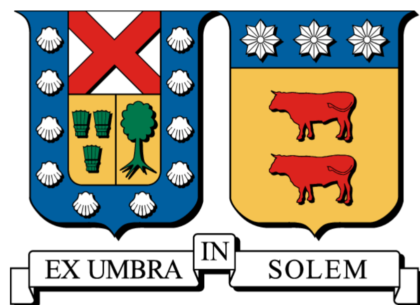
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE



PLAN DE CALIDAD DE SOFTWARE PARA EL SISTEMA INTRANET DE UNA UNIDAD ACADÉMICA

Víctor Antonio Zúñiga Millán

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Carlos Buil Aranda
Profesor Correferente: Cecilia Reyes Covarrubias

Noviembre - 2018

Agradecimientos

Primero, agradezco a mi hermosa familia. A lo largo de los años he descubierto lo privilegiado que soy de poder atender una educación superior de calidad, y no podría haberlo hecho sin las decisiones y el apoyo de mi familia.
Además, agradezco a mis amigos: a los del Imperio que conozco desde el colegio, mis amigos que he conocido en línea, y los amigos que he encontrado en la universidad, en especial a los Jorges, Orlando, Patricio, Ignacio, Fernando, Pablo, Reynaldo, Alejandro, Alondra e Inti, con quienes he trabajado, he reído y he pasado muy buenas experiencias.
Y también agradezco a los amigos que he conocido a lo largo de la realización de esta práctica: Eduardo, Adbías, Geordy, Herman, Diego, Aldo y Matías, con quienes he compartido buenos momentos, y de quienes he recibido ayuda innumerables veces.
Finalmente, agradezco a Yonathan Dossow, Miguel Varas, Carlos Buil, Cecilia Reyes y a los miembros del Departamento de Informática de la UTFSM Casa Central, quienes me han dado el espacio y las herramientas para llevar esta memoria al éxito.

Resumen

Esta memoria trata la formulación e implementación de un plan de calidad de software para el desarrollo del sistema web Intranet del Departamento de Informática de la Universidad Técnica Federico Santa María. Intranet es un sistema que busca agilizar varios de los procesos internos que involucran a estudiantes, profesores y funcionarios del departamento antes mencionado. Las funcionalidades son desarrolladas, en general, por estudiantes en práctica y ayudantes bajo una metodología iterativa incremental, pero existen varios problemas en ésta que conllevan a la baja satisfacción de los usuarios finales. Se realizó una reseña sobre las metodologías de desarrollo de software que se podrían aplicar para resolver el problema, junto con las herramientas técnicas que van a complementar tales metodologías. Luego, se presenta una comparativa de éstas según los requerimientos del proyecto y las circunstancias de éste, para luego proceder a una implementación. A partir del trabajo realizado, se logra definir un esquema de trabajo que no solo permite un aseguramiento de calidad al proyecto Intranet, sino que también llega a definir un estándar para el desarrollo del software de uso interno por parte del Departamento de Informática.

Palabras clave: Aplicación web, Plan de Calidad, DevOps, Integración Continua, Pruebas automáticas

Abstract

The following memoir presents the formulation and implementation of a Software Quality Assurance plan (SQA plan) for the Intranet web system of the Departamento de Informática (Department of Informatics) at Universidad Técnica Federico Santa María. Intranet is a web application designed to help streamline many of the processes involving students, teachers, and staff of this department. The features of this application are developed, in general, by internship students and assistant students under an incremental iterative methodology, from which many problems have been discovered, thus leading to low end-user satisfaction with the system. In this document, many of the software development methodologies that could be applied to the project, together with the technical tools that complement them, are reviewed and compared. Then, an SQA Plan is prepared and implemented using the chosen options. This plan has succeeded in creating a workflow that not only provides the Intranet project with it's needed quality assurance, but also serves as a new standard for use in the development of other software projects tailored to this department.

Keywords: Web application, SQA Plan, DevOps, Continuous Integration, Test automation

Glosario

API: *Application Programming Interface*, o interfaz de programación de aplicación, es un conjunto de funciones o subrutinas que un software o biblioteca ofrece a otro software o biblioteca, para que puedan interactuar entre sí. De esta forma, se puede "llamar" a una API para obtener cierta información o efectuar un cambio sin tener que conocer el código del software o biblioteca que tiene tal API.

