UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Programa de Apoyo al Egreso de Profesionales en Actividad

TÍTULO: Tests automatizados en aplicaciones web. Caso de Aplicación en empresa de turismo internacional.

AUTOR: Andrea Paola Gómez Del Mónaco.

DIRECTOR ACADÉMICO: Silvia Esponda.

DIRECTOR PROFESIONAL: Néstor Fabián Oviedo.

CARRERA: Licenciatura en Sistemas.

Resumen

Se describe el proceso de capacitación del personal, generación y ejecución de pruebas automatizadas en el marco del desarrollo de software de la empresa *Despegar.com.* El objetivo es mostrar cómo los **tests automatizados** aportan **calidad** al proceso de integración continua del producto de software y del monitoreo de las aplicaciones.

Palabras Clave

test automation, control de calidad del software, E2E, integración contínua, testing contínuo, aplicación web.

Conclusiones

Las **pruebas automatizadas** como parte del proceso de **integración contínua** han reducido el intervalo de tiempo entre un deploy a producción y otro, pues ya no es necesario un proceso validación manual QA. Las mismas proporcionan **seguridad ante refactorizaciones**. Y si bien se pierde ese ”testeo global” que aportan las pruebas manuales, garantizan la **calidad** y cumplimiento de las especificaciones cuando se aplican metodologías como TDD o BDD.

Trabajos Realizados

Se presentaron **experiencias** de aplicación de **test automatizados** en desarrollos de software. Los **tests unitarios y de componentes** de las aplicaciones backend desarrollados, utilizando las librerías JUnit y TestNG, pasaron a formar parte del proceso de Integración contínua del proyecto. Para comprobar el correcto funcionamiento de las aplicaciones se crearon suites de **tests E2E** utilizando los frameworks: Protractor, Nightwatch y Selenium.

Trabajos Futuros

Se pretende aplicar todo el **conocimiento** y las **lecciones aprendida**s sobre testing automatizado y continuo en la reingeniería de la extranet de Hoteles, aplicación encargada de la administración de alojamientos. Para lograr el objetivo con éxito es necesario poner a punto los tests actuales e **Investigar estrategias** de tests automatizados sobre operaciones en **bases de datos**.

Fecha de la presentación: Julio 2021

Tests automatizados en aplicaciones web. Caso de aplicación en empresa de turismo internacional

Andrea Paola Gómez Del Mónaco
Universidad Nacional de La Plata
Facultad de Informática

Directores
Director Académico: Silvia Esponda
Director profesional: Néstor Oviedo

Julio 2021

Índice general

Agradecimientos 3

Índice de figuras 4

Abreviaciones 5

Capítulo 1 5
- **1.2 Objetivo** 6
- **1.3 Motivación** 6
- **1.4 Desarrollos propuestos** 7
- **1.5 Estructura de la tesina** 8

Capítulo 2 Calidad en el Software 9
- **2.1 Introducción** 9
- **2.2 Definición de calidad** 9
- **2.2.3 Conceptos de calidad** 11
- **2.3 Calidad en los sistemas de información** 11
- **2.4 Aseguramiento de la calidad del Software (SQA)** 12
  - **2.4.1 Control de la Calidad de Software (SQC)** 13
  - **2.4.2 Relación entre las actividades de SQA y SQC** 14
- **2.5 Resumen** 14

Capítulo 3 Filosofía “ágil” 16
- **3.1 Introducción** 16
- **3.2 Hacia el cambio de paradigma** 16
- **3.3 El momento del testing** 18
- **3.4 Equipos ágiles** 19
- **3.5 Resumen** 20

Capítulo 4 Técnicas y tipos de pruebas de software 21
- **4.1 Introducción** 21
- **4.2 Tipos de pruebas de software** 21
  - **4.2.1 Caja blanca o caja de cristal** 21
  - **4.2.2 Caja negra o de comportamiento** 22
- **4.3 Clasificación de los tipos de pruebas de software** 23
- **4.4 Técnicas según la generación de pruebas** 28
- **4.5 Clasificación de los defectos** 32
- **4.6 Cuadrantes de las Pruebas Ágiles** 34
- **4.7 Resumen** 38

Capítulo 5 Testing automatizado y continuo 40
- **5.1 Introducción** 40
- **5.2 Test automatizados: los beneficios** 40

5.3 Posibles aspectos negativos de las pruebas automatizadas 41
5.4 Automatizaciones en equipos ágiles 42
5.5 Pirámide de la automatización del Test 43
5.6 Herramientas para automatizar pruebas en aplicaciones web 46
5.7 La integración continua en el proceso de desarrollo 50
5.8 Herramientas para testing con integración contínua 52
5.9 Resumen 54

Capítulo 6 Casos de aplicación de tests automatizados en empresa de turismo online 55
6.1 Introducción 55
6.2 El proceso de testing 55
6.3 El arquetipo para automatizaciones 56
6.4 Experiencia de tests automatizados en Hoteles de Agencias Afiliadas 58
6.5 Experiencia de tests automatizados en la aplicación Special Clients 63
6.7 Experiencia de tests automatizados en la aplicación Sc-Services 68
6.8 Casos de pruebas automatizadas sobre el sitio público de ventas 70
6.9 Resumen 72

Capítulo 7 Conclusiones y trabajos futuros 75
7.1 Conclusiones 75
7.2 Trabajos futuros 77

Referencias bibliográficas 79

Anexo I Entrevista al Proyect Manager Lucio Loiacono 82

Agradecimientos

A mis padres, por enseñarme la importancia del estudio y por darlo todo para que pueda cumplir el objetivo.

A mis hermanos que me dijeron las palabras justas en los momentos correctos. A Male por su amor. A Lily y a Michu, fieles compañeras.

A la familia que acompañó cada pasito.

A mi esposo por el aliento incondicional.

A Milo, quién me metió más presión que mis padres.

A mis tíos Pablo y Jale. Ellos confiaron y apostaron por mí.

A los amigos de toda la vida y a los que conocí en este camino.

A Silvia y a Nestor por la predisposición, el compromiso y por el tiempo que han dedicado para dirigir la tesina. Gracias a Seba Bottero por aceptar el desafío en primera instancia.

A mis compañeros de trabajo que me ayudaron a recopilar mails y archivos. En especial a: Lucio, Silvi, Tomy, Juán y Marcos; quienes participaron con sus testimonios.

A las personas de la Facultad de Informática que me brindaron sus conocimientos.

A la gente que aporta su dinero para que exista una Universidad Pública y también para mantener los Centros Universitarios que albergan chicos del interior, como lo hizo el CUP conmigo.

Índice de figuras

2.1 Componentes de calidad según Stylianou y Kumar
3.1 Ciclo de desarrollo de software bajo una metodología lineal.
3.2 Principales características de la metodología ágil.
3.3 Ciclo de desarrollo de software bajo metodología ágil.
4.1 Tipos de pruebas de software de Caja blanca.
4.2 Tipos de pruebas de software de Caja negra.
4.3 Tipos de pruebas según ISTQB.
4.4 Técnicas de tipos de pruebas.
4.5 Historia del término “bug” en la informática.
4.6 Cuadrante de las pruebas ágiles.
5.1 Pirámide para la automatización de pruebas.
6.1 Infraestructura propuesta para automatizar tareas de testeo.
6.2 Configuración de semáforos en Henry.
6.3 Un mail enviado por Henry alertando un cambio en el estado de un semáforo.
6.4 Test que verifica disponibilidad y formas de pago en las habitaciones en la página de detalle de un hotel.
6.5 Diagrama UML entre las aplicaciones Special Clients y Sc-Watchdog.
6.6 Reporte de alerta enviado por Sc-Watchdog al grupo de Telegram que tenía el equipo.
6.7 Modelado de test automatizados utilizando el patrón Page Object.
6.8 Captura de pantalla tomada por la automatización ante un fallo por timeout en el flujo de vuelos.
6.9 Flujo de desarrollo de software en Sc-Services.
6.10 Información de salida de las automatizaciones desarrolladas utilizando Protractor

Abreviaciones

IT	Information Technology (Tecnología de Información)
SW	Software
QA	Quality Assurance (Aseguramiento de la Calidad)
QC	Quality Control (Control de Calidad)
RC	Release Candidate (Versión candidata)
TDD	Test-Driven Development (Desarrollo Orientado a Pruebas)
BDD	Behavior Driven Development (Desarrollo Orientado al Comportamiento)
ISTQB	International Software Testing Qualifications Board (Junta internacional de calificaciones de pruebas de software)
UAT	User Acceptance Tests (Pruebas de aceptación del usuario)
API	Application Programming Interface (Interfaz de programación de aplicaciones)
GUI	Graphical User Interface (Interfaz gráfica del usuario)
E2E	End-To-End (principio a fin)
SQA	Software Quality Assurance (Aseguramiento de la Calidad del Software)
SQC	Software Quality Control (Control de Calidad del Software)
IC	Integración Contínua
IDE	Entorno de Desarrollo Integrado
ROI	Retorno de inversión
SDE	Entorno de desarrollo de software
UML	Lenguaje Unificado de Modelado

Capítulo 1

1.2 Objetivo

Describir el proceso de generación y ejecución de pruebas automatizadas en el marco del desarrollo de software de la empresa *Despegar.com,* incluyendo los conceptos de test automatizado, herramientas de aplicación y capacitación de los recursos humanos.

Presentar un caso de aplicación de uso de las pruebas en el sitio público de dicha empresa.

1.3 Motivación

El desarrollo ágil se basa en producir software de alta calidad en un marco de tiempo que maximice su valor para el negocio[1]. El testing ágil es una metodología para realizar pruebas indispensable para el desarrollo ágil en la que programadores y testers escriben pruebas antes y durante la codificación del software. Los resultados que arrojan dichos tests actúan como “inputs” para la nueva iteración. Todos los integrantes del equipo multifuncional tienen la responsabilidad de velar por la calidad del producto a entregar, por tal motivo se considera que los requerimientos del cliente se han alcanzado cuando las pruebas dan un resultado positivo.

Martin Fowler, escritor y orador experto en desarrollo de software para aplicaciones empresariales, especifica que cada integración se verifica con un build automático (que incluye la ejecución de pruebas) para detectar errores de integración tan pronto como sea posible y obtener aplicaciones en lapsos más cortos de tiempo.

La integración continua propone la creación de diversos ambientes de desarrollo que posibilitan realizar tipos de pruebas como las de performance y carga, en las que es fundamental contar con un ambiente aislado.

Con cada integración, el equipo de desarrollo se apoya en la ejecución de las pruebas automatizadas para verificar el comportamiento, la codificación de los

proyectos y normas de diseño, y que el resultado sea un producto funcional comprobable. También son de vital importancia a la hora de realizar refactorizaciones de código.

Las métricas de pruebas automatizadas, pueden ayudar a la organización en la realización de evaluaciones, en cuanto al progreso, la productividad y la calidad de los testeos. Es importante comprender qué partes de la automatización de pruebas funcionan, cuáles hay que mejorar y cuales hay que desechar.

Las definiciones anteriores permiten entender el papel que deben ocupar las pruebas automatizadas en un ambiente de integración contínua para garantizar la calidad en el proceso de desarrollo de software escalable.

En la actualidad, la posibilidad de propagación de la pandemia ha modificado las formas de trabajo de las empresas, que se han visto afectadas económicamente al no poder ofrecer sus servicios con normalidad.

En este contexto, dichas empresas ven una fuerte disminución en sus ingresos y deben reducir su staff profesional y apoyarse en las automatizaciones para disminuir la cantidad de procesos manuales repetitivos propensos a errores y así ahorrar tiempo, costos y esfuerzo. Los equipos de desarrollo pueden controlar la calidad interna y parte de la calidad externa del software mediante los test automatizados y así seguir garantizando un servicio de calidad a los usuarios.

1.4 Desarrollos propuestos

- Descripción del concepto Testing Ágil, ventajas y desventajas del mismo.
- Análisis y descripción de tipos de pruebas automatizadas.
- Descripción del proceso de desarrollo donde se implantaron las pruebas, y sus características identificatorias.
- Muestra de resultados de su aplicación en el sitio público de la empresa www.despegar.com, para lo cual se diseñará un set de pruebas automatizadas:
    - Test para aplicaciones web backend.
    - Pruebas de regresión en aplicación web con interfaz GUI.
    - Test en herramienta de deploy.
    - Test automatizados con integración de alertas.

1.5 Estructura de la tesina

La presente tesina está organizada en siete capítulos, siendo el presente en el que se exponen los objetivos, la motivación y la presentación del tema.

En el capítulo 2 se ponen de manifiesto diferentes interpretaciones acerca de la definición de Calidad. Se hace mención de la importancia de la Calidad en los Sistemas de Información. El segundo capítulo finaliza señalando el papel que ocupan los profesionales que se encargan del Control de la Calidad del Software (SQC) como actividad para garantizar el Aseguramiento de la Calidad en todo el proceso del desarrollo del Software.

En el capítulo 3 se describe el concepto de metodología ágil y se destacan los beneficios que le aporta a los equipos de desarrollo de software.

Los tipos de pruebas de software y las técnicas para aplicarlas son actividades fundamentales para el SQC, por tal motivo se les dedica el capítulo 4.

En el capítulo 5 se explica de forma detallada el concepto de test automatizado en el ámbito de la integración contínua mencionando los pros y contra que trae al equipo de desarrollo su implementación.

En el capítulo 6 de la presente Tesina se exponen ejemplos de desarrollos de testing automatizados que se realizaron en la empresa Despegar.com

En el capítulo 7 se presenta la conclusión a la que se ha llegado y posibles líneas de trabajo futuro sobre el proyecto de software Suppliers Hotels de la empresa dedicada al turismo online.

Capítulo 2 Calidad en el Software

2.1 Introducción

Corría la década de 1940 cuando surgió el concepto de “programación” como actividad independiente de los programas diseñados para un ordenador de un fabricante específico. Desde aquel tiempo de tarjetas perforadas y de turnos de espera para utilizar la “sala de máquina” hasta la actualidad, la ciencia informática extendió su área de presencialidad hasta formar parte de la cotidianeidad de las personas, y como consecuencia, de la sociedad.

La *ISO/IEC 2382:2015* define a la Ingeniería de Software como una aplicación sistemática de conocimientos, métodos y experiencia científica y tecnológica al diseño, implementación, prueba y documentación de software[2]. Los profesionales que practican la ingeniería de software buscan enriquecer su conocimiento a través de la investigación y la experiencia obtenida a lo largo de los años para entregar a sus clientes software de manera eficiente, eficaz y de calidad.

2.2 Definición de calidad

A lo largo de la historia referentes de la industria han dado su interpretación del significado de Calidad.

1951

“La Calidad se implementa como un sistema total conectado con los clientes y proveedores”

Armand Vallin Feigenbaum

1961

“Cero defectos”

Philip B. Crosby

1985

"En su interpretación más estrecha, calidad significa calidad del producto. En su interpretación más amplia, calidad significa calidad del trabajo, calidad del servicio, calidad de la información, calidad del proceso, calidad de la división, calidad de las personas incluyendo a los trabajadores, ingenieros, gerentes y ejecutivos, calidad del sistema, calidad de la empresa, calidad de los objetivos, etcétera. Nuestro enfoque básico es controlar la calidad en todas sus manifestaciones"

Kaoru Ishikawa

1985

"La calidad es la pérdida que un producto causa a la sociedad después de ser entregada… además de las pérdidas causadas por su función intrínseca."

Gen'ichi Taguchi

1988

"La Calidad es responsabilidad de todos"

William Edwards Deming

1995

"Calidad es el conjunto de características de un producto que satisfacen las necesidades de los clientes y, en consecuencia, hacen satisfactorio el producto"

Joseph Moses Juran

1997

"El cumplimiento de los requerimientos funcionales y de performance explícitamente definidos, de los estándares de desarrollo explícitamente documentados y de las características implícitas esperadas del desarrollo de software profesional."

Roger Pressman

La norma internacional *ISO 9000:2015[3]* define Calidad como "el grado en el que un conjunto de características inherentes de un objeto cumple con los requisitos". Las "características" que señala la norma pueden entenderse como una cualidad que diferencia a cada producto del resto. Los tipos de características pueden ser: mecánicas, eléctricas, sensoriales, de comportamiento, ergonómicas y funcionales.

2.2.3 Conceptos de calidad

Es posible señalar que la percepción de calidad de las sociedades por la prestación de un servicio o el uso de un producto, puede ser relativa y variar a lo largo de los años, ya que están presentes factores como: la edad de las personas que participan en el intercambio comercial, el avance tecnológico, el tiempo, el contexto en el que se vive y la actividad industrial.

2.3 Calidad en los sistemas de información

Stylianou y Kumar[13] plantean una "*visión holística*" de la calidad de los sistemas de información en la que consideran a la empresa u organización como un todo, como un conjunto de componentes de calidad, dividida en dos grandes grupos: calidad de los procesos de negocio y calidad de los sistemas de información.

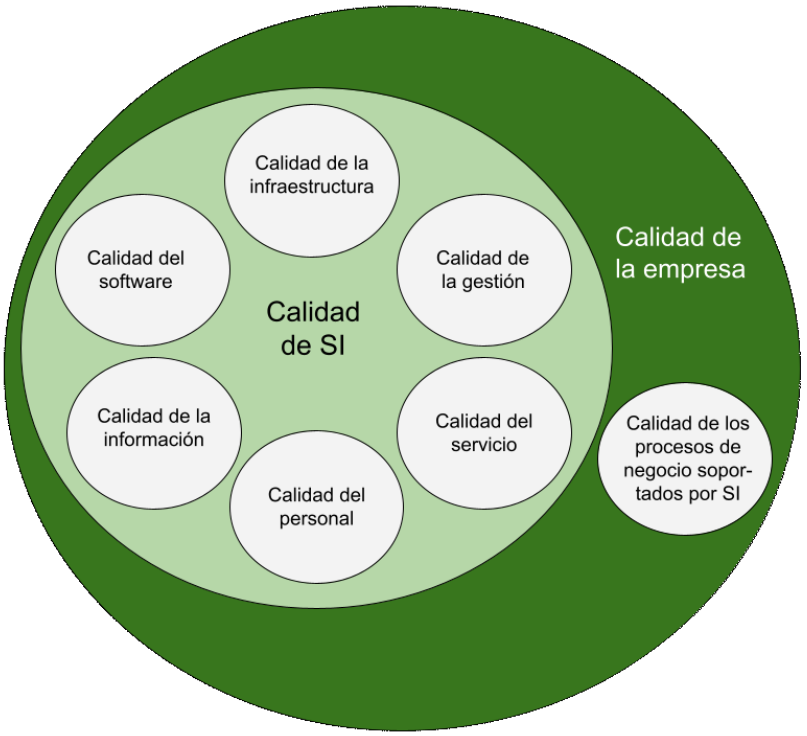


Figura 2.1 Componentes de calidad según Stylianou y Kumar

Desde el punto de vista tecnológico es relevante la calidad del grupo de los sistemas de información. Lograr la calidad de la empresa u organización está directamente relacionado con la obtención de la calidad de todos sus componentes:

- **Calidad de la infraestructura:** hardware, software habilitado, redes, etc. mantenido y soportado por el SI.
- **Calidad de la gestión:** que incluye los aspectos de gestión relacionado con la función de los SI como el presupuesto, la planificación y la programación.
- **Calidad del servicio:** que aborda la calidad desde el punto de vista de los servicios que ofrecen los SI, incluyendo la calidad de los procesos de soporte, de los clientes y del personal implicado.
- **Calidad de datos:** se refiere a la información ingresada al SI y el resguardo de la misma.
- **Calidad de la información:** hace referencia a la calidad de los datos y de la información resultante de los SI. En los casos en los cuales los datos de salida de un sistema es la entrada de otro, la calidad de la información es relativa a la calidad de los datos.
- **Calidad del software:** de las aplicaciones de software construidas, o mantenidas, o soportadas por el SI. A su vez se divide en:
    - **Calidad del producto obtenido**
    - **Calidad del proceso de desarrollo**

2.4 Aseguramiento de la calidad del Software (SQA)

La *IEEE 730-2014*[7] determina el concepto de Aseguramiento de la Calidad del Software (SQA) como un conjunto de actividades que evalúan el cumplimiento y la adecuación de los procesos de software utilizados para desarrollar y modificar productos de software. SQA también determina el grado en que se obtienen los resultados deseados del control de calidad del software. En el mismo estándar se menciona a la objetividad como un atributo clave de la función SQA con respecto al proyecto. La función SQA también puede ser organizativamente independiente del proyecto; es decir, libre de presiones técnicas, administrativas y financieras. SQA

tiene la responsabilidad de controlar que los procesos y tareas que se realicen en un proyecto cumplan con los estándares organizacionales.

Bajo el mismo lineamiento, en la quinta edición de *Software Extension to the PMBOK Guide[4]* se define el concepto de SQA como: un conjunto de actividades que evalúan el cumplimiento y la idoneidad de los procesos de software utilizados para desarrollar y modificar productos de software. El aseguramiento de la calidad del software es un proceso continuo que audita otros procesos del software para garantizar que se sigan esos procesos (incluidos, entre otros, la planificación y el seguimiento de los planes de gestión de la calidad del software).

Los criterios de aceptación detallados a nivel de historia se definen como parte de la preparación, acumulación y cambios en los requisitos para el equipo de desarrollo, por lo tanto, el equipo de SQA está continuamente involucrado con el equipo de desarrollo desde el análisis hasta la aceptación de los incrementos entregables de software.

El personal de SQA y el gerente del proyecto comparan datos de desempeño, como el esfuerzo de trabajo, el tiempo transcurrido y el costo hasta la fecha para medir la variación entre lo planificado y los resultados actuales para determinar dónde pueden ser necesarios cambios en los procesos o en los cronogramas y/o los recursos para mejorar la calidad de la planificación.

Para verificar la validez de los resultados de las pruebas, el equipo de SQA utiliza las simulaciones analíticas que predicen el número más probable de defectos que se esperan en el código, basándose en resultados de pruebas anteriores, la complejidad del software y la experiencia del equipo de desarrollo de software. Para los proyectos de software que utilizan ciclos de vida adaptativos, los detalles de los planes de prueba, incluidos los criterios de aceptación específicos, se elaboran progresivamente junto con los requisitos[4].

2.4.1 Control de la Calidad de Software (SQC)

En el Software Extension to the *PMBOK Guide[4]* se puede encontrar una definición de SQC orientada al ámbito de los sistemas de software en la cual se lo define como: “un conjunto de actividades que miden, evalúan e informan sobre la

calidad de los artefactos del proyecto de software a lo largo del ciclo de vida del proyecto". El control de calidad del software (SQC) se ocupa de aplicar métodos, herramientas y técnicas para garantizar que los productos de trabajo de software (incluido, entre otros, el código de software) satisfagan los requisitos de calidad para un producto de software en desarrollo o en proceso de refactorización. Los métodos de SQC comúnmente utilizados son: revisiones, inspecciones y pruebas de software.

