PONTIFICIA UNIVERSIDAD CATÓLICA DEL ECUADOR

FACULTAD DE INGENIERÍA

DISERTACIÓN PREVIA A LA OBTENCIÓN DEL TÍTULO DE INGENIERO DE SISTEMAS Y COMPUTACIÓN

"AUTOMATIZACIÓN DE PRUEBAS UNITARIAS EN EL MARCO DE DESARROLLO ÁGIL PARA CARTERA DE TICKETS ELECTRÓNICOS DE PARQUEADERO"

AUTOR

ANDRÉS ASIMBAYA

DIRECTOR:

ANDRÉS JIMÉNEZ

QUITO, ABRIL 2019

Dedicatoria

A las personas mas importantes en mi vida, mis padres cuyo esfuerzo y dedicación se han reflejado en todo lo que soy y puedo llegar a ser, ellos me han enseñado que hay un solo núcleo que es para siempre, la familia.

Agradecimiento

A mis mentores y amigos, Guillermo Carrillo e Iann Yanes cuyas enseñanzas me ayudaron a formar lo que soy profesionalmente y lo que quisiera llegar a ser.

A las personas que me enseñaron que la universidad nos ayuda a alinear nuestro pensamiento, pero lo que sabes de verdad lo aprendes día a día (Webcreekzuela).

Índice General

Índice de Figuras

CAPÍTULO 1
INTRODUCCIÓN

1.1. Tema

AUTOMATIZACIÓN DE PRUEBAS UNITARIAS EN EL MARCO DE DESARROLLO ÁGIL, CASO DE ESTUDIO: CARTERA DE TICKETS ELECTRÓNICOS DE PARQUEADERO.

1.2. Justificación

El desarrollo ágil al momento es de ese tipo de cosas que las personas quieren usar, pero nadie sabe exactamente cómo, basado en esto nace la necesidad de entender el desarrollo ágil en general y lograr profundizar en una rama importante como son las pruebas unitarias, en la mayoría de software que se desarrolla se plantean casos de uso y posteriormente casos de prueba, los "tester" (aseguradores de calidad) se encargan de ejecutar los casos de prueba pero el objetivo del desarrollo ágil es que todo sea mucho más rápido, la automatización de pruebas unitarias es un cambio de pensamiento en donde los casos de prueba nacen junto con el código esto quiere decir que las pruebas unitarias también deben ser programadas disminuyendo la posibilidad a fallas y agilizando el desarrollo desde el punto de vista en que las pruebas ya no necesitan de los "tester" para ser ejecutadas, esta automatización debe tomar en cuenta consideraciones para que funcione como un método de agilizar el proceso de desarrollo, caso contrario; se torna en un obstáculo en lugar de una ayuda.

1.3. Alcance

Se realizará una investigación descriptiva para estudiar las características de las pruebas unitarias en el marco del desarrollo ágil y como se lo puede incluir en el marco de DevOps para el despliegue y la arquitectura de las aplicaciones. Se usará la investigación correlacional entre las pruebas unitarias automatizadas y el desempeño del desarrollo ágil con datos obtenidos del desarrollo del caso de estudio.

1.4. Planteamiento del problema

Las pruebas en general y las pruebas unitarias específicamente consumen tiempo del proceso de desarrollo de software al tratar de ser ágil se necesita disminuir al máximo el

tiempo que cada proceso consume, es por esto que la forma de agilizar las pruebas unitarias es proponer una automatización de las mismas, generando un método especifico que funcione dentro del marco de desarrollo de software ágil aplicándolo a un caso de estudio con el uso del framework de desarrollo SCRUM y para la aplicación práctica de la automatización de las pruebas unitarias se usara la misma herramienta que se usara para desarrollar el caso de estudio con AngularJS. Generalmente, se puede encontrar problemas cuando se trata de automatizar pruebas y no se tiene una arquitectura de software adecuada, ya que si la arquitectura no es modular las pruebas tampoco lo serán y una de las ventajas principales de automatizar pruebas es que pueden correrse y verificar el estado de un módulo, con la modularidad se logra que el software sea extensible es decir que si se aumenta una funcionalidad siguiendo la arquitectura propuesta la nueva funcionalidad no tiene por qué alterar el funcionamiento de las funcionalidades previas y las nuevas pruebas solo deben extender las comprobaciones de las pruebas previas. Se propondrá entender el desarrollo ágil en líneas generales y comprobar qué lugar ocupa la automatización de pruebas unitarias dentro de la propuesta ágil. También se tratará de discutir el prejuicio de que el desarrollo ágil no genera productos de calidad debido a que se disminuyen los tiempos en el proceso de desarrollo, la propuesta de la automatización de pruebas unitarias comprobara que a pesar de ser desarrollo ágil el aseguramiento de la calidad del software a partir de las pruebas unitarias con esta propuesta es mayor que implementar dichas pruebas de manera manual. En el contexto actual, no existe una receta clara y practica de cómo usar la automatización de pruebas para agilizar el proceso de desarrollo usando herramientas innovadoras y actuales, es por esto que se busca proponer este método y así comprender que las pruebas unitarias deben nacer desde la arquitectura misma y que el valor que pueden generar para la calidad del software es mayor que el tiempo que se emplea en implementar las pruebas.

1.5. Objetivos

1.5.1 Objetivo General

Automatizar pruebas unitarias para el desarrollo del prototipo de la cartera de tickets electrónicos de parqueadero.

1.5.2 Objetivos Específicos

- Exponer qué lugar ocupa la automatización de pruebas unitarias en el desarrollo ágil.

- Proponer el uso de herramientas actuales de desarrollo, arquitectura, infraestructura y entrega continua (DevOps) con el fin de agilizar el proceso de desarrollo de software.
- Lograr encajar las herramientas actuales de desarrollo, arquitectura, infraestructura y entrega continua (DevOps) con la automatización de pruebas unitarias.
- Implementar un método de automatización de pruebas unitarias en el marco del desarrollo ágil.

CAPÍTULO 2
MARCO TEÓRICO

2.1. Desarrollo ágil

El desarrollo ágil al momento es de ese tipo de cosas que las personas quieren usar, pero nadie sabe exactamente cómo, basado en esto nace la necesidad de entender el desarrollo ágil en general y lograr profundizar en una rama importante como son las pruebas unitarias, en la mayoría de software que se desarrolla se plantean casos de uso y posteriormente casos de prueba, los "tester" (aseguradores de calidad) (Molinero, 2018, pág. 253), los cuales se encargan de ejecutar los casos de prueba pero el objetivo del desarrollo ágil es que todo sea mucho más rápido, la automatización de pruebas unitarias es un cambio de pensamiento en donde los casos de prueba nacen junto con el código esto quiere decir que las pruebas unitarias también deben ser programadas disminuyendo la posibilidad a fallas y agilizando el desarrollo desde el punto de vista en que las pruebas ya no necesitan de los "tester" para ser ejecutadas, esta automatización debe tomar en cuenta consideraciones para que funcione como un método de agilizar el proceso de desarrollo, caso contrario; se torna en un obstáculo en lugar de una ayuda.

Es por este argumento que en este subcapítulo se abordará el tema del desarrollo ágil luego de esta breve explicación de su importancia.

La idea del desarrollo ágil nace del pensamiento de un grupo de individuos que mediante la observación de las problemáticas que presentaban los equipos de desarrollo de software en distintas empresas (Nathaly, 2015). Establecieron principios y lineamientos para su correcto funcionamiento. Estos individuos se autodenominaban Agile Alliance.

El trabajo de este equipo se enfoca en realizar una declaración de valores, que bautizaron como el Manifiesto para el desarrollo de software ágil.

Al respecto, lo exponen de la siguiente manera:

> Yague (2018) menciona que se estamos descubriendo mejores formas de desarrollo de software haciéndolo y ayudando a otros hacerlo.
>
> **Individuos e interacciones** sobre procesos y herramientas.
>
> **Software de trabajo** sobre documentación completa.
> **Colaboración del cliente** en la negociación de contratos.
> **Respondiendo al cambio** siguiendo un plan.
>
> Es decir, si bien hay valor en los artículos en a la derecha, valoramos más los artículos de la izquierda.

A continuación, analizaremos cada uno de estos presupuestos, para manejar conceptos claros que nos permitirán entender el trabajo a realizarse.

a) Individuos e interacciones sobre procesos y herramientas

Las metodologías ágiles valoran el recurso humano por sobre el recurso material por eso hacen énfasis en considerar la importancia de las personas en los procesos de desarrollo, hablamos de que: i) personas capacitadas; ii) que puedan trabajar en grupo iii) se adapten al entorno y iv) sean eficaces, pueden triunfar en el desarrollo de un proyecto, siempre y cuando los procesos en los que se desarrolle el trabajo, también sean eficaces. (Evaluando software, 2018)

Afirman que contar con un equipo calificado, que además tiene talento para interactuar con el usuario brinda mayores probabilidades de éxito que contar con herramientas y procesos altamente inflexibles.

Es menor la importancia que se le atribuye la implementación de un bagaje de herramientas y entornos adecuados, sin embargo, de lo cual, a la par, es necesario tener en cuenta que estos implementos no se consideren “sobre enfatizados” puesto que el exceso de herramientas, así como su manejo complejo, puede constituir más perjudicial que una ayuda al proyecto.

En este primer aspecto la recomendación de la doctrina se resume a conformar un equipo de trabajo con todas las competencias y capacidad necesarias, para luego proveerle de las herramientas que conforme el avance del proyecto el equipo, llegase a necesitar.

Para garantizar una mayor productividad, las metodologías ágiles valoran el recurso humano como el principal factor de éxito. Para lograr esto son los procesos los que se tienen que adaptar al equipo y no al revés. (The Blokehead, 2016)

b) Software de trabajo sobre documentación completa

La página web Cátedra Viewnext USAL (2018) menciona que "La documentación, en las metodologías ágiles procura mecanismos más dinámicos y menos costosos como son la comunicación personal, el trabajo en equipo, la auto documentación y los estándares".

Este grupo también destaca la importancia de una documentación, clara y precisa que permita la comunicación, entre el equipo, mantener estándares, clarificar conceptos y requerimientos que sean legibles para los seres humanos y no basarse únicamente en la producción del lenguaje de máquina. Sin embargo, estos documentos deben ser concisos y cortos, ya que una documentación innecesariamente cargada podría constituir varios problemas para el equipo, que pueden ser de sincronización y/o producción. Esta documentación buscar ordenar y organizar procesos.

Cuando los documentos son estrictamente los necesarios y estos contienen justificaciones y estructuras básicas se nos presenta una primera problemática que se resume en actualizar a los nuevos miembros del equipo, ya que al no contener especificaciones amplias, estos no pueden realizar una autoalimentación, por lo que será obligación de los miembros del equipo prestar la ayuda necesaria para que el nuevo integrante se encuentre en las capacidades de entender la documentación por sí mismo, de esta manera volvemos a afirmar la necesidad de contar con excelentes integrantes en el equipo.

Los dos documentos que son los mejores para transferir información a los nuevos miembros del equipo son el código y el equipo. El código no miente sobre lo que hace. Puede ser difícil extraer las razones y la intención del código, pero el código es la única fuente de información no ambigua. El equipo tiene la hoja de ruta siempre cambiante del sistema en la cabeza de sus miembros. La forma más rápida y eficiente de poner esa hoja de ruta en el papel y transferirla a otros es a través de la interacción de persona a persona.

Es importante que los equipos, no suspendan el trabajo y en la elaboración de los documentos, sino que es importante que produzcan la documentación estrictamente necesaria. (The Blokehead, 2016)

c) Colaboración del cliente en la negociación de contratos

En términos generales, es el cliente quien realiza las indicaciones y solicitudes sobre las bases de necesidades y requerimientos para poder aprobar el software terminado, a su entera satisfacción y en los tiempos acordados.

Al respecto nos aporta la siguiente observación:

Es tentador para los gerentes de la compañía decirle a su personal de desarrollo cuáles son sus necesidades y luego esperar que el personal se vaya por un tiempo y regrese con un sistema que satisfaga esas necesidades. Pero este modo de operación conduce a la mala calidad y al fracaso.

Es ciertamente un hecho que en el desarrollo del producto los clientes y el equipo, asumen una posición distante y defensiva que no permite un correcto avance del trabajo del equipo. (The Blokehead, 2016)

La solución a este inconveniente es claramente, involucrar al cliente en cada etapa del desarrollo, para obtener una retroalimentación constante y correcta de sus necesidades o cambios. Es necesario asumir que el trabajo que se realiza entre cliente y equipo, más que una posición de rivalidad, debe ser un aporte de beneficio común.

El trabajo entre grupos humanos constituye la presencia del usuario, pues él es el llamado a hacer conocer lo que se necesita corregir, aumentar y recomendar, es por ellos que su trabajo debe ser constante desde el inicio hasta la entrega.

Un ejemplo de un contrato exitoso es el que negocié para un proyecto grande, multianual, de medio millón de líneas en 1994. Nosotros, el equipo de desarrollo, recibimos una tarifa mensual relativamente baja. Se nos hicieron grandes pagos cuando entregamos ciertos bloques grandes de funcionalidad. Esos bloques no fueron especificados en detalle por el contrato. Más bien, el contrato estipulaba que el pago se haría por un bloqueo cuando el bloqueo pasara la prueba de aceptación del cliente. Los detalles de esas pruebas de aceptación no se especificaron en el contrato.

Durante el transcurso de este proyecto, trabajamos muy de cerca con el cliente. Le lanzamos el software casi todos los viernes. Para el lunes o martes de la semana siguiente, tenía una lista de cambios para que los pongamos en el software. Priorizamos esos cambios juntos y luego los programamos en semanas subsiguientes. El cliente trabajó tan estrechamente con nosotros que las pruebas de aceptación nunca fueron un problema. Sabía cuándo un bloque de funcionalidad satisfacía sus necesidades, porque lo veía evolucionar de una semana a otra.