***Build*:** Es la tarea que "construye" cierta versión de un software. En los sistemas de Integración Continua, esta construcción puede implicar la ejecución de *scripts*, la realización de pruebas y la compilación/despliegue de esta versión del software, con tal de que quede lista para su uso.

Deprecado: Anglicismo, del inglés *deprecated*, son las partes de un software que están consideradas obsoletas y deben ser desechadas o reemplazadas en un futuro cercano.

Modularidad: Es la capacidad de un sistema de funcionar como un conjunto de elementos o subsistemas, donde la separación de éstas está bien definida y con poca interdependencia.

Parche: Un conjunto pequeño de cambios de código que se despliegan en un software para solucionar un problema, actualizar componentes o agregar mejoras.

***Pipeline*:** En la práctica de Integración Continua modernos, es el proceso que se se debe seguir para realizar un *build*. Está conformado por *stages* o fases, donde cada fase tiene uno o más *jobs* o trabajos. Un *stage* va después de otro, y se pasa al siguiente solo cuando todos los *jobs* del *stage* actual se hayan completado exitosamente. Los *jobs* de un mismo *stage* se pueden realizar en paralelo.

Producción: Es la modalidad del software en el cual éste se configurada para su uso por parte de los usuarios finales. Generalmente se hace la distinción entre el ambientes o modo de producción respecto al modo de desarrollo y al modo de pruebas (aunque otros pueden existir).

Repositorio VCS: Es una localización remota (generalmente un servidor), donde se aloja el código de uno o más proyectos. Este código es manejado por los VCS de los desarrolladores, los cuales podrán subir y descargar cambios.

***Runtime*:** Es un software diseñado para ejecutar programas escritos en algún lenguaje de programación o algún framework. Puede incluir elementos que estos programas utilizan, como bibliotecas y funciones.

Servidor réplica: En un software bajo una arquitectura distribuida, el servidor réplica es un servidor que comparte el mismo software (o parte de éste) respecto a un servidor primario, y que recibe órdenes o peticiones de este último. A veces la relación primario/réplica es llamada maestro/esclavo o primario/secundario.

Software *self-hosted*: Es un software que funciona como un servidor en la arquitectura cliente-servidor, y que puede ser instalado manualmente en una máquina de preferencia para su uso privado. Esto contrasta el SaaS (*software as a service*) el cual es instalado y alojado en servidores remotos de propiedad de los propietarios de este software, los cuales en general cobran una subscripción por su uso.

***Suite* de pruebas:** Es un conjunto de varios casos de prueba que validan todas las funcionalidades del sistema, o un conjunto de funcionalidades importantes. Cuando son pruebas automatizadas que se acumulan a través del desarrollo, también se les llama "banco de pruebas".

***Suite* de software:** Es un conjunto de programas que se distribuye y/o comercializa en conjunto, y que en general tiene funcionalidades relacionadas.

VCS: Version Control System, es un software encargado de administrar los cambios en el código de un proyecto de software. En general, generan registros con los cambios realizados en el código dentro de un período, indicando quién hizo los cambios y en qué momento se agregaron al código. Permiten además el trabajo colaborativo al poder unir cambios de varios colaboradores.

Versionado: Es la acción de asignar un número y/o nombre a un software en un momento dado según su desarrollo a través del tiempo, con tal de diferenciarlo respecto al estado del software en otros momentos.