2.4.2 Relación entre las actividades de SQA y SQC

Hay dos niveles de SQA y SQC en la mayoría de las organizaciones que desarrollan software: SQA y SQC internos que ocurren dentro del equipo o equipos de desarrollo de software; y SQA y SQC externos que ocurren a nivel de la unidad organizativa en la que reside un proyecto de software. SQA a nivel organizacional examina métodos, herramientas, técnicas y resultados de SQC internos y externos para proyectos de software dentro de la organización[4].

La SQA externa participa en la identificación de áreas para la mejora del procesos como la identificación de cuellos de botella, los retrasos y las fuentes de errores, el desarrollo del proyecto inicial y el alcance del producto, la identificación y participación de las partes interesadas, la inclusión de un cliente informado y disponible de manera continua, la cantidad y las habilidades adecuadas de los equipos y miembros del equipo.

Dentro de los equipos de software, la SQA interna toma la forma de introspección, reuniones retrospectivas y revisiones de lecciones aprendidas para determinar si se están siguiendo o no los procesos especificados y encontrar oportunidades de mejora. Los niveles de desempeño alcanzados también se revisan y se comparan con las normas y expectativas.

Para el SQC interno, los miembros del equipo de software que no son los que produjeron un componente de software realizan revisiones y pruebas entre pares.

2.5 Resumen

A lo largo del capítulo se presentaron distintas interpretaciones del concepto de Calidad, el cual ha ido cambiando a lo largo del tiempo. La norma internacional *ISO*

9000:2015[3] define Calidad como “el grado en el que un conjunto de características inherentes de un objeto cumple con los requisitos“. Para lograr esa Calidad en las aplicaciones de software en desarrollo o en mantenimiento, es necesario alcanzar la Calidad en toda la organización.

El Aseguramiento de Calidad del Software es un área presente en las organizaciones que persigue el objetivo de auditar los requisitos de calidad y los resultados obtenidos a partir de medidas de control de calidad, a fin de garantizar que se utilicen definiciones operacionales y normas de calidad adecuadas. SQA comprende un conjunto completo de actividades planificadas y sistemáticas que pueden demostrarse para brindar confianza en que un producto o servicio cumplirá con sus requisitos de calidad.

Las actividades de SQC corresponden a un subconjunto complementario de tareas de las que debe realizar la SQA. El rol de SQC es el de verificar y detectar inconvenientes al validar que cada resultado que provenga de los procesos implementados cumplan con los estándares de calidad especificados por la organización. El SQA auditará al SQC, como a todos los equipos de la organización para corroborar que cada uno de ellos sigue los procedimientos, estándares y procesos acordados estableciendo métricas para medir la efectividad alcanzada.

Los ciclos de vida de los proyectos de software adaptativos que se basan en iteraciones frecuentes para producir software funcional, probado y entregable son muy adecuados para planificar un enfoque integrado de SQA y SQC. Ambos equipos pueden participar en el análisis de defectos y otros problemas y hacer recomendaciones pertinentes, revisiones continuas por pares recorridos, inspecciones, pruebas de regresión automatizadas y análisis[4].

Capítulo 3 Filosofía “ágil”

3.1 Introducción

La filosofía “ágil” puede interpretarse como una metodología de trabajo y de organización para el desarrollo de software basada en la rapidez y flexibilidad, capaz de brindar productos y servicios de calidad a clientes con requerimientos cambiantes. Divide el desarrollo de un proyecto en “entregables” que tienen que completarse en un lapso de pocas semanas y en los que el cliente tiene una participación activa.

En el presente capítulo se destacan los beneficios que la metodología ágil aporta a los equipos de desarrollo.

3.2 Hacia el cambio de paradigma

Antes del surgimiento de las metodologías “blandas” (denominación con la que se conoce a los modelos ágiles), el desarrollo de software solía estar orientado hacia un enfoque sistemático, secuencial, que comienza en un nivel de sistemas y progresa con el análisis, diseño, codificación, pruebas y mantenimiento[10]. Al tratarse de un modelo lineal, como se ilustra en la Figura 3.1, el inicio de una etapa dependía de la terminación de una anterior, de modo que el producto se entrega cuando está finalizado, lo que puede tardar años dependiendo del proyecto, lapso de tiempo en el que los requerimientos del cliente pueden haber cambiado. Este modelo predictivo es

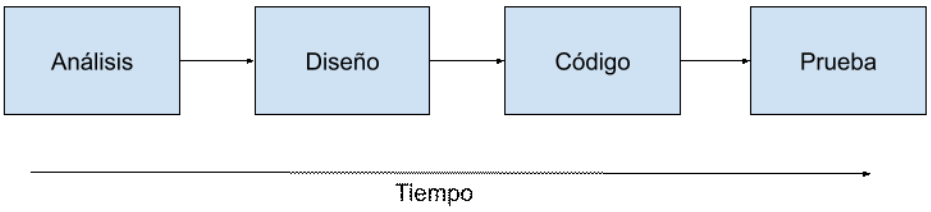


Figura 3.1 Ciclo de desarrollo de software bajo una metodología lineal

ideal en un ámbito en el que se cuenta con mucha documentación, el relevamiento de la información y los requerimientos se han realizado correctamente y se mantienen

estables. En caso contrario puede ocurrir el riesgo de entregar un producto de mala calidad que puede no ser lo que el cliente espera.

The New New Product Development Game[19], escrito en 1986 por *Ikujiro Nonaka* en conjunto con *Hirotaka Takeuchi*, fue uno de los primeros artículos en los que se narró la experiencia de un nuevo modelo de trabajo. En la década de los 90 's surgieron metodologías como *Extreme Programming, SCRUM, DSDM, Adaptive Software Development, Crystal, Feature-Driven Development, Pragmatic Programming* que se diferenciaban de las tradicionales por sus características ágiles. Fue en febrero de 2001 cuando sus autores se reunieron en Utah, EE.UU. para unificar criterios y crear el "*Manifiesto Agile*[20]". En dicho documento se describe un modelo de desarrollo de software rápido y de mejora constante que se centra en planificar, crear, validar el resultado con el cliente y mejorar el producto obtenido en cada entrega. En la figura 3.2 se ilustran los aspectos más importantes que persigue el paradigma ágil.

Figura 3.2 Principales características de la metodología ágil

En un proyecto ágil el cliente participa activamente con el equipo de desarrollo en la división del trabajo a realizar, en "historias" que se describen en un lenguaje no

técnico. Juntos asignan una puntuación a cada historia teniendo en cuenta su complejidad, establecen las dependencias entre ellas y definen las fechas tentativas de cada entregable.

Cuando una organización enmarca un desarrollo de software bajo una metodología incremental e iterativa como es la ágil, tiene como ventaja el poder visualizar el estado de lo producido en cada iteración sin tener que esperar a que el software esté finalizado.

Figura 3.3 Ciclo de desarrollo de software bajo metodología ágil

3.3 El momento del testing

El modelo predictivo secuencial y las metodologías ágiles tienen en común etapas el ciclo de vida: La prueba del producto realizado es una de ellas. En el caso del primero el “momento del testing” no se da sino hasta el final del proyecto, antes de entregarlo al cliente. De modo que el equipo responsable de las pruebas puede tener más carga de trabajo hacia el final del proyecto.

El enfoque del desarrollo ágil es producir software de alta calidad en un marco de tiempo que maximice su valor para el negocio[1]. En los productos ágiles cada iteración da como salida una retroalimentación constante como se ilustra en la Figura 3.3. Los resultados que arrojan las pruebas en el software actúan como “inputs” para

la nueva iteración, lo que da la posibilidad de una identificación temprana de los errores, aumentando la calidad del producto.

Todos los integrantes del equipo de desarrollo orientado al modelo iterativo e incremental tienen la responsabilidad de velar por la calidad del producto a entregar. De modo que los desarrolladores y testers escriben pruebas antes y durante la codificación del software. Cuando se ejecutan esas pruebas y dan un resultado positivo, el equipo de desarrollo puede considerar que los requerimientos de la historia se han alcanzado.

3.4 Equipos ágiles

Las organizaciones que eligen realizar sus desarrollos con metodologías “blandas” pueden realizar reasignaciones de los integrantes de los equipos de desarrollo entre los proyectos que se encuentren activos teniendo en cuenta la carga de trabajo que cada uno tiene. Uno de los mayores desafíos que presenta la filosofía ágil a la organización misma y por lo tanto a sus integrantes es la capacidad de adaptación a las circunstancias y necesidades que se presenten.

La metodología ágil fomenta el intercambio de conocimiento constante entre los miembros y entre los equipos multidisciplinarios. Cada uno de ellos puede cambiar de rol al solicitar asesoramiento y feedback o al brindar capacitación. Esta retroalimentación de conocimiento entre las partes de la organización tiene como fin generar productos de manera eficaz y eficiente, que cumplan con los requerimientos solicitados en el menor tiempo posible.

El dinamismo que rodea a los proyectos ágiles y la puja por completar con lo acordado para cada iteración, puede conllevar en la postergación de la documentación del sistema. Los desarrolladores pueden no encontrar el tiempo de plasmar los conocimientos del negocio en una fuente común; y esa información va quedando en cada una de las personas que van formando parte de cada desarrollo.

Los miembros de los equipos ágiles organizan reuniones frecuentes con distintos objetivos. Al menos una vez al día cada integrante hace saber a todo el equipo sobre el estado de la “historia” en la que se encuentra trabajando. Es fundamental saber si pudo finalizar su tarea para que pueda asignársele otra. En los casos en los que tenga inconvenientes con la historia y no haya podido avanzar lo

esperado, debe informar para que los demás colaboradores puedan asistirlo, a fin de alcanzar la resolución o al menos avanzar en la incidencia.

Es común que los equipos ágiles realicen reuniones una vez finalizada una iteración o un proyecto. Esos encuentros son nombrados generalmente como *"retros"* y tienen como finalidad resaltar los aspectos positivos que se encontraron y también hablar sobre los inconvenientes que se presentaron para tomar acciones que permitan al equipo encontrar las oportunidades de mejora necesarias a aplicar en los procesos y prácticas en un futuro.

3.5 Resumen

El concepto *"ágil"* aplicado como metodología de desarrollo de software consiste en buscar una mejora continua que se centra en planificar y desarrollar software "entregable" en lapsos cortos de tiempo. Cada división de los requerimientos como la validación de los resultados esperados se hace con un cliente involucrado a lo largo del ciclo de vida del proyecto.

Como el modelo "ágil" pretende entregar software funcional al cliente en cada iteración; todos los miembros del equipo participan en la creación y ejecución de pruebas y validaciones. Es decir, la etapa de prueba se ejecuta tantas veces como incrementos de software se hagan a diferencia del modelo de desarrollo de software tradicional, en el que la etapa de prueba se posterga hasta las fases finales del desarrollo.

Capítulo 4 Técnicas y tipos de pruebas de software

4.1 Introducción

Las pruebas de software son una actividad fundamental en el proceso de control de calidad. Son un elemento crítico para la garantía de calidad del software y representan una revisión final de las especificaciones, del diseño y de la codificación[9]. Son investigaciones empíricas y técnicas cuyo objetivo es proporcionar información sobre la calidad del producto.

Los profesionales de SQC aplican técnicas para lograr realizar los tipos de pruebas necesarias para validar la salud de las aplicaciones de software durante todo su ciclo . En este capítulo se clasifican los tipos de pruebas según el objetivo que persiguen y las técnicas para lograrlas.

4.2 Tipos de pruebas de software

Existen dos grandes categorías de validaciones que se pueden realizar sobre una aplicación de software teniendo en cuenta en si se basa o no en la información de cómo está implementado el código fuente. A continuación se detallan cada uno de los dos enfoques: las pruebas de caja blanca y las pruebas de caja negra.

4.2.1 Caja blanca o caja de cristal

Una prueba de software se considera de “caja blanca” cuando se basa en la forma en cómo se ha diseñado o codificado el software[9]. Mediante los métodos de prueba de caja blanca, el ingeniero del software puede obtener casos de prueba que (1) garanticen que se ejercita por lo menos una vez todos los caminos independientes de cada módulo; (2) ejerciten todas las decisiones lógicas en sus vertientes verdadera y falsa; (3) ejecuten todos los bucles en sus límites y con sus límites operacionales; y (4) ejerciten las estructuras internas de datos para asegurar su validez[10]. En la Figura 4.1 se presenta una representación de una prueba en el que se puede

observar el funcionamiento interno de la aplicación, entonces las entradas y las salidas están condicionadas al conocimiento de la estructura interna de la aplicación.

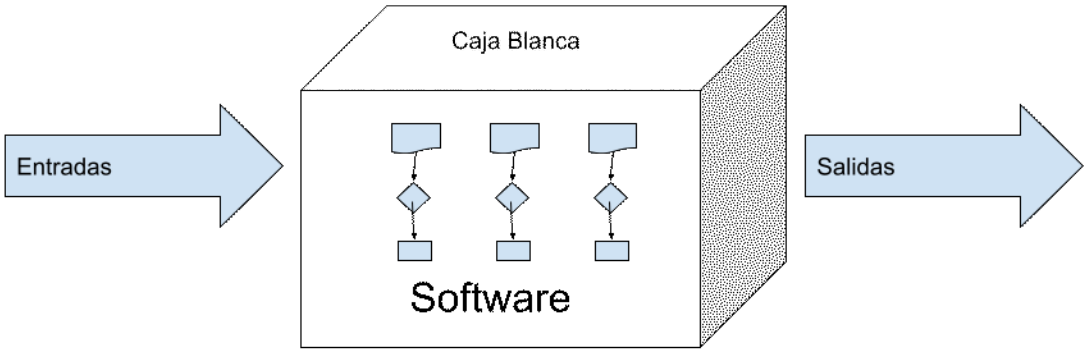


Figura 4.1 Tipos de pruebas de software de Caja blanca.

4.2.2 Caja negra o de comportamiento

Las pruebas de caja negra tienen el objetivo de validar el cumplimiento de los requisitos funcionales del software. El ingeniero del software crea conjuntos de condiciones de entrada, y a través de las salidas, evalúa el comportamiento, la validez funcional, el rendimiento, los volúmenes y niveles de datos que podrá tolerar el programa para encontrar errores de las siguientes categorías: (1) funciones incorrectas o ausentes, (2) errores de interfaz, (3) errores en estructuras de datos o en accesos a bases de datos externas, (4)errores de rendimiento y (5) errores de inicialización y de terminación[10]. En la Figura 4.2 se expone una representación de este tipo de pruebas en el que, ante la llegada de valores de entrada se obtienen valores de salida sin conocer el funcionamiento interno de la aplicación.

Las pruebas de caja negra están enfocadas en identificar los tipos de errores que no se detectan con las pruebas de caja blanca, por tal motivo son complementarias.

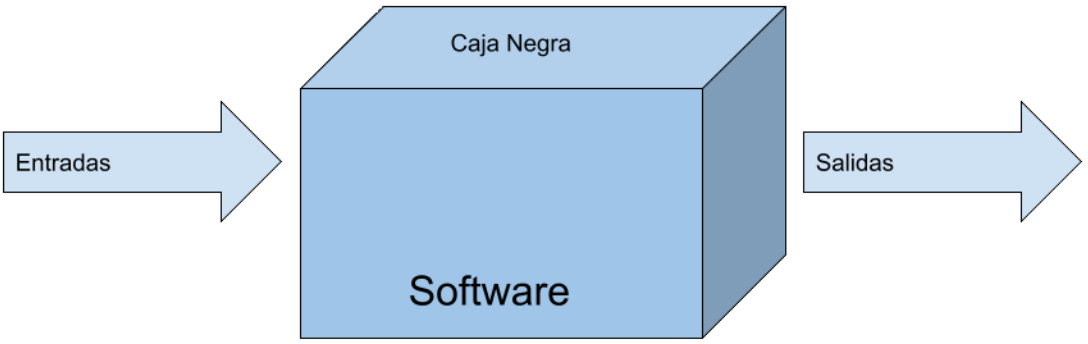


Figura 4.2 Tipos de pruebas de software de Caja negra.

4.3 Clasificación de los tipos de pruebas de software

El *International Software Testing Qualifications Board* (*ISTQB*)[18], establece la definición de los tipos de pruebas de software en función del objetivo en el que se centran y el comportamiento del sistema (interno o externo) a la que está dirigida. Teniendo en cuenta las definiciones de los tipos de pruebas que se presentaron anteriormente, y la representación propuesta por *ISTQB* (Figura 4.3) sería apropiado asociar a los subtipos: *funcionales, estructurales y de mantenimiento* con a los tipos de prueba de caja de cristal o caja blanca, mientras que los tipos *no funcionales* corresponden en general, a los tipos de prueba de caja negra.

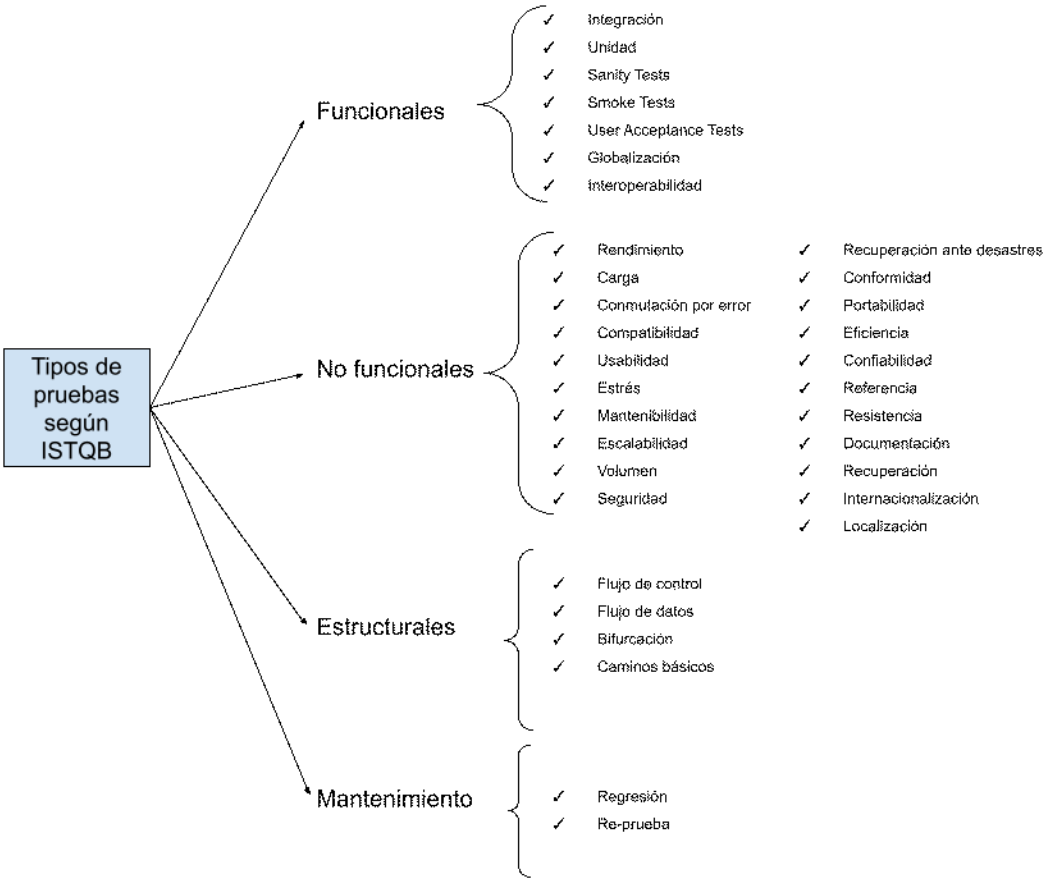


Figura 4.3 Tipos de pruebas según ISTQB

Funcionales:

Se trata de un tipo de testeo que puede ejecutarse en todos los niveles de pruebas (componentes, integración, sistema, entre otros). Para realizar este tipo de pruebas, los responsables pueden acudir a las especificaciones de los requerimientos que se desarrollaron para la aplicación; a la documentación del sistema; al código fuente o el comportamiento que se espera del sistema, subsistema o componente en funcionamiento. La experiencia adquirida de los miembros del equipo de SQC debido a la interoperabilidad con otros sistemas específicos similares también puede contribuir al momento de realizar las pruebas funcionales sobre la aplicación.

Teniendo en cuenta la definición de pruebas de caja blanca o de cristal, se puede pensar a las pruebas funcionales como un subconjunto de las pruebas de Caja blanca. Entre los tipos de prueba funcionales se encuentran:

- **Pruebas unitarias:** Verifican la funcionalidad y el comportamiento de un pequeño subconjunto del sistema, como un objeto o método.
- **Pruebas de componentes:** Chequean el comportamiento de una parte del sistema, como, por ejemplo, un grupo de clases o métodos que brindan algún servicio.
- **Sanity Tests:** Se utilizan para tener la certeza de que una parte de la funcionalidad de la aplicación continúa funcionando como lo esperado después de realizar una modificación menor. Para realizarla se seleccionan un subconjunto de entre las funcionalidades que se testean, y se las evalúa en profundidad. Se trata de un tipo de prueba que abarca un número reducido de casos de prueba y los realiza de manera profunda.
- **Smoke Tests:** Los equipos de desarrollo las ejecutan para corroborar que las funcionalidades más importantes de la aplicación no han sido afectadas por cambios que se hayan efectuado al software. Para realizarlas, se selecciona un conjunto reducido con las funcionalidades más relevantes del sistema a testear. Tal conjunto es más amplio que el que se seleccionaría para las sanity tests.
- **Test de integración:** Su objetivo es verificar la interacción entre dos o más sistemas con el fin de corroborar que los componentes se comunican exitosamente a través de sus interfaces y satisfacen los requisitos y funcionalidades esperadas.
- **UAT (User Acceptance Tests):** Son las pruebas, ejecutadas y validadas por el usuario/cliente quien ha definido los requerimientos de la aplicación. En determinados desarrollo de software dicho integrante se lo denomina bajo el nombre de: *Product Owner*.

No funcionales:

Las Pruebas Software no Funcionales tienen como objetivo validar, verificar y realizar mediciones sobre las características del software que establecen cómo responde el sistema.

Persiguen el objetivo de aumentar la usabilidad, la eficiencia, el mantenimiento y la portabilidad, entre otros atributos; y de reducir los riesgos y costos del producto de software desarrollado.

Los outputs que arrojan las pruebas no funcionales pueden ser utilizados para ampliar el conocimiento del comportamiento del producto y realizar investigaciones y desarrollo internos.

Pueden abarcar aspectos del comportamiento externo e internos del sistema, por que se utilizan tanto técnicas de pruebas de caja blanca y pruebas de caja negra.