Los requisitos para este proyecto se encontraban en un estado continuo de flujo. Los grandes cambios no fueron infrecuentes. Se eliminaron bloques enteros de funcionalidad y se insertaron otros. Y, sin embargo, el contrato, y el proyecto, sobrevivieron y tuvieron éxito. La clave de este éxito fue la intensa colaboración con el cliente y un contrato que rige dicha colaboración en lugar de intentar especificar los detalles del alcance y el cronograma a un costo fijo. (Cátedra Viewnext USAL, 2018)

d) Responde al cambio sobre el siguiente plan

"La capacidad de responder al cambio a menudo determina el éxito o el fracaso de un proyecto de software. Cuando elaboramos planes, debemos asegurarnos de que sean flexibles y estén listos para adaptarse".

Es de entenderse, que, en la dinámica del desarrollo de la sociedad actual, un proyecto se puede enfrentar a cambios constantes e inesperados, ya sean desde temas de personalización hasta la realización de nuevos productos. Es por esta razón que las estrategias que se elaboren para la realización del proyecto no sean estrictamente planificadas puesto que esto evita poder reaccionar a tiempo con los cambios requeridos.

Es en este punto de la discusión donde las metodologías pesadas fracasan, puesto que su planificación conserva una dinámica tan estrecha que casi imposibilita responder ante estos imprevistos.

Las recomendaciones doctrinaras para superar este factor, es básicamente realizar planes a corto plazo, con cumplimiento de objetivos en ese tiempo, de tal forma que para las siguientes estrategias pueda mutar, al respecto ------ nos comenta lo siguiente.

Una mejor estrategia de planificación es hacer planes detallados para la próxima semana, planes aproximados para los próximos 3 meses y planes extremadamente crudos más allá de eso. Debemos conocer las tareas individuales en las que trabajaremos la próxima

semana. Deberíamos conocer los requisitos en los que estaremos trabajando durante los próximos 3 meses. Y deberíamos tener solo una vaga idea de lo que hará el sistema después de un año.

Se puede notar que el hecho importante en esta recomendación es que los planes a futuro son más flexibles y estrictos que los que se realizarán inmediatamente, lo que evidentemente nos ayudará con la consecución del objetivo.

2.2. Principios del manifiesto ágil

Miguel (2018) presenta ciertos principios a ser aplicados juntamente con el esquema del desarrollo ágil, a continuación, realizaremos un breve resumen de cada uno:

a) Nuestra mayor prioridad es satisfacer al cliente mediante entregas tempranas y continuas de software.

Es una característica principal de los sistemas ágiles, la entrega de software funcional en el menor tiempo posible, esto nos brinda como resulta un cliente satisfecho al cual le hemos ahorrando largas esperas para ver pequeños avances del producto.

Es evidente que en métodos como estos se requiere de la constante participación del cliente, pues es él quien evaluará en el menor tiempo el resultado del producto. (Miguel, 2018)

b) Bienvenido el cambio de requisitos tardíos

Los procesos ágiles aprovechan el cambio para la ventaja competitiva del cliente. Esta es una declaración de actitud. Los participantes en un proceso ágil no tienen miedo al cambio. Consideran que los cambios en los requisitos son cosas buenas, porque esos cambios significan que el equipo ha aprendido más sobre lo que se necesitará para satisfacer al cliente.

Evidenciamos que los requerimientos que realiza el cliente, so a partir de su perspectiva de usuario y no como un desarrollador, es por eso que sus solicitudes seguirán mutando según la evolución del proyecto, es por eso que se debe asumir como un proceso de maduración del producto final.

Nuestra capacidad para responder ante estos imprevistos, es decir, responder bien y con agilidad amentará la satisfacción del usuario

Un equipo ágil trabaja muy duro para mantener la estructura de su software flexible, de modo que cuando los requisitos cambian, el impacto en el sistema sea mínimo. Más adelante en este libro, analizaremos los principios, patrones y prácticas de diseño orientado a objetos que nos ayudan a mantener este tipo de flexibilidad. (Miguel, 2018)

c) Ofrezca software de trabajo con frecuencia

La entrega de un producto funcional debe realizarse con la mayor frecuencia posible, es decir, tanto planes como documentos, no son verdaderas entregas, sino aquellas que el cliente pueda valorar en su funcionalidad y pueda en algún punto satisfacer las necesidades del cliente. (Miguel, 2018)

d) Las personas del negocio y los desarrolladores deben trabajar juntos a lo largo del proyecto

El usuario es quien nos aportará críticas en torno a la funcionalidad del proyecto y el cumplimiento de sus requerimientos.

Los empresarios y desarrolladores deben trabajar juntos todos los días a lo largo del proyecto. Para que un proyecto sea ágil, los clientes, los desarrolladores y las partes interesadas deben tener una interacción importante y frecuente. (Miguel, 2018)

e) Construye proyectos alrededor de individuos motivados.

Es conveniente que el ambiente de trabajo genere ánimos de pertenencia, disposición y motivación para el equipo pues es este quien está a cargo del éxito o fracaso del proyecto a entregarse.

f) El método más eficiente y efectivo de transmitir información hacia y dentro de un equipo de desarrollo es la conversación cara a cara.

El desarrollo del proyecto debe basarse en la comunicación, entre miembros del equipo y el quipo con el usuario y/o cliente y los doctrinarios afirman que la mejor forma de realizarlo es hablando personalmente, de esta forma evitamos intervinientes extras en la comunicación que en algún punto puedan distorsionar el mensaje. (Miguel, 2018)

g) El software de trabajo es la principal medida del progreso.

Los proyectos ágiles miden su progreso al medir la cantidad de software que actualmente satisface las necesidades del cliente. No miden su progreso en términos de la

fase en que se encuentran o por el volumen de documentación que se ha producido o por la cantidad de código de infraestructura que han creado. (Miguel, 2018)

h) Los procesos ágiles promueven el desarrollo sostenible.

Se debe manejar al quipo de manera que puedan mantener un ritmo constante, mediante el cual todos tengan tareas asignadas desde el principio y puedan cumplirlas al mismo ritmo.

Correr demasiado rápido conduce a agotamiento, atajos y debacle. Los equipos ágiles se apresuran No se dejan cansar demasiado. No toman prestada la energía del mañana para hacer un poco más hoy. Trabajan a una velocidad que les permite mantener los estándares de más alta calidad durante la duración del proyecto. (Miguel, 2018)

i) La atención continua a la excelencia técnica y el buen diseño mejoran la agilidad.

Al respecto, entendemos que la agilidad y la calidad se refleja desde la perspectiva del usuario y del equipo, en cuanto a la perspectiva del equipo, hablamos de un código que se limpia continuamente, es decir a la vez que es creado. (Miguel, 2018)

j) La simplicidad, el arte de maximizar la cantidad de trabajo no hecho, es esencial

Es importante priorizar las mejoras que deben o no realizarse, pues existen ciertas funcionalidades que serán aprovechadas y otras que no serán necesarias, es por esto que se debe dedicar tiempo importante a aquellas requeridas por el cliente. (Miguel, 2018)

k) Las mejores arquitecturas, requisitos y diseños surgen de los equipos autoorganizados

Las responsabilidades no se entregan a los miembros individuales del equipo desde el exterior, sino que se comunican al equipo en su totalidad. El equipo determina la mejor manera de cumplir esas responsabilidades. (Miguel, 2018)

l) En intervalos regulares, el equipo reflexiona sobre cómo ser más efectivo, luego ajusta y ajusta su comportamiento como corresponde

Un equipo ágil ajusta continuamente su organización, reglas, convenciones, relaciones, etc. Un equipo ágil sabe que su entorno está cambiando continuamente y sabe que debe cambiar con ese entorno para seguir siendo ágil. (Miguel, 2018)

En cuanto a la implementación del manifiesto ágil, existen varias propuestas, como lo son, SCRUM, DSDM, ASD, FDD Y LD, a continuación, nos enfocaremos en un breve análisis de SCRUM como metodología ágil.

2.2.1 SCRUM

SCRUM, es una de las metodologías ágiles que engloba un concepto de marco de trabajo, en el cual los equipos pueden crear proyectos de forma "interativa e incremental". El desarrollo de esta metodología se basa en los denominados "sprints" en los cuales el equipo, logrará objetivos, trazará prioridades y realizará entregas de productos funcionales. (Subra & Vannieuwenhuyse, 2018)

El tiempo de duración del sprint, será establecido por el equipo, generalmente empiezan con ciclos amplios y mientras van mejorando los acortan lo que aumenta su eficiencia.

El sprint, empieza con la realización de una lista de prioridades y requerimientos que podrán ser cumplidos por el equipo con la entrega de avances funcionales y terminados por completo, se entiende que mientras dure el sprint, este no podrá asumir cambios ni ser modificado, sino hasta el siguiente sprint, recordemos que al ser una metodología ágil este es susceptible de adaptación a los cambios. Al reunirse el equipo y los interesados en el producto, se muestran los objetivos cumplidos y se obtienen retroalimentaciones a ser implementadas en el siguiente sprint.

Los miembros del equipo SCRUM son: el dueño del producto, equipo y scrum master, a saber: i) la función del dueño del producto es priorizar las funcionalidades del producto en una lista organizada para así, poder definir el siguiente sprint, es decir enlistar los elementos con más y menos valor, para el desarrollo del proyecto, todo mediante su apreciación y de la retroalimentación de los interesados en el producto; ii) equipo, es aquel que desarrollo lo que indica el dueño del producto, al respeto, entendemos que el equipo es cross funcional y autogestionado, cada miembro tiene diferentes capacidades y experticias para llegar a obtener lo que se indica, nutrirse del conocimiento del equipo y también puede realizar

recomendaciones al dueño del producto; iii) el scrum master, es el encargado de ayudar al equipo a implementar correctamente SCRUM. (Coremain, 2019)

Esta metodología viene acompañada de un método de organización denominado backlog de producto, mediante el cual se priorizará y enlistará los requerimientos desde la perspectiva del cliente, el backlog, puede contener, nuevas funcionalidades, mejoras en la ingeniería, investigaciones a realizarse, optimización de defecto desconocidos, entre otros.

El backlog, debe ser: i) detallado, al menos en sus elementos más priorizados pues son los que están próximos a realizarse ii) estimados, los elementos deben estar estimados y debe ser siempre revisados para ser corregidos en cada sprint, según la nueva información que se obtiene, iii) emergente, el backlog, debe ser actualizado de acuerdo con los nuevos requerimientos y necesidades del cliente; iv) priorizado, la lista debe estar enumerada desde los elementos más priorizados a los menos priorizados, se entiende que los más priorizados son los que tienen más valor para el negocio a un menor costo.

Por último, es necesario que el equipo como el dueño del producto, establezcan claramente la definición de producto terminado en cada sprint, es decir un avance funcional al final del ciclo, de esta manera el equipo planificara su actuar para conseguir lo que se denominó "terminado."

2.2.2 DevOps (Development+Operations)

Se refiere a un conjunto de procesos y métodos para pensar a cerca de la comunicación y la colaboración entre los departamentos de Desarrollo y TI. Se describe comúnmente como una relación más colaborativa y productiva entre los equipos de desarrollo y los equipos de operaciones. Los DevOps requieren de mano de obra igualmente la gestión de bases de datos y la administración de sistemas. (Atlassian , 2019)

La doctrina suele calificarlo como una cultura de organización basada principalmente en la comunicación de los equipos para obtener una rápida entrega de un producto eficiente.

Al respecto Lucas (2016) manifiesta que "DevOps es un profesional IT cuyo objetivo es ayudar al desarrollo y producción de productos creando un puente entre el departamento encargado del desarrollo y de las operaciones"

El objetivo de DevOps es cambiar y mejorar la relación promoviendo una mejor comunicación y colaboración entre estas dos unidades de negocio.

En la empresa es necesario desglosar los silos, donde las unidades de negocios operan como entidades individuales dentro de la empresa donde la administración, los procesos y la información están protegidos. En el lado del desarrollo de software, y para aquellos que trabajan en operaciones de TI, es necesario que haya una mejor comunicación y colaboración para atender mejor las necesidades de TI de la organización.

Entonces, entendemos que durante el ciclo de desarrollo de un producto este se ve influenciado por la elaboración de diferentes actividades generalmente a cargo de distintos departamentos. La elaboración de datos, recopilación de requisitos, el desarrollo, elaboración de código, las pruebas realizadas, se encuentran a cargo de varios miembros del equipo, sin embargo, estos no siempre mantenían unidos, sino más bien en instalaciones diferentes, lo que ocasionaba una comunicación distorsionada, un desequilibrio en el avance del proyecto, problemas de funcionalidad y verificación que debía resolverse en cada departamento ocasionando que el proyecto tome más tiempo del necesario.

La solución que nos brinda Devops, es la aplicación de un conjunto de herramientas y estrategias que permitirá la comunicación del equipo Mendoza (2018). Una estrategia holística de DevOps, en el nivel más básico, debe responder las siguientes preguntas:

¿Cuáles son nuestros objetivos y metas comerciales?

¿Cómo planeamos la hoja de ruta? ¿Por dónde empezamos?

¿Cómo debemos canalizar nuestros esfuerzos?

¿Qué estamos tratando de lograr?

¿Cuál es el horario para esto?

¿Cuál es el impacto para el negocio?

¿Cómo ven nuestros grupos de interés el valor?

¿Cuáles son los beneficios y costos de hacerlo?

DevOps se enfoca a ser un programa que abarque todos los aspectos del desarrollo y maduración del software, que intenta integrar, aria plataformas, equipos, tecnologías y herramientas. Mendoza (2018)

Según Mendoza (2018) una organización puede considerar la introducción de DevOps para atender a propósitos específicos, como los siguientes:

- Automatización de la infraestructura y gestión de la configuración del flujo de trabajo.
- Automatización de repositorios de código, compilaciones, pruebas y flujos de trabajo.
- Integración y despliegue continuo.
- Virtualización, contenedorización y balanceo de carga.
- Big data y proyectos de redes sociales.
- Proyectos de aprendizaje automático.