Índice General

Glosario **3**

1. Introducción **13**

1.1. Motivación y problema a abordar 13

1.2. Objetivos de la memoria 14

1.2.1. Objetivo general 14

1.2.2. Objetivos específicos 14

1.3. Alcance 14

1.4. Estructura del Documento 15

2. Definición del Problema **17**

2.1. Qué es Intranet 17

2.2. Breve historia y contexto del desarrollo de Intranet 18

2.3. El equipo de trabajo 18

2.4. Aspectos técnicos de Intranet 19

2.4.1. Servidor de producción 19

2.4.2. Conexiones a otros sistemas 20

2.4.3. Ambiente de desarrollo 21

2.4.4. Recursos disponibles 21

2.5. Metodología de desarrollo actual 22
2.6. Problemas específicos en la metodología actual 23
2.7. Conclusiones 24

3. Estado del Arte **25**

3.1. La tecnología detrás de la Intranet: Ruby on Rails 25
3.1.1. Características y ventajas 25
3.2. Calidad de software 28
3.2.1. Qué es calidad: dos perspectivas diferentes 28
3.2.2. Aseguramiento de la calidad del software (SQA) 28
3.3. Plan de Calidad de Software 31
3.4. Pruebas de software automatizadas 32
3.4.1. Tipos de pruebas automatizadas 33
3.4.2. Formas de automatizar las pruebas 35
3.4.3. Herramientas para su implementación en aplicaciones RoR . . . 35
3.4.4. Resumen comparativo de los tipos de pruebas 38
3.5. Prácticas modernas en el desarrollo de software 40
3.5.1. Desarrollo guiado por pruebas (TDD) 40
3.5.2. Desarrollo guiado por comportamiento (BDD) 40
3.5.3. Integración Continua (CI) 41
3.5.4. DevOps y tendencias actuales 42
3.5.5. Herramientas para su implementación en aplicaciones RoR . . . 43
3.6. Desarrollo Ágil 50
3.6.1. Historia y definición 50
3.6.2. Metodologías de Desarrollo Ágil 52

3.6.3. Resumen comparativo de metodologías ágiles 61

3.7. Desarrollo de código abierto 62

3.8. Conclusiones 64

4. Propuesta de Solución 65

4.1. Metodología de desarrollo 65

4.1.1. Análisis de las metodologías existentes 66

4.1.2. Actividades escogidas desde metodologías existentes 69

4.1.3. Resumen de la metodología 70

4.2. Metodología para pruebas de software 71

4.2.1. Análisis de las metodologías existentes 71

4.2.2. Metodología escogida 72

4.3. Comparativa y selección de herramientas 72

4.3.1. Herramientas de gestión de configuración del software 72

4.3.2. Herramientas de desarrollo 75

4.3.3. Herramientas de pruebas 75

4.3.4. Herramientas para Integración y Entrega Continua 77

4.3.5. Herramientas de comunicación 77

4.3.6. Herramientas de comunicación con el cliente 77

4.4. Conclusiones 78

5. Implementación de la Solución 79

5.1. Trabajo realizado 80

5.2. Desarrollo de *suite* de pruebas 81

5.2.1. Uso de VCR como herramienta de captura/repetición 82

5.3. Despliegue de servidor de pruebas: Servidor de puesta en escena o *staging* 84

5.4. Despliegue de servidor de CI/CD: Jenkins 86
5.5. Integraciones y configuraciones 89
5.6. Conclusiones 90

6. Validación de la Solución 92

6.1. Caso de evaluación: Sistema de *tickets* en Intranet 92
6.2. Resultados y métricas del despliegue 93
6.3. Resultados de la aplicación de la metodología 94
6.4. Observaciones en el equipo de trabajo 95
6.5. Conclusiones de la validación de la solución 96

7. Conclusiones 97

7.1. Conclusiones generales 97
7.2. Cumplimiento de objetivos 97
7.3. Sobre el desarrollo de las pruebas 99
7.4. Trabajo a futuro 101

Referencias 102

A. Plan de Calidad de Software para Intranet 104

A.1. Objetivo 104
A.2. Documentos adjuntos 104
A.3. Administración 105
A.4. Documentación 106
A.5. Estándares, prácticas, convenciones y métricas 107
A.5.1. Flujo de trabajo 107
A.5.2. Comunicación entre el equipo 109