- **Seguridad**: Detectan cómo se protege un sistema contra ataques deliberados y repentinos de fuentes internas y externas. Se utilizan para identificar posibles vulnerabilidades, amenazas y riesgos en sistema de software que podrían ocasionar pérdidas de información, monetarias o reputación de la organización. Toda organización debe garantizar la confidencialidad e integridad de su software. Ejemplos de ellas son:
    - Escaneo de vulnerabilidades
    - Escaneo de seguridad
    - Prueba de penetración
    - Evaluación de riesgos
    - Auditoría de seguridad
    - Hacking Ético (Ethical hacking)
    - Pruebas de evaluación de la postura (Posture Assessment)

- **Confiabilidad**: Chequea que el sistema realice una determinada función sin fallas durante un período de tiempo bajo un entorno determinado y que cada vez que se ejecute esa tarea se obtenga el mismo resultado. Existen 3 niveles de pruebas de confiabilidad:
    - Modelado
    - Medición
    - Mejora

- **Recuperación:** Test que evalúa la capacidad que tiene el sistema de recuperarse en caso de fallo de software/hardware/red, etc. Su objetivo es volver al sistema al último punto de integridad y re-procesar las transacciones incompletas.

- **Estabilidad:** Prueba que mide la eficiencia y la capacidad de una aplicación de software para funcionar continuamente más allá de su capacidad operativa normal.
- **Usabilidad:** Es un tipo de prueba que se aplica con el fin de conocer cómo es la experiencia del usuario con la aplicación. Para ello se convoca a un grupo de personas y se las observa interactuando con el sistema.
- **Escalabilidad:** Prueba la capacidad del sistema para responder a las crecientes necesidades de aumento de tráfico, volumen de datos, transacciones por segundo, uso de CPU, memoria, entre otros.
- **Interoperabilidad:** Se utilizan para comprobar la comunicación entre interfaces de un sistema de software con otros.
- **Portabilidad:** Mide la flexibilidad del software para ser transferido desde su entorno actual de hardware o software a otros.
- **Reutilización:** Evalúa el porcentaje del sistema de software se puede utilizar en otra aplicación.

Estructurales:

Tipo de prueba en la que se examina el comportamiento interno de la aplicación de software, por tal motivo se las puede agrupar dentro de las "prueba de caja blanca" o "prueba de caja de cristal". Es un método de diseño de test que se deriva a partir del conocimiento de la arquitectura e implementación del software.

Para expresar el alcance o extensión de la estructura del código que se examina por el conjunto de pruebas se utiliza el concepto de "Cobertura", que significa la cantidad de código que se ve alcanzado por la prueba. Normalmente el índice de la cobertura alcanzada se expresa en forma de porcentaje. En los casos en los que se obtienen resultados con un valor menor al 100%, entonces es posible diseñar pruebas adicionales hasta lograr la totalidad de la cobertura.

- **Pruebas de flujo de datos:** Este tipo de pruebas tiene como objetivo analizar el flujo de las variables de un programa. Se trata de buscar la mejor selección de los caminos de prueba de un programa para llegar a una resolución correcta de las definiciones y de los usos de las variables del programa.

- **Pruebas de caminos básicos:** La complejidad ciclomática del grafo de flujos del código fuente del software determinará el número de casos de prueba que deben ejecutarse para garantizar que todas las sentencias de un componente han sido ejecutadas al menos una vez[10].
- **Prueba de Condición:** También son conocidas bajo el nombre de: "flujo de control". Su objetivo es ejecutar las condiciones lógicas simples o complejas para encontrar posibles errores en los operadores lógicos o relacionales, en operaciones aritméticas, en variables o al escribir código entre paréntesis.
- **Prueba de bucles:** Su finalidad es ejecutar pruebas en 3 tipos diferentes de bucles: bucles simples, bucles concatenados y los bucles anidados.

Regresión y Re-pruebas

También son conocidas como las "Pruebas de mantenimiento" y corresponden a la última división de los tipo de pruebas que propone el *ISTQB*. Cuando un error es encontrado y posteriormente corregido es imprescindible comprobar que ese bug ya no está presente en la aplicación, pero también que ese cambio que se efectuó para resolver la incidencia no haya ocasionado efectos adversos involuntarios. Por tal motivo los miembros del equipo de desarrollo ejecutarán las pruebas de regresión una vez que un error detectado haya sido corregido para confirmar que la aplicación funcione correctamente.

Cuando el equipo decide correr el set de pruebas de regresión también lo hace para garantizar que el software siga cumpliendo con las funcionalidades que solicitó el cliente.

4.4 Técnicas según la generación de pruebas

Las técnicas de pruebas son diversos mecanismos que aplican los integrantes de SQC y los desarrolladores para intentar descubrir inconsistencias en las aplicaciones de software. Las técnicas intentan "romper" un programa siendo lo más sistemático posible en la identificación de insumos que producirán comportamientos representativos del programa, por ejemplo, considerando subclases del dominio de entrada, escenarios, estados y flujos de datos[9].

En el *SWEBOK*[9] se presenta una división de las técnicas de detección de errores y defectos (Figura 4.4) en base a cómo se generan las pruebas que pueden ser: a partir de la intuición y experiencia de la persona que la realiza, a las especificaciones del desarrollo, a la estructura del código, los tipos de fallas reales o imaginarias que se descubrirán, el uso previsto, a los modelos o la naturaleza de la aplicación, entro otros aspectos.

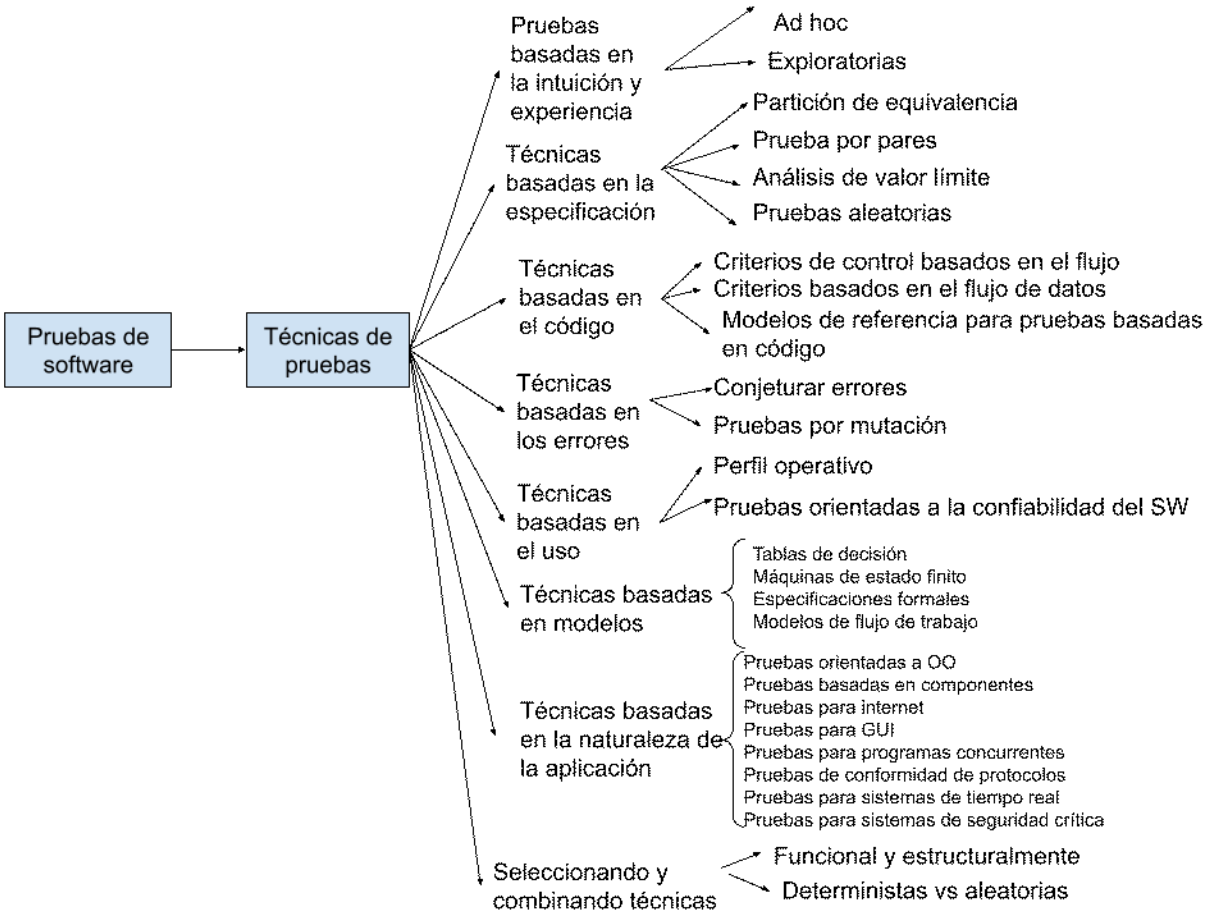


Figura 4.4 Técnicas de tipos de pruebas.

Basadas en la intuición y experiencia

- **Ad hoc (específicas):** Los encargados de realizar las pruebas utilizan su habilidad, intuición y experiencia con programas similares.
- **Exploratorias:** Se diseñan, ejecutan y modifican dinámicamente. El responsable de realizar las pruebas se apoya en el comportamiento observado del producto durante la prueba, familiaridad con la aplicación, la plataforma, el proceso de falla, entre otros conocimientos

Basadas en la especificación

- **Partición de equivalencia:** implica la partición del dominio de entrada en una colección de subconjuntos (o clases equivalentes) según un criterio o relación especificados. Este criterio o relación pueden ser resultados computacionales diferentes, una relación basada en el flujo de control o flujo de datos, o una distinción hecha entre entradas válidas que son aceptadas y procesadas por el sistema y entradas inválidas, como valores fuera de rango, que son no aceptado y debería generar un mensaje de error o iniciar el procesamiento de errores.
- **Prueba por pares:** Los casos de prueba se obtienen combinando valores interesantes para cada par de un conjunto de variables de entrada en lugar de considerar todas las combinaciones posibles. Se denominan t -wise, por lo que todas las combinaciones posibles de t variables de entrada se consideran.
- **Análisis de valor límite:** Los casos de prueba se eligen en o cerca de los límites del dominio de entrada de las variables, ya que muchas fallas tienden a concentrarse cerca de los valores extremos de las entradas.
- **Pruebas aleatorias:** El dominio de entrada de la prueba debe ser conocido para poder elegir puntos aleatorios dentro de él. Pueden ser simples de automatizar. Ej: Fuzz testing o fuzzing

Técnicas basadas en código

- **Criterios de control basados en el flujo:** Tienen como objetivo cubrir todas las declaraciones, bloques de declaraciones o combinaciones específicas de declaraciones en un programa. El más fuerte de los criterios basados en el flujo de control es la prueba de ruta, cuyo objetivo es ejecutar todas las rutas de flujo de control de entrada a salida en el gráfico de flujo de control de un programa.
- **Criterios basados en el flujo de datos:** El grafo de flujos de control tiene anotaciones con información acerca de cómo las variables del programa se definen, usan y destruyen. Todos sus caminos de uso-definición, requieren que para cada variable, se ejecute cada uno de los segmentos del camino del flujo de control de esa variable a un uso de esa definición.
- **Modelos de referencia para pruebas basadas en código (gráfico de flujos, gráfico de llamadas):** No es una técnica en sí misma. La estructura de control

de un programa se representa usando gráficos de flujos en las técnicas de pruebas basadas en código. Un gráfico de flujo es un gráfico dirigido cuyos nodos y arcos se corresponden con pares del programa, los nodos podrían representar líneas de código o secuencia de líneas ininterrumpidas y los arcos la transferencia de control entre otros.

Técnicas basadas en errores

- **Conjeturar errores:** Se intenta predecir cuáles pueden ser los errores más probables, basándose en otras aplicaciones similares y la experiencia del tester.
- **Pruebas por mutación:** Se crean “mutantes”, que son versiones del programa con modificaciones sintácticas y cada caso de prueba se aplica al SW original y al “mutante” y si una prueba logra identificarlo, se dice que se ha “matado” al mutante.

Técnicas basadas en el uso

- **Perfil operativo:** El objetivo es deducir la confiabilidad fuera del SW cuando se encuentre en su entorno operativo. Lo que se intenta hacer es que el entorno de pruebas sea lo más parecido al entorno operativo real que tendrá el SW.
- **Heurística de observación del usuario:** El objetivo es detectar posibles problemas en el diseño de las interfaces de usuarios. En este tipo de pruebas pueden realizarse a los usuarios: recorridos cognitivos, análisis de afirmaciones, observaciones de campo, pensamiento en voz alta, cuestionarios y entrevistas.

Técnicas de prueba basadas en modelos:

- **Tablas de decisión:** Los casos de prueba se derivan sistemáticamente considerando cada combinación posible de condiciones (entradas) y sus correspondientes acciones (salidas) resultantes.
- **Máquinas de estado finito:** Al modelar un programa como una máquina de estados finitos, se pueden seleccionar pruebas para cubrir los estados y transiciones.
- **Especificaciones formales:** Al expresar las especificaciones en un lenguaje formal, por ejemplo: *Testing and Test Control Notation versión 3* (TTCN3) es

posible la derivación automática de casos de prueba funcionales y la verificación de los resultados de las pruebas.

- **Modelos de flujo de trabajo:** Especifican una secuencia de actividades (escenarios) realizadas por humanos y/o aplicaciones de software. Cada una de ellas, que puede estar representada mediante símbolos gráficos, constituye un flujo de trabajo que debe verificarse.

Técnicas basadas en la naturaleza de la aplicación: Se trata de pruebas que deben aplicarse a aplicaciones específicas. Ej:

- Pruebas orientadas a OO
- Pruebas basadas en componentes
- Pruebas para internet
- Pruebas para GUI
- Pruebas para programas concurrentes
- Pruebas de conformidad de protocolos
- Pruebas para sistemas de tiempo real
- Pruebas para sistemas de seguridad crítica

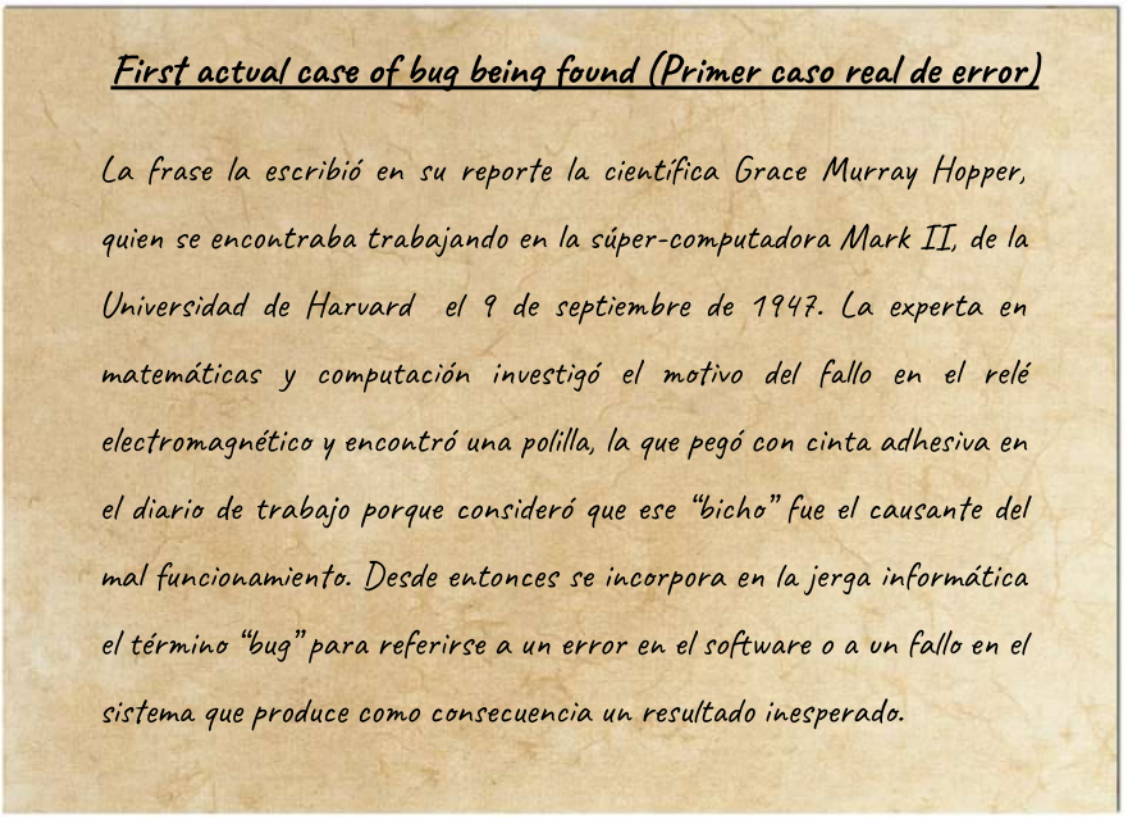
Seleccionando y combinando técnicas

- **Funcional y estructuralmente:** Son complementarias a las pruebas basadas en las especificaciones y el código.
- **Deterministas vs aleatorias:** Para lograr la aleatoriedad de las entradas de datos se les asigna una probabilidad de distribución o perfil teniendo en cuenta la frecuencia en que se dan en la realidad.

4.5 Clasificación de los defectos

Las actividades de SQC están presentes en las etapas del desarrollo y el mantenimiento del software. En cada una de ellas, los profesionales pueden encontrar diversos tipos de defectos que deben identificarlos e informarlos con precisión a los miembros del equipo de desarrollo o plasmarlos en las auditorías y revisiones. Por tal motivo, el *SWEBOK*[9] realiza la siguiente clasificación entre los tipos de defectos:

- ***Error Computacional***: Diferencia entre una condición o valor computado, observado o medido y el verdadero, especificado, o teóricamente correcto valor o condición.
- ***Error***: Acción humana que produce un resultado incorrecto.
- ***Defecto***: Una “imperfección o deficiencia en un producto de trabajo donde ese producto de trabajo no alcanza sus requerimientos o especificaciones y necesita ser reparado o reemplazado.” Un defecto es causado por una persona cometiendo un error.
- ***Falta***: Un defecto en el código fuente. La codificación de un error humano en el código fuente. En la jerga informática se lo conoce como: *bug*.
- ***Falla***: Un “evento en el cual un sistema o componente no lleva a cabo una función requerida dentro de límites específicos.” Una falla es producida cuando una falta es encontrada por el procesador bajo condiciones específicas.


First actual case of bug being found (Primer caso real de error)

La frase la escribió en su reporte la científica Grace Murray Hopper, quien se encontraba trabajando en la súper-computadora Mark II, de la Universidad de Harvard el 9 de septiembre de 1947. La experta en matemáticas y computación investigó el motivo del fallo en el relé electromagnético y encontró una polilla, la que pegó con cinta adhesiva en el diario de trabajo porque consideró que ese “bicho” fue el causante del mal funcionamiento. Desde entonces se incorpora en la jerga informática el término “bug” para referirse a un error en el software o a un fallo en el sistema que produce como consecuencia un resultado inesperado.

Figura 4.5 Historia del término “bug” en la informática

En el ámbito informático el error o falla en un sistema se lo menciona como “bug”. En la Figura 4.5 se presenta una reseña sobre el origen de ese significado.

4.6 Cuadrantes de las Pruebas Ágiles

Según el libro *“Agile Testing”* de *Lisa Crispin* y *Janet Gregory*[1], existen tipos de pruebas que cumplen propósitos específicos y que pueden representarse mediante una matriz de cuadrantes. Cada una de las partes que la componen ayuda al equipo de control de calidad a asegurarse de que han considerado todos los tipos de pruebas necesarias para la aplicación.

Como se puede observar en la Figura 4.6, la matriz de pruebas ofrece a los equipos involucrados un panorama que refleja qué parte de la aplicación está cubierta por cada test, cuál falta verificar y quienes son los responsables en cada uno de los cuadrantes.

Cada sección de pruebas de la matriz es independiente de las restantes y pueden realizarse las pruebas que se especifican en cada una de las partes sin necesidad de seguir un orden determinado, la cantidad de veces que sea conveniente para el equipo.

La naturaleza de cada proyecto de software, los requerimientos del cliente, la cantidad de recursos asignados y si tiene o no la llamada: “deuda técnica”[12], es decir, tareas técnicas de mantenimiento de la aplicación, van a determinar el tiempo estimado que debe invertirse a los distintos tipos de pruebas que propone cada cuadrante.

El desarrollo ágil comienza con las pruebas de cara al negocio (Q2), porque son las pruebas que le indican al equipo de desarrollo el qué y el cómo debe codificar. Es durante cada una de las etapas de cada una de las iteraciones por las que transitan las aplicaciones que es posible desarrollar, incorporar y ejecutar cualquiera de las pruebas especificadas en los cuadrantes y tomar los outputs de esos tests para retroalimentar tanto al código como así también a los miembros de los equipos involucrados.

Cada sección de pruebas de la matriz es independiente de las restantes y pueden realizarse las pruebas que se especifican en cada una de las partes de la

matriz sin necesidad de seguir un orden determinado y la cantidad de veces que sea conveniente para el equipo.

Figura 4.6 Cuadrante de las pruebas ágiles

Cuadrante inferior izquierdo Q1

El cuadrante inferior izquierdo está integrado por los tests unitarios y de componentes que se consideran: “pruebas de tecnología o del programador”, El propósito principal de los tests del Q1 se centran en aplicar el *desarrollo basado en pruebas (TDD)* o el *desarrollo dirigido por comportamiento (BDD)*[11]

Dependiendo de la naturaleza de cada proyecto, las pruebas del Q1 pueden no ser los primeros tests que se escriban para cada historia, pero ayudan a los desarrolladores a guiar el diseño y el desarrollo de cada historia que el equipo emprende hasta lograr la funcionalidad requerida.

Cuando el equipo decide utilizar la técnica TDD, las pruebas son codificadas utilizando el mismo lenguaje de programación de la funcionalidad a testear, “las pruebas de los programadores” son normalmente parte de un proceso automatizado que se ejecuta con cada registro de código, lo que brinda al equipo una retroalimentación instantánea y continua sobre su calidad interna[1]. En Extreme Programming Explained, Kent Beck escribe: "Existe una extraña relación entre la calidad interna y externa. La calidad externa es la calidad medida por el cliente. La calidad interna es la calidad medida por los programadores"[15]. Todas las actividades

del cuadrante 1 están orientadas a garantizar la mayor calidad interna posible en el producto de software a entregar.

Cuadrante superior izquierdo Q2

Las pruebas del cuadrante 2 respaldan el trabajo del equipo de desarrollo a un nivel más alto que las pruebas del cuadrante 1. Estas pruebas de cara al negocio, también denominadas pruebas de cara al cliente o pruebas de cliente, definen la calidad externa y las características que los clientes están esperando del software. En equipos ágiles en los que se espera una participación activa del cliente, las pruebas del Q2 son redactadas utilizando un lenguaje del dominio empresarial.

En determinados proyectos, es el equipo del cliente el que proporciona ejemplos del software que necesita al equipo de desarrollo Teniendo en cuenta estos ejemplos, se realizan pruebas y se ejecutan a nivel funcional y cada una de ellas verifica una condición de satisfacción empresarial.

Las pruebas de *maquetas y wireframes* son un grupo de pruebas que también pertenece a este cuadrante y que se utilizan para comunicar los diseños de la GUI consensuada con el cliente a los desarrolladores antes de que comiencen a codificarlas.

Cuadrante superior derecho Q3

El cuadrante 3 clasifica las pruebas de cara al negocio que ejercitan el software en funcionamiento para validar que el software satisfaga las expectativas de los clientes. Las pruebas comerciales que critican el producto tienen una baja posibilidad de automatizarse, ya que en dichas validaciones se requiere de las habilidades y experiencia humana. Sin embargo, el equipo de control de calidad puede apoyarse en las herramientas de automatización para aliviar tareas, como por ejemplo, la configuración de datos requeridos para realizar una prueba.

Las pruebas de aceptación del usuario (UAT) son utilizadas por los usuarios y clientes porque brinda la oportunidad de experimentar y validar las nuevas funciones que el equipo de desarrollo ha incorporado al sistema. También sirven para ir incorporando ideas para posibles cambios y funcionalidades a incorporar en nuevas historias futuras.

En el tercer cuadrante también se encuentran las pruebas de usabilidad que critican el producto. Evaluar o criticar al producto desde una orientación de negocio consiste en manipular el sistema bajo prueba y tratar de recrear las experiencias reales de los usuarios finales. En este tipo de tests, el profesional de control de calidad analiza las características de la aplicación con el usuario final en mente. Se nutre de las especificaciones de cada historia, pero también de su experiencia en utilización de otras aplicaciones similares para determinar si el producto de software es fácil e intuitivo. Pero para obtener el conocimiento real de cómo los usuarios utilizan el software en desarrollo se convoca a un target de personas y se las estudia mientras interactúan con la aplicación. De esa experiencia se sacarán las conclusiones que serán un feedback para todos los equipos involucrados en el desarrollo de la aplicación.

Las pruebas exploratorias son un enfoque reflexivo fundamental para el tercer cuadrante. El objetivo es corroborar que el equipo de desarrollo ha desarrollado el software con el valor comercial requerido y esperado por los clientes. Para ello los testers intentan emular el comportamiento y las acciones que un usuario real realizaría con la aplicación cada vez que intente realizar una tarea determinada. Si bien se trata de una prueba que debe realizar un humano, puede automatizarse la parte de configuración de datos a través de scripts para agilizar con el proceso.

Cuando el equipo de desarrollo trabaja bajo metodologías ágiles, los responsables del control de calidad del software pueden realizar pruebas exploratorias cada vez que se termina una iteración.

Cuadrante inferior derecho Q4

Las pruebas que componen el cuarto y último cuadrante son imprescindibles para cualquier aplicación de software, independientemente de la metodología de desarrollo que la guíe y tienen como objetivo criticar las características del producto como el rendimiento, la solidez y la seguridad, desde un punto de vista tecnológico.

La validación de los requisitos no funcionales tienen como finalidad ayudar al equipo de control de calidad a determinar el nivel de calidad necesaria que debe alcanzar el producto Los aspectos que se deben incluir son: problemas de configuración, seguridad, rendimiento, administración de memoria, confiabilidad,

interoperabilidad y escalabilidad, recuperación y conversión de datos. La información que brindan estas pruebas puede resultar en nuevas historias y tareas cuando se completa el ciclo de retroalimentación desde las pruebas que critican el producto hasta las pruebas que impulsan cambios que mejorarán los aspectos no funcionales del producto.

La creación y ejecución de las pruebas que realizan los miembros de SQC pueden incluir tareas manuales y automatizaciones. Las personas que se encargan de realizar cada prueba no funcional, deben contar con un conjunto de habilidades y herramientas especializadas. Pueden ser miembros del equipo que adquieren una capacitación especial o directamente la organización decide contratar a terceros idóneos para tal fin.

4.7 Resumen

Las pruebas de software son una actividad fundamental en el proceso de control de calidad. Son un elemento crítico para la garantía de calidad del software y representan una revisión final de las especificaciones, del diseño y de la codificación[9].

Los tipos de pruebas para validar una aplicación se diferencian entre sí por la información en que se basan para realizar la validación. En el tipo de pruebas de “caja blanca” se utilizan los datos que se tiene sobre el diseño del software o sobre su código fuente, mientras que en los tipos de “caja negra” se pretende observar los valores de salida en función de valores de entrada para comprobar la funcionalidad del sistema sin conocer su estructura interna.

El *ISTQB*[18] expone una clasificación de los tipos de pruebas en base al objetivo que persiguen como: funcionales, no funcionales, estructurales y de mantenimiento. Los subtipos de mantenimiento y no funcionales forman parte del grupo de los tipos de prueba de caja negra; mientras que las pruebas estructurales y funcionales corresponden al tipo de prueba de caja blanca.

Los profesionales de SQC aplican durante todo el ciclo de vida del software las técnicas que consideran más adecuadas para lograr efectuar los tipos de pruebas necesarias para validar los requerimientos y los aspectos que consideren más relevantes de tester de cada una de aplicaciones de software.

El Cuadrante de las pruebas ágiles[1] proporciona un panorama real de las pruebas desarrolladas y ejecutadas por el equipo de desarrollo y SQC. Las pruebas de los cuadrantes 1 y 2 que define la matriz son de tipo de pruebas de caja blanca y las del 3 y 4 corresponden al tipo de prueba de caja negra.

Conocer las técnicas y tipos de pruebas que se realizan al software es de suma importancia para que los profesionales del área de SQC realicen su labor teniendo en cuenta la naturaleza de cada aplicación y los objetivos para los cuales fue creada[4].

Capítulo 5 Testing automatizado y continuo

5.1 Introducción

Surgida en los primeros años del nuevo milenio bajo el objetivo de ampliar la cobertura de los test y de reducir los tiempos y el número de recursos involucrados en la validación de las aplicaciones, la automatización de pruebas, es considerada como una actividad de control de calidad que debe ser diseñada, implementada y también testeada.

En un proceso de prueba manual, cada una de las personas responsables de verificar el correcto funcionamiento de un software debe seguir una serie de pasos para obtener un resultado. Cada vez que necesita volver a comprobar este resultado, debe volver a ejecutar esas mismas acciones. Esa y cualquier tarea tediosa o repetitiva involucrada en el desarrollo de software es candidata a la automatización. Los tests o pruebas automatizadas tienen como objetivo convertir, parcial o totalmente, los procesos de verificación de las funcionalidades que se realizan de forma manual en software reutilizable.

En el presente capítulo se desarrolla el concepto de “test automatizado” subrayando los pros y contras de su adopción en el marco de la integración continua.

5.2 Test automatizados: los beneficios

La automatización proporciona consistencia al proceso de testeos, ya que cuando se corren las pruebas automatizadas se hacen de manera sistemática. Es decir que todas las veces que se corre un test automatizado ejecuta las mismas instrucciones con los mismos valores de entrada. Por lo que ya se conoce cuáles deben ser las salidas esperadas para tales testeos. Cuando un tester hace la misma prueba manualmente, puede equivocarse u omitir pasos; tomar acciones diferentes y llegar a resultados no esperados o equivocados.

Los responsables de realizar tareas de mantenimiento de software necesitan la certeza de que los cambios realizados no afectarán la funcionalidad de la aplicación. Los test automatizados proporcionan seguridad y respaldo al equipo de desarrollo al

momento de reducir la "*deuda técnica*"[12] y de enfrentar las refactorizaciones de código. La ejecución de un conjunto automatizado de pruebas cada vez que se realiza un cambio en el código ayuda a garantizar que los errores de regresión se detecten de manera temprana, lo que posibilita que los errores detectados ocasionen la pérdida monetaria de menor impacto posible.

La ejecución de las tareas automatizadas consumen menos tiempo en relación al que tardaría en efectuarse la misma tarea en forma manual, es por tal motivo que el equipo de control de calidad puede enfocarse en llevar adelante otras actividades de chequeos que antes no hubiesen podido realizarlas por falta de tiempo o por la complejidad de las pruebas.

Los equipos que tienen una buena cobertura de las pruebas de regresión y la configuración de las pruebas exploratorias pueden enfrentar con seguridad la creación de pruebas sofisticadas y complejas que muestran información de la aplicación e investigan las partes potencialmente débiles del sistema. La retroalimentación constante que brinda la automatización permite que la resolución de los problemas sea más rápida.

Las pruebas automatizadas son siempre una imagen precisa de cómo funciona y se comporta el código de la aplicación, por eso se dice que se convierten en documentación "viva" del sistema[1].

5.3 Posibles aspectos negativos de las pruebas automatizadas

Comenzar a sumergirse en el mundo de las pruebas automatizadas requiere de tiempo y aptitudes para incorporar el conocimiento necesario para realizarlas y también para configurar las herramientas para ejecutarlas y monitorearlas. Cada una de las aplicaciones que se desee automatizar puede implicar volver a incorporar nuevos conocimientos.

Los tests automatizados pueden requerir una inversión importante de horas de desarrollo y el equipo puede no contar con ese tiempo disponible, dicha circunstancia puede llegar a considerarse como una característica negativa de las pruebas automatizadas.

Otro aspecto desfavorable que presentan las automatizaciones de pruebas se da en aplicaciones que presentan gran dinamismo, ya que las pruebas automatizadas quedan obsoletas o desactualizadas en forma parcial o en su totalidad cada vez que se agrega una funcionalidad o se modifica una existente, lo que implica que los encargados de ellas deban mantenerlas actualizadas al mismo ritmo que cambia la aplicación que se desea testear.

Cuando los equipos de desarrollo trabajan en una incidencia nueva, las automatizaciones de pruebas pueden presentar desventajas, una de ellas es que, para contar con la prueba automatizada del nuevo desarrollo que aún no esté presente en un ambiente productivo, la misma debe realizarse utilizando los ambientes de prueba en los que se encuentra el nuevo código el cual va variando constantemente, lo que implica que, para tener la prueba lista para cuando se dispone la nueva funcionalidad, se debe ir actualizando el test al mismo tiempo, lo que conlleva una inversión de tiempo extra.

Otro aspecto negativo se puede presentar cuando las automatizaciones fallan y entender el porqué de ese fallo puede requerir de un análisis para detectar si la prueba falló porque se realizaron cambios en la aplicación o porque la automatización efectivamente detectó un error.

5.4 Automatizaciones en equipos ágiles

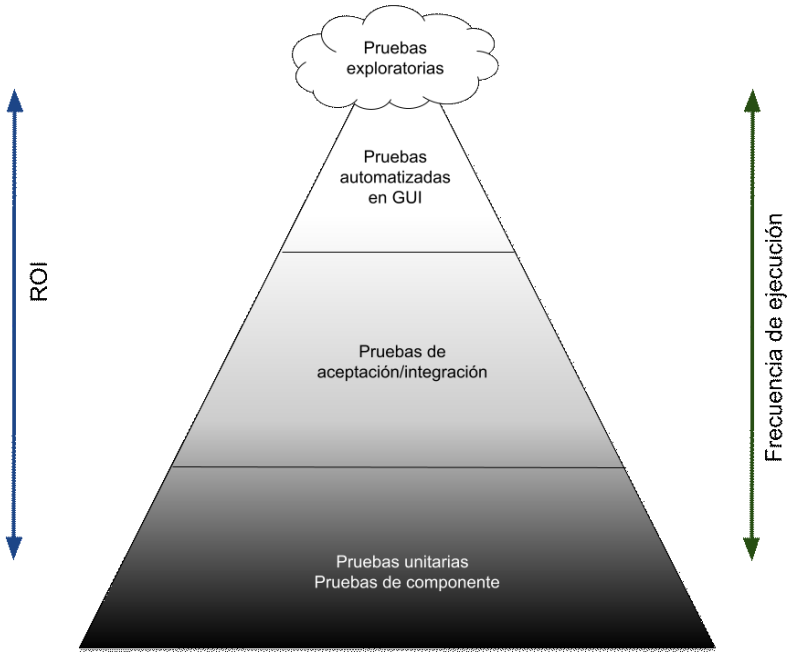
Para trabajar por la calidad interna del software se debe llevar a cabo una estrategia adecuada de automatización de pruebas. Es por ello que los equipos ágiles pueden sumar a la estimación de tiempo y/o esfuerzo de cada historia el tiempo que les lleve la automatización de pruebas vinculadas a esas incidencias.

Un entorno o ambiente de desarrollo de software (SDE) es el conjunto de herramientas de hardware y de software que se utilizan para construir aplicaciones de software. Cada ambiente de desarrollo persigue un propósito específico. Las organizaciones implementan sus ambientes de desarrollo dependiendo de sus necesidades, pero generalmente se dispone de: un ambiente local; uno de pre-producción o sandbox y uno de producción. Antes de deployar al ambiente productivo la nueva iteración, es indispensable que se ejecute un conjunto completo de pruebas de regresión. Para que todas las funcionalidades que se deben testear

estén alineadas con el código de la aplicación que se actualiza constantemente, los programadores deben tomarse un tiempo para ayudar con las pruebas de regresión manual o el equipo tiene que contratar más testers. Por tal motivo desde el punto de vista de negocio, se puede obtener un buen retorno de inversión (ROI) si se invierte tiempo en el desarrollo de testing automatizado a conciencia, es decir, si se implementa las pruebas necesarias.

5.5 Pirámide de la automatización del Test

En su libro *Succeeding with Agile, Mike Cohn*[14] propuso un gráfico en forma de pirámide en el que describió cómo deberían estructurarse los distintos niveles de test automatizados. Como se presenta en la Figura 5.1, la clasificación de automatización de pruebas ágiles muestra tres capas diferentes de pruebas automatizadas. En el nivel inferior se encuentran las pruebas unitarias y/o de componentes, posteriormente les siguen las pruebas de API (Application Programming Interface) y/o servicios web; y por último las pruebas de GUI (Graphics User Interface) o E2E (End To End) en aplicaciones Web. Cada una de los tipos de pruebas persigue un objetivo, exige un grado de esfuerzo y un retorno de inversión determinado.



5.1 Pirámide para la automatización de pruebas

La primera clasificación corresponde al **nivel inferior**, que debe ser una base sólida de automatizaciones. La componen principalmente las pruebas unitarias y pruebas de componentes, que son las pruebas tecnológicas que respaldan al equipo.

Por lo general, los tests que forman parte del nivel inferior de la pirámide están escritos en el mismo lenguaje que la aplicación de software y representan la mayor parte de las pruebas automatizadas porque son las automatizaciones más rápidas de ejecutar y de desarrollar, brindan el nivel de ROI más importante y son menos costosas de implementar para el equipo de desarrollo. En los equipos de desarrollo ágil se intenta impulsar la mayor cantidad de pruebas en esta capa como sea posible. Cada vez que falla una prueba unitaria, el desarrollador puede encontrar el error específico de manera más eficiente que cuando ese mismo error es reportado por un miembro del equipo de control de calidad, lo que permite ahorrar tiempo y dinero a toda la organización.

El nivel intermedio de la pirámide propuesta por *Cohn* es el nivel en el que se encuentran la mayor parte de las pruebas automatizadas de cara al negocio escritas para apoyar al equipo[1]. Se trata del conjunto de pruebas funcionales que verifican lo especificado en una historia. Cubren el conjunto de pruebas de funcionalidad más grandes que la capa inferior de la pirámide, la que estaba compuesta por las pruebas unitarias y de componentes. Operan a nivel de la interfaz de programación de aplicaciones (API), una colección de funciones que pueden ser ejecutadas por otras aplicaciones o componentes de software.

Estas validaciones generalmente se escriben en un lenguaje específico de dominio que los clientes sean capaces de comprender. Los casos de prueba tienen que ver con configurar valores de entrada y comparan los outputs que se obtuvieron como respuesta con los resultados esperados.

Las llamadas a la API y sus tests pueden comenzar a desarrollarse en la primera etapa del desarrollo del software. Todas las aplicaciones se componen de varios servicios. Un servicio es algo que la aplicación hace en respuesta a alguna entrada o conjunto de entradas. La prueba de nivel de servicio consiste en probar los servicios de una aplicación por separado de su interfaz de usuario[14].

Al comprobar la funcionalidad directamente sin pasar por la GUI son pruebas menos costosas de escribir y mantener que las pruebas que utilizan la interfaz gráfica.

Las pruebas de la API abarcan un campo más amplio de validación de la aplicación y genera una interacción entre componentes, lo que requiere de un mayor lapso de tiempo para su ejecución que las pruebas de unidad.

Se considera que las pruebas del nivel intermedio de la pirámide de automatización brindan un nivel de retroalimentación intermedio, por lo que el valor de ROI que aportan es moderado.

En el nivel superior de la pirámide de los tests se encuentran las pruebas de interfaz de usuario automatizadas. Se desarrollan después de que se completa el código de la aplicación, por lo que se incorporan a la suite de regresiones. Proporcionan información importante, pero también brindan el ROI más bajo

Cohn afirma que en la cúspide se deben situar las validaciones que el equipo debería invertir menos esfuerzo porque tienen los siguientes atributos negativos:

- Frágil/quebradizas. Un pequeño cambio en la interfaz de usuario puede romper muchas pruebas. Cuando esto se repite muchas veces en el transcurso de un proyecto, los equipos simplemente se rinden y dejan de corregir las pruebas cada vez que cambia la interfaz de usuario.
- Es caro de escribir. Un enfoque rápido de captura y reproducción para grabar pruebas de interfaz de usuario puede funcionar, pero las pruebas grabadas de esta manera suelen ser las más frágiles. Escribir una buena prueba de interfaz de usuario que siga siendo útil y válida lleva tiempo.
- Consume mucho tiempo. Las pruebas que se ejecutan a través de la interfaz de usuario suelen tardar bastante en ejecutarse. En ocasiones tardan tanto en ejecutarse que no se pueden ejecutar todas las noches, y mucho menos varias veces al día.

Los tests manuales como pruebas exploratorias

Cohn considera que las pruebas manuales deben verse principalmente como: “una forma de realizar pruebas exploratorias”. Este tipo de prueba implica un ciclo rápido a través de los pasos de planificación de pruebas, diseño de pruebas y ejecución de pruebas. Las pruebas exploratorias deben incluir ciclos cortos de generación de retroalimentación a través de estos pasos de una manera análoga a

ciclo corto de desarrollo impulsado por pruebas de refactorización de código de prueba.

Las pruebas exploratorias pueden ser útiles para identificar casos de prueba faltantes, que posteriormente se pueden incorporar al nivel piramidal correspondiente. También pueden verse como una oportunidad para descubrir requerimientos que faltaron especificar en la historia del usuario.

5.6 Herramientas para automatizar pruebas en aplicaciones web

Pasaron tres décadas desde los inicios de las pruebas automatizadas, y desde ese entonces hasta la actualidad, el software para realizar la tarea de automatización es cada vez más fácil de utilizar con conocimientos técnicos limitados. La evolución de las herramientas convierte a las pruebas automatizadas en un proceso cada vez más adoptado por las organizaciones. Existen software de automatización para crear datos de prueba, configurar escenarios para pruebas manuales, pruebas de carga, pruebas unitarias, de rendimiento, de seguridad, análisis de resultados y automatización de los procesos de integración continua.

Cada equipo de desarrollo seleccionará el software para automatizar teniendo en cuenta el conjunto de habilidades y conocimientos de sus integrantes, la tecnología que utiliza la aplicación, la prioridad que le asigna a la tarea de automatizar. Los aspectos como el presupuesto asignado y el tiempo disponible son variables que también se sumarán en la balanza.

A continuación se presentan ejemplos de herramientas utilizadas para realizar distintos tipos de pruebas sobre aplicaciones web.

JUnit es una plataforma que se utiliza para ejecutar frameworks de prueba en la JVM[22]. Se trata de un conjunto de librerías para hacer pruebas unitarias para aplicaciones en el lenguaje Java. Está publicada bajo Licencia Common Public License.

Su objetivo es evaluar el funcionamiento esperado de las clases y los métodos y exponer los resultados (runners) en modo texto, gráfico o como tarea. Es una

alternativa para automatizar las pruebas de regresión. Se puede integrar a los entornos de desarrollo de los entornos de desarrollo integrados (IDEs) más populares (IntelliJ IDEA, Eclipse, NetBeans y Visual Studio Code) y herramientas de compilación (consulte Gradle, Maven y Ant).

Mockito[23] es un framework para simular datos que permite escribir pruebas a través de una API.
La comunidad de Massive StackOverflow votó a Mockito como el mejor marco de simulación para Java

Está basado en expect-run-verify como EasyMock o jMock. Algunas de las características por las cuales la comunidad de Massive StackOverflow votó a Mockito como el mejor framework de simulación para Java:

- Simula clases concretas e interfaces
- Tiene soporte para la sintaxis sugar - @Mock
- Muestra en forma clara los errores de verificación. Basta con hacer clic en el seguimiento de la pila para ver la verificación fallida en la prueba; luego puede hacer clic en la causa de la excepción para navegar a la interacción real en el código.
- Permite una verificación flexible y en orden.
- Admite el chequeo del número exacto de veces y al menos una vez (exact-number-of-times y at-least-once verification)
- Verificación flexible o apéndice utilizando comparadores de argumentos (anyObject(), anyString() o refEq() para la coincidencia basada en la reflexión)
- Permite crear comparadores de argumentos personalizados o usar comparadores de Hamcrest existentes

Spring[26] es un framework Open Source diseñado para el desarrollo de aplicaciones para la plataforma Java. Sus funciones básicas son: la inyección de dependencia y la inversión de Control. Contiene un conjunto de librerías (Spring Testing) integradas directamente con JUnit y diseñadas para crear las pruebas de integración y pruebas unitarias fomentando la metodología de TDD.

Spring Test provee:

- Acceso al factory de beans directamente desde los test unitarios.
- Inyección automática de la clase bajo test en el TestCase.
- Rollback automático de transacciones contra bases de datos.
- Acceso a la conexión jdbc para realizar querys de comprobación.

Postman[27] es una herramienta gratuita para investigar y testear APIs. Está disponible como: extensión de Chrome, aplicación de escritorio, CLI y una API. Postman permite realizar llamadas a la API (REST, SOAP o HTTP simple) e inspeccionar las respuestas de cualquier tamaño. También brinda soporte para formatos de datos como OpenAPI GraphQL y RAML. Mediante la CLI los testers y desarrolladores pueden ejecutar las colecciones directamente.

TestNG[16]es un framework para escribir y ejecutar pruebas para diversos propósitos: unitarias, funcionales, end-to-end, integración. Utiliza Java y está basado en JUnit y NUnit. A la hora de escribir las suites de tests provee funcionalidades como: soporte para anotaciones (JDK 5 , JDK 1.4 y JavaDoc annotations), pruebas para data-driven testing (with @DataProvider) y pasaje de parámetros. Para realizar una prueba en TestNg se requiere realizar sólo tres pasos:

- Escribir la prueba con las anotaciones TestNG que apliquen al propósito del test,
- Crear un archivo de configuración con la información sobre el/los test/s que se necesiten ejecutar,
- Ejecutar TestNG.

Selenium[8] es un conjunto de software (Selenium QA) utilizado para realizar pruebas en aplicaciones web en diferentes navegadores y plataformas. Es de código abierto y se distribuye bajo la licencia de

Apache 2,0. Los Test Scripts de Selenium pueden escribirse en varios lenguajes de programación como Java, C # y Python.

La lista de herramientas que la componen son:

- Selenium Integrated Development Environment (IDE): Está implementado como una extensión de Firefox y permite grabar, editar y depurar pruebas.
- Selenium Remote Control (RC): Creado por Paul Hammant, se trata de un servidor escrito en Java que actúa como un proxy HTTP para enmarcar a Selenium Core y la aplicación web que se está probando bajo el mismo dominio.
- WebDriver: Framework de prueba multiplataforma que controla el navegador web desde el nivel del sistema operativo.
- Selenium Grid: Es un servidor que permite ejecutar pruebas en paralelo en múltiples máquinas remotas y manejar diferentes versiones y configuraciones de manera centralizada.

Jasmine[31] es un framework para el desarrollo de pruebas en JavaScript impulsado por el comportamiento para código (*Behavior Driven Development)*. Su objetivo es automatizar el comportamiento real del usuario sobre la página web sin utilizar el DOM. Es de código abierto, no tiene dependencias con otros frameworks y cuenta con una sintaxis intuitiva que intenta asemejarse a un lenguaje natural.

KarmaJS[17] es un *test-runner,* es decir, una herramienta simple que permite correr las pruebas en un entorno local como así también en el servidor de integración continua. Su objetivo es el de ejecutar código JavaScript en múltiples navegadores con diferentes resoluciones (escritorio, móvil, tablets, etc.). Está configurado con Jasmine.js como framework de testing por defecto. Es posible obtener informes de cobertura mediante el framework istanbul.

Protractor[15] es un framework de prueba de un extremo a otro (E2E) para aplicaciones desarrolladas en Angular y AngularJS. La herramienta corre los tests contra su aplicación que es ejecutada en un navegador real.

Construido sobre Selenium[40] WebDriverJS, Protractor utiliza eventos nativos y controladores específicos del navegador para interactuar con la aplicación como lo haría un usuario.

La característica más importante con la que cuenta Protractor es la: “espera automática”, por lo que no se requiere usar las sentencias de waits y sleeps en los tests para esperar a que se ocurra la sincronización entre el test y la página web.

Protractor admite estrategias de localizador específicas de Angular, característica que ahorra tiempo y esfuerzo a las personas que quieran automatizar software desarrollado en ese lenguaje.

5.7 La integración continua en el proceso de desarrollo

En los equipos ágiles, cuando una historia cumple con las etapas de análisis y desarrollo de la nueva funcionalidad, es necesario integrar el código que se generó en el entorno local del desarrollador con el código de la aplicación para generar una release candidate (RC).

La versión de la aplicación con los cambios de la nueva iteración debe subirse en los ambientes de prueba y, si las pruebas dan el resultado esperado, ese nuevo requerimiento está listo para pasar al ambiente productivo, donde deben continuar las pruebas que sean posibles de realizar. La integración continua, es una práctica de desarrollo de software, en la que los miembros de un equipo integran su trabajo con frecuencia.

Cada integración, es verificada por un sistema automatizado de construcción (incluyendo las pruebas), para detectar errores de integración lo más rápido posible. Este enfoque ayuda a reducir los problemas de integración y permite desarrollar el producto de forma más rápida. La IC conduce a problemas de integración significativamente reducidos y permite que un equipo desarrolle software cohesivo más rápidamente[11]. La integración continua establece un proceso de compilación consistente y repetible que se inicia automáticamente cuando se detecta un cambio

en el repositorio de control. Su finalidad es garantizar la calidad del software mediante la retroalimentación inmediata.

Paul M. Duvall considera que la IC ofrece al equipo de desarrollo los siguientes aspectos positivos[11]:

- **Reduce riesgos:** La IC facilita la detección de defectos en el código fuente de la aplicación, la medición del estado del software y la reducción de supuestos o malentendidos entre miembros del equipo. Los siguientes son algunos de los riesgos que IC ayuda a mitigar.
    - Falta de software cohesivo e implementable.
    - Descubrimiento tardío de defectos.
    - Software de baja calidad.
    - Falta de visibilidad del proyecto.
- **Reduce los procesos manuales repetitivos:** Reducir los procesos repetitivos ahorra tiempo, costos y esfuerzo. Estos procesos repetitivos pueden ocurrir en todas las actividades del proyecto, incluida la compilación de código, la integración de la base de datos, las pruebas, la inspección, la implementación y la retroalimentación. Al automatizar la IC, tiene una mayor posibilidad de garantizar que:
    - El proceso se ejecuta de la misma manera cada vez.
    - Se sigue un proceso ordenado.
    - Los procesos se ejecutarán cada vez que se produzca una confirmación en el repositorio de control de versiones.
- **Genera software implementable en cualquier momento y en cualquier lugar:** El software implementable es el activo más tangible para los "externos", como clientes o usuarios.
- **Permite una mejor visibilidad del proyecto:** IC proporciona la capacidad de notar tendencias y tomar decisiones efectivas, y ayuda a proporcionar el valor para innovar con nuevas mejoras.
- **Establecer una mayor confianza en el producto de software del equipo de desarrollo:** Con cada compilación, el equipo sabe que se ejecutan pruebas contra el software para verificar el comportamiento, que se cumplen los

estándares de diseño y codificación del proyecto y que el resultado es un producto funcionalmente comprobable.

La integración continua es un proceso de compilación consistente y repetible que se inicia automáticamente cuando se detecta un cambio en el repositorio del código fuente de la aplicación. Los tests automatizados forman parte de la compilación y el equipo de desarrollo debe tener disciplina para implementarlos y mantenerlos.

Paul Duvall cree que el código fuente es tan confiable como la cobertura de las pruebas, y las pruebas son tan valiosas como su frecuencia de ejecución. Por tal motivo establece una categoría de pruebas automatizables: unidad, componente, sistema y funcional. Las pruebas unitarias se pueden ejecutar durante los registros; pruebas funcionales, de componentes y de sistemas en un intervalo regular, como puede ser una compilación secundaria.

La confiabilidad del sistema se logra garantizando la confiabilidad a nivel de objeto individual, es decir, asegurar y medir la confiabilidad en el nivel más bajo de la aplicación, que solo puede lograrse desarrollando y ejecutando pruebas unitarias.

5.8 Herramientas para testing con integración contínua

A continuación se presentan ejemplos de herramientas utilizadas para la ejecución de automatizaciones de test en un marco de integración continua.

Jenkins[24] es un servidor diseñado para automatizar tareas relacionadas con los procesos de: creación, prueba y entrega o implementación de software.. Es open source (licencia MIT), y está escrito en lenguaje Java. Está basado en el proyecto Hudson.

Jenkins se integra con herramientas de control de versiones (CVS, Subversion, Git, Mercurial, Perforce y Clearcase). Sus funcionalidades se pueden ampliar mediante la instalación de complementos para: construcción, implementación y

automatización de proyecto de software o a través del desarrollo propio de un complemento custom.

Git[32] es un software de control de versiones diseñado para trabajar de manera eficiente el trabajo colaborativo sobre archivos compartidos en un repositorio de código de un proyecto de software. Presenta las siguientes características:

- Requiere de poco tiempo de aprendizaje.
- Ramificación y fusión de código: Git permite tener múltiples versiones del código fuente divididos en “ramificaciones” locales que pueden ser completamente independientes entre sí. Esta característica de Git brinda a los desarrolladores investigar, implementar ideas y probarlas sin preocuparse por el estado en que quedan los archivos del proyecto.

  Una vez que considere finalizada una tarea, el desarrollador puede seleccionar qué y de cuáles archivos desea subir su versión del código para ser fusionada con el código que se encuentra en el servidor.
- Distribuido.
- Aseguramiento de datos: Git garantiza la integridad criptográfica de cada parte de su proyecto.
- Área de ensayo: “staging” es un área intermedia donde cada cambio se pueden formatear y revisar antes de completar el commit.
- Libre y de código abierto.

New Relic[33] es un servicio de software pago destinado a monitorear aplicaciones, registro de logs, configuración de métricas y de notificaciones que se activen cuando se cumple cierto umbral, por ejemplo ERROR RATE mayor al 5% para una aplicación específica enviara una notificación a las personas o grupo owner que se haya especificado para tal aplicacion.

5.9 Resumen

Los tests o pruebas automatizadas tienen como objetivo convertir, parcial o totalmente, los procesos de verificación de las funcionalidades que se realizan de forma manual en software reutilizable. Se trata de un proceso sistemático que busca proporcionar seguridad y respaldo cuando se producen cambios en el código de la aplicación. Los tests automatizados pueden requerir una inversión importante de horas de desarrollo y corren el riesgo de quedar obsoletas o desactualizadas en forma parcial o en su totalidad cada vez que se agrega una funcionalidad o se modifica una existente.

Mike Cohn propuso un gráfico en forma de pirámide en el que describió cómo deberían estructurarse los distintos niveles de test automatizados. En el nivel inferior se encuentran las pruebas unitarias y/o de componentes, posteriormente les siguen las pruebas de API (Application Programming Interface) y/o servicios web; y por último las pruebas de GUI (Graphics User Interface) o E2E (End To End) en aplicaciones Web. Cada una de los tipos de pruebas persigue un objetivo, exige un grado de esfuerzo y un retorno de inversión determinado.

La integración continua es una práctica de desarrollo de software, en la que los miembros de un equipo integran constantemente sus desarrollos al código estable de la aplicación. Cuando se detecta un cambio en el repositorio de control se da comienzo a un proceso de compilación consistente y repetible. Su finalidad es garantizar la calidad del software mediante la retroalimentación inmediata.

Capítulo 6 Casos de aplicación de tests automatizados en empresa de turismo online

6.1 Introducción

El año 1999 estaba llegando a su fin cuando *Roberto Souvirón* y cuatro amigos: *Martín Rastellino, Christian Vilate, Alejandro Tamer* y *Mariano Fiori* dejaron sus compromisos laborales, reunieron inversores y fundaron una puntocom dedicada a un rubro innovador: la venta de tickets de vuelos a través de la web.

Desde aquella startup de raíces argentinas a exponer sus acciones en la bolsa de valores de *Wall Street, Despegar* se ha convertido en una agencia de viajes 100% online que opera en 21 países ofreciendo paquetes de viaje, vuelos, hoteles, alquiler de autos, buses, tickets, tours, actividades y servicios adicionales, como asistencia al viajero, traslados y hasta tests PCR para el diagnóstico de COVID-19.

A partir del año 2007 los administradores de *Despegar* deciden que la empresa argentina deje de consumir productos de software externos y comience a desarrollar sus propias soluciones tecnológicas. Es entonces cuando se crea un sector de IT que hoy cuenta con más de 600 desarrolladores de software.

El área de Tecnología de *Despegar* es responsable por la creación y mantenimiento de las aplicaciones internas y externas que utiliza la empresa para ofrecer turismo online a viajeros de América Latina. Otra de las responsabilidades que tiene la dirección de IT es la capacitación constante de sus colaboradores actuales y futuros para lo cual creó el equipo I*T University* en el año 2012.

En el presente capítulo se describen experiencias de automación de pruebas implementadas en equipos de desarrollo de software de *Despegar.com* que utilizan metodologías ágiles.

6.2 El proceso de testing

Con el afán de detectar los errores en una etapa temprana, de optimizar los tiempos y los recursos que la empresa le asignaba a las pruebas manuales, se

impulsó a los equipos a adoptar una metodología basada en la planificación, desarrollo y ejecución de pruebas automatizadas para asegurar el control de calidad.

A lo largo del año 2012 el equipo de *IT University* comenzó a investigar sobre los conceptos de testing ágil, de herramientas y otras estrategias para automatizar, teniendo en cuenta los lenguajes de programación que más se utilizaban en los equipos de desarrollo. El objetivo era capacitar a todos los empleados que se encontraban en los equipos de QA (Aseguramiento de la calidad) realizando pruebas manuales en las oficinas de las ciudades de Buenos Aires y La Plata. Una vez completada la recopilación y elaboración del material, se dio comienzo a la etapa de capacitación mediante un cronograma con clases presenciales y entrega de actividades prácticas, tanto grupales como individuales.

Los temas en los que se hizo hincapié en el curso fueron:

- Carga de los scripts de ejecución y del monitoreo de pruebas automatizadas en Henry, una aplicación interna que estaba encargada de ejecutar y monitorear tareas (procesos y pruebas). Hoy en día fue reemplazada por jobs en Jenkins.
- Creación de test automatizados en TestNg[16] para las API, y Selenium Webdriver para las GUIs.
- Uso de patrones de diseño para el desarrollo de pruebas automatizadas.
- Monitoreo de aplicaciones mediante el servicio NewRelic[33] (todas las aplicaciones de Despegar pueden configurarlo)
- Proceso de integración continua utilizando la herramienta Jenkins.

Cuando culminó el proceso de capacitación, los miembros de los equipos de QA que estaban encargados de la realización de tareas manuales y semi-automatizadas, pasaron a formar parte de cada uno de los equipo de desarrollo bajo el perfil de: automation tester y, desde ese momento, su función pasó a ser la de desarrollar, mantener y monitorear todas las suites de pruebas automatizadas para las aplicaciones backend y frontend del proyecto asignado.

6.3 El arquetipo para automatizaciones

Con el fin de facilitar la tarea de automatización a los miembros del equipo de QA que pasaban de realizar validaciones manuales a desarrollar tests y tenían poca o

nula experiencia en tareas de programación, el equipo de *IT University* creó un arquetipo de automatización en Maven[25] para que funcione como un marco de referencia estándar; se reutilicen las librerías y funcionalidades en común, como por ejemplo las librerías para el mapeo del formato Json a una clase de objeto enviado como parámetro, y se apliquen sobre él las configuraciones correspondientes a cada proyecto. Las principales características de este arquetipo eran las siguientes:

- Posibilidad de soportar diversos ambientes, datos por oficina, y distintos tipos de archivos de prueba.
- Permitir la actualización de archivos de configuración en tiempo de ejecución.
- Facilitar la división e independencia entre componentes.
- Permitir la división de la carga de trabajo del equipo de automatización.
- Brindar facilidad para ser ejecutado por cualquier tipo de framework (tests o procesos).

Como se ilustra en la Figura 6.1, la estructura para la ejecución de los tests estaba conformada por:

- Un servidor de integración continua a través de Jenkins[24] para ejecutar automatizaciones de procesos de build, releases, ejecución de set de pruebas unitarias y de integración, deploys automatizados y generación de reportes.
- Un servidor de archivos en el cual se alojan los artefactos generados por el proceso de integración continua.
- Un ejecutor de procesos y tareas compuesto por workers con configuraciones específicas tanto para las ejecuciones que requerían GUI, como para las que validaba APIs directamente, incluyendo además capacidades de monitoreo

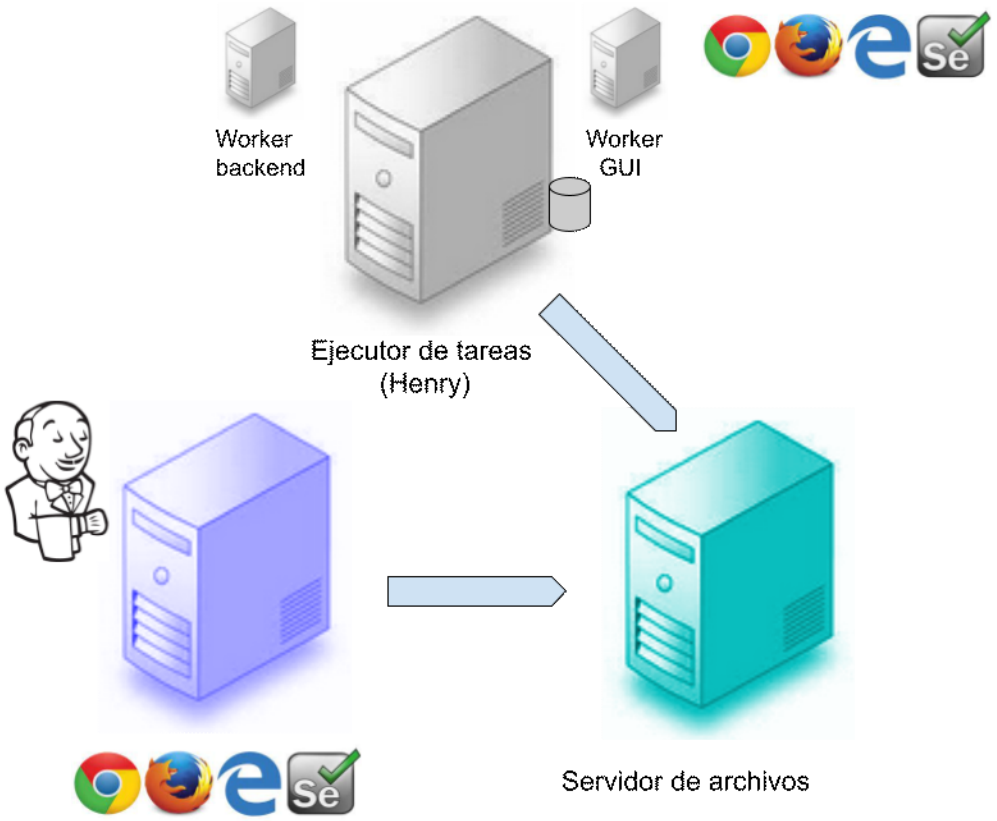


Figura 6.1 Infraestructura propuesta para automatizar tareas de testeo

6.4 Experiencia de tests automatizados en Hoteles de Agencias Afiliadas

Agencias Afiliadas es una unidad de negocio de la empresa que provee de contenido turístico (hoteles, vuelos, paquetes y autos) a aquellas agencias de viajes particulares que tienen una oficina física. A cada una de ellas se les brinda acceso a una plataforma por la que pueden establecer sus condiciones de venta y también un sitio web público que pueden personalizar.

Hoteles era un proyecto de *Agencias Afiliadas*, que tenía a su cargo proveer el flujo completo de compra de alojamientos. Desde el año 2018 esa responsabilidad pasó a ser el equipo de *WL-shopping.*

El proyecto *Hoteles* tenía dos aplicaciones backend desarrolladas en Java: Jano y Júpiter; ambas estaban encargadas de exponer servicios a la aplicación frontend llamada *Vulcano*, desarrollada en Freemarker[28] y Angular[29]. *Jano* debía atender y responder las solicitudes de búsqueda y detalle del flujo de compra de un

hotel; mientras que *Júpiter* era responsable del checkout y de la compra. Para automatizar los testeos de las tres aplicaciones se utilizó como base el arquetipo de automatización provisto por el equipo de *IT University*. Para el caso de Vulcano, se programaron flujos de compra E2E utilizando Selenium[40].

A continuación se describen ejemplos de test automatizados que se realizaron para corroborar el estado de las aplicaciones del proyecto *Hoteles* de *Agencias Afiliadas*.

Proceso de compra de un hotel

Al hecho de garantizar que los clientes tengan una experiencia positiva en la plataforma se le sumó un proceso de migración de API interna, por lo que se desarrolló una suite de tests que, además de generar los Ids de compra teniendo en cuenta todos los nuevos flujos que exige un cambio de API, generase un escenario de compra que permitiese el envío de parámetros para emular tanto una compra exitosa como también una en la que se requiera al usuario completar el pago con otra tarjeta, entre otros escenarios. La herramienta TestNG[16] y sus aspectos positivos se pusieron en la balanza a la hora de realizar las automatizaciones.

Por cada incremento del desarrollo de las validaciones se generaba una nueva versión del artefacto por medio de una tarea de Jenkins y se utilizaba la herramienta interna *Henry* para programar la ejecución de los tests, crear los distintos archivos con la información que se enviaba como parámetro a cada prueba y crear semáforos para monitorear las ejecuciones como se muestra en la Figura 6.2.

Henry también contaba con la funcionalidad de envíos de mails ante cambios de los estados de los semáforos. En la Figura 6.3 se muestra un mail enviado por la aplicación informando del fallo de un test que realizaba un booking exitoso en el ambiente de Producción. La aplicación detectó que se produjo el cambio del estado del semáforo de Off a Error y envió la notificación a las casillas de correo que tenía configuradas.