En virtud de lo anteriormente expuesto podemos aportar los beneficios que nos puede otorgar Devops, son:

- Acortar el ciclo de desarrollo
- Los procesos de DevOps se pueden implementar de forma independiente o como una combinación de otros procesos.
- El manual de operaciones está preparado en desarrollo para ayudar a las operaciones.
- Buscar acuerdos sin reducir la calidad
- En los procesos DevOps, el monitoreo específico de la aplicación es parte del proceso de desarrollo.
- Disminuir los fallos en el reléase.
- El equipo de operaciones de DevOps y los desarrolladores están familiarizados con el uso de marcos de registro.
- Los requisitos no funcionales de operabilidad, mantenimiento y monitoreo reciben la atención suficiente, junto con las especificaciones de desarrollo del sistema.
- Reducir tiempo de restaurar servicios

2.3. Test Driven Development (TDD)

Incluso después de varias décadas de avances en la industria del software, la calidad del software producido sigue siendo un problema. Teniendo en cuenta el enfoque de los últimos años en el tiempo de comercialización, el crecimiento en el gran volumen de software que se está desarrollando, y el flujo de nuevas tecnologías para absorber, no es de extrañar que las organizaciones de desarrollo de software hayan seguido enfrentando problemas de calidad. (Reyes, 2018)

Los defectos crean costos no deseados al hacer que el sistema sea inestable, impredecible o potencialmente completamente inutilizable. Reducen el valor del software a entregar, a veces hasta el punto de crear más daño que valor.

Es por esto que encontramos la forma para desarenos de los defectos y es a través de las pruebas: vemos si el software funciona, y luego intentamos dañarlo de alguna manera. Las pruebas se han establecido como un ingrediente crítico en el desarrollo de software, pero la forma en que se realizan las pruebas tradicionalmente, configura una fase de prueba prolongada después de que el código se haya estancado deja mucho espacio para mejora.

El desarrollo guiado por pruebas o TDD (por su sigla en inglés) emergió conjuntamente con el auge del proceso ágil Latevaweb (2018).Ambos tienen sus raíces en los modelos de procesos iterativos, incrementales y evolutivos utilizados como a principios de los años cincuenta. Además, las herramientas han evolucionado para configurar un papel importante en el apoyo a TDD.

Primero escribimos una prueba, luego escribimos código para pasar la prueba. Este enfoque para crear software fomenta un buen diseño, produce código comprobable y nos mantiene alejados de sobre-ingeniería nuestros sistemas debido a suposiciones erróneas y todo esto se logra con el simple hecho de conducir nuestro diseño en cada paso del camino con pruebas ejecutables que nos mueven hacia la implementación final.

TDD es una técnica por la cual se escribe un caso de prueba antes de escribir cualquier implementación de código

Tradicionalmente, las pruebas de la unidad ocurrían después de que los desarrolladores codificaran la unidad. Esto puede tomar desde unos pocos minutos a unos pocos meses. Con TDD, las pruebas unitarias podrán se escritas por el mismo programador o por un probador designado. Con TDD, el programador escribe pruebas unitarias previas al código bajo prueba.

TDD es una forma de programación que fomenta el buen diseño y nos ayuda a evitar errores de programación. Se enfoca en hacernos escribir pruebas pequeñas y automatizadas, que eventualmente construyen un sistema de alarma muy efectivo para proteger nuestro código de la regresión.

Esta técnica fomenta el desarrollo de código repitiendo ciclos cortos, hay diferentes actividades que se realizan entre la creación de pruebas unitarias y la refactorización de códigos. Estas actividades son:

- Desarrollo de pruebas unitarias: los desarrolladores deben desarrollar pruebas unitarias automatizadas basadas en requisitos.
- Ejecución de la prueba: todas las pruebas escritas se ejecutan y el resultado se comprueba para la prueba recién agregada como debería fallar.
- Escritura de código: el código se desarrolla para la prueba de modo que cuando se ejecuta, debe pasar. Este proceso también se puede llamar "implementación de escritura para la prueba unitaria".
- Prueba de código: después de implementar la prueba unitaria, todas las pruebas se ejecutan de nuevo y se garantiza que las pruebas se escriban
- Para nuevas funcionalidades debe pasar por la refactorización de código: este paso es garantizar la limpieza del código de cualquier duplicación

A manera de resumen podemos considerar que el ciclo corto de TDD simplifica en los siguientes enunciados: (i) escribir una prueba de unidad para una funcionalidad o comportamiento no implementado; (ii) suministro mínimo cantidad de código de producción para hacer pasar las pruebas unitarias; (iii) aplicar refactorización donde y cuando sea necesario; y (iv) comprobar que todas las pruebas aún se están superando después de la refactorización.

El breve ciclo de desarrollo que promueve TDD está bien orientado hacia la escritura de un código de alta calidad desde el principio. El ciclo corto es diferente de la forma en que estamos acostumbrados a la programación pues en principio hemos diseñado, luego hemos implementado el diseño para posteriormente probar esa implementación, por lo general no demasiado a fondo.

TDD cambia esta forma de pensar y dice que deberíamos escribir la prueba primero y solo luego escribir código para alcanzar ese objetivo claro. El diseño es lo que hacemos. Miramos el código que tenemos y encontramos el diseño más sencillo posible.

El último paso en el ciclo se llama refactorización y es una forma disciplinada de transformar el código de un estado o estructura a otro, eliminando la duplicación, y moviendo gradualmente el código hacia un mejor diseño.

TDD nos ayuda a acelerar al reducir el tiempo que toma arreglar los defectos. Generalmente podemos demorarnos en reparar un defecto dos meses después de su introducción en el sistema lo que conllevará, tiempo y dinero, mucho más que arreglarlo el mismo día que se presentó.

2.4. Pruebas Unitarias

Prueba unitaria: Una prueba unitaria, o "unit test", es un método que prueba una unidad estructural de código. Apiumhub (2019)

Contrariamente a lo que piensan muchos desarrolladores –que el desarrollo de pruebas unitarias resta tiempo a tareas más importante– las pruebas unitarias por lo general son simples y rápidas de codificar, el desarrollo de una prueba unitaria no debería tomar más de cinco minutos.

Debido a la diversidad de definiciones, convendremos que una "buena" prueba unitaria tiene las siguientes características:

- Unitaria: Prueba solamente pequeñas cantidades de código.
- Independiente: No debe depender ni afectar a otras pruebas unitarias.
- Prueba métodos públicos: de otra forma la prueba sería frágil a cambios en la implementación y no se podría utilizar en pruebas de regresión.
- Automatizable: La prueba no debería requerir intervención manual.
- Repetible y predecible, no debe incidir el orden y las veces que se repita la prueba, el resultado siempre debe ser el mismo.
- Profesionales: Las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc.

Respecto al último punto y contrariamente a lo que piensan muchos desarrolladores –que el desarrollo de pruebas unitarias resta tiempo a tareas más importante– las pruebas unitarias por lo general son simples y rápidas de codificar, el desarrollo de una prueba unitaria no debería tomar más de cinco minutos (Ragonha, 2013).

Por lo tanto, podemos definir a la prueba unitaria como: Prueba unitaria es prueba a través de la cual unidades del código fuente se prueba para su funcionalidad deseada. Una unidad se compone de módulo completo, función o procedimiento en alguna programación de procedimiento, o una interfaz completa como una clase en un objeto orientado.

En general, los desarrolladores de software desarrollan y ejecutan pruebas unitarias para probar el diseño esperado y el comportamiento del código generado. A diferencia del desarrollo de software tradicional,

TDD, como se mencionó en líneas anteriores, hace hincapié en el desarrollo de la primera prueba y se basa en la repetición de pruebas y desarrollo cortos o ciclos Beck, quien presentó patrones para escribir pruebas de unidad buenas y efectivas para TDD. Descripciones breves

De algunos patrones TDD son los siguientes:

- Prueba n: las pruebas unitarias se ejecutan muchas veces y necesitan más tiempo de ejecución iterativo (ejecutado por el desarrollador y más a menudo) para automatizar las pruebas unitarias.
- Prueba aislada: ejecute la prueba de la unidad de forma independiente para encontrar su comportamiento real y uno debe ser capaz de reordenarla.
- Prueba de afirmación: la prueba de unidad debe aclarar su propósito. Debe abordar la funcionalidad de pertenencia, gastos esperados.
- Datos de prueba: en la ejecución de prueba unitaria, utilice datos que los hacen fáciles de leer y comprender.
- Datos evidentes: uso de datos reales durante la prueba según sea necesario.

En un ciclo de desarrollo basado en pruebas, una prueba unitaria debe pasar por dos estados. Según beck estos estados son:

- Rojo: prueba el desarrollo y haz que falle.
- Verde: pasa la prueba por cualquier medio deseado

Hay requisitos mínimos para la elaboración de pruebas unitarias, entre las cuales se encuentran, quién debe hacerlo, cómo debe hacerlo, cuál será la documentación mínima necesaria, entre otros,

Debido a que las pruebas unitarias se centran principalmente en la implementación, y requiere un entendimiento de la intención del diseño, es mucho más eficiente que sea realizada por diseñadores

La documentación necesaria para la elaboración de pruebas unitarias:

- Debe ser revisable. Es decir, los registros deben ser suficientes para que otros revisen la adecuación de las pruebas.
- Debe ser suficiente para que las pruebas sean repetibles esto es importante para la regresión.
- La repetibilidad es también es importante para analizar fallos, tantas fallas durante la prueba inicial, y fallas posteriores. Sabiendo exactamente qué fue y no fue probado, y exactamente qué pasó y que falló durante la prueba, es una ayuda invaluable para aislar fallas de campo difíciles de producir. La repetibilidad no solo implica la necesidad de grabar en razonable detalle cómo se ejecuta la prueba y qué datos son utilizados, sino que también implica la identificación de la versión de código bajo prueba.
- Los registros deben ser archivables. Es decir, ellos deben estar suficientemente bien mantenidos e identificados, de tal forma de que se pueden encontrar si es necesario, en un momento posterior

2.4.1 Ventajas de las pruebas unitarias

Apiumhub (2019) menciona que las pruebas unitarias buscan aislar cada parte del programa y mostrar que las partes individuales son correctas, proporcionando varias:

- Fomentan el cambio, las pruebas unitarias facilitan la reestructuración del código (refactorización), puesto que permiten hacer pruebas sobre los cambios y verificar que las modificaciones no han introducido errores (regresión).
- Simplifican la integración, permiten llegar a la fase de integración asegurando que las partes individuales funcionan correctamente. De esta manera se facilitan las pruebas de integración.
- Documentan el código, las propias pruebas pueden considerarse documentación, ya que las mismas son una implementación de referencia de cómo utilizar el código.
- Separación de la interfaz y la implementación, la única interacción entre los casos de prueba y las unidades bajo prueba son las interfaces de estas últimas, se puede cambiar cualquiera de los dos sin afectar al otro (ver pruebas mock).
- Pueden mejorar el diseño, la utilización de prácticas de diseño y desarrollo dirigida por las pruebas (Test Driven Development o TDD) permite definir el comportamiento esperado en un paso previo a la codificación.

- La utilización de pruebas unitarias permitirá mejorar progresivamente la calidad del código en la medida que los desarrolladores aumenten la calidad de las pruebas y la cobertura del mismo, por ejemplo, "en MicroGestion durante el año 2011 se dedicaron 1160 horas a la resolución de errores en la fase de desarrollo y en el período de garantía, y en 2012 se registraron 710 horas" (Hahn, 2013).
- Dedicar menos tiempo a la depuración y pues se tendrá más confianza de los cambios en su código, esta confianza nos permitirá ser más agresivos con el código de refactorización y agregar nuevas características. Sin pruebas, es fácil volverse paranoico acerca de refactorizar o agregar nuevas funciones.
- Sin pruebas unitarias, el tiempo que lleva depurar o resolver una prueba funcional que puede haber fallado toma mucho tiempo para localizarla. Sin embargo, con la prueba unitaria, el alcance de la prueba se mantiene al mínimo y el punto en el que se puede desencadenar una falla se puede aislar más rápido.
- Las pruebas de unidad de escritura obligan a los desarrolladores a pensar cómo su código se va a utilizar y generalmente ha resultado en un mejor diseño y aún mejor documentación.
- Mejor mecanismo de retroalimentación: cuando todas las pruebas de unidad para un sistema se ejecutan en conjunto, el estado del sistema en su conjunto puede ser medido. Las pruebas unitarias también proporcionan a otros desarrolladores un mecanismo para evaluar si otras partes del código base cumplen con sus requisitos o todavía están en desarrollo. Los cambios en los entornos de prueba a veces pueden causar problemas con el código base y el informe continuo de los conjuntos de pruebas de unidad en su conjunto puede ayudar a indicar el estado del entorno de prueba.
- Un conjunto completo de pruebas de unidades permite refactorizaciones importantes y la reestructuración del código base se completará con mayor confianza mientras pruebas de unidad continúan pasando, los desarrolladores pueden estar seguros de que sus cambios en el código no tienen un impacto negativo en la funcionalidad de la aplicación en su conjunto.
- Muchos estudios han demostrado que cuesta mucho más dinero reparar un error encontrado más tarde en su lanzamiento que antes en su desarrollo. Un buen conjunto de pruebas de unidad descubrirá errores básicos al principio del ciclo de

desarrollo, reduciendo el potencial de otros errores y reduciendo costo de mantenimiento futuro.

CAPÍTULO 3
PRUEBAS UNITARIAS EN EL MARCO DEL DESARROLLO ÁGIL

3.1 Arquitectura orientada a integración continua

3.1.1 Integración continua

El desarrollo de software está lleno de mejores prácticas para lo cual es tener un proceso automatizado para ensamblar y probar versiones ejecutables de nuestro software, de manera que el equipo de desarrollo pueda construir y probar varias veces al día el software en que están trabajando. Este concepto es conocido como "integración continua" y es una de las prácticas de Extreme Programming, sin embargo no es necesario estar siguiendo Extreme Programming para aplicarla. (Fernández, 2017)

Martin Fowler es uno de los autores más reconocidos en el ámbito de prácticas ágiles y define la integración continua de la siguiente manera:

> "La integración continua es una práctica en la que miembros de un equipo integran su trabajo frecuentemente, típicamente cada persona íntegra al menos una vez al día, generando varias versiones por día. Cada versión ejecutable es verificada por un sistema automático de integración y pruebas para detectar errores de integración lo más rápido posible."

La integración continua consta de una serie de propuestas (o buenas prácticas) a seguir para que se desarrolle una aplicación software de la mejor manera posible.

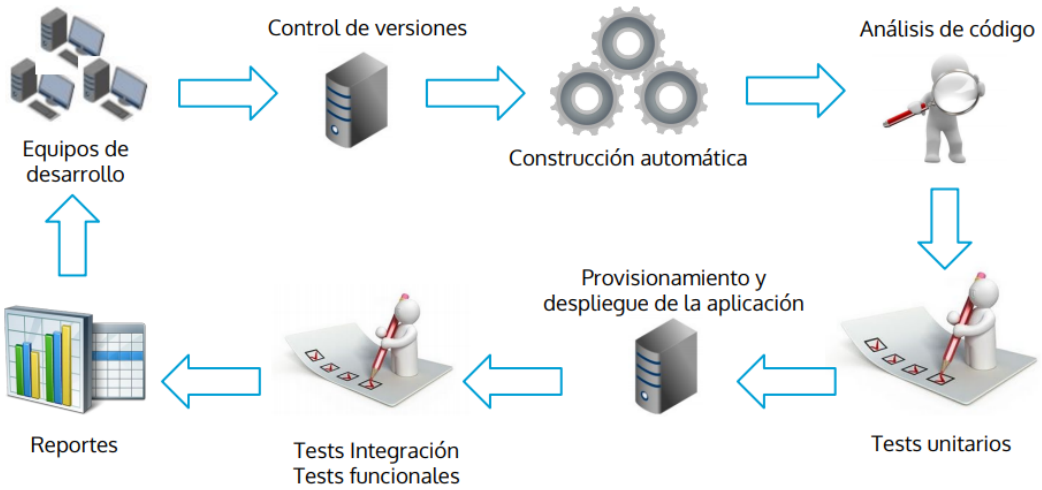

Figura 1. Flujo de trabajo de la integración continua

Fuente: (Capgemini, 2015)

Como se puede observar en la figura anterior en cada integración, se realiza lo siguiente: construcción automática de la aplicación, ejecución de pruebas automáticas y comprobación de la calidad del software a través de métricas predefinidas. Además abarca todo el ciclo de vida de construcción de la solución: compilación -> Test Unitarios -> Test Integración -> Test Funcionales - > Test QA -> Despliegue (puesta producción). (Capgemini, 2015)

3.1.2 Características y requisitos de la integración continua

La integración continua tiene una serie de propuestas(o buenas prácticas) a seguir para que se desarrolle las aplicaciones de mejor manera posible. Martin Fowler precursor de la integración continua destaca.

- Mantener un único repositorio de código fuente
- Automatizar la construcción del proyecto
- Hacer que la construcción del proyecto ejecute sus propios tests
- Entregar los cambios a la línea principal todos los días
- Construir la línea principal en la máquina de integración
- Mantener una ejecución rápida de la construcción del proyecto
- Probar en una réplica del entorno de producción
- Hacer que todo el mundo pueda obtener el último ejecutable de forma fácil
- Publicar qué está pasando
- Automatizar el despliegue

3.1.3 Beneficios de la integración continua

Según Capgemini (2015) afirma que el principal beneficio de la integración continua es la reducción del riesgo. Se puede predecir el tiempo de integración, puesto que es algo que se realiza de forma continua. También permite reducir la aparición de bugs, puesto que la realización constante de pruebas permite su pronta detección y corrección antes de que entren en producción. Ya que se dispone en todo momento de ejecutables del proyecto, esto

permite la rápida adopción por parte de los usuarios de las nuevas características añadidas al proyecto. Esto también permite que los usuarios valoren estos cambios y sugieran cambios nuevos de forma rápida.

3.2 Automatización de pruebas unitarias en el desarrollo ágil

Iniesta (2016) menciona que la automatización de pruebas unitarias en el desarrollo ágil, se le conoce como Agile Testing y se refiere a la práctica de pruebas de software que sigue los principios del desarrollo ágil de software que involucra a todos los miembros de un equipo ágil multifuncional, el cual debe garantizar el valor de negocio deseado por el cliente a un ritmo sostenible y continuo.

3.2.1 Características de Agile Testing

- El equipo es responsable de la calidad, y en cierto modo del testing con la finalidad de integrar la calidad al desarrollo del producto, al contrario de un enfoque de primero elaborar el producto y luego determinar su nivel de calidad.
- No se considera al testing como una fase separada del desarrollo software, sino como parte integral al igual que la programación.
- Todo el equipo realiza pruebas: los analistas de negocio y programadores de software también ejecutan pruebas, no sólo los testers como en las metodologías convencionales.
- Proporciona retroalimentación continua, permitiendo corregir el rumbo continuamente durante el desarrollo de software.
- Reduce el tiempo para recibir retroalimentación: los equipos del área de negocio (el cliente) están involucrados en cada iteración, no solo al final durante la fase de aceptación, con esto se reduce el tiempo de retroalimentación y el coste de correcciones.
- Incorpora una serie de prácticas tales como:
  - Integración continua
  - Test Driven Development (TDD): una técnica de diseño e implementación de software guiado por pruebas.
  - Desarrollo guiado por comportamiento Behaviour Driven Development (BDD).
  - Desarrollo guiado por pruebas de aceptación Acceptance Test Driven Development (ATDD). (Iniesta, 2016)

La siguiente pirámide ayuda a explicar las diferencias del testing trabajando con metodologías convencionales y trabajando con metodologías ágiles.

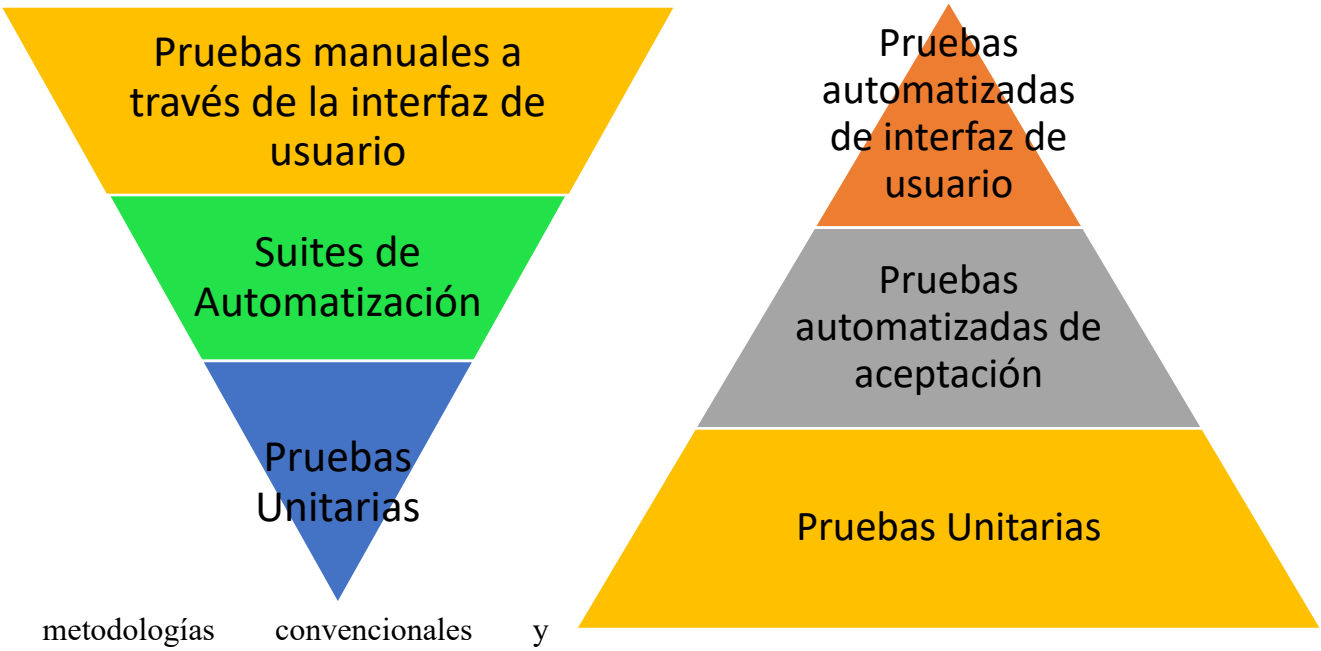

Figura 2. Diferencias del testing trabajando con metodologías convencionales y metodologías ágiles

Fuente: (Iniesta, 2016)

En la figura anterior se puede observar que la pirámide de la izquierda representa el *testing* en metodologías tradicionales, en la que la gran mayoría de las pruebas son manuales y funcionales, pudiendo existir algún pequeño grado de automatización y de pruebas unitarias.

La pirámide de la derecha representa como se debe enfocar el *testing* en metodologías ágiles, donde la mayoría de las pruebas son unitarias, siempre buscando reducir al mínimo las pruebas manuales.

3.2.2 Cuadrantes definidos en Agile Testing

La siguiente figura tiene como objetivo asegurar que todos los tipos de pruebas necesarias son cubiertas y en el que cada cuadrante nos ayuda detectar qué pruebas debemos realizar.

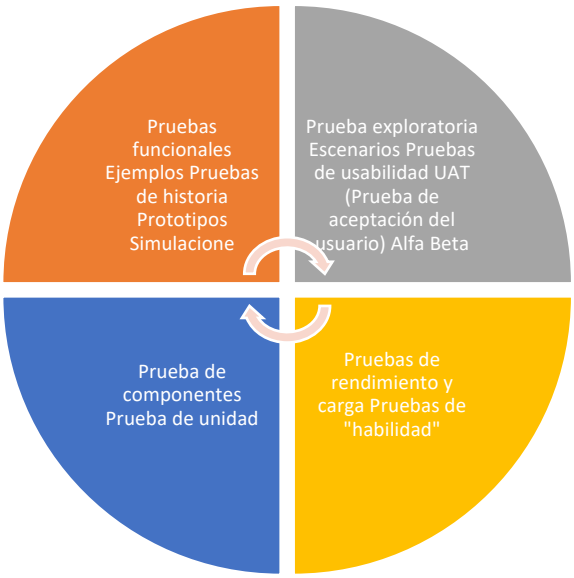

Figura 3. Cuadrantes definidos en Agile Testing

Fuente: (Iniesta, 2016)

Según Iniesta (2016) en calidad de *software* existen diferentes dimensiones, las cuales requieren un enfoque de pruebas diferente.

Cuadrante 1

- El propósito de este cuadrante es el desarrollo guiado por pruebas (TDD). Este proceso de escribir pruebas principalmente ayuda a los programadores a diseñar el código y además proporciona confianza al seguir avanzando con nuevas historias.
- Las pruebas unitarias verifican la funcionalidad de una pequeña parte del código, como métodos, objetos etc.
- Las pruebas de componentes verifican el comportamiento de una parte del sistema, como un grupo de clases que proporcionan algún servicio.

Cuadrante 2

- Está constituido por las pruebas llevadas a cabo por el equipo de desarrollo, pero a más alto nivel. Estas pruebas llamadas *"business-facing tests"*, las cuales prueban principalmente las especificaciones dadas por el cliente, es decir, lo que el cliente quiere. Escritas en lenguaje de "negocio".

Cuadrante 3

- Corresponde a las pruebas exploratorias.
- Pruebas manuales que solo un humano realiza.
- Normalmente, usuarios o clientes realizan este tipo de pruebas. Las pruebas de aceptación de usuario (UAT) dan a los clientes la oportunidad de proporcionar *feedback*.

Cuadrante 4

- Incluyen pruebas relacionadas con el rendimiento, carga, seguridad, etc.

3.3 Correlación entre DevOps y pruebas unitarias automatizadas

3.3.1 La cultura DevOps

Digital Business Assurange (2018) menciona que la cultura DevOps se ha convertido en el camino que permite optimizar el rendimiento de un proyecto en un contexto en el que los cambios se producen a un ritmo más rápido que los propios desarrollos. El concepto DevOps viene definido por varios factores, tales como trabajo en equipo, creatividad y automatización, siendo este último uno de los más relevantes, ya que la automatización afecta directamente a todos los procesos del proyecto.

Los diferentes proyectos DevOps utilizan procesos estandarizados, lo que permite automatizarlos para mejorar su fiabilidad y la calidad del software. De esta manera, la

automatización, combinada con otros principios de DevOps, facilita que los equipos puedan focalizarse en proporcionar valor a la entrega del software, siguiendo el primer principio del Manifiesto Agile: “Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor”.

3.3.2 Procesos de automatización

Según Digital Business Assurange (2018) los procesos de automatización se desarrollan en cuatro áreas:

3.3.2.1 Construcción automatizada

Se basa en técnicas de integración continua, con generación de builds automatizados que incluirán tanto los desarrollos de software como las pruebas que permitan localizar lo antes posible errores de código o en la propia integración de este. Aquí se utilizan herramientas como Jenkins, Bamboo o Team Foundation Build, apoyadas en otras herramientas de gestión y construcción de proyectos, como Ant, Maven, Nant o MSBUIL. (Digital Business Assurange, 2018)

3.3.2.2 Testing automatizado

La automatización de las pruebas es una de las partes básicas de los proyectos DevOps. En este aspecto no debemos pensar solo en la automatización de las pruebas funcionales, sino a todos los niveles.

Automatización de pruebas unitarias: Son las más voluminosas. Cada clase o método tendrá que disponer de su prueba automatizada.

Automatización de Servicios: Pruebas serán muy estables, existirán pocas modificaciones y asegurarán la integración del software. Herramientas como SoapUI o JMeter.

Automatización de pruebas funcionales: Se definirán regresiones para asegurar el cumplimiento de los requisitos de negocio solicitados.

Automatización de pruebas de rendimiento: Automatizan pruebas de carga y estrés, observando la respuesta del software ante un número de peticiones determinado y ante situaciones extremas

3.3.2.3 Despliegue automatizado

Para poder llevar a cabo la política en los proyectos DevOps es necesario que estos procesos sean automatizados. Hay que pensar que serán despliegues constantes, con pequeños cambios, y que deben ser rápidos y fiables. Es preciso automatizar tanto las subidas de código como las modificaciones en base de datos. Con este fin aparecen herramientas como XL Deploy o Clarive Lean Application Delivery. (Digital Business Assurange, 2018)

3.3.2.4 Aprovisionamiento automatizado

El aprovisionamiento automatizado de los sistemas permite que todos los interlocutores dispongan del mismo sistema en el mismo momento en el tiempo. Para ello se vigila la construcción de las máquinas y se mantiene el control sobre los paquetes instalados, cuentas de usuario, configuraciones, etc. En este caso, se utilizan herramientas como Chef, Puppet o Salt. (Digital Business Assurange, 2018)

Cabe destacar que la automatización es uno de los pilares fundamentales en los que se basan los principios de DevOps. Invertir en la automatización de los procesos, liberando a los equipos de tareas que no aportan valor y que les permitan orientar sus esfuerzos hacia la mejora del producto final, sería un buen comienzo para las organizaciones que tienen como objetivo llegar a ser DevOps. (Digital Business Assurange, 2018)

3.4 De historias de usuario a pruebas unitarias

Villamizar, Tabares, & Zapata (2015) afirman que una historia de usuario es el resultado de conversaciones entre los interesados del proyecto, los analistas de negocios, los encargados de pruebas y los desarrolladores.

> Una historia de usuario es una representación de un requisito de software escrito en una o dos frases utilizando el lenguaje común del interesado. Las historias de

usuario se utilizan en las metodologías de desarrollo ágil para la especificación de requisitos (acompañadas de las discusiones con los usuarios y las pruebas de validación). Cada historia de usuario debe ser limitada, pues se debería poder escribir sobre una nota adhesiva pequeña. Dentro de la metodología XP, los clientes deberían escribir las historias de usuario.

Las historias de usuario son utilizadas en los métodos ágiles para especificar los requisitos de una aplicación de software. El desarrollo dirigido por pruebas (TDD Test Driven Development) que consiste en una técnica usada en los métodos ágiles para generar pruebas unitarias automáticas basadas en las historias de usuario.

A partir de las historias de usuario se generan unas pruebas de aceptación (preguntas restringidas para validar la información con el interesado), que son afirmaciones en lenguaje natural. Luego, el equipo de desarrollo hace el esfuerzo de conectar esas frases con los puntos de entrada y salida del código. (Villamizar, Tabares, & Zapata, 2015)

CAPÍTULO 4
IMPLEMENTACIÓN ORIENTADA AL CASO DE ESTUDIO

4.1 Análisis del diseño del caso de estudio

La parte fundamental del diseño de este proyecto se basa en el uso de archivos de diseño y especificaciones, para este análisis tomaremos como ejemplo la funcionalidad "repositorioDeTarjetas" que se encuentra dentro de la capa de dominio.

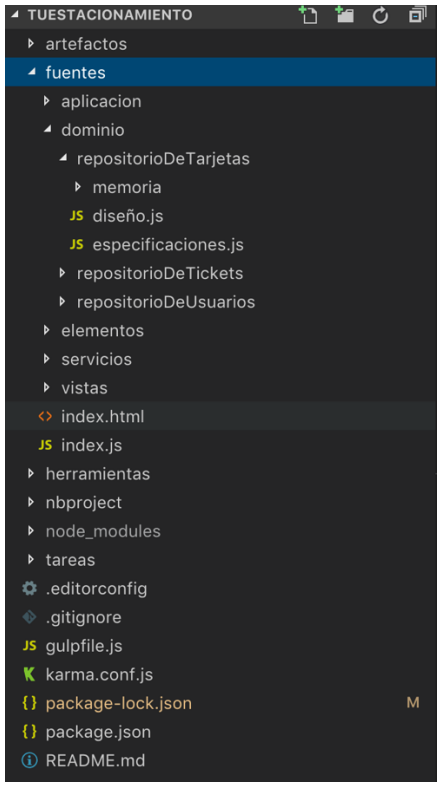

Figura 4. Ubicación de los archivos diseño.js y especificaciones.js a ser analizados

Los archivos de diseño nos permiten definir la estructura de las clases a ser implementadas, se las puede considerar clases base a ser extendidas con una implementación, estos archivos se los puede asociar con interfaces en lenguajes orientados a objetos para el caso de este proyecto el lenguaje utilizado es Javascript al no ser un lenguaje puramente orientado a objetos los diseños e implementaciones son definiciones de prototipos

que las clases (objetos) de implementación usaran como prototipo para su construcción, por ejemplo:

```
1.  var TuEstacionamiento;
2.  (function (TuEstacionamiento) {
3.      var Dominio;
4.      (function (Dominio) {
5.          var Repositorios;
6.          (function (Repositorios) {
7.              var Tarjetas;
8.              (function (Tarjetas) {
9.
10.                 Tarjetas.Diseño = {
11.                     obtenerTarjeta: function (datosDeTarjeta) {
12.                         throw Error('Método no implementado');
13.                     },
14.                     salvarTarjeta: function (tarjetaASalvar) {
15.                         throw Error('Método no implementado');
16.                     }
17.                 };
18.
19.             })(Tarjetas = Repositorios.Tarjetas || (Repositorios.Tarjetas = {}));
20.         })(Repositorios = Dominio.Repositorios || (Dominio.Repositorios = {}));
21.     })(Dominio = TuEstacionamiento.Dominio || (TuEstacionamiento.Dominio = {}));
22. })(TuEstacionamiento || (TuEstacionamiento = {}));
```

Figura 5. Código del archivo tuEstacionamiento/fuentes/dominio/repositorioDeTarjetas/diseño.js

El archivo diseño.js define un objeto de Javascript cuyo prototipo tendrá la estructura TuEstacionamiento.Dominio.Repositorios.Tarjetas.Diseño esta es la misma estructura que las carpetas que lo contienen, esto lo podemos evidenciar en todo el código del presente proyecto y además de una buena práctica es un concepto que nos ayuda a la mantenibilidad del proyecto.

Dentro del objeto TuEstacionamiento.Dominio.Repositorios.Tarjetas.Diseño se definen los métodos obetenerTarjeta (línea 10.) y salvarTarjeta (línea 14.) estos dos métodos son las funciones que el repositorioDeTarjetas debe implementar posteriormente, es así que el diseño en su definición solo tiene un “throw” de una excepción de “Método no implementado” en el caso de que el método no se sobrescriba en la implementación, con esto un arquitecto puede diseñar sus historias de usuarios en un alto nivel en conjunto con las especificaciones de las pruebas y delegar la implementación a sus desarrolladores.

Por otro lado, los archivos de especificaciones nos permiten definir la estructura de las pruebas a ser implementadas, este es un archivo de vinculo entre el diseño y las pruebas que cada método debe implementar y cumplir, las especificaciones se reflejan como casos de prueba de las historias de usuario, estos casos de prueba pueden ser casos triviales como el comportamiento esperando de cierta funcionalidad, así como casos bordes, por ejemplo:

```
1.  var TuEstacionamiento;
2.  (function (TuEstacionamiento) {
3.      var Dominio;
4.      (function (Dominio) {
5.          var Repositorios;
6.          (function (Repositorios) {
7.              var Tarjetas;
8.              (function (Tarjetas) {
9.
10.                 Tarjetas.Especificaciones = {
11.                     nombre: 'TuEstacionamiento.Dominio.Repositorios.Tarjetas',
12.                     obtenerTarjeta: {
13.                         nombre: 'obtenerTarjeta',
14.                         DeberiaNoPoderObtenerUnaTarjetaNoExistenteEnLaDataDelRepositorio:
    'Deberia no poder obtener una tarjeta no existente en la data del repositorio',
15.                         DeberiaPoderObtenerUnaTarjetaExistenteEnLaDataDelRepositorio:
    'Deberia poder obtener una tarjeta existente en la data del repositorio'
16.                     },
17.                     salvarTarjeta: {
18.                         nombre: 'salvarTarjeta',
19.                         DeberiaPoderSalvarUnaTarjetaNoExistenteEnLaDataDelRepositorio:
    'Deberia poder salvar una tarjeta no existente en la data del repositorio',
20.                         DeberiaPoderActualizarUnaTarjetaExistenteEnLaDataDelRepositorio:
    'Deberia poder actualizar una tarjeta existente en la data del repositorio'
21.                     }
22.                 };
23.
24.             })(Tarjetas = Repositorios.Tarjetas || (Repositorios.Tarjetas = {}));
25.         })(Repositorios = Dominio.Repositorios || (Dominio.Repositorios = {}));
26.     })(Dominio = TuEstacionamiento.Dominio || (TuEstacionamiento.Dominio = {}));
27. })(TuEstacionamiento || (TuEstacionamiento = {}));
```

Figura 6. Código archivo tuEstacionamiento/fuentes/dominio/repositorioDeTarjetas/especificaciones.js

El archivo de especificaciones declara un objeto para ser usado como prototipo dentro de la clase que lo implemente es así como cada método del diseño tiene definidas las pruebas que deberán implementarse, el método obtenerTarjeta en la línea 12. declara que debe tener

dos especificaciones DeberiaNoPoderObtenerUnaTarjetaNoExistenteEnLaDataDelRepositorio y DeberiaPoderObtenerUnaTarjetaExistenteEnLaDataDelRepositorio.

Estas declaraciones deben ser lo mas claras y precisas a razón de que el implementador de dichas pruebas entienda cual es el propósito de las mismas, para el primer caso es claro que un repositorio de tarjetas no puede obtener una tarjeta que no existe en el repositorio de datos y en el segundo caso es lo contrario, si una tarjeta existe en el repositorio este debe ser capaz de obtener dicha tarjeta.

Con estas especificaciones y diseños se puede pasar a la implementación de los métodos definidos en los archivos de diseño y también a la implementación de las pruebas definidas en los archivos de especificaciones, el uso de este diseño nos permite tener varias implementaciones de un mismo diseño/especificación en este caso de análisis el repositoriodeTarjetas tendrá una implementación en memoria, así:

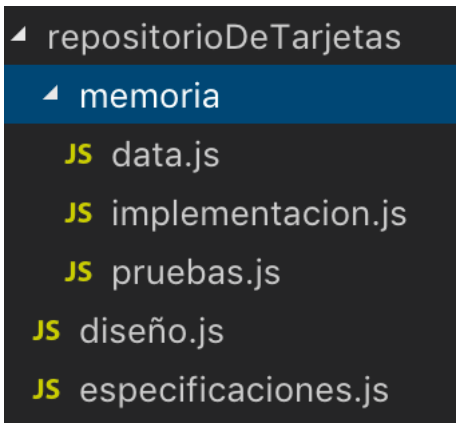

Figura 7. Estructura de la implementación de la funcionalidad repositorioDeTarjetas

Cada implementación deberá tener sus respectivas pruebas y en el caso de necesitarse archivos adicionales para la implementación estos deberán ser parte de la carpeta que contiene la implementación. El mantener organizado el código de esta manera nos permite estar preparados para en el caso de que una funcionalidad cambie sus requerimientos no es necesario desechar y sobrescribir la anterior implementación en lugar de eso se puede crear una implementación en paralelo que cumpla con las nuevas reglas del negocio y dichos cambios serán fáciles de integrar.

4.2 Planteamiento de la arquitectura

Para el caso de estudio de cartera de tickets de parqueadero se utilizó un diseño de 3 capas: dominio, aplicación y vistas.

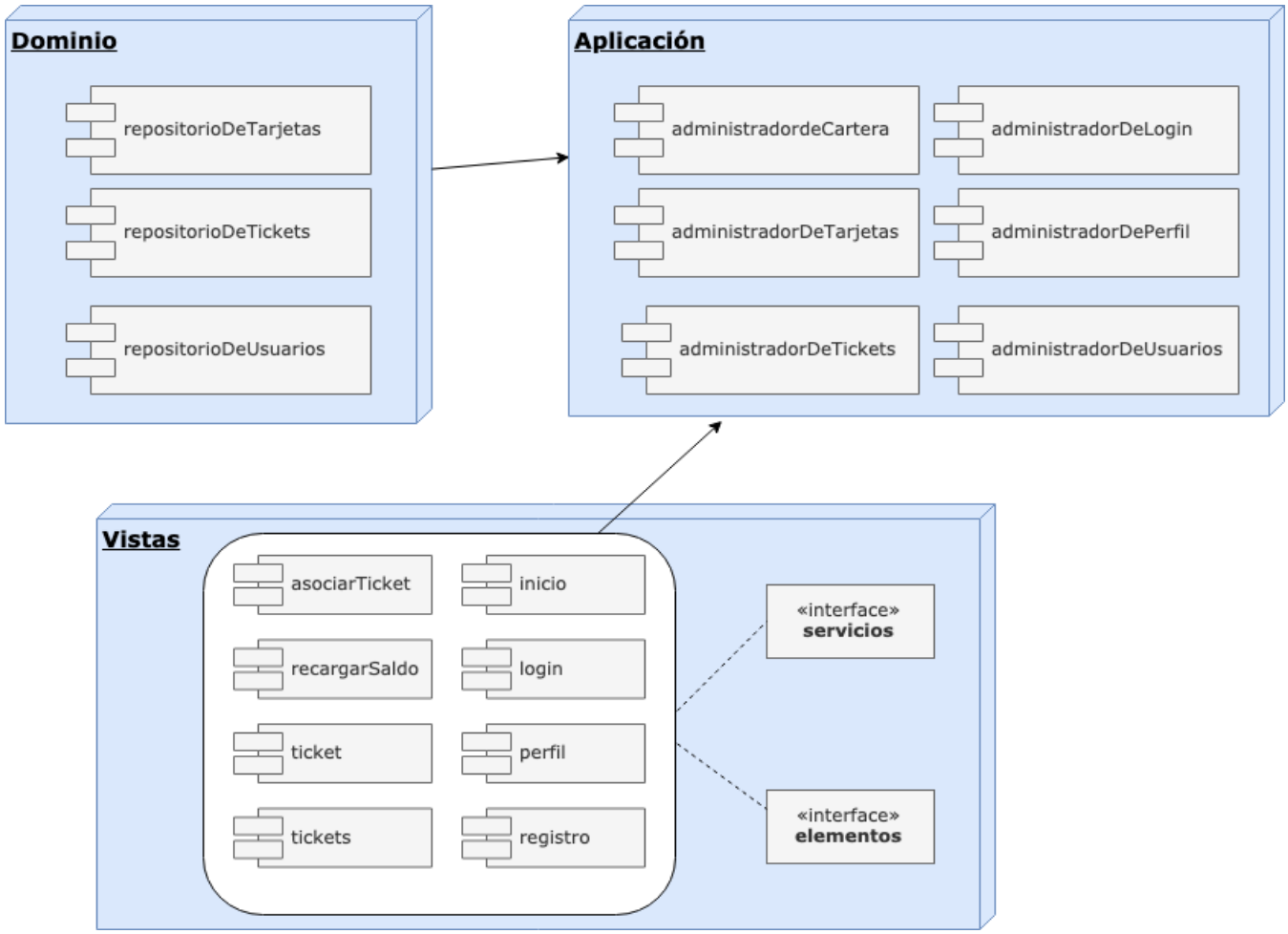

Figura 8. Diagrama de paquetes de la arquitectura del Sistema de cartera de tickets de parqueadero

La capa de dominio se encarga de la interacción con el/los orígenes de datos, la implementación de un repositorio consiste en definir la funcionalidad de los métodos descritos en el diseño usando el repositorio de datos para persistir las acciones a realizarse, por ejemplo:

```
1. var TuEstacionamiento;
2. (function (TuEstacionamiento, angularJs) {
3.     var Dominio;
4.     (function (Dominio) {
5.         var Repositorios;
6.         (function (Repositorios) {
7.             var Tarjetas;
8.             (function (Tarjetas) {
9.                 (function (Implementaciones) {
10.
11.                     Implementaciones.Memoria = Implementacion;
12.
13.                     function Implementacion(dataDeTarjetas) {
14.                         this.obtenerTarjeta = obtenerTarjeta;
15.                         this.salvarTarjeta = salvarTarjeta;
16.
17.                         function obtenerTarjeta(datosDeTarjeta) {
18.                             for (var i = 0; i < dataDeTarjetas.length; i++) {
19.                                 if (tarjetaExisteEnElRepositorio(i, datosDeTarjeta)) {
20.                                     return dataDeTarjetas[i];
21.                                 }
22.                             }
23.                             throw Error('Tarjeta inexistente');
24.
25.                             function tarjetaExisteEnElRepositorio(indice,
    datosDeTarjetaAComprobar) {
26.                                 return dataDeTarjetas[indice].datos.numeroDeTarjeta ===
    datosDeTarjetaAComprobar.numeroDeTarjeta
27.                                     &&
    dataDeTarjetas[indice].datos.nombreDelTarjetaHabiente ===
    datosDeTarjetaAComprobar.nombreDelTarjetaHabiente
28.                                     && dataDeTarjetas[indice].datos.codigoDeSeguridad
    === datosDeTarjetaAComprobar.codigoDeSeguridad
29.                                     && dataDeTarjetas[indice].datos.fechaDeVencimiento
    === datosDeTarjetaAComprobar.fechaDeVencimiento;
30.
31.                             }
32.                         }
33.                         function salvarTarjeta(tarjetaASalvar) {
34.                             try {
35.                                 var tarjetaEnRepositorio =
    obtenerTarjeta(tarjetaASalvar.datos);
```

```
                                tarjetaEnRepositorio = tarjetaASalvar;

                            } catch (error) {
                                dataDeTarjetas.push(tarjetaASalvar);
                            }
                        }
                    }

                    Implementacion.prototype = Object.create(Tarjetas.Diseño);

                    if (angularJs) {
                        angular.module(Tarjetas.Especificaciones.nombre, [
                            'TuEstacionamiento.Dominio.Repositorios.Tarjetas.Data'
                        ]).service('RepositorioDeTarjetas', Implementacion);

                        Implementacion.$inject = ['DataDeTarjetas'];
                    }

                })(Implementaciones = Tarjetas.Implementaciones ||
    (Tarjetas.Implementaciones = {}));
            })(Tarjetas = Repositorios.Tarjetas || (Repositorios.Tarjetas = {}));
        })(Repositorios = Dominio.Repositorios || (Dominio.Repositorios = {}));
    })(Dominio = TuEstacionamiento.Dominio || (TuEstacionamiento.Dominio = {}));
})(TuEstacionamiento || (TuEstacionamiento = {}), angular);

```

Figura 9. Código archivo tuEstacionamiento/fuentes/dominio/repositorioDeTarjetas/memoria/implementacion.js

En este archivo se puede define la implementación de la funcionalidad de los métodos obtenerTarjeta y salvarTarjeta con el uso de un repositorio de datos en memoria, en el momento que se finaliza con las definiciones se define la propiedad prototype usando el diseño del archivo diseño.js correspondiente, luego dicho repositorio se lo puede inyectar como parte de un módulo.

La capa de aplicación es la capa encargada de implementar las historias de usuario con el uso de administradores que son especies de servicios que usando repositorios definen funciones que reciben una entrada, aplican una función (lógica del negocio) y generan una salida la cual cumpliría con la historia de usuario.

La capa de vistas es la capa encargada de implementar las interfaces del usuario, esta capa usa dos componentes importantes los elementos que se los puede asociar con componentes de UI los cuales definen sus propios controladores y exponen dichos

controladores a través de un modelo, en esta capa también se definen las plantillas HTML de dichos componentes.

4.3 Implementación de historias de usuario a pruebas unitarias.

Una vez definida una historia de usuario que refleje precisamente los requerimientos del usuario se procede a la definición del diseño y las especificaciones para este análisis tomaremos como ejemplo la historia de usuario "Administrar Tickets":

Historias de usuario administrador de tickets:

Como administrador de la cartera de tickets de parqueadero quiero asociar un ticket a un usuario para persistir el vinculo entre un ticket existente y un usuario existente. Esta funcionalidad deberá cumplir con el siguiente diseño:

```
asociarTicketAUnUsuario: function (idDelTicketAAsociar, idDelUsuario)
```

Criterios de aceptación:

- Se debe validar que el ticket tenga estado "ACTIVO" para poder ser asociado a un usuario.
- Se debe validar que el usuario exista utilizando el parámetro idDelUsuario se lo puede buscar en el repositorio.

Como administrador de la cartera de tickets de parqueadero quiero procesar el pago de un ticket para generar el consumo en la cartera del usuario y procesar el estado del ticket de a "PAGADO". Esta funcionalidad deberá cumplir con el siguiente diseño:

```
pagarTicket: function (idDelTicket, idDelUsuario)
```

Criterios de aceptación:

- La funcionalidad debe usar el administrador de carteras de usuarios para generar el consumo en la cartera.
- Una vez generado el consumo se debe actualizar el estado del ticket a "PAGADO" y persistirlo.
- Se debe validar que no se pueda pagar un ticket que no tenga estado "ACTIVO".
- Se debe validar que se pueda procesar el consumo si y solo si el usuario tiene saldo suficiente en su cartera.

El archivo de diseño debe reflejar exactamente las funcionalidades que la historia de usuario debe cumplir, así:

```
1.  var TuEstacionamiento;
2.  (function (TuEstacionamiento) {
3.      var Aplicacion;
4.      (function (Aplicacion) {
5.          var AdministradorDeTickets;
6.          (function (AdministradorDeTickets) {
7.
8.              AdministradorDeTickets.Diseño = {
9.                  asociarTicketAUnUsuario: function (idDelTicketAAsociar, idDelUsuario) {
10.                     throw Error('Método no implementado');
11.                 },
12.                 buscarTodosLosTicketsDeUnUsuarioPorEstatus: function (idDelUsuario,
    estatusDelTicket) {
13.                     throw Error('Método no implementado');
14.                 },
15.                 obtenerTicket: function (idDelTicket) {
16.                     throw Error('Método no implementado');
17.                 },
18.                 pagarTicket: function (idDelTicket, idDelUsuario) {
19.                     throw Error('Método no implementado');
20.                 }
21.             };
22.
23.         })(AdministradorDeTickets = Aplicacion.AdministradorDeTickets ||
    (Aplicacion.AdministradorDeTickets = {}));
24.     })(Aplicacion = TuEstacionamiento.Aplicacion || (TuEstacionamiento.Aplicacion = {}));
25. })(TuEstacionamiento || (TuEstacionamiento = {}));
26.
```

Figura 10. Código archivo tuEstacionamiento/fuentes/aplicacion/administradorDeTickets/ diseño.js

Como se puede observar entre la línea 9 a la 20 se definen 4 funciones como parte del diseño: asociarTicketAUnUsuario, buscarTodosLosTicketsDeUnUsuarioPorEstatus, obtenerTciket, pagarTciket.

Una vez definido el diseño el siguiente paso seria definir las especificaciones, anteriormente se mencionó que las especificaciones son el vínculo entre las hostorias de usuario y las pruebas unitarias, por ejemplo:

```
var TuEstacionamiento;
(function (TuEstacionamiento) {
    var Aplicacion;
    (function (Aplicacion) {
        var AdministradorDeTickets;
        (function (AdministradorDeTickets) {

            AdministradorDeTickets.Especificaciones = {
                nombre: 'TuEstacionamiento.Aplicacion.AdministradorDeTickets',
                asociarTicketAUnUsuario: {
                    nombre: 'asociarTicketAUnUsuario',
                    DeberiaNoPoderAsociarUnTicketNoActivoAUnUsuarioExistente: 'Deberia no poder asociar un ticket no activo a un usuario existente',
                    DeberiaPoderAsociarUnTicketActivoAUnUsuarioExistente: 'Deberia poder asociar un ticket activo a un usuario existente'

                }, buscarTodosLosTicketsDeUnUsuarioPorEstatus: {
                    nombre: 'buscarTodosLosTicketsDeUnUsuarioPorEstatus',
                    DeberiaPoderBuscarLosTicketsAsociadosAUnUsuarioPorStatus: 'Deberia poder buscar los tickets asociados a un usuario por status'

                }, obtenerTicket: {
                    nombre: 'obtenerTicket',
                    DeberiaPoderObtenerUnTicketExistente: 'Deberia poder obtener un ticket existente'

                },
                pagarTicket: {
                    nombre: 'pagarTicket',
                    DeberiaNoPoderPagarUnTicketNoActivo: 'Deberia no poder pagar un ticket no activo',
                    DeberiaPoderPagarUnTicketActivoSiElUsuarioTieneSaldoSuficienteEnSuCartera: 'Deberia poder pagar un ticket activo si el usuario tiene saldo suficiente en su cartera'

                }
            };

        })(AdministradorDeTickets = Aplicacion.AdministradorDeTickets || (Aplicacion.AdministradorDeTickets = {}));
    })(Aplicacion = TuEstacionamiento.Aplicacion || (TuEstacionamiento.Aplicacion = {}));
})(TuEstacionamiento || (TuEstacionamiento = {}));

```

Figura 11. Código del archivo tuEstacionamiento/fuentes/aplicacion/administradorDeTickets/especificaciones.js

Como se puede observar las pruebas que se especifican para cada método son:

1. **asociarTicketAUnUsuario:**
   1.1. DeberiaNoPoderAsociarUnTicketNoActivoAUnUsuarioExistente
   1.2. DeberiaPoderAsociarUnTicketActivoAUnUsuarioExistente
2. **buscarTodosLosTicketsDeUnUsuarioPorEstatus:**
   2.1. DeberiaPoderBuscarLosTicketsAsociadosAUnUsuarioPorStatus
3. **obtenerTicket:**
   3.1. DeberiaPoderObtenerUnTicketExistente
4. **pagarTicket:**
   4.1. DeberiaNoPoderPagarUnTicketNoActivo

Cada una de las pruebas tiene definido su mensaje de error el cual debe ser claro y descriptivo ya que este será el mensaje para usar en el caso de que la prueba falle al momento de ejecutar dichas pruebas. Una vez definidas se puede pasar a la implementacion de las pruebas unitarias, así:

```
. (function (especificaciones, administradorDeTicketsEstandar, dependencias) {
.
.    describe(especificaciones.nombre, function () {
.        describe(especificaciones.asociarTicketAUnUsuario.nombre, function () {
.
 it(especificaciones.asociarTicketAUnUsuario.DeberiaNoPoderAsociarUnTicketNoActivoAUnUsuarioExistente,
 function () {
.                //Arrange
.                var repositorioDeTicketsMock = Object.create(dependencias.repositorioDeTickets),
.                        repositorioDeUsuariosMock = Object.create(dependencias.repositorioDeUsuarios),
.                        administradorDeTickets = new
 administradorDeTicketsEstandar(repositorioDeTicketsMock, repositorioDeUsuariosMock),
0.                                          dataDePrueba = {
1.                                              idDelUsuario: 'gach87',
2.                                              idDeTicket: '0'
3.                                          };
4.                                  spyOn(repositorioDeTicketsMock,
 'obtenerTicket').and.callFake(function () {
5.                                      return {
6.                                          id: "0",
7.                                          status: 'pagado',
8.                                          horaDeEntrada: new Date().toTimeString(),
9.                                          horaDePago: new Date().toTimeString(),
0.                                          horaDeSalida: '',
1.                                          total: '290',
2.                                          moneda: 'Bs'
3.                                      };
4.                                  });
5.                                  //Act
6.                                  //Assert
7.                                  expect(function () {
```

```
                                    return
 administradorDeTickets.asociarTicketAUnUsuario(dataDePrueba.idDeTicket, dataDePrueba.idDelUsuario);
                                }).toThrowError('Solo pueden asociarse tickets activos');

 expect(repositorioDeTicketsMock.obtenerTicket).toHaveBeenCalledWith(dataDePrueba.idDeTicket);
                                expect(repositorioDeTicketsMock.obtenerTicket.calls.count()).toBe(1);
                            });

 it(especificaciones.asociarTicketAUnUsuario.DeberiaPoderAsociarUnTicketActivoAUnUsuarioExistente,
 function () {
                                //Arrange
                                var repositorioDeTicketsMock =
 Object.create(dependencias.repositorioDeTickets),
                                    repositorioDeUsuariosMock =
 Object.create(dependencias.repositorioDeUsuarios),
                                    administradorDeTickets = new
 administradorDeTicketsEstandar(repositorioDeTicketsMock, repositorioDeUsuariosMock),
                                    dataDePrueba = {
                                        idDelUsuario: 'gach87',
                                        idDeTicket: '0'
                                    };
                                spyOn(repositorioDeTicketsMock,
 'obtenerTicket').and.callFake(function () {
                                    return {
                                        id: "0",
                                        status: 'activo',
                                        horaDeEntrada: new Date().toTimeString(),
                                        horaDePago: '',
                                        horaDeSalida: '',
                                        total: '',
                                        moneda: 'Bs'
                                    };
                                });
                                spyOn(repositorioDeTicketsMock, 'salvarTicket');
                                //Act
                                //Assert
                                expect(function () {
                                    return
 administradorDeTickets.asociarTicketAUnUsuario(dataDePrueba.idDeTicket, dataDePrueba.idDelUsuario);
                                }).not.toThrow();

 expect(repositorioDeTicketsMock.obtenerTicket).toHaveBeenCalledWith(dataDePrueba.idDeTicket);
                                expect(repositorioDeTicketsMock.obtenerTicket.calls.count()).toBe(1);
                                expect(repositorioDeTicketsMock.salvarTicket).toHaveBeenCalled();
                                expect(repositorioDeTicketsMock.salvarTicket.calls.count()).toBe(1);
                            });
                        });
                        describe(especificaciones.buscarTodosLosTicketsDeUnUsuarioPorEstatus.nombre,
 function () {
```

```
6.
 it(especificaciones.buscarTodosLosTicketsDeUnUsuarioPorEstatus.DeberiaPoderBuscarLosTicketsAsociadosAU
 nUsuarioPorStatus, function () {
7.                                  //Arrange
8.                                  var repositorioDeTicketsMock =
 Object.create(dependencias.repositorioDeTickets),
9.                                      repositorioDeUsuariosMock =
 Object.create(dependencias.repositorioDeUsuarios),
0.                                      administradorDeTickets = new
 administradorDeTicketsEstandar(repositorioDeTicketsMock, repositorioDeUsuariosMock),
1.                                      dataDePrueba = {
2.                                          idDeUsuario: 'gach87',
3.                                          status: 'activo'
4.                                      };
5.                                  spyOn(repositorioDeTicketsMock, 'buscarTicketsPorUsuarioYEstatus');
6.                                  //Act
7.                                  //Assert
8.                                  expect(function () {
9.                                      return
 administradorDeTickets.buscarTodosLosTicketsDeUnUsuarioPorEstatus(dataDePrueba.idDeUsuario,
 dataDePrueba.status);
0.                                  }).not.toThrow();
1.
 expect(repositorioDeTicketsMock.buscarTicketsPorUsuarioYEstatus).toHaveBeenCalledWith(dataDePrueba.idD
 eUsuario, dataDePrueba.status);
2.
 expect(repositorioDeTicketsMock.buscarTicketsPorUsuarioYEstatus.calls.count()).toBe(1);
3.                              });
4.                          });
5.                          describe(especificaciones.obtenerTicket, function () {
6.                              it(especificaciones.obtenerTicket.DeberiaPoderObtenerUnTicketExistente,
 function () {
7.                                  //Arrange
8.                                  var repositorioDeTicketsMock =
 Object.create(dependencias.repositorioDeTickets),
9.                                      repositorioDeUsuariosMock =
 Object.create(dependencias.repositorioDeUsuarios),
0.                                      administradorDeTickets = new
 administradorDeTicketsEstandar(repositorioDeTicketsMock, repositorioDeUsuariosMock), dataDePrueba = {
1.                                          idDeTicket: '0'
2.                                  };
3.                                  spyOn(repositorioDeTicketsMock, 'obtenerTicket');
4.                                  //Act
5.                                  //Assert
6.                                  expect(function () {
7.                                      return
 administradorDeTickets.obtenerTicket(dataDePrueba.idDeTicket);
8.                                  }).not.toThrow();
9.
 expect(repositorioDeTicketsMock.obtenerTicket).toHaveBeenCalledWith(dataDePrueba.idDeTicket);
```

```
00.                                 expect(repositorioDeTicketsMock.obtenerTicket.calls.count()).toBe(1);
01.                             });
02.                         });
03.                         describe(especificaciones.pagarTicket.nombre, function () {
04.                             it(especificaciones.pagarTicket.DeberiaNoPoderPagarUnTicketNoActivo,
 function () {
05.                                 //Arrange
06.                                 var repositorioDeTicketsMock =
 Object.create(dependencias.repositorioDeTickets), repositorioDeUsuariosMock =
 Object.create(dependencias.repositorioDeUsuarios), administradorDeTickets = new
 administradorDeTicketsEstandar(repositorioDeTicketsMock, repositorioDeUsuariosMock), dataDePrueba = {
07.                                     idDeTicket: '0'
08.                                 };
09.                                 spyOn(repositorioDeTicketsMock,
 'obtenerTicket').and.callFake(function () {
10.                                     return  {
11.                                         id: "0",
12.                                         status: 'pagado',
13.                                         horaDeEntrada: new Date().toTimeString(),
14.                                         horaDePago: '',
15.                                         horaDeSalida: '',
16.                                         total: '',
17.                                         moneda: 'Bs'
18.                                     };
19.                                 });
20.                                 //Act
21.                                 //Assert
22.                                 expect(function () {
23.                                     return
 administradorDeTickets.pagarTicket(dataDePrueba.idDeTicket);
24.                                 }).toThrowError('Solo pueden pagarse tickets activos');
25.
 expect(repositorioDeTicketsMock.obtenerTicket).toHaveBeenCalledWith(dataDePrueba.idDeTicket);
26.                                 expect(repositorioDeTicketsMock.obtenerTicket.calls.count()).toBe(1);
27.                             });
28.
 it(especificaciones.pagarTicket.DeberiaPoderPagarUnTicketActivoSiElUsuarioTieneSaldoSuficienteEnSuCart
 era, function () {
29.                                 //Arrange
30.                                 var repositorioDeTicketsMock =
 Object.create(dependencias.repositorioDeTickets),
31.                                     repositorioDeUsuariosMock =
 Object.create(dependencias.repositorioDeUsuarios),
32.                                     administradorDeTickets = new
 administradorDeTicketsEstandar(repositorioDeTicketsMock, repositorioDeUsuariosMock),
33.                                     dataDePrueba = {
34.                                         usuario: {
35.                                             id: 'gach87',
36.                                             contraseña: '18942bakuryu',
37.                                             perfil: {
```

```
                                                imagenDePerfil:
 "https://media.licdn.com/mpr/mpr/shrinknp_200_200/p/4/000/14f/16a/1076d85.jpg",
                                                nombre: "Guillermo Carrillo",
                                                correoElectronico: "gach87@gmail.com",
                                                telefonoDeContacto: "0414-2058311"
                                            },
                                            cartera: {
                                                saldoDisponible: 10000000,
                                                moneda: 'Bs'
                                            }
                                        },
                                        ticketAPagar: {
                                            id: "0",
                                            status: 'activo',
                                            horaDeEntrada: new Date().toTimeString(),
                                            horaDePago: '',
                                            horaDeSalida: '',
                                            total: '550',
                                            moneda: 'Bs',
                                            usuarioAsociado: 'gach87'
                                        }
                                    };
                                spyOn(repositorioDeTicketsMock,
 'obtenerTicket').and.callFake(function () {
                                    return dataDePrueba.ticketAPagar;
                                });
                                spyOn(repositorioDeTicketsMock, 'salvarTicket');
                                spyOn(repositorioDeUsuariosMock,
 'obtenerUsuario').and.callFake(function () {
                                    return dataDePrueba.usuario;
                                });
                                spyOn(repositorioDeUsuariosMock, 'salvarUsuario');
                                //Act
                                //Assert
                                expect(function () {
                                    return
 administradorDeTickets.pagarTicket(dataDePrueba.ticketAPagar.id, dataDePrueba.usuario.id);
                                }).not.toThrow();

 expect(repositorioDeTicketsMock.obtenerTicket).oHaveBeenCalledWith(dataDePrueba.ticketAPagar.id);
                expect(repositorioDeTicketsMock.obtenerTicket.calls.count()).toBe(1);

 expect(repositorioDeUsuariosMock.obtenerUsuario).toHaveBeenCalledWith(dataDePrueba.usuario.id);

 expect(repositorioDeUsuariosMock.obtenerUsuario.calls.count()).toBe(1);
                                expect(repositorioDeTicketsMock.salvarTicket).toHaveBeenCalled();
                                expect(repositorioDeTicketsMock.salvarTicket.calls.count()).toBe(1);
                            });
                        });
                    });
```

```
81.
82.             })(TuEstacionamiento.Aplicacion.AdministradorDeTickets.Especificaciones,
83.
 TuEstacionamiento.Aplicacion.AdministradorDeTickets.Implementaciones.Estandar,
84.                     {
85.                         repositorioDeUsuarios:
 TuEstacionamiento.Dominio.Repositorios.Usuarios.Diseño,
86.                         repositorioDeTickets:
 TuEstacionamiento.Dominio.Repositorios.Tickets.Diseño
87.                     });
```

Figura 12. Código del archivo tuEstacionamiento/fuentes/aplicacion/administradorDeTickets/estandar/pruebas.js

Para la definición de las pruebas se usa la función *"describe"* para declarar el conjunto de pruebas sobre las funciones individuales usando el nombre definido en las especificaciones como identificador, la definición de la función *"describe"* del marco de trabajo de pruebas Jasmine es la siguiente:

```
function describe(description: string, specDefinitions: () =>
void): void

Create a group of specs (often called a suite).

@param description — Textual description of the group

@param specDefinitions — Function for Jasmine to invoke that will define inner
suites a specs
```

Figura 13. Definición de la función *"describe"* del marco de trabajo de pruebas Jasmine

Como podemos observar *"describe"* recibe como parámetros un string que es la descripción textual de lo que llaman una "suite" o grupo de pruebas y como segundo parámetro un objeto de funciones que son las *"specDefinitions"* las cuales son el conjunto de pruebas individualmente definidas.

Cada una de las pruebas se define usando la función *"it"* la cual es una función de segundo orden que recibe como parámetros el nombre de la prueba y la implementación que es el parámetro tipo función. La definición de la función *"it"* de el marco de trabajo de pruebas Jasmine es la siguiente:

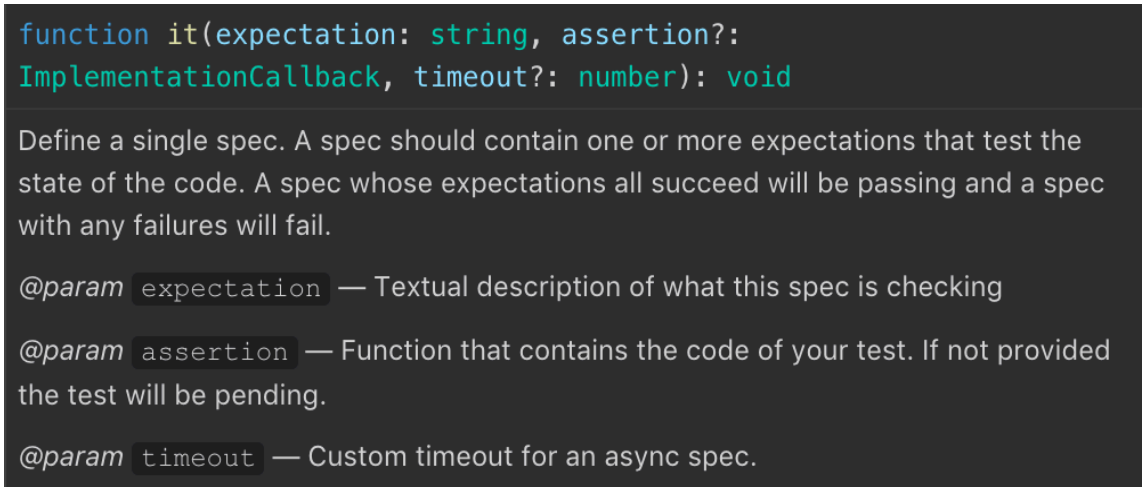

Figura 14. Definición de la función *"it"* del marco de trabajo de pruebas Jasmine

Como se puede observar la función *"it"* requiere un parámetro obligatorio y dos opcionales, el primer parámetro es *"expectation"* que es un string de la descripción textual de lo que esta especificación prueba, el segundo parámetro es *"assertion"* que es una función la cual contiene la implementación del código de la prueba en el caso de que no se provea esta función la prueba quedará pendiente en tiempo de ejecución.

Cada función de implementación de una prueba unitaria tiene 3 partes principales la primera parte se llama "Arrange" y es la parte encargada de la definición de todas las variables y datos "mock" a ser usados dentro de la prueba, aquí también se define una instancia del módulo que contiene la función a ser probada a manera de variable. La segunda parte es llamada "Act" y se encarga de hacer las llamadas a las funciones a ser probadas proveyéndolas de los datos definidos en "Arrange", la ultima parte es llamada "Assert" y se encarga de definir todas las expectativas que el resultado de la ejecución de "Act" debería cumplir, es decir, se definen todos los requisitos que el resultado debe cumplir para que la función pase la prueba exitosamente.

4.4 Configuración de compilación, despliegue y CI.

La configuración de la ejecución de las pruebas, así como el proceso de compilación del proyecto deben estar ligados es así como en el presente caso de estudio utiliza tareas de NodeJS para ejecutar los procesos de ejecución del framework de pruebas y el framework de compilación del proyecto, así:

```
{
    "name": "tuestacionamientousm",
    "version": "1.1.1",
    "description": "TuEstacionamientoUSM: An Ionic project",
    "dependencies": {
        "angular": "^1.6.9",
        "angular-mocks": "^1.6.9",
        "bower": "^1.8.4",
        "browser-sync": "^2.23.6",
        "cordova": "^8.0.0",
        "del": "^3.0.0",
        "gulp": "^3.9.1",
        "gulp-autoprefixer": "^5.0.0",
        "gulp-cache": "^1.0.2",
        "gulp-concat": "^2.6.1",
        "gulp-cssnano": "^2.1.3",
        "gulp-eslint": "^4.0.2",
        "gulp-flatten": "^0.4.0",
        "gulp-htmlmin": "^4.0.0",
        "gulp-if": "^2.0.2",
        "gulp-imagemin": "^4.1.0",
        "gulp-inject": "^4.3.2",
        "gulp-install": "^1.1.0",
        "gulp-load-plugins": "^1.5.0",
        "gulp-newer": "^1.4.0",
        "gulp-sass": "^3.2.1",
        "gulp-size": "^3.0.0",
        "gulp-sourcemaps": "^2.6.4",
        "gulp-uglify": "^3.0.0",
        "gulp-uncss": "^1.0.6",
        "gulp-useref": "^3.1.5",
        "ionic": "^3.20.0",
        "ionic-framework-v1": "^1.3.1",
        "jasmine-core": "^3.1.0",
        "karma": "^2.0.0",
        "karma-coverage": "^1.1.1",
        "karma-jasmine": "^1.1.1",
        "karma-jasmine-matchers": "^3.7.0",
        "karma-phantomjs2-launcher": "^0.5.0",
        "merge-stream": "^1.0.1",
        "ng-cordova": "^0.1.27-alpha",
```

```
42.         "require-dir": "^1.0.0",
43.         "run-sequence": "^2.2.1",
44.         "shelljs": "^0.8.1",
45.         "stream-series": "^0.1.1"
46.     },
47.     "scripts": {
48.         "construir:android:ionic": "ionic build android",
49.         "ejecutar:android:ionic": "ionic run android",
50.         "construir": "gulp",
51.         "servir": "gulp servir",
52.         "test": "karma start",
53.         "start": "npm run servir"
54.     },
55.     "cordovaPlugins": [
56.         "cordova-plugin-device",
57.         "cordova-plugin-console",
58.         "cordova-plugin-whitelist",
59.         "cordova-plugin-splashscreen",
60.         "cordova-plugin-statusbar",
61.         "ionic-plugin-keyboard",
62.         {
63.             "locator": "https://github.com/wymsee/cordova-
    imagePicker.git",
64.             "id": "cordova-plugin-image-picker"
65.         },
66.         {
67.             "locator": "https://github.com/hazemhagrass/phonegap-base64",
68.             "id": "com-badrit-base64"
69.         }
70.     ],
71.     "cordovaPlatforms": [
72.         "android"
73.     ]
74. }
75.
```

Figura 15. Código del archivo tuEstacionamiento/package.json

Las tareas que vemos definidas entre las líneas 48 a la 53 permiten ejecutar, probar, servir y empaquetar el proyecto es así como en este nivel de abstracción estas tareas pueden ser ejecutadas dentro de cualquier ambiente de integración continua ya que hacer un build del proyecto entero solo requiere correr el siguiente comando “npm run construir”.

La configuración del framework de karma para pruebas se la define en un archivo ubicado al nivel jerárquico mas alto del proyecto, así:

```
1. // Karma configuration
2. // Generated on Wed Apr 13 2016 19:16:47 GMT-0430 (Venezuela Standard Time)
3.
4. module.exports = function (config) {
5.     config.set({
6.         // base path that will be used to resolve all patterns (eg. files, exclude)
7.         basePath: '',
8.         // frameworks to use
9.         // available frameworks: https://npmjs.org/browse/keyword/karma-adapter
10.        frameworks: ['jasmine', 'jasmine-matchers'],
11.        // list of files / patterns to load in the browser
12.        files: [
13.            'node_modules/angular/angular.js',
14.            'node_modules/angular-mocks/angular-mocks.js',
15.            'fuentes/servicios/navegacion/**/*.js',
16.            'fuentes/servicios/mensajesDeUsuario/**/*.js',
17.            'fuentes/dominio/**/*.js',
18.            'fuentes/aplicacion/**/*.js',
19.            'fuentes/elementos/**/*.js'
20.        ],
21.        // list of files to exclude
22.        exclude: [
23.        ],
24.        // preprocess matching files before serving them to the browser
25.        // available preprocessors: https://npmjs.org/browse/keyword/karma-
   preprocessor
26.        preprocessors: {
27.            'fuentes/dominio/**/*.js': 'coverage',
28.            'fuentes/aplicacion/**/*.js': 'coverage',
29.            'fuentes/elementos/**/*.js': 'coverage',
30.            'fuentes/servicios/**/*.js': 'coverage'
31.        },
32.        // test results reporter to use
33.        // possible values: 'dots', 'progress'
34.        // available reporters: https://npmjs.org/browse/keyword/karma-reporter
35.        reporters: ['progress', 'coverage'],
36.        // web server port
37.        port: 9876,
38.        // enable / disable colors in the output (reporters and logs)
39.        colors: true,
40.        // level of logging
41.        // possible values: config.LOG_DISABLE || config.LOG_ERROR || config.LOG_WARN
   || config.LOG_INFO || config.LOG_DEBUG
42.        logLevel: config.LOG_INFO,
43.        // enable / disable watching file and executing tests whenever any file
   changes
44.        autoWatch: true,
```

```
45.         // start these browsers
46.         // available browser launchers: https://npmjs.org/browse/keyword/karma-
    launcher
47.         browsers: ['PhantomJS2'],
48.         // Continuous Integration mode
49.         // if true, Karma captures browsers, runs the tests and exits
50.         singleRun: true,
51.         // Concurrency level
52.         // how many browser should be started simultaneous
53.         concurrency: Infinity,
54.         coverageReporter: {
55.             type: 'html',
56.             dir: 'artefactos/pruebas/',
57.             reporters: [
58.                 // reporters not supporting the `file` property
59.                 { type: 'html', subdir: '.' },]
60.         }
61.     });
62. };
63.
64.
```

Figura 16. Código del archivo tuEstacionamiento/karma.conf.js

CAPÍTULO 5
CONCLUSIONES Y RECOMENDACIONES

Conclusiones:

- La automatización de pruebas unitarias permite agilizar el proceso de desarrollo sin dejar de lado la calidad, ya que evita la realización de pruebas manuales que generalmente toman mucho tiempo y son bloqueantes para la continuidad de la codificación.
- Usar pruebas automatizadas integra el aseguramiento de la calidad dentro del proceso de codificación y genera un flujo de trabajo con el uso de integración continua que nos permite garantizar que una funcionalidad no será incluida dentro de el código base a no ser que cumpla con los casos de prueba descritos.
- Automatizar las pruebas de un sistema aporta y fortalece cuatro de los seis componentes de la calidad de un producto software: funcionalidad, fiabilidad, eficiencia y mantenibilidad.
- La automatización de pruebas unitarias genera una visibilidad del estado de las funcionalidades de un proyecto ya que se puede saber cuando una funcionalidad ha sido implementada en el momento en el que cumple con las pruebas.
- En proyectos medianos a grandes la automatización de pruebas unitarias es indispensable ya que nos permite generar las pruebas una sola vez y conforme el proyecto crece podemos continuar probando las funcionalidades existentes en cuestión de minutos a diferencia de las pruebas manuales que tomarían mucho mas tiempo.
- La integración continua genera una manera de exponer los resultados de las pruebas en el momento de la compilación lo cual nos ayuda a tener transparencia de el estado de las funcionalidades a integrarse en cada “commit”.
- La transparencia y concordancia entre las historias de usuario, arquitectura por funcionalidad e implementación ayudan a que el proyecto sea extensible y fácil de mantener.
- La interacción del arquitecto al momento de la definición de los diseños de las pruebas es indispensable debido a que es la persona indicada de conocer las historias de usuario y definir la estructura de los casos de pruebas correspondientes.

Recomendaciones:

- El uso de herramientas para la automatización de las pruebas y para la integración continua (CI) es muy importante, se recomienda hacer un análisis al inicio del proyecto para definir dichas herramientas dependiendo del tipo de proyecto y de las necesidades de este.
- La arquitectura es una parte fundamental de un proyecto de software por lo que se recomienda manejar una arquitectura bien estructurada y basada en patrones como la planteada en el presente trabajo, esto permitirá hacer el proceso de desarrollo más predecible y organizado.
- Se recomienda utilizar un lenguaje de programación maduro e independiente para la implementación de las pruebas con esto se pueden reusar las pruebas independientemente de que el marco de trabajo del proyecto cambie.
- Se recomienda hacer una propia configuración de compilación y despliegue ya que aparte de garantizar el funcionamiento de este, permite utilizarlo en cualquier ambiente independiente de herramientas de terceros.
- El uso de marcos de trabajo de interfaces graficas hace el desarrollo mas rápido, pero reduce la mantenibilidad del proyecto por lo que se recomienda mantener el marco de trabajo de interfaces graficas lo más aislado posible de la implementación de las historias de usuario, es decir, las interfaces gráficas deben encargarse si y solo si de las interacciones del usuario con la interfaz gráfica de usuario.
- Se recomienda mantener la modularidad de las funcionalidades lo mas granular posible ya que esto permite generar pruebas unitarias mas precisas y efectivas.
- Se recomienda hacer una integración de las corridas de pruebas hasta el punto en el que un “pull request” a la rama principal de repositorio no pueda ser aprobado a no ser que dicha rama de funcionalidad haya pasado un “build" exitoso en nuestro ambiente de CI.

BIBLIOGRAFÍA

Apiumhub . (2019). *Beneficios de las pruebas unitarias*. Obtenido de https://apiumhub.com/es/tech-blog-barcelona/beneficios-de-las-pruebas-unitarias/

Arizmendi, P. (2018). *AngularJS: Conviértete en el profesional que las compañías de software necesitan.* Paiminix.

Atlassian . (2019). *DevOps: cómo romper la barrera entre Desarrollo y Operaciones*. Obtenido de https://es.atlassian.com/devops

Capgemini. (2015). *Testing + Integración continua* . Obtenido de https://www.uv.es/capgeminiuv/documents/Presentacion_Testing_IC_2015_11_05.pdf

Coremain. (2019). *Desarrollo software ágil con SCRUM: qué es y cómo funciona.* Coruña: http://www.coremain.com/desarrollo-software-agil-scrum/.

Digital Business Assurange. (2018). *Sin automatización no hay DevOps*. Obtenido de https://www.mtp.es/sin-automatizacion-no-hay-devops/

Evaluando software. (2018). *¿Qué es desarrollo de software ágil?* Obtenido de https://www.evaluandosoftware.com/desarrollo-software-agil/

Fernández, J. (2017). *Implntación de un sistema de integración contínua en una metodología consolidada.* España: Universidad de Castilla - La Mancha.

Iniesta, G. (2016). *Introducción a Agile Testing para el desarrollo de software*. Obtenido de Introducción a Agile Testing para el desarrollo de software

Instituto de Marketing Digital. (2017). *¿Qué es el Desarrollo Ágil?* Obtenido de https://www.institutodemarketingagil.com/single-post/Que-es-el-Desarrollo-agil

Latevaweb. (2018). *Metodologia TDD para la programación de webs a medida*. Obtenido de https://www.latevaweb.com/metodologia-tdd-programacion-web-a-medida

Mendoza, M. (2018). *DevOps y análisis de perfonance automáticos.* España.

Noriega, R. (2017). *El proceso de desarrollo de software.* IT Campus Academy.

Reyes, M. (2018). *La Metodología TDD*. Obtenido de http://mreysei.es/blog/metodologia-tdd-20180807

Subra, J.-P., & Vannieuwenhuyse, A. (2018). *Scrum n método ágil para sus proyectos.* Ediciones ENI.

Villamizar, K., Tabares, J., & Zapata, C. (2015). Mejora de historias de usuario y casos de prueba de metodologías ágiles con base en TDD. *Cuaderno activa*.