A.5.3. Comunicación con el cliente 109

A.6. Reporte de problemas y acciones a realizar 110

A.7. Revisiones e inspecciones 111

A.8. Gestión de configuración del software 111

A.9. Herramientas, técnicas y tecnologías 113

A.9.1. Editores de código y/o IDEs 113

A.10. Metodología de las pruebas de software 113

Índice de Figuras

2.1. Diagrama de componentes de alto nivel de Intranet 20

3.1. Componentes de Ruby on Rails 27

3.2. Interfaz de usuario convencional de Jenkins 45

3.3. Arte conceptual del proyecto BlueOcean 45

3.4. Arte promocional de TeamCity 46

3.5. Arte promocional de GitLab 47

3.6. Las tres fases de un proceso de desarrollo Scrum 54

3.7. Las dos dimensiones de las metodologías Crystal 55

3.8. Una iteración o “incremento” en las metodologías Crystal 57

3.9. Un ejemplo de *kanban board* 59

3.10. Captura de pantalla de *builds* del software de código abierto Dolphin . 63

5.1. Diagrama de alto nivel que muestra el nuevo flujo de trabajo 79

5.2. Un ejemplo de la estructura de las pruebas en RSpec con Capybara . . 82

5.3. Configuración y ejemplo de VCR 83

5.4. Diagrama de alto nivel mostrando cómo se configuró el servidor *staging* 84

5.5. Captura de pantalla de la aplicación web Intranet en el servidor *staging* 85

5.6. Captura de pantalla de Jenkins instalado 86

5.7. Captura de pantalla de *plugin* de credenciales de Jenkins 87

5.8. Captura de pantalla de la interfaz BlueOcean de Jenkins 88

5.9. Captura de pantalla de VCS BitBucket 89

A.1. Organigrama I&T 105

Índice de Tablas

3.1. Comparativa de tipos de pruebas de software 39

3.2. Comparativa de herramientas CI/CD *self-hosted* 48

3.3. Comparativa de metodologías ágiles 61

Capítulo 1

Introducción

En esta sección se hablará de la motivación del trabajo realizado, junto con una introducción al proyecto de software analizado y su problema. Se indicarán los objetivos de la memoria, y la estructura general del documento.

1.1. Motivación y problema a abordar

Hoy en día los sistemas de computación ayudan a las organizaciones a cumplir muchas de las tareas repetitivas y procesos que requieren operar con datos e información, a tal punto que muchas de las industrias de nuestra sociedad llegan a depender de estos sistemas. Por lo tanto, es importante tener en cuenta la importancia del rubro de desarrollo de software, y la cantidad de personas que son afectas a ésta. Un proceso de desarrollo de software problemático puede producir software de mala calidad, que ocasionará problemas a los usuarios finales. Lamentablemente, las metodologías y prácticas que pueden evitar estos problemas no son utilizados por estudiantes universitarios en sus proyectos de pregrado, y conlleva a que los proyectos realizados por éstos no se ajusten a los estándares de la industria.

A partir de lo anterior es que se toma el caso de estudio de este documento, el cual es el proyecto Intranet del Departamento de Informática (DI) de la Universidad Técnica Federico Santa María. Este proyecto trata del desarrollo de un sistema para agilizar varios de los procesos que profesores, funcionarios y alumnos del DI podrían necesitar, los cuales normalmente requieren trabajo burocrático de parte de las secretarías y funcionarios. Estos procesos son distintos entre sí, pero muchas veces comparten datos y no se justifica el tener que crear un proyecto nuevo para cada uno. Así, se busca que desde el sistema Intranet los usuarios del departamento puedan realizar estos procesos utilizando una misma interfaz, y compartiendo el mismo sistema de credenciales y autorización. Por ello es que Intranet es desarrollada en base a módulos, los cuales se

construyen y mejoran a través del tiempo.

El sistema Intranet funciona como una aplicación web desarrollada en *Ruby on Rails* por varios estudiantes en práctica, ayudantes administrativos y funcionarios de la Unidad de Infraestructura y Tecnología (I&T) del departamento. Sin embargo, dado a que el fuerte de esta unidad es la administración de sistemas y no el desarrollo de software, es que se le deja el trabajo principalmente a estudiantes. En los últimos meses se han encontrado varios problemas en la metodología de desarrollo (o la falta de ésta), lo cual ha ocasionado numerosos problemas en los procesos de prueba y despliegue de Intranet, e incluso se han encontrado errores graves en producción por parte de los usuarios finales. Se tiene, por lo tanto, un problema en la calidad del software final.

1.2. Objetivos de la memoria

1.2.1. Objetivo general

Generar e implementar un plan de aseguramiento de la calidad del software en el sistema web Intranet del Departamento de Informática de la Universidad Técnica Federico Santa María

1.2.2. Objetivos específicos

1. Conocer las características del equipo de desarrollo de la Intranet, y las circunstancias de su trabajo.
2. Comparar metodologías de desarrollo y herramientas técnicas que puedan formar un plan de calidad para el software Intranet.
3. Implementar el Plan de Calidad en el ambiente de desarrollo y despliegue del software Intranet.
4. Proponer recomendaciones para otros proyectos de software del Departamento de Informática.

1.3. Alcance

Esta memoria se basa en la creación de un Plan de Calidad de software en un proyecto de baja envergadura y de uso interno en el Departamento de Informática, donde el equipo de desarrollo cumple características especiales por ser, en general, estudiantes

de pregrado. Por ello, la metodología de desarrollo, las prácticas y las herramientas a implementar son a baja escala, y no buscan cumplir con algún estándar de madurez organizacional de tipo ISO o CMM. Las métricas de calidad se basarán en la reducción de defectos de software, cumplimiento de requerimientos y cobertura de las pruebas.

La implementación en sí será a lo largo del desarrollo de uno de los módulos nuevos de Intranet, el cual se asimila a la creación de proyectos de software de baja envergadura dentro del departamento, con un uso principalmente interno y baja complejidad. Se busca que esto sirva de prueba para refinar el plan, y poder demostrar su efectividad ante las limitaciones de este tipo de proyectos.

1.4. Estructura del Documento

En esta memoria comenzará con una **Definición del Problema**, donde se verá a profundidad cuál es el contexto en el cual se desarrolla Intranet, junto con los involucrados, las herramientas utilizadas y el proceso de desarrollo de software que se sigue. A partir de esto, se podrán encontrar los problemas específicos que conllevan a la reducción de calidad en el software, y así poder comenzar a definir un plan para enfrentarlos.

Luego, se realizará el **Estado del Arte**, donde se analizarán varios de los conceptos, metodologías y herramientas de desarrollo de software que son parte de los estándares actuales para asegurar la calidad en proyectos de envergadura y tecnologías similares. Así, al poder conocer cómo se trabaja en la industria, se podrán encontrar soluciones a los problemas del proyecto, y se podrá comenzar a definir los elementos del plan de calidad.

Se continuará con la **Propuesta de Solución**, donde se definirá una metodología de trabajo a partir de la comparación de distintas acciones y herramientas investigadas anteriormente. Esta comparativa tomará en cuenta las características y limitaciones del equipo de trabajo del proyecto Intranet, y funcionará como una primera versión del Plan de Calidad a generar.

Con esto podremos pasar a la **Implementación de la Solución**, donde se detallarán las acciones realizadas con tal de aplicar el plan de calidad en el desarrollo de la Intranet. Esto va a implicar la definición formal del nuevo flujo de desarrollo, junto con la instalación y preparación de las herramientas técnicas, lo cual va a incluir código nuevo, pruebas o *tests* automatizados, y servicios que complementen las operaciones de desarrollo (lo que se conoce como *DevOps*).

Luego de esta implementación, se presentará la **Validación de la Solución**, donde se mostrarán y analizarán las consecuencias de la aplicación de la metodología en el desarrollo de un nuevo módulo de Intranet que se desarrolló a lo largo del 2018.

Finalmente, en las **Conclusiones** del documento se entregarán correcciones y recomendaciones para afinar el Plan de Calidad, y así tener una guía para éste y otros proyectos similares.

Todo este trabajo se sostiene en la tesis de que, aplicando una metodología de desarrollo de software que se sostenga en el uso de las herramientas correctas, se podrá asegurar la calidad del sistema sin tener que rehacer todo el proyecto (lo que se conoce como "re-ingeniería"). Esto beneficiaría a todos los miembros del Departamento de Informática, y se espera que deje un legado importante en los estándares de calidad de los sistemas internos desarrollados en los laboratorios del Departamento de Informática.

Capítulo 2

Definición del Problema

En este capítulo se explicará en detalle las características del proyecto Intranet, el cual es desarrollado y desplegado a través del tiempo con nuevos módulos, pero sin una metodología que pueda asegurar un estándar de calidad que no sea propenso a error humano. Los problemas del proyecto se asemejan a los problemas comunes en la industria, pero las características técnicas del software y las características del equipo de trabajo no permitirán el uso de metodologías de desarrollo muy estructuradas, o que asuman desarrolladores con experiencia.

2.1. Qué es Intranet

El caso de estudio de este documento es el sistema Intranet del Departamento de Informática (DI), de la Universidad Técnica Federico Santa María. Este es un sistema web, accesible desde cualquier navegador, desde el cual se pueden realizar trámites, peticiones y procesos internos del departamento, los cuales normalmente requerirían hablar con secretarías (de éste y otros departamentos) o efectuar peticiones a la Unidad de Infraestructura y Tecnología (I&T). Su acceso se restringe, por ahora, a profesores, funcionarios y alumnos del departamento, los cuales podrán entrar utilizando su cuenta del DI (administrada por un directorio LDAP) a la que se le darán acceso a las funcionalidades según ciertos roles asignados por los administradores del sistema (lo que es conocido como RBAC, o *Role Based Access Control*).

El Intranet que se tiene hoy en día es un sistema relativamente nuevo, que en sus inicios buscaba reemplazar al sistema "Personas" del Departamento de Informática, encargado de efectuar cambios en el directorio LDAP de éste. Se vio que este último fue desarrollado con pocos estándares de código, seguridad e interfaz, y añadir nuevas funcionalidades sería muy costoso. Sin embargo, la aparición de variados requerimientos por parte del departamento hicieron que el proyecto Intranet cambiara de rumbo hacia

un sistema de múltiples funcionalidades.

Ante ello, se decidió crear este sistema basado en Ruby on Rails 5.0, el cual es un *framework* web pensado para el desarrollo ágil, la rápida construcción de prototipos funcionales, e ideal para sistemas que no requieren atender un número elevado de peticiones.

2.2. Breve historia y contexto del desarrollo de Intranet

La necesidad de una Intranet para el Departamento de Informática surge en el año 2009, cuando el entonces Director de Departamento Carlos Castro propone como trabajo de título para algún estudiante de Pregrado de las carreras de Informática, el desarrollo de un sistema web para manejar ciertos procesos: la subida y descarga de documentos, el llenado de formularios para permisos de viajes, y el llenado de formularios de contabilidad. Este sistema logró ser desarrollado, pero su implementación dejó mucho que desear.

Además, Castro propuso que el sistema podría ser capaz de generar informes anuales sobre indicadores relevantes a la estrategia del departamento (por ejemplo, tasas de titulación, gastos por área, etc.). Sin embargo, esta idea fue ignorada en ese momento, y no sería tomada en cuenta hasta el año 2016 (en forma separada a la Intranet).

Esto llevó a que en el año 2014 se comenzó a preparar una re-ingeniería de la Intranet. Se desechó el sistema anterior, y se eligió un desarrollo basado en trabajo iterativo sobre módulos. El desarrollo comenzó con el trabajo de tres personas (dos ayudantes administrativos y un estudiante en práctica) y ha ido avanzando lentamente debido a la rotativa de estudiantes, los cuales serán desarrolladores y mantenedores del sistema, pero que por su carga académica, solo podrán entregar al proyecto una cantidad limitada de horas por semana.

2.3. El equipo de trabajo

El trabajo en Intranet ha continuado con su modalidad histórica, donde en todo momento el equipo no supera cuatro personas, y se conforma por un conjunto de:

- **Estudiantes en práctica**, los cuales trabajan por dos meses de verano, en modalidad de tiempo completo (de 9:00 a 18:00 horas). Son estudiantes de pregrado de Informática o Telemática en la universidad, y por ende, tienen poca experiencia.

- **Ayudantes administrativos de I&T**, que en general son estudiantes que anteriormente tuvieron su práctica en la Unidad de Infraestructura y Tecnología, y que desean continuar el trabajo en Intranet a lo largo de uno o más semestres. Trabajan en promedio 15 horas semanales. Tienen algo de experiencia, pero tampoco la suficiente para ser considerados “desarrolladores profesionales”.

- **Funcionarios de I&T**, los cuales trabajan tiempo completo pero que en general, tienen varias otras responsabilidades, y no pueden atender a Intranet en forma prioritaria: