron analizas con profundidad en el capítulo de *Comparación de las metodologías*. Para el caso del desarrollo guiado por comportamiento, una de sus ventajas inmediatas es el aporte en la comunicación de todos los actores dentro del desarrollo de software. Los criterios de aceptación son

particularmente importantes, ya que ayudan a manejar expectativas del cliente, y adicionalmente muestran las metas al equipo de desarrollo.

Por otro lado, el desarrollo guiado por pruebas clásico posee ventajas como la baja barrera de entrada, y el limitado costo de mantención. Lo anterior lo transforma en un candidato ideal a un punto de entrada a las metodologías orientadas a la calidad.

Finalmente, es importante analizar en que situaciones es recomendable usar cada una de las metodologías. A partir de lo detallado en este Trabajo de Título, tanto la barrera de entrada, como el hecho que es una metodología exclusiva de desarrollo, hacen de Test Driver Development clásico la más interesante (y simple) de implementar de las dos. TDD es una metodología con un flujo sencillo, y cuya implementación puede resultar beneficiosa siempre y cuando se entienda que posee costos asociados, a nivel de tiempo y mantenimiento. Ahora bien, es importante aclarar que las metodologías no son excluyentes, es decir, se pueden implementar simultáneamente realizando pequeños ajustes a cada una. Por ejemplo, es interesante considerar BDD para crear criterios de aceptación, los que servirán como metas u objetivos de negocio. Luego, al momento de implementar el funcionamiento interno del sistema, se puede optar por TDD clásico. De ésta forma, se puede tener componentes modulares en el sistema, y a la vez estar alineado con el negocio y así mismo entregando transparencia desde el equipo de desarrollo al usuario.

Lo anterior no quiere decir que BDD sea inviable por sí sólo, pero en la experiencia de éste Trabajo de Título, posee costos que pueden resultar elevados si no se ha trabajado con metodologías similares. Es más, en escenarios donde no se tenga un cliente definido, o bien éste no cuente con el tiempo ni la dedicación necesaria para

ser parte integral del proyecto, su implementación puede resultar una carga y no un aporte.

Finalmente, es necesario explicitar que ninguna de éstas metodologías asegura calidad en el software por si solas, sino que únicamente disminuyen la probabilidad de introducir errores al realizar modificaciones en el código de la solución. Es responsabilidad de los desarrolladores utilizar estas herramientas para mejorar continuamente la implementación y mantenerla lo suficientemente flexible, de forma que sea capaz de enfrentar los cambios en el dominio de negocio que indudablemente se presentarán en el futuro.

Bibliografía

1. **Feathers, Michaels.** *Working Effectively with Legacy Code.* s.l. : Prentice Hall, Octubre 2, 2004.

2. **Poppendieck, Tom and Poppendieck, Mary.** *Implementing Lean Software Development: From Concept to Cash.* s.l. : Addison-Wesley Professional, Septiembre 17, 2006.

3. Quora. [Online] http://www.quora.com/Test-Driven-Development/Why-does-Kent-Beck-refer-to-the-rediscovery-of-test-driven-development.

4. **Beck, Kent.** *Extreme Programming Explained.* s.l. : Addison-Wesley Professional, 1999.

5. **Freeman, Steve and Pryce, Nat.** *Growing Object-Oriented Software, Guided by Tests.* s.l. : Addison-Wesley Professional, Octubre 22, 2009.

6. **Royce, Winston W.** Managing the development of large software systems. [Online] 1970. http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf.

7. **Cockburn, Alistair.** Design as Knowledge Acquisition. *Alistair Cockburn.* [Online] http://alistair.cockburn.us/Design+as+Knowledge+Acquisition.

8. Do The Simplest Thing That Could Possibly Work. *Extreme Programming Roadmap.* [Online] http://xp.c2.com/DoTheSimplestThingThatCouldPossiblyWork.html.

9. **Beck, Kent.** *Test Driven Development: By Example.* s.l. : Addison-Wesley Professional, Noviembre 18, 2002.

10. **Martin, Robert C.** *Agile Software Development, Principles, Patterns, and Practices.* s.l. : Prentice Hall, Octubre 25, 2002.

11. —. *Clean Code: A Handbook of Agile Software Craftsmanship.* s.l. : Prentice Hall, Agosto 11, 2008.

12. **Dijkstra, Edsger W.** The Humble Programmer, ACM Turing Lecture 1972. [Online] http://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html/.

13. NATO Software enginering conference, pag 32. [Online] Octubre 1968. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

14. **Shore, James.** *The Art of Agile Development .* s.l. : O'Reilly Media, 2007. 0596527675.

15. *Does Test-Driven Development Really Improve Software Design Quality?* **Janzen, David and Saiedian, Hossein.** March 2008, IEEE Software, pp. 77-84.

16. **Martin, Robert C.** The Transformation Priority Premise. *8th light.* [Online]

17. **Beck, Kent.** Simple Smalltalk Testing: With Patterns. *xprogramming.com.* [Online] http://www.xprogramming.com/testfram.htm.

18. **Fowler, Martin.** Xunit. *Martin Fowler.* [Online] http://www.martinfowler.com/bliki/Xunit.html.

19. **North, Dan.** Introducing BDD. *Dan North & associates.* [Online] Septiembre 20, 2006. http://dannorth.net/introducing-bdd/.

20. **Bache, Emily.** *The Coding Dojo Handbook.* s.l. : leanpub.com, 2013.

21. **Parnas, David L.** A rational design process: How and why to fake it. [Online] http://www.win.tue.nl/~mchaudro/sa2007/ParnasRationalDesign.pdf.

22. rspec-core. *Github.* [Online] https://github.com/rspec/rspec-core.

23. Home. *Cucumber Wiki.* [Online] https://github.com/cucumber/cucumber/wiki.

24. *Cucumber.* [Online] http://cukes.info/.

25. Specflow, pragmatic BDD for .net. [Online] http://www.specflow.org/.

26. Jbehave. [Online] http://jbehave.org/.

27. **Bache, Emily.** The London School of Test Driven Development. *coding is like cooking.* [Online] Abril 10, 2013. http://emilybache.blogspot.com/2013/04/the-london-school-of-tdd-verify-behaviour.html.

Anexos

Anexo A: Implementación *Bowling Kata*:

```ruby
# lib/bowling_game.rb
class BowlingGame

  def initialize
    @rolls  = []
  end

  def roll(pins)
    @rolls << pins
  end

  def score

    score = 0
    first_in_frame = 0
    10.times do |frame|

      if is_strike(first_in_frame)
        score += score_for_strike(first_in_frame)
        first_in_frame += 1
      else
        if is_spare(first_in_frame)
          score += score_for_spare(first_in_frame)
        else
          score += frame_score(first_in_frame)
        end
        first_in_frame += 2
      end

    end

    score
  end

  private

  def is_spare(first_in_frame)
    frame_score(first_in_frame) == 10
  end

  def is_strike(first_in_frame)
    @rolls[first_in_frame] == 10
  end

  def frame_score(first_in_frame)
    @rolls[first_in_frame] + @rolls[first_in_frame+1]
  end
```

```
  def score_for_spare(first_in_frame)
    10 + @rolls[first_in_frame+2]
  end

  def score_for_strike(first_in_frame)
    10 + @rolls[first_in_frame+1] + @rolls[first_in_frame+2]
  end

end
```

```
# test/bowling_game_test.rb
require "test/unit"

class TestBowlingGame < Test::Unit::TestCase

  def setup
    @game = BowlingGame.new
  end

  def test_all_rolls_missed
    roll(20, 0)
    assert_equal(0, @game.score)
  end

  def test_all_ones
    roll(20, 1)
    assert_equal(20, @game.score)
  end

  def test_one_spare
    roll_spare
    @game.roll(3)
    roll(17, 0)

    assert_equal(16, @game.score)
  end

  def test_one_strike
    roll_strike
    @game.roll(4)
    @game.roll(3)
    roll(16, 0)

    assert_equal(24, @game.score)
  end

  def perfect_game
    roll(12,10)
    assert_equal(300, @game.score)
  end

  private

  def roll(n, pins)
    n.times { @game.roll pins}
```

```
  end

  def roll_spare
    @game.roll(5)
    @game.roll(5)
  end

  def roll_strike
    @game.roll(10)
  end

end
```

Anexo B: Implementación *Game of life Kata*

```
# lib/game_of_life.rb

class GameOfLife

  attr_reader :alive_cells

  def initialize(seed=[[0,0]])
    @alive_cells = seed
  end

  def next
    @alive_cells = survivors + births
  end

  def survivors
    @alive_cells.select { |cell| alive_neighbours(cell).count.between?(2,3) }
  end

  def alive_neighbours(cell)
    neighbours(cell) & @alive_cells
  end

  def births
    @alive_cells.flat_map do |cell|
      dead_neighbours(cell).select do |neighbour|
        alive_neighbours(neighbour).count == 3
      end
    end.uniq
  end

  def dead_neighbours(cell)
    neighbours(cell) - @alive_cells
  end

  def neighbours(cell)

    x, y = cell
    delta = [
```

```
      [-1, 1], [ 0, 1], [ 1, 1],
      [-1, 0],          [ 1, 0],
      [-1,-1], [ 0,-1], [ 1,-1]
    ]

    delta.map {|dx, dy| [x + dx, y + dy] }
  end

end
```

```
# test/game_of_life_test.rb

require "test/unit"

require_relative "../lib/game_of_life"

class TestGameOfLife < Test::Unit::TestCase

  def test_blank_board
    game = GameOfLife.new

    assert_equal([], game.next)
  end

  def test_game_stores_alive_cells
    seed = [[0,0]]
    game = GameOfLife.new(seed)

    assert_equal(seed, game.alive_cells)
  end

  def test_blank_board_with_origin_seed
    game = GameOfLife.new([[0,0]])

    assert_equal([], game.next)

  end

  def test_alive_neighbours
    seed = [[0,0]]
    game = GameOfLife.new(seed)

    assert_equal([], game.alive_neighbours([0,0]))
  end

  def test_alive_neighbours_with_two_alive
    game = GameOfLife.new([
      [0,0],
      [1,0],
      [1,1]
      ])

    assert_equal([[1,0],[1,1]].sort.uniq, game.alive_neighbours([0,0]).sort.uniq)
  end
```

```
  def test_neighbours_in_origin
    origin = [0,0]
    game = GameOfLife.new([origin])
    neighbours = game.neighbours(origin)

    assert_equal(8, neighbours.size)
    assert(!neighbours.include?([0,0]))
    assert(neighbours.include?([1,0]))
    assert(neighbours.include?([-1,-1]))
  end

  def test_survivors

    # .**.
    # .**.

    seed = [
      [0,0],
      [0,1],
      [1,0],
      [1,1]
    ]

    game = GameOfLife.new(seed)

    assert_equal(seed, game.next)
    assert_equal(seed, game.next)
  end

  def test_dead_neighbours
    seed = [
      [-1, 0],
      [ 1, 1],
      [-1,-1]
    ]

    game = GameOfLife.new(seed)

    expected_dead_neighbours = [
      [-1,  1],
      [ 0,  1],
      [ 1,  0],
      [ 0, -1],
      [ 1, -1],
    ]

    assert_equal(expected_dead_neighbours.sort, game.dead_neighbours([0,0]).sort)

  end

  def test_births

    # .*
    # **

    game = GameOfLife.new([
      [0,0],
```

```
      [1,0],
      [1,1]
      ])

    assert_equal([[0,1]], game.births)
  end

  def test_blinker

    # .*.
    # .*.
    # .*.
    #
    # -->
    #
    # ...
    # ***
    # ...

    step_one = [
      [1,0],
      [1,1],
      [1,2]
    ]

    step_two = [
      [0,1],
      [1,1],
      [2,1]
    ]

    game = GameOfLife.new(step_one)

    assert_equal(step_two.sort, game.next.sort)
    assert_equal(step_one.sort, game.next.sort)
  end

end
```

Anexo C: Implementación *Tienda de videojuegos*

```
# controllers/game_controller.rb

class GamesController < ApplicationController

  def new
    @game = Game.new
  end

  def create
    @game = Game.new(params[:game])
    if @game.save
      redirect_to games_path
```

```
    else
      flash[:alert] = 'The game already exists'
      redirect_to new_game_path
    end
  end

  def index
    @games = Game.all
  end

end
```

```
# models/game.rb

class Game
  include Mongoid::Document

  attr_accessible :name, :description

  field :name, type: String
  field :description, type: String

  validates_uniqueness_of :name
end
```

```
<!-- views/index.html.erb -->

<div>
  <ul>
    <% @games.each do |game| %>
        <li>
            <h3>
              <%= game.name %>
            </h3>
            <p>
              <%= game.description %>
            </p>
        </li>
    <% end  %>
  </ul>
</div>
```

```
<!-- views/new.html.erb -->

<%= form_for @game do |f| %>
    <%= f.label :name %>
    <%= f.text_field :name, id: 'name' %>
    <%= f.label :description %>
    <%= f.text_field :description, id: 'description' %>
```

```
    <%= f.submit 'Create' %>
<% end %>
```

```
# config/routes.rb

VaporStore::Application.routes.draw do

  resources :games, only: [:new, :create, :index]

 end
```

```
# spec/models/game_spec.rb

require 'spec_helper'

describe Game do

  describe 'name' do

    let(:game) {{ name: 'Grand Theft Auto IV',
                  description: 'Open world action-adventure video game' }}

    before { Game.create(game) }

    it 'has to be unique' do
      expect{Game.create(game)}.not_to change(Game, :count)
    end

  end

end
```

```
# spec/controllers/games_controller_spec.rb

require 'spec_helper'

describe GamesController do

  describe '#new' do

    before { get :new }

    it 'is successful' do
      expect(response).to be_success
    end

    it 'creates a new game object' do
      expect(assigns(:game)).not_to be_nil
    end
  end
```

```
  describe '#create' do

    before { post :create, game: {name: 'Starcraft', description: 'Military science
fiction real-time strategy'} }

    it 'creates a new game' do
      expect(assigns(:game)).not_to be_nil
      expect(assigns(:game).name).to eq 'Starcraft'
    end

    it 'redirects to the book list page' do
      expect(response).to redirect_to games_path
    end

  end

  describe '#index' do

    it 'is successful' do
      get :index
      expect(response).to be_success
    end

    context 'there is at least a game in the store' do

      before { Game.create(name: 'Deus Ex: Human Revolution', description:
'Cyberpunk-themed action role-playing video game') }

      it 'sends games to the view' do
        get :index
        expect(assigns(:games)).not_to be_nil
        expect(assigns(:games).size).to be 1
      end
    end

  end

end
```

```
# features/user_manages_games.feature

Feature: User manages games

Scenario: User adds a game to the list
  Given A new game with name The Witcher 3 and description Open world action RPG
  When I go to the add game page
  And I fill in the game’s name
  And I fill in the game’s description
  When I press 'Create'
  Then I'm redirected to games list page
  And I see The Witcher 3

Scenario:
  Given This games exists:
```

```
    | name          | description |
    | The Witcher 3 | Open world action RPG |
  When I go to the add game page
  And I fill in name with The Witcher 3
  And I fill in description with Open world action RPG
  When I press 'Create'
  Then I see and error message
  When I go to the games list page
  Then The game is listed only once
```

```
# features/steps/steps.rb

Given(/^A new game with name (.*) and description (.*)$/) do |name, description|
  @name = name
  @description = description
end

When(/^I go to the add game page$/) do
  visit new_game_path
end

And(/^I fill in the game’s name$/) do
  fill_in :name, with: @name
end

And(/^I fill in the game’s description$/) do
  fill_in :description, with: @description
end

When(/^I press 'Create'$/) do
  click_button 'Create'
end

Then(/^I'm redirected to games list page$/) do
  expect(current_path).to eq games_path
end

And(/^I see The Witcher 3$/) do
  expect(page).to have_content 'The Witcher 3'
end

Given(/^This games exists:$/) do |table|
  Game.create table.hashes
end

And(/^I fill in (.*) with (.*)$/) do |field, value|
  fill_in field, with: value
end

Then(/^I see and error message$/) do
  expect(page).to have_selector '.alert', text: 'The game already exists'
end

When(/^I go to the games list page$/) do
  visit games_path
```

```
end

Then(/^The game is listed only once$/) do
  expect(page).to have_content 'The Witcher 3', count: 1
end
```