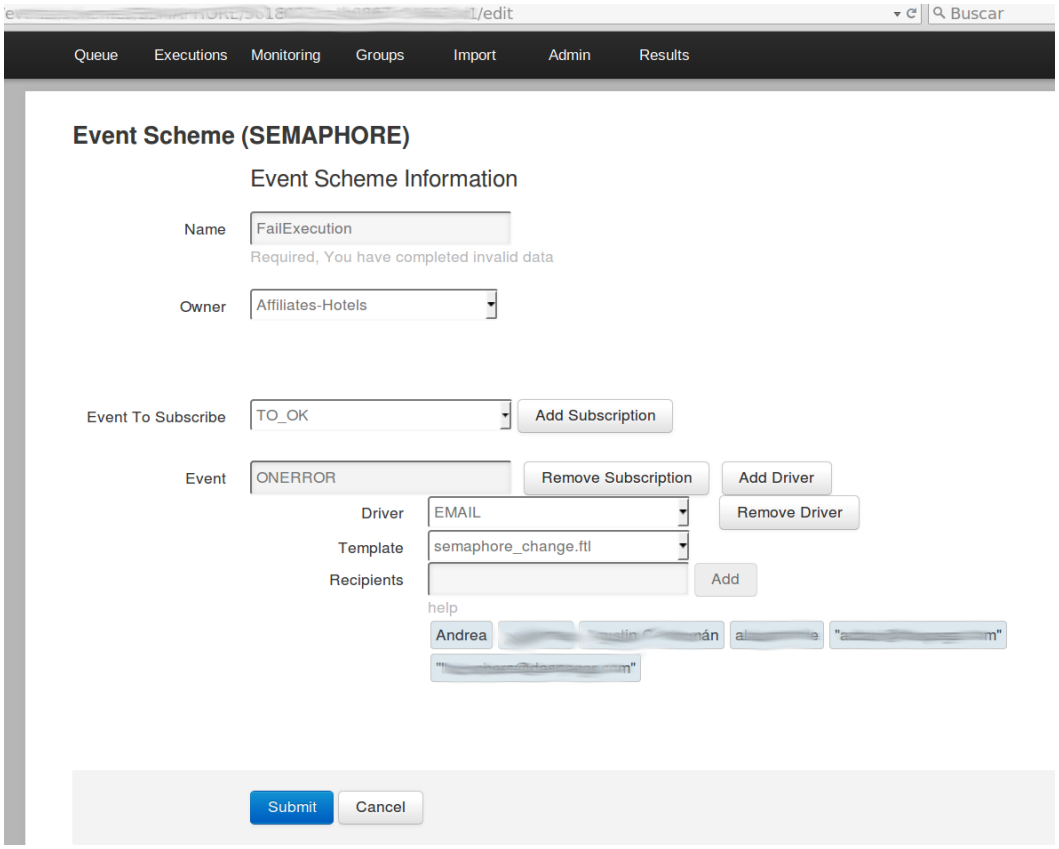


Figura 6.2 Configuración de semáforos en Henry

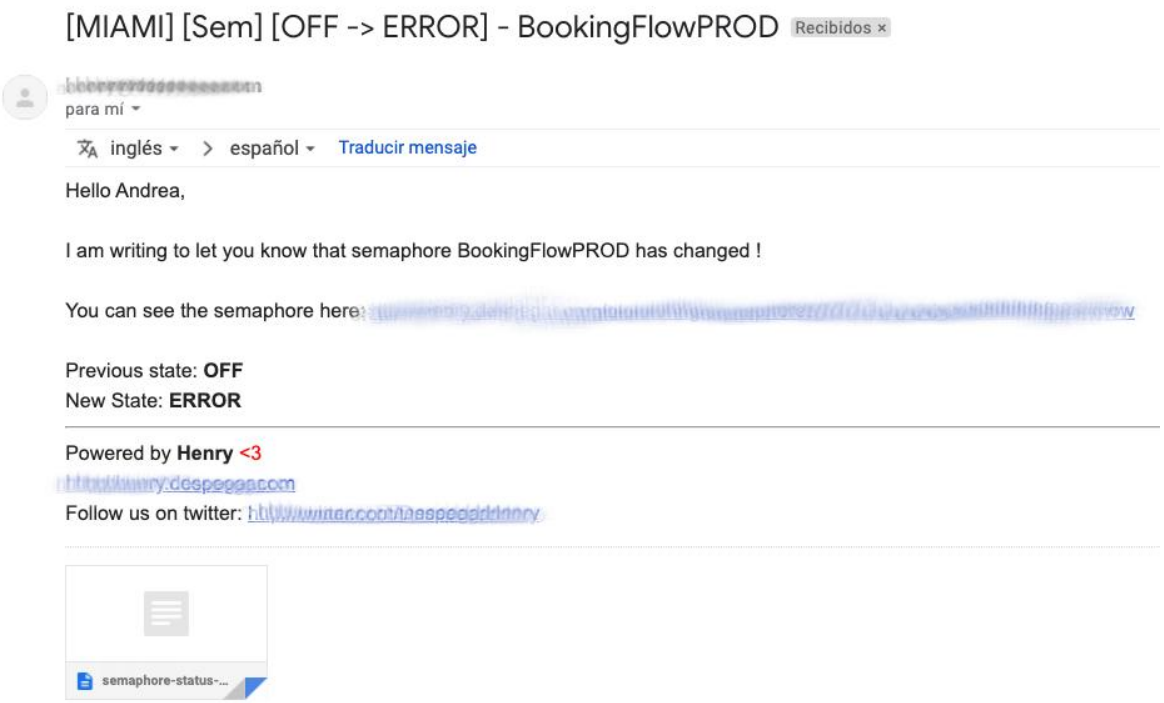


Figura 6.3 Un mail enviado por Henry alertando un cambio en el estado de un semáforo

Validaciones en configuraciones de las agencias

Otro de los desafíos principales era mostrar correctamente los precios a lo largo de todo el flujo de compra para cada una de las agencias, teniendo en cuenta la moneda con el símbolo que corresponda al país al cual pertenece y su descripción. Para automatizar la validación se creó un test sobre el servicio que exponía las configuraciones para cada uno de los países en los que operan las agencias.

La automatización de esta validación fue importante para corroborar que se continúe mostrando el símbolo, la moneda y la localización correspondiente durante todo el flujo de compra. De no contar con el tests, se hubiese tenido que acceder constantemente a cada uno de los dominios de los sitios en los que operan las agencias para validar la información, ya que una incorrecta notificación de las tarifas puede causar incertidumbre y confusión en el usuario.

Chequeo de disponibilidad

La aplicación Jano era la responsable de solicitar disponibilidad, tarifas e información de hoteles a otras aplicaciones internas de la empresa para luego customizar la información y enviársela a Vulcano, la aplicación frontend.

Un chequeo importante que se realizaba sobre Jano tenía que ver con comprobar que la información sobre la disponibilidad ofrecida se hubiera generado teniendo en cuenta la configuración de cada una de las agencias por las cuales se realizaban las solicitudes. Tal motivo llevó al equipo de desarrollo a generar el correspondiente test de unidad para automatizar la tarea de testeo y así evitar realizar la validación de manera manual.

En la Figura 6.4 se expone un test destinado a validar que exista disponibilidad al llegar a la página del detalle de un hotel. Si existe, entonces verifica que se haya generado un valor único que representa la búsqueda; que existan formas de pago y que el tipo de moneda sea el correspondiente a la configuración del país al que pertenece el hotel.

```
public class RoomsAvailabilityServiceTest {


  private RoomsAvailabilityService roomsAvailabilityService;

  private static DetailsTrait detailsTrait;
  private static CountryConfiguration countryConfig;
  private static AgencyData agencyData;
  private static final Locale LOCALE = new Locale("es", "AR");
  private static final String CURRENCY_CODE = "ARS";
  private static final String DISTRIBUTION = "2";
  private static final String HOTEL_ID = "346550";
 private static AgencyData agencyData;
  private static DetailsTrait detailsTrait;
  private static CountryConfiguration countryConfig;
  private String checkin;
  private String checkout;


  @Before

  public void setUp() throws Exception {

     agencyData = new AgencyData();
     agencyData.setCountry("AR");
     Calendar calendar = Calendar.getInstance();
           calendar.add(Calendar.MONTH, 3);
     SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd");
     this.checkin = format.format(calendar.getTime());
     calendar.add(Calendar.DATE, 4);
     this.checkout = format.format(calendar.getTime());
     countryConfig = new CountryConfiguration();
     Fare fare = new Fare();
     fare.setPriceToShow(PriceToShow.SUBTOTAL);
     fare.setShowOriginal(true);
     countryConfig.setFare(fare);
     detailsTrait = new DetailsTrait(HOTEL_ID, this.checkin, this.checkout, DISTRIBUTION);

      }


@Test
  public void getRoomsAvailabilityTest() throws JanoCommissionException, NotFoundException, ServiceException
{
     RoomsAvailability roomsAvailability = this.roomsAvailabilityService.getRoomsAvailability(detailsTrait, LOCALE,
agencyData, countryConfig);
     Assert.assertEquals(HOTEL_ID, roomsAvailability.getId());
     if (roomsAvailability.getRoomClusters().size() > 0) {
        Assert.assertNotNull(roomsAvailability.getAvailabilityToken());
        HotelRoomPack roomPack =
         roomsAvailability.getRoomClusters().iterator().next().getRoompacks().iterator().next();
           Assert.assertNotEquals(0, roomPack.getPaymentTypes().size());
           Assert.assertEquals(CURRENCY_CODE, roomPack.getPriceDetail().getCurrency());
           }
  }
```

Figura 6.4 Test que verifica disponibilidad y formas de pago en las habitaciones en la página de detalle de un hotel

De la experiencia de implementación, mantenimiento y monitoreo de los test automatizados para las aplicaciones encargadas del flujo de *Hoteles* de *Agencias*

Afiliadas es posible destacar que si bien las alertas de *Henrry* llegaban a todo el equipo, era responsabilidad del automatizador chequear las alertas y validar aquellas que informan de un error, ya que pueden resultar en casos de falsos positivos. En las automatizaciones E2E se presentaba con mucha frecuencia ese caso. Generalmente se recibían mails de alertas que, cuando se acudía a los registros de error, se comprobaba que habían fallado por timeout al intentar encontrar un elemento que aún no estaba disponible en el DOM del sitio.

Otro caso de fallos de tests que ocurrían con frecuencia estaban relacionados con la actualización de los elementos del sitio por nuevos desarrollos o cambios en los requerimientos. Entonces las automatizaciones tampoco encontraban los elementos en el DOM y había que actualizarlas constantemente, lo que requería de una inversión de tiempo extra. El aspecto positivo que se destacaba de este tipo de pruebas es que eran pruebas que testeaban la aplicación como un todo. Es decir verificaban el estado de la aplicación frontend, pero también el de las aplicaciones backend. Un ejemplo de ello se presentó un día en que la automation E2E falló porque detectó que se comenzó a visualizar un valor NULL en un lugar del sitio donde debería visualizarse un dato relevante del negocio.

6.5 Experiencia de tests automatizados en la aplicación Special Clients

El proyecto *Special Clients* era responsable de proveer sitios personalizados para empresas externas interesadas en brindar a sus clientes servicios turísticos con un descuento especial o por medio de un canje de puntos. Los sitios personalizados podían requerir autenticación para sus clientes o ser de acceso público y contar con una característica especial, como por ejemplo que sólo se pueda pagar con una tarjeta de crédito específica de un banco particular.

A partir de junio del año 2016 cada una de las personas que se desempeñaban como *automation tester* pasaron a ser desarrolladores estándar. Es decir que todos los integrantes del proyecto pasaron a tener la responsabilidad de crear, mantener y monitorear las pruebas automatizadas. A continuación se describen experiencias de automatizaciones de tests.

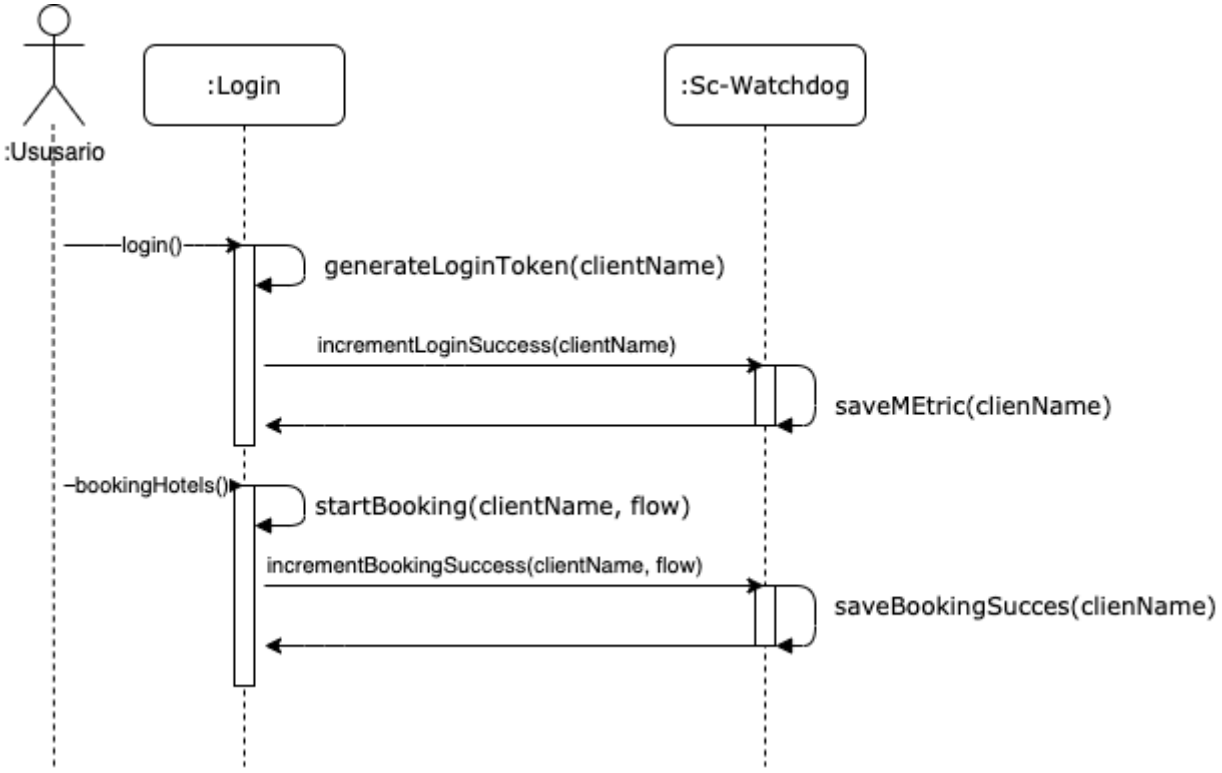


Figura 6.5 Diagrama UML entre las aplicaciones Special Clients y Sc Watchdog

Para comprobar el estado de la aplicación *Special Clients*, el equipo desarrolló *Sc-Watchdog*, también en Java, con la responsabilidad de recibir y cuantificar notificaciones de intentos de compras exitosos y fallidos. Asimismo también recibía notificaciones de login de usuarios. Cada vez que *Sc-Watchdog* recibía notificaciones vía API, realizaba un registro interno para poder determinar si debía enviar una alerta al equipo. En la Figura 6.5 se presenta un diagrama UML que muestra el flujo de comunicación exitoso entre las aplicaciones *Special Clients* y *Sc-Watchdog* ante las acciones de login y compra de un producto por parte de un usuario. Cada vez que un usuario de un banco o una empresa intenta ingresar al sitio web, *Special Clients* verifica los datos de autenticación y envía una notificación a *Sc-Watchdog* para que actualice el conteo. El mismo flujo de comunicación se recorría cuando el usuario intentaba efectuar la compra de un producto turístico, por ejemplo un Hotel. *Special Clients* recibía la notificación y enviaba un pedido a *Sc-Watchdog*, para que incremente sus variables.

Sc-Watchdog utilizaba la información de eventos recibidos para alertar al equipo de desarrollo cuando el registro de errores registrados superaba el umbral establecido, utilizando la aplicación de mensajería Telegram[30] como canal de comunicación. Un ejemplo de notificación de casos de error en el flujo de compra de paquetes se muestra en la Figura 6.6.

Otra de las responsabilidades que tenía *Sc-Watchdog* era elaborar reportes, que sólo debía enviar los días de semana y en horario laboral, de la cantidad de intentos de venta exitosos y fallidos que se efectuaban por cliente.

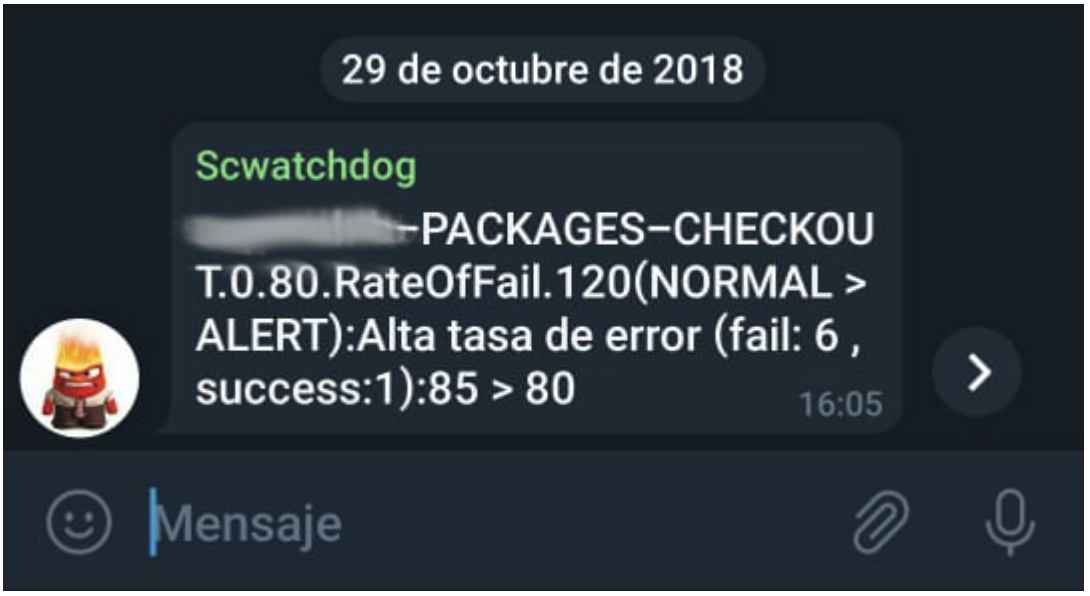


Figura 6.6 Reporte de alerta enviado por Sc-Watchdog al grupo de Telegram que tenía el equipo

Además de contar con tests unitarios desarrollados en JUnit[22], el equipo de *Special Clients* tenía una suite de tests E2E llama: *Sc-Test*, implementada utilizando Nightwatch.js[5], un framework escrito en el lenguaje de programación Node.Js que utiliza la API W3C WebDriver para controlar navegadores webs y ejecutar comandos y validaciones en los elementos del DOM.

Las ejecuciones recorrían todo el flujo de compra de cada uno de los productos turísticos que brindaba la plataforma de *Special Clients*: hoteles, vuelos, autos, paquetes, actividades y buses; hasta llegar a la página de Gracias, que es la última que visualiza el usuario cuando realiza una compra exitosa. Otras pruebas de regresiones estaban pensadas para completar los datos del checkout y forzar determinados mensajes de error.

Para mantener los tests en un flujo continuo de monitoreo sobre los ambientes Beta y Producción, el equipo de desarrollo generó un Job en Jenkins[24]. que disparaba la suite de tests para cada uno de los clientes.

La estructura del proyecto *Sc-Test* fue pasando por procesos de refactorización hasta lograr tener módulos de leyendas y configuraciones de uso común, que aplicaban a todos los tests de todos los clientes. y también específicos a cada uno de ellos.

Las automatizaciones fueron evolucionando hasta que se logró implementar los test utilizando el patrón de diseño para pruebas automatizadas: *Page Object*, el que aportó una separación entre la implementación de cada página y el comportamiento que se espera de cada test; la reutilización de código, legibilidad y mayor mantenibilidad ante los constantes cambios que se presentaban en el sitio. En la Figura 6.7 se presenta una representación del modelo implementado utilizando el patrón *Page Object*.

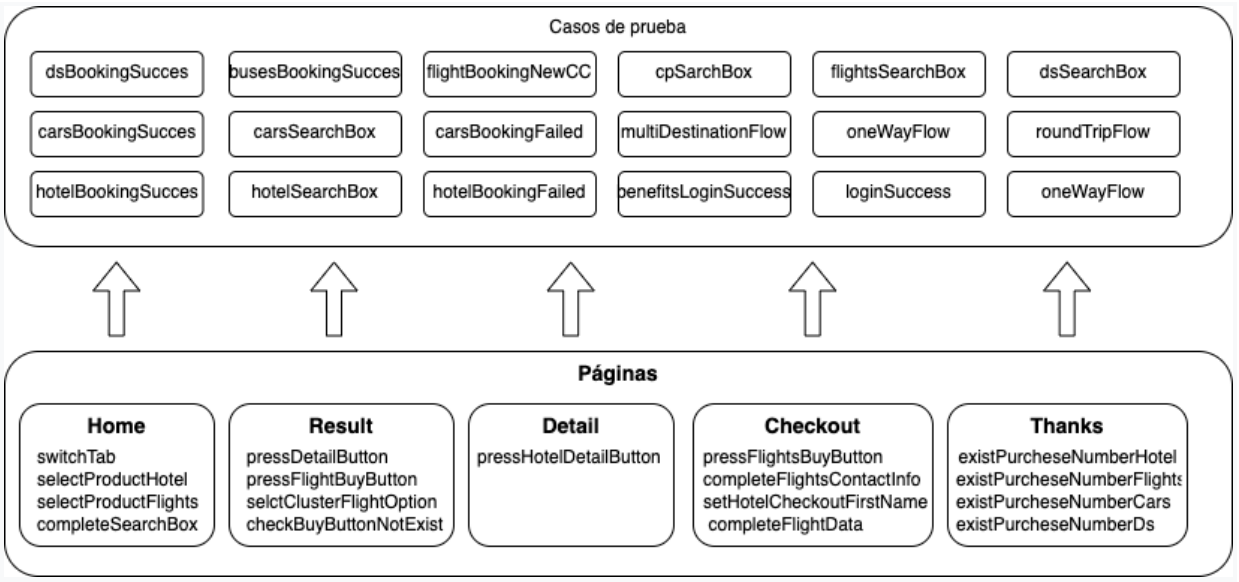


Figura 6.7 Modelado de test automatizados utilizando el patrón Page Object.

Otras de las funcionalidades que se implementó tenía que ver con la generación de un reporte en el que consten los datos del test y el resultado de la ejecución. Los reportes generados a partir de corridas de tests en los ambientes de Beta y Producción eran enviados a *BigBrother*, una aplicación interna que tenía la responsabilidad de recibir los resultados y enviar una alerta al equipo en los casos en que detectase que la cantidad de fallos recibidos era mayor a los configurados. En la

Figura 6.8 se puede observar una captura de pantalla que tomó el framework ante un fallo por no encontrar un elemento luego de superar el tiempo de timeout de tolerancia configurado.

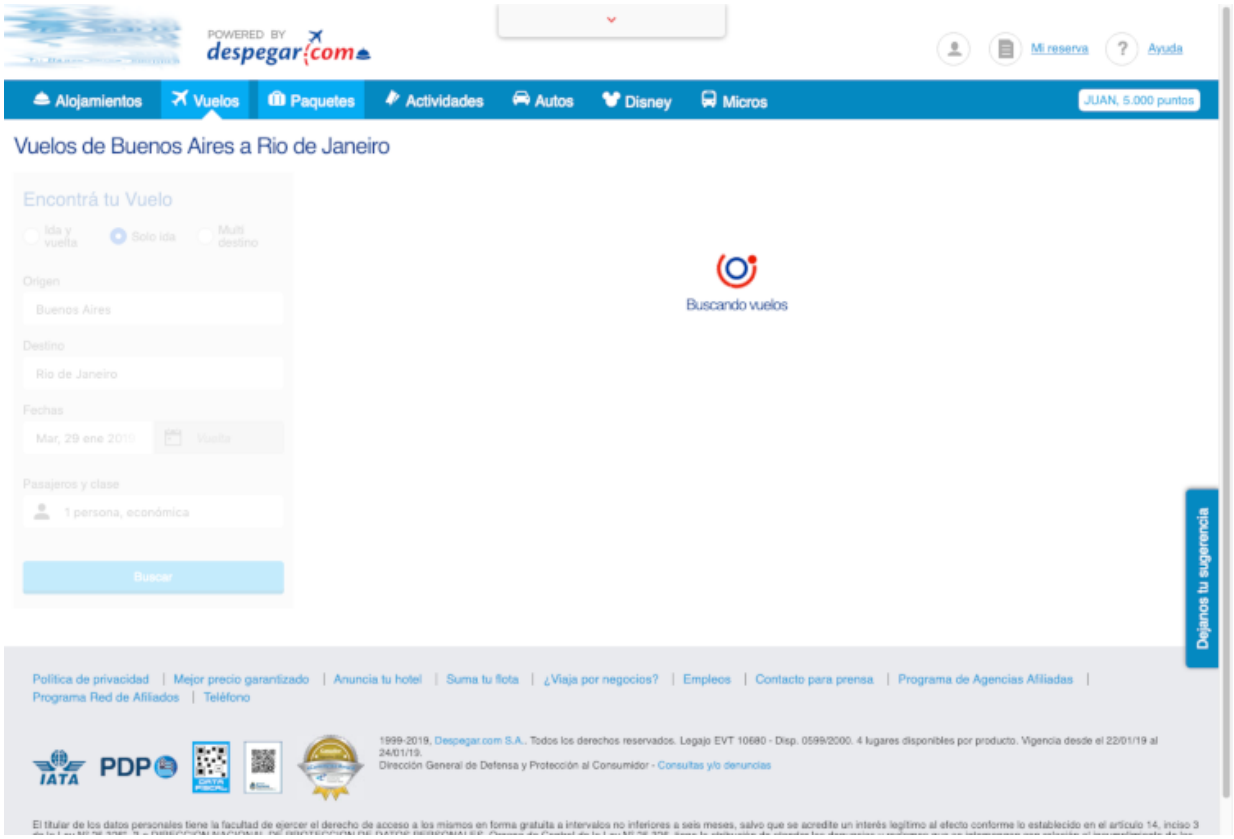


Figura 6.8 Captura de pantalla tomada por la automatización ante un fallo por timeout en el flujo de vuelos.

El rol en el proyecto *Special Clients* fue muy diferente al experimentado en Hoteles. Si bien estuvieron presentes las tareas de generar y mantener tests automatizados, como la integración de *Sc-Watchdog* con el cliente de Telegram[30] y tests para clientes que se fueron integrando al negocio, también existieron otras tareas orientadas al desarrollo en general que hicieron que las actividades que emprendiera un ex “tester automation” sean más heterogéneas y que le permitieran mayor crecimiento profesional.

La responsabilidad de que los tests automatizados se mantengan vigentes y que arrojen resultados sin falsos positivos era compartida con el resto del equipo. Éste último punto puede considerarse como un aspecto positivo que tuvo el cambio del rol de “automation” hacia todos los integrantes, pero también presentó dificultades al momento de monitorear esas automatizaciones cuando fallaban, ya que la

responsabilidad de esa tarea no tenía un nombre y apellido y cuando los tests fallan no todos los integrantes del equipo se presentan colaborativos y dejan de hacer sus actividades para chequear por qué fallaron los tests.

6.7 Experiencia de tests automatizados en la aplicación Sc-Services

La arquitectura monolítica en la que estaba construida *Special Clients* y la complejidad del negocio, impulsaron una reestructuración de arquitectura y responsabilidades que provocó una división de la aplicación y de los colaboradores en 4 equipos. Uno de ellos fue: *Sc-Services,* cuya responsabilidad es la de proveer una capa de abstracción de servicios para brindar:

- Cotizaciones de descuento/puntos.
- Efectivización de descuentos/puntos.
- Manejo de sesiones.

Sc-Services es una aplicación backend que cuenta con un conjunto de actividades que miden, evalúan e informan sobre la calidad de sus artefactos aplicando las técnicas de prueba de software funcionales y de mantenimiento que ejecutan tipos de tests unitarios, de componentes, pruebas de regresión y de re prueba.

El testing en el proceso de desarrollo de Sc-Services

Cada vez que un miembro del equipo toma una tarea la transiciona al estado: “En desarrollo”. Dicha tarea puede provenir de “Lista para desarrollar” o pudo haber estado “bloqueada”, y su situación ha cambiado. Cuando el desarrollador considera que ha culminado con los requerimientos y ha realizado las validaciones pertinentes en su ambiente local, utiliza la herramienta Git[32] para versionar el código mediante una nueva ramificación del proyecto (branch) y así subir esa nueva funcionalidad al repositorio del proyecto. Posteriormente pasa su tarea al estado de To Review, y para garantizar la calidad del código, solicita la validación de los cambios al resto del equipo (pull request) antes de unificar la rama con el nuevo desarrollo y el código

estable de la aplicación. En la Figura 6.9 se puede observar el diagrama de flujo con los posibles estados por los que se transiciona una tarea en el equipo de Sc-Services.

El último paso es crear las releases y deployar en los ambientes de prueba: Beta y Sandbox. Para automatizar el proceso de build, el equipo creó 3 jobs en Jenkins[24]. Cada uno de lo cuales persigue objetivos distintos:

- **sc-services-quality-check:** Se ejecuta automáticamente cada vez que se genera un pull request, y valida que el código cumpla con las reglas de calidad propuestas por SonarQube[34], la herramienta para el análisis estático de código.
- **sc-services-release:** Este job es ejecutado manualmente por los desarrolladores del equipo al momento de generar una release que luego se deploya en los ambientes de: Beta, Sandbox y Producción. Está programado para unificar el branch develop con el branch master (la rama que contiene el mismo código que producción), crear un tag en master con cambio de versión, y sacar una release. El job también está integrado con una aplicación interna de la empresa encargada de realizar diversos chequeos de seguridad y generar el build de la aplicación, construir el artefacto final y almacenarlo en un repositorio interno de código para luego ser localizado a la hora del deploy. Hay equipos de desarrollo en la empresa que actualmente prefieren optar por el esquema Git Flow[31].
- **sc-services-test:** Se trata de un job automatizado configurado para ejecutar los tests del proyecto cada 30 minutos sobre el branch de pre-producción llamado: *develop*.

Los jobs generados en Jenkins contribuyen al proceso de integración contínua de la aplicación Sc-services. La ejecución de cada uno de ellos otorga seguridad, respaldo y calidad en cada deploy.

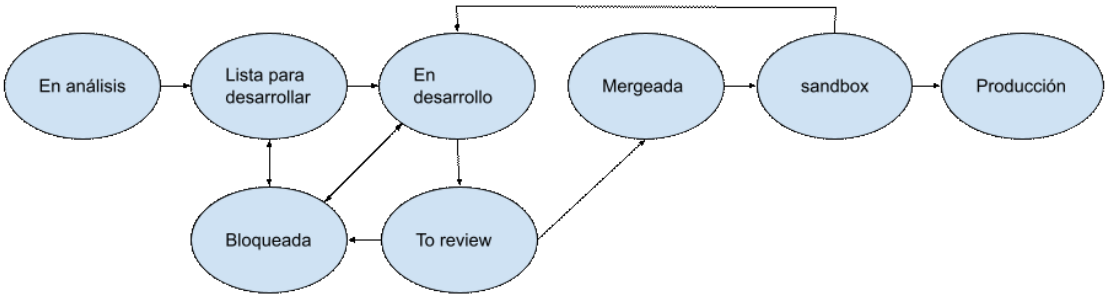


Figura 6.9 Flujo de desarrollo de software en Sc-Services

6.8 Casos de pruebas automatizadas sobre el sitio público de ventas

Las pruebas automatizadas reducen considerablemente el tiempo necesario para realizar validaciones en relación a lo que demanda la misma prueba a una persona. También aportan consistencia al proceso de verificación de las aplicaciones, ya que realizan la misma secuencia de instrucciones cada vez que se ejecutan. Con el fin de mostrar cómo un set de pruebas automatizadas puede asegurar calidad en el proceso de testing de una aplicación, se creó un conjunto de pruebas utilizando el framework Protractor[41] sobre el sitio público de venta de productos turísticos www.despegar.com y se mostrará su ejecución durante la exposición de la presente tesina.

Una de las automatizaciones desarrolladas se creó con el objetivo de verificar: el título del sitio web, la localización, y la visualización de los links de cada uno de los productos que se deben mostrar dependiendo el país, dado que cada uno de los países en que opera *Despegar* tiene su propio acuerdo legal.

Una segunda automatización E2E fue realizada sobre el flujo de compra de Hoteles y tiene como objetivo validar el precio por noche y el nombre de los alojamientos durante todo el flujo de compra. El flujo comienza en la página de “Resultados” a la que se llega por medio del armado de la URL. Se eligió esta metodología para evitar los falsos positivos que se generan en los tests al fallar por errores de timeout en los campos de autocompletado y en el selector de fechas que forman parte de la página principal.

```
Hotel Name: Broadway Hotel & Suites
Hotel Price: $ 5.120

  workspace-project App
    ✓ hotels details validation

Expected text: //www.despegar.com.ar/vuelos/ - Actual text: //www.despegar.com.ar/vuelos/
Expected text: Vuelos - Actual text: Vuelos
Expected text: //www.despegar.com.ar/paquetes/ - Actual text: //www.despegar.com.ar/paquetes/
Expected text: Paquetes - Actual text: Paquetes
Expected text: //www.despegar.com.ar/ofertas-de-viajes - Actual text: //www.despegar.com.ar/ofertas-de-viajes
Expected text: Ofertas - Actual text: Ofertas
Expected text: //www.despegar.com.ar/escapadas/ - Actual text: //www.despegar.com.ar/escapadas/
Expected text: Escapadas - Actual text: Escapadas
Expected text: //www.despegar.com.ar/actividades/ - Actual text: //www.despegar.com.ar/actividades/
Expected text: Actividades - Actual text: Actividades
Expected text: //www.despegar.com.ar/autos/ - Actual text: //www.despegar.com.ar/autos/
Expected text: Autos - Actual text: Autos
Expected text: //www.despegar.com.ar/actividades/d-DY_ORL/entradas+para+walt+disney+world+resort-en-orlando?from=nav - Ac
Expected text: Disney - Actual text: Disney
Expected text: //www.despegar.com.ar/asistencia-al-viajero/ - Actual text: //www.despegar.com.ar/asistencia-al-viajero/
Expected text: Seguros - Actual text: Seguros
Expected text: //www.despegar.com.ar/traslados/ - Actual text: //www.despegar.com.ar/traslados/
Expected text: Traslados - Actual text: Traslados
  workspace-project App
    ✓ Test AR site
Expected text: //www.decolar.com/passagens-aereas/ - Actual text: //www.decolar.com/passagens-aereas/
Expected text: Passagens - Actual text: Passagens
Expected text: //www.decolar.com/pacotes/ - Actual text: //www.decolar.com/pacotes/
Expected text: Pacotes - Actual text: Pacotes
Expected text: //www.decolar.com/ofertas-de-viagens - Actual text: //www.decolar.com/ofertas-de-viagens
Expected text: Ofertas - Actual text: Ofertas
Expected text: //www.decolar.com/viagens-curtas/ - Actual text: //www.decolar.com/viagens-curtas/
Expected text: Escapadas - Actual text: Escapadas
Expected text: //www.decolar.com/atracoes-turisticas/ - Actual text: //www.decolar.com/atracoes-turisticas/
Expected text: Ingressos - Actual text: Ingressos
Expected text: //www.decolar.com/carros/ - Actual text: //www.decolar.com/carros/
Expected text: Carros - Actual text: Carros
Expected text: //www.decolar.com/atracoes-turisticas/d-DY_ORL/ingressos+para+walt+disney+world+resort-orlando?from=nav -
Expected text: Disney - Actual text: Disney
Expected text: //www.decolar.com/seguro-viagem/ - Actual text: //www.decolar.com/seguro-viagem/
Expected text: Seguros - Actual text: Seguros
Expected text: //www.decolar.com/transfer/ - Actual text: //www.decolar.com/transfer/
Expected text: Transfers - Actual text: Transfers
    ✓ Test BR site
Expected text: //www.despegar.com.do/actividades/ - Actual text: //www.despegar.com.do/actividades/
Expected text: Actividades - Actual text: Actividades
Expected text: //www.despegar.com.do/autos/ - Actual text: //www.despegar.com.do/autos/
Expected text: Autos - Actual text: Autos
Expected text: //www.despegar.com.do/actividades/d-DY_ORL/entradas+para+walt+disney+world+resort-en-orlando?from=nav - Ac
Expected text: Disney - Actual text: Disney
Expected text: //www.despegar.com.do/seguro-de-viaje/ - Actual text: //www.despegar.com.do/seguro-de-viaje/
Expected text: Seguros - Actual text: Seguros
Expected text: //www.despegar.com.do/traslados/ - Actual text: //www.despegar.com.do/traslados/
Expected text: Traslados - Actual text: Traslados
    ✓ Test DO site
Expected text: //www.us.despegar.com/flights/ - Actual text: //www.us.despegar.com/flights/
Expected text: Flights - Actual text: Flights
Expected text: //www.us.despegar.com/packages/ - Actual text: //www.us.despegar.com/packages/
Expected text: Packages - Actual text: Packages
Expected text: //www.us.despegar.com/activities/ - Actual text: //www.us.despegar.com/activities/
Expected text: Tickets - Actual text: Tickets
Expected text: //www.us.despegar.com/cars/ - Actual text: //www.us.despegar.com/cars/
Expected text: Cars - Actual text: Cars
    ✓ Test US site

Executed 5 of 5 specs SUCCESS in 37 secs.
[00:42:39] I/launcher - 0 instance(s) of WebDriver still running
[00:42:39] I/launcher - chrome #01 passed
```

Figura 6.10 Información de salida de las automatizaciones desarrolladas utilizando Protractor

Una vez en la página de resultados, se guarda el valor del precio final de la noche de hotel del primero de la lista y continúa hasta la página de detalle en una nueva pestaña donde selecciona otra habitación distinta y compara los precios para validar que hayan cambiado, ya que el precio de cada habitación depende de las comodidades que ofrezca, del tipo de desayuno y de la forma de pago. La

automatización vuelve a elegir la primera habitación y presiona el botón para continuar el flujo de compra.

Antes de llegar a la página de Checkout donde se cargan los datos de los huéspedes y la forma de pago, el sitio muestra una página intermedia en la que se ofrecen otros productos para agregar al carrito.

Ya en el Checkout, la automatización verifica que el nombre del hotel y el precio total a pagar sean los mismos que se mostraron durante todo el flujo de compra.

En la Figura 6.10 se muestra el resultado de la ejecución de la prueba automatizada en la que es posible observar que tardó sólo 37 segundos en realizar todas las verificaciones descritas en los párrafos anteriores.

6.9 Resumen

La empresa *Despegar.com* genera su propio software desde el año 2007 y desde ese entonces ha implementado diversos mecanismos para alcanzar la calidad en todos sus componentes{13]. El cambio hacia una metodología de testing ágil con tests automatizados como herramienta principal del testeo de las aplicaciones logró el aseguramiento de la calidad del proceso de desarrollo de software.

Ante el cambio en la metodología de generación de software, la gerencia *Canales* decidió capacitar en nuevas herramientas y estrategias para automatizaciones a cada uno de los integrantes de los equipos de QA para que pasaran de realizar tareas manuales a desempeñar el rol de *automation tester* en cada uno de los proyectos de desarrollo de software que tenía la gerencia.

Con el fin de facilitar la tarea de automatización a los miembros del equipo de QA que tenían poca o nula experiencia en tareas de programación, el equipo de *IT University* creó un arquetipo de automatización en Maven[25] como un marco de referencia estándar.

El *automation tester* asignado al equipo de Hoteles de Agencias Afiliadas se basó en el arquetipo para automatizar los casos de prueba más relevantes para el proyecto. Para la aplicación frontend programó flujos de compra E2E utilizando Selenium[40]; y para las aplicaciones backend desarrolló pruebas funcionales, de regresión y de re-prueba. Era total responsabilidad del *automation tester* chequear las

alertas que se configuraban sobre las pruebas, y validar aquellas que informaban casos de error.

El punto beneficioso que brindaron las técnicas de pruebas para la GUI fue el hecho de poder testear a la aplicación como un todo. Un aspecto negativo que se encontró en las automatizaciones E2E tuvo que ver con la inversión de tiempo extra para actualizarlas cada vez que se realizan cambios en el sitio web.

Las pruebas unitarias de las aplicaciones backend del equipo de *Hoteles*, fueron de suma importancia porque aportaron consistencia y seguridad al equipo a la hora de generar el build, al alertar sobre posibles errores antes de que el nuevo código llegase al ambiente productivo.

Un año y medio más tarde la gerencia *Canales* eliminó el rol específico: *automation tester* y todo el grupo de trabajo pasó a tener la responsabilidad de generar, mantener y monitorear los tests automatizados. El colaborador que desarrollaba tal rol pasó a realizar tareas orientadas al desarrollo en general con actividades heterogéneas que le permitieron un mayor crecimiento profesional. Éste punto puede considerarse como un aspecto positivo que tuvo el cambio de rol, pero también presentó dificultades al momento de monitorear las automatizaciones, puesto que la responsabilidad de esa tarea no tenía un nombre y apellido; y cuando los tests fallaban no todos los integrantes del grupo se mostraban colaborativos y dejaban de hacer sus actividades para chequear el por qué de las fallas en los tests.

El proyecto *Special Clients*, estaba dedicado al desarrollo de sitios personalizados para bancos y empresas interesados en ofrecer productos turísticos a sus usuarios con descuentos o por canje de puntos. En el presente capítulo se describió la aplicación Sc-Watchdog creada por el equipo con la función de recibir notificaciones de eventos, almacenarlos, detectar umbrales de fallos y enviar alertas al grupo de Telegram del proyecto. La calidad de la aplicación Special Client también se aseguraba mediante pruebas unitarias, además de implementaciones de suites de tests E2E desarrolladas en el framework Nightwatch.js[5] que se ejecutaban constantemente por medio de un job en Jenkins.

A mediados de enero de 2019 se emprendió una reingeniería en *Special Clients* que lo dividió en 4 aplicaciones, uno de ellos se denominó *Sc-Services*. Las actividades de aseguramiento de calidad en Sc-Services estuvieron orientadas a la

aplicación de técnicas de pruebas funcionales y a la generación de tres jobs en Jenkins para asegurar la calidad en el proceso de deploy de la aplicación.

Capítulo 7 Conclusiones y trabajos futuros

7.1 Conclusiones

Las experiencias de implementación de pruebas automatizadas en aplicaciones web que se describieron a lo largo del capítulo 6 contribuyeron a mejorar la calidad de los sistemas de información de la empresa Despegar.com. Sus sitios brindan servicios de calidad porque están siendo monitoreadas y testeadas por pruebas automatizadas que validan que el funcionamiento sea el esperado.

Antes de las automatizaciones, era el equipo de QA quien tomaba una historia y validaba su calidad manualmente, pero no era un recurso dedicado. Entonces la gerencia decidió disolverlo y, previa capacitación, asignar a cada uno de sus integrantes en cada uno de los equipos de desarrollo. QA no sólo tomaba historias para validar su calidad, sino que también recibía, investigaba y daba respuestas a consultas por posibles bugs. También acordaba con el equipo de *Producto* los parámetros de aceptación para cada historia. Cuando se eliminó el área de QA parte de esas responsabilidades pasaron al equipo de desarrollo.

Los testeos automatizados otorgaron mayor dinamismo al proceso de subida a producción de las aplicaciones de cada equipo de la gerencia *Canales. Silvina Maneglia*, quien es Líder técnico del equipo de *Backoffice* de *Agencias Afiliadas* (BO) desde 2015, considera que disolver el equipo de QA fue beneficioso y recuerda anécdotas de cuando iba personalmente a hablar con QA para solicitarles que prioricen las historias su proyecto porque necesitaban deployar en producción, destacando que se formaba un “cuello de botella” en el área de QA ante tantos requerimientos. Actualmente es el mismo equipo quien maneja sus tiempos de deploy a producción apoyado en las automatizaciones de los tests que ha implementado.

Marcos Godoy, Líder Técnico de *Special Clients* desde el año 2013, apoya la teoría de la responsable del *BO* sobre la desintegración de su equipo de QA dedicado, pero su argumento es otro: los testers tenían un intervalo de tiempo ocioso entre un deploy y el siguiente. Ese margen también pudo haber estado relacionado con el hecho de que *Special Clients* era una aplicación monolítica extremadamente

compleja con subidas a producción cada 15 días.

A mediados de 2016 la gerencia *Canales* eliminó el rol específico del automatizador dedicado y sus responsabilidades pasaron a todo el equipo. El ex-tester comenzó a realizar actividades heterogéneas que le permitieron un mayor crecimiento profesional y el chequeo de las automatizaciones quedó sin nombre y apellido. *Godoy* considera que, en base a las combinaciones de estrategias de testing que experimentó en *Despegar*, la que mejor resultado aporta es crear el rol del "*encargado de las automatizaciones*" en el equipo y que sea el rol con sus responsabilidades el que vaya rotando entre los integrantes. Entonces de esta manera no hay miembros del equipo desmotivados o poco interesados en su labor porque eventualmente todos realizan tareas diferentes y la responsabilidad de crear, monitorear y controlar las pruebas automatizadas es grupal.

Tomás Arce, quien era Líder Técnico del equipo de *Paquetes* de *Agencias Afiliadas* cuando se hizo el cambio de metodología y actual responsable del equipo de *Fulfillment As a Service*, señala que, cuando él se sentó a analizar las estrategias de automatización que se iban a implementar con el *automation teste*r de su equipo definieron que cada test sería lo más reducido y específico posible, priorizaron tener "poco, pero bueno". *Arce* considera que tal estrategia les ahorró el inconvenientes de recibir múltiples mails de las aplicaciones de monitoreo informando por tests que no fallaron ante errores reales.

Tomando como base los testimonios de los líderes técnicos y del responsable del equipo de IT University, Lucio Loiacono (ver Anexo I) encargado del proceso de capacitación, se puede concluir que la implementación de pruebas automatizadas es beneficiosa, contando de igual manera con aspectos positivos y conflictivos dependiendo del proyecto y del equipo en que se apliquen.

Conclusiones en común:

- El equipo de QA validaba el estado de las aplicaciones durante el horario laboral, mientras que los tests automatizados pueden ser configurados a través de una herramienta que los mantiene en ejecución y monitoreo 24/7.
- Si las pruebas de tests E2E forman parte de una cola de ejecución integrada con envío de alertas, los responsables pueden llegar a recibir

notificaciones ante casos “falsos positivos”. Los tests de estas características son poco recomendados para validar flujos largos de navegación; tampoco son aconsejables para páginas en las que su tiempo de carga depende de servicios que administran otros equipos o contienen formularios que están compuestos por campos con autocompletado a partir del ingreso de determinada cantidad de caracteres.

- Las pruebas unitarias dentro del flujo de integración continua, se consideran de vital importancia, puesto que aportan consistencia, seguridad y alertan al equipo sobre posibles errores antes de que el nuevo código llegue al ambiente productivo.
- Las pruebas automatizadas respaldan al equipo de desarrollo ante procesos de refactorización. Para mantener consistentes sus validaciones con respecto a la aplicación, se requiere de inversión de tiempo y recursos.
- Existen casos específicos en los que es costoso, o no es posible, generar el test automatizado para validar la funcionalidad desarrollada. Es importante que el equipo reconozca estas situaciones para que no invierta tiempo en vano, y realice tales chequeos de forma manual.
- La posibilidad de parametrizar las suites de tests con datos específicos por ambientes de desarrollo brinda dinamismo y agilidad al procesos de testeo.
- En empresas como *Despegar* en las que sus desarrolladores cambian de equipos constantemente, cada vez que se invierte tiempo en generar un test, también se está dejando una documentación sobre el funcionamiento de la aplicación. Por lo tanto, cuando otros desarrolladores se hacen cargo del proyecto, las pruebas automatizadas también aportan información sobre las funcionalidades que debe cumplir la aplicación.

7.2 Trabajos futuros

El proyecto de *Extranet Hoteles* es un conjunto de aplicaciones que permiten la administración de los alojamientos que la empresa *Despegar.com* ofrece a sus clientes.

Como trabajo futuro de la presente tesina se propone realizar un conjunto de migraciones y refactorizaciones a fin de reducir la complejidad general de la plataforma hotelera, lo que representa una buena oportunidad para aplicar todo el conocimiento adquirido, poner en práctica las técnicas asociadas a la automatización de testeos y desarrollar nuevas estrategias para asegurar la calidad de las aplicaciones que la conforman.

En particular esta iniciativa de migración también presenta un desafío importante, ya que sería necesario iniciar líneas de investigación para atacar la problemática de automatización sobre aplicaciones con bases de datos, debiendo asegurar características como consistencia de los datos entre ejecuciones, entre otras.

Referencias bibliográficas

Las siguientes referencias se han tomado como base para la elaboración de la presente tesina.

[1] Crispin Lisa, Gregory Janet. Agile testing : a practical guide for testers and agile teams. - 1st ed. Pearson Education, Inc. Boston, 2009.

[2] ISO/IEC 2382 INFORMATION TECHNOLOGY - VOCABULARY, International Organization for Standardization, 2015. Dirección URL: <https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:ed-1:v1:en> [Consulta: 20 de febrero 2021].

[3] ISO 9000 Sistemas de gestión de la calidad — Fundamentos y vocabulario, 2015. Dirección URL <https://www.iso.org/obp/ui/#iso:std:iso:9000:ed-4:v1:es:term:3.6.2> [Consulta 21 de febrero de 2021]

[4] Project Management Institute. Software Extension to the PMBOK® Guide Fifth Edition. Foundational Standards, 2013.

[5] Nightwatch.js - Node.js powered End-to-End testing framework. Dirección URL: <https://nightwatchjs.org/>. [Consulta: 23 de junio 2021].

[6] Jira | Software de seguimiento de proyectos e incidencias. Dirección URL: <https://www.atlassian.com/es/software/jira>. [Consulta: 25 de junio 2021].

[7] IEEE Standard for Software Quality Assurance Process. The Institute of Electrical and Electronics Engineers, Inc, 2014.

[8] The Selenium Browser Automation Project :: Documentation for Selenium. Dirección URL: <https://www.selenium.dev/documentation/en/> [Consulta 25 de junio de 2021].

[9] Bourque Pierre, Fairley Richard E. Guide to the Software Engineering Body of Knowledge, Version 3.0, IEEE Computer Society, 2014.

[10] Pressman Roger S. Ingeniería del software. Un enfoque práctico. - 7ma edición. A Subsidiary of The McGraw-Hill Companies, Inc. 2010.

[11] Duvall Paul, Matyas Steve, Glover Andrew. Continuous Integration: Improving Software Quality and Reducing Risk -1st ed. Addison-Wesley Professional, Massachusetts, 2007.

[12], Cunningham Ward. The WyCash Portfolio Management System. OOPSLA' 92 Experience Report, 1992.

[13] Stylianou, A.C., Kumar, R.L. An integrative framework for IS quality management. Communications of ACM, 43(9), 99-104, 2000.

[14] Succeeding with Agile: Software Development Using Scrum - 1st Edición Pearson Education, Inc. 2010

[15] Protractor end-to-end testing for AngularJs. Dirección URL: < https://www.protractortest.org/#/api>. [Consulta 10 de junio de 2021]

[16] TestNG Documentation. Dirección URL: < https://testng.org/doc/documentation-main.html>. [Consulta: 25 de abril 2021].

[17] Karma - Spectacular Test Runner for JavaScript. Dirección URL: <https://karma-runner.github.io/latest/index.html>. [Consulta 5 de junio de 2021].

[18] International Software Testing Qualifications Board ISTQB, [en línea]. Certifying Software Testers Worldwide - ISTQB® International Software Testing Qualifications Board. Dirección URL: <https://www.istqb.org/>. [Consulta: 3 de enero de 2021].

[19] The New New Product Development Game (1986). Dirección URL: <https://hbr.org/1986/01/the-new-new-product-development-game>. [Consulta: 25 de abril de 2021].

[20] Manifiesto ágil, [en línea]. Manifiesto por el desarrollo Ágil de Software Ágil. Dirección URL: <http://agilemanifesto.org/iso/es/manifesto.html>. [Consulta: 21 de abril de 2021]

[21] CasperJS a navigation scripting & testing for PhantomJS and SlimerJS. Dirección URL: <www.casperjs.org>. [Consulta: 4 de mayo de 2021].

[22] JUnit 5 User Guide. Dirección URL: <https://junit.org/junit5/docs/current/user-guide>. [Consulta: 4 de abril de 2021].

[23] Mokito. Dirección URL: <https://site.mockito.org/>. [Consulta: 5 de abril de 2021].

[24] Jenkins User Documentation. Dirección URL: <https://www.jenkins.io/doc>. [Consulta: 25 de abril de 2021].

[25] Maven Documentation. Dirección URL: <https://maven.apache.org>. [Consulta: 25 de abril de 2021].

[26] Spring Documentation. Dirección URL: <https://docs.spring.io/spring-framework/docs/current/reference/html/testing.html>. [Consulta: 7 de junio de 2021].

[27] Postman. Dirección URL: <https://www.postman.com>. [Consulta: 25 de abril de 2021].

[28] Apache FreeMarker Manual. Dirección URL: <https://freemarker.apache.org/docs/index.html>. [Consulta: 2 de mayo de 2021].

[29] Angular Documentation. Dirección URL: <https://angular.io/docs>. [Consulta: 21 de junio de 2021].

[30] Telegram APIs. Dirección URL: <https://core.telegram.org/>. [Consulta: 10 de junio de 2021].

[31] Jasmine Documentation. Dirección URL: <https://jasmine.github.io/>. [Consulta 3 de mayo de 2021].

[32] Git Documentation. Dirección URL: <https://git-scm.com/doc>. [Consulta: 2 de mayo de 2021].

[33] New Relic Documentation. Dirección URL: <https://docs.newrelic.com/>. [Consulta: 25 de abril de 2021].

[34] SonarQube Documentation. Dirección URL: <https://docs.sonarqube.org/latest/>. [Consulta: 18 de junio de 2021].

[35] Docker Documentation. Dirección URL: <https://docs.docker.com/>. [Consulta: 1 de junio de 2021].

[36] Apache JMeter. Dirección URL: <hhttps://jmeter.apache.org>. [Consulta: 13 de mayo de 2021].

Anexo I Entrevista al Proyect Manager Lucio Loiacono

Entrevista al Ingeniero en Sistemas, Lucio Loiacono, Project Manager del equipo de University IT, quien tuvo a su cargo la responsabilidad de capacitar al equipo de QA sobre tests automatizados en la empresa Despegar.com.

¿Cómo era el día a día antes de incorporar la Integración Contínua?

La gente dependía de pasar por un ambiente de preproducción donde hacíamos un testeo de compatibilidad. Como era el único ambiente estable que teníamos, los desarrolladores lo usaban para testear y en vez de estar 4 horas testeando la compatibilidad con las aplicaciones testeaban también funcionalidad. El proceso hacia que nosotros vayamos a las 4 de la mañana a subir los paquetes de manera cuasi manual. Había muy pocas cosas automatizadas.

La idea era a las 7 de la mañana tener todo deployado, porque después de las 8, 9 de la mañana la gente empezaba a usar el sitio. El sitio no podía estar bajo. La gente inclusive no cumplía ese horario. Lo que pasaba era que había errores en producción, más allá de que había usado preproducción para testear. A veces los errores llegaban a producción y subíamos el paquete hasta 5 veces en el mismo día.

La gente no se hacía cargo de lo suyo. Los desarrolladores pensaban que era: desarrollar y que QA lo testee. Y QA no llegaba a testear todo, porque QA detectaba algo y el equipo volvía a deployar todo. Entonces no era testear esa partecita, era testear todo de vuelta.

¿La empresa expone un lineamiento a seguir sobre el proceso de testing?

La bajada de línea de la empresa no es hagan o no hagan testing, o hagan o no hagan automatizaciones. La bajada es: “che cuando subas no rompas nada”. O sea a mi no me puede venir a decir Damian Scokin que no estamos vendiendo nada porque le avisó un comercial, no. Me tengo que dar cuenta yo de que la curva de ventas bajó y conocer el por qué.

Desde la Dirección de IT tampoco te dicen cómo hay que desarrollar. Un equipo también decide qué motor de base de datos va a usar. Lo que pasa que tenés que elegir bien qué usar dependiendo de lo que tengas que hacer. Con el testing pasa lo mismo, tenés la libertad en la medida en que vos no rompas todo. La bajada de línea existe, no de manera directa, pero si existe. Si vos testeas también podés llegar a romper producción, pero tenés más conocimiento de lo que está pasando. Hay que hacer un análisis post mortem. Las pruebas existen siempre de una manera u otra.

Las pruebas también están en el proceso del paso de los Jiras[6], pero hay equipos que no lo tienen tan obligatorio. Existe la columna de *Testing in progress*, pero el testing puede darse en cualquier momento del desarrollo. Lo importante es que el testing se haga no necesariamente cuando se completa el campo de la evidencia de prueba.

¿Cuáles fueron los motivos por los cuales se decidió incorporar el Testing ágil?

Por el año 2009 se había empezado a automatizar en Ruby, pero eso nunca llegó demasiado bien a los equipos. Era muy complicada esa gestión. Existía un área de QA que hacía una parte de testing manual y una parte automatizada. Se juntaba muchas cosas: errores en producción porque no se testeaba bien y los desarrolladores no se hacían cargo de su testing. Se pensaba: “¿para qué lo voy a probar yo si lo prueba el QA?”

Lo que se quiso hacer es que los equipos de QA estén dentro de los equipos de desarrollo, donde un rol lo asume un desarrollador o lo asume otra persona que sólo hace testing. Se pensó en delegar la responsabilidad a cada equipo de cada gerencia de que se hagan cargo de dos cosas: de todo el desarrollo y de la subida a producción del código. Entonces como el equipo no se ponía a probar manual, la idea era que automaticen en un lenguaje que ellos conozcan. Lo que trajo aparejado que cada equipo haga su propio testing.

Entonces ¿qué se hace con esto? Mejorar las herramientas primero, porque con los años fueron apareciendo mejores herramientas y el equipo de automation de QA no se renovaba; y hacer que los equipos de desarrollo estén más cerca de poder mantener su testing: usando Jenkins[24] o automatizando en Java, buscando otras

maneras de automatizar, usando mock (Mokito[23]), TestNg[16] para hacer testing mediante pruebas unitarias, usando Sonar[34]. etc. un montón de herramientas que tienen que ver con eso: con que el mismo desarrollador se auto-testee lo que hace. Y el otro punto es que si una aplicación rompe producción, la responsabilidad es de ese equipo que subió a producción sin probar. Cada equipo tiene que subir a producción las cosas de manera sana, que no rompan todo y que las cosas sigan funcionando. La idea es que sean dueños de lo que diseñan como plataforma de prueba.

¿Cómo se llevó a cabo ese cambio de paradigma?

Como todo en Despegar: “esto no funciona, probemos otra cosa”. El tema de lo ágil. Lo ágil hace que pruebes cosas que a veces es probarlas para ver si funcionan. No necesitas tener algo previo que lo valide o lo verifique. El tema es acomodar a que las cosas funcionen a una cosa intuitiva de sentido común y hacer que las cosas funcionen sin atarnos a las cosas teóricas de cómo debería ser.

El tema de QA automation como área separada no funcionaba, así como se crearon verticales por producto, Ej: vertical Gerencia Hotel, Vertical Gerencia Vuelos, Canales. Cada gerencia era responsable de su subida a producción, cada gerencia era responsable de su testing. Si las cosas funcionan o no, las tenían que probar cada equipo.

Vengo de un palo de que un proyecto tardaba 2 años en salir a producción, no en Despegar, sino en otras empresas. Lo que veo desde el 2011 a la fecha es que lo ágil le ganó a lo estático. Lo ágil consiste en hacer cosas más pequeñas y en cada paso que das lo vas mejorando y vas viendo cómo funciona. Despegar siempre tuvo esa mentalidad. Todas las decisiones que tuvieron que ver con cambios a nivel organizativo o a nivel de cómo vamos a laburar, también es ágil.

La idea de que cada equipo automatice o testee sus cosas o use sus propias herramientas o decida cuáles va a usar para probar eso, era una decisión del equipo; les daba libertad y funcionó. Con los años nunca volvimos para atrás en eso. Creo que los equipos evolucionaron mucho en eso de: “soy dueño de, me tengo que hacer cargo de”

¿Cómo fue el proceso de la capacitación del personal de QA para que realice las automatizaciones?

El tema de la automatización era que se automatizaba en Ruby, pero no está mal eso, sino que después era muy costoso mantenerlo.

Surgió la idea de por qué no enseñarles a los equipos a automatizar en lo posible el mismo código en el que desarrollan. Entonces así se empezaron a armar algunos equipos dentro de desarrollo que hacían una parte de definición de toda la parte que era automation y el testing, pero ya con herramientas propias que usaban los equipos. De ahí se empezó a usar Java con Selenium[8] para automatizar, porque el 90% de los equipos usaba Java; o desarrollaban es Scala pero sabían Java. Lo que se trataba de hacer era que el que definía los procesos en testing de las aplicaciones de ese equipo sea una persona asignada para eso, pero después que al mantenimiento de eso lo hagan todos, porque no había un equipo de testing tan grande; pero si había un equipo grande de desarrollo.

En el año 2013, como era muy difícil conseguir gente que automatice es decir, que tenga experiencia en automatización, la idea fue que desde *IT University* tengamos un equipo de dos personas que conozcan del tema y armar una capacitación. En paralelo fuimos articulando lo que es hoy en día el programa de "Jóvenes de Alto Vuelo", donde los chicos tienen un módulo de todo lo que es automation, pero no para que salgan de ahi y vayan a equipos a automatizar, sino para que aprendan todas las cosas que hay que hacer para asegurar su calidad del proceso de desarrollo.

Hubo equipos que, al momento de la capacitación, eran nuevos entonces tenían escasas automatizaciones hechas en Ruby, es el caso de *Agencias Afiliadas*. Su gerente me pidió que le diera una mano porque no estaba consiguiendo gente que supiera automatizar. Entonces le comenté que estábamos haciendo que los equipos de desarrollo automaticen y le ofrecí capacitarlos. Nosotros intentamos darle un panorama de cuáles eran las herramientas que había en ese entonces, pero cada uno de ellos decidía cómo hacerlo. Así que hoy en día los equipos fueron evolucionando y el arquetipo que se utilizaba en ese entonces para automatizar ya no se utiliza más. Hace unos años archivé el proyecto porque estaba trayendo errores.

¿Qué criterios se tomaron para las capacitaciones?

Había muchos equipos en Java y los equipos tenían que hacer y mantener su automatización. Entonces se trató de ir para el lado de Jenkins-java. Explicar cómo hacer para que se ejecuten de forma automática los tests. La idea fue salir con java y después cada equipo fue adoptando otras cosas.

Una de las ideas era tomar gente que por ahí tenga poco skill en automation para capacitarse y después asignarla a un equipo que necesite automation. Que se incorporen al equipo de desarrollo y que les ayude a hacer un mejor testing.

¿Se basaron en alguna experiencia de otra empresa?

En la consultora donde trabajé antes que en Despegar también era Project Manager y hacia las implementaciones de PeopleSoft[1] de Oracle. Todo ese proceso también requería de automatización y entonces logré mucha experiencia al usar una herramienta que se llama pi screen.

El dinamismo de *Despegar* ayudó a que lo empujemos, pero yo venía de hacer eso en una consultora. El equipo de Qa que hacía automation en Ruby, era la misma iniciativa que yo arranqué en 2005 con *Mercado Libre*.

Mercado Libre necesitaba un equipo que lo automatice, nosotros en la consultora resolvimos que podíamos automatizar en Ruby, que en ese momento era la única tecnología fácil para automatizar cosas. Era el código que se ajustaba más a tener capacidades para hacer testing. Por ejemplo Selenium[8] no existía, sino que apareció unos años después.

Esas herramientas que usábamos no eran trasladables a *Mercado Libre*. Entonces dijimos: “che, por qué no hacemos algo que sirva para un sitio desarrollado en tales plataformas”. Y así fue cómo empezamos a buscar qué tecnología podíamos usar y surgió lo de Ruby. Armamos una estrategia y luego surgió la posibilidad desde la misma consultora de aplicarlo en *Despegar*. Usamos la misma idea, pero tuvimos que adaptarnos al sitio, porque eran sitios diferentes; vender cosas de supermercado o electrónica no es lo que el turismo online. La experiencia la traíamos.

[1] Aplicaciones Oracle PeopleSoft https://www.oracle.com/ar/applications/peoplesoft

En el año 2011, cuando llegué a *Despegar* como *Project Manager* en IT, ese área de QA la cual le brindamos servicios, se había independizado y tenía su propio equipo de automatización. Dos años después surge esto de que: “los equipos tienen que aprender a hacer su testing, aprender a pensar en ello” y entonces empezamos a capacitar.

Lo que logramos en *Despegar* es comprobar que no necesariamente tenés que estar en un equipo de QA para testear, sino que puede hacerlo cualquier persona que tuviese las herramientas correctas. A parte las herramientas han evolucionado mucho tanto para testing como para monitoreo.

¿Cuáles cree que fueron los pros y contras que se vieron en ese momento?

Los contras eran un equipo de QA automation que desarrollaba sólamente en una tecnología, no avanzaba en modernizarse con eso.

Un punto positivo es que cada gerencia, cada equipo de desarrollo se haga cargo de lo suyo. Era un poco difícil dividir las aguas: ”vos sos hoteles, medimos hoteles y hacete cargo”. El tema de dividir y que cada aplicación pase a ser micro aplicación hizo que los problemas se dividieran también.

Otro aspecto positivo que encuentro es que divide las responsabilidades. En 2011 empezamos a dividir un monolítico que era todo hoteles, todo vuelos, todo lo que se te ocurra estaba ahí adentro. y vuelos subía bien y hoteles subía mal y el sitio se caía. Al empezar a dividir en aplicaciones más chicas hizo que el testing también pueda ser dividido.

Cada equipo se hacía cargo del testing de sus aplicaciones que cada vez se trataban de aplicaciones más pequeñas de lo que eran originalmente. Es más mantenible. La frase: “divide y triunfarás” sería lo que queríamos hacer nosotros: “sigamos haciendo lo mismo, que cada equipo se haga cargo y que las cosas sean de un tamaño que las podamos manejar”. Cuando algo es muy grande, hay que dividirlo.

Hoy en día hay miles de aplicaciones en *Despegar* muy chicas que cada una cumple el objetivo que tiene que resolver. Cuando las aplicaciones empezaban a ser muy grandes y se mezclaban y no se sabía si una lógica funcional iba en una o en otra, entonces se creaba otra aplicación. Se tomaba esa lógica funcional y se sacaba

afuera. Porque se trataba de evitar que se haga una aplicación enorme que sea inmantenible y que el conocimiento encima esté en 4 o 5 personas.

¿Por qué se eliminaron definitivamente los equipos dedicados únicamente al testing?

Como el área de Automation QA no tenía el ownership de los proyectos, la gente no testeaba. Entonces así como se automatizó la subida a producción, que sería una automatización del proceso de deploy, se empezó a delegar la responsabilidad del testing a los equipos.

El equipo de QA manual se eliminó totalmente porque no andaba. El testing manual no se eliminó. Hoy hay gente que hace testing manual, inclusive los mismos desarrolladores hacen testing manual. La gente de producto hace un *UAT*. Un desarrollador que no tiene algo automatizado, la primera vez que hace el testing, es manual. Cuando se cansa de hacer algo manual o sabe que va a repetir el proceso todas las veces que toque algo en ese código, hace los 10 casos de prueba borde y los automatiza. Al principio, el testing, siempre es manual. Al hacer la automatización necesitas hacer el testing manual para saber que estás automatizando bien.

Se elimina el equipo porque se necesita que la responsabilidad que tenía el equipo de QA, la pase a tener el equipo que desarrolla. Después cada gerencia adopta si tiene un equipo a parte que hace testing o si los mismos desarrolladores testean, según la aplicación y según lo que se necesite. Por ejemplo, FUlfitnes es muy grande y tiene un equipo de QA que hace automation, pero se encargan un poco de E2E de eso. No se encargan tanto del testing de las aplicaciones de forma particular. Ellos verifican que todos los datos que salen de un lado y se meten en el otro sean consistentes. Asumen la responsabilidad de la regresión, no tanto del testeo en sí.

¿Encontraron alguna contra de que la responsabilidad del testeo de las aplicaciones pasó a ser de los desarrolladores y testers?

No, la verdad que no porque el problema que teníamos muchas veces era que la persona que hacía testing manual no era del palo de IT. Entonces en esos casos el

testing manual consistía en decir: “hay un error en el sitio”. Pero no tenía las habilidades para meterse y ver cómo probar el servicio, ver si está respondiendo mal en tal campo. o si el servicio devolvía el valor en el campo equivocado. No tenían esos skills. Entonces el error volvía con un screenshot que decía: “veo esto mal”. No te explicaban cómo lo replicaban, cómo llegabas a esa situación. Algunos de los Qas que luego pasaron a desarrollo tenían skills técnicos y otros se reconvirtieron, pero no pasaba lo mismo con todos los casos.

Hasta muchas veces eran ambientes distintos los que usaban desarrolladores y testers con distintos datos. El desarrollador tiene los propios ambientes dentro del equipo, sabe los casos bordes, está metido en el tema y cada vez que toca algo sabe qué es lo que puede romper y lo que no. Es como que logras que se involucre más.

Lo que sí se logró es que se haga el testeo manual con más skils, porque hoy lo hace el equipo de IT. Al momento que detectan un error, lo pueden arreglar. y no estamos dependiendo de un proceso de testing en el que se reporta un caso, lo arreglan, vuelve. Eso es muy tedioso. Creo que ahí se ahorra mucho tiempo.

El testing mejoró en su calidad del cómo hacerlo, no en la calidad de qué encuentra, qué no encuentran. El hecho de sumar más herramientas hace que posiblemente encuentres más cosas en el mismo lapso de tiempo.

¿Qué pasa con los casos que no se pueden testear en ambientes de prueba o son costosos de automatizar?

Con el monitoreo en producción, la gente sabe si subió algo que rompe o no rompe. Con el movimiento que tenemos hoy en el sitio a veces no se puede automatizar algo con casos reales y o emular situaciones que pueden llegar a suceder. Muchas veces el testing se traduce a un seguimiento y monitoreo en producción donde se arma un propio dashboard y se ve cómo varían las curvas. Entonces en realidad estás probando con el usuario que está utilizando el sitio. En los casos en los que no hay riesgo alto, el testing se termina derivando a producción porque es más barato que intentar reproducir o generar un caso en un ambiente de prueba.

¿Continúan las capacitaciones al personal para crear pruebas automatizadas?

Las capacitaciones específicas que se dieron durante los años 2013 a 2015 ya no se dan más porque después dejamos de tomar a la gente de automation nosotros desde el equipo de *IT University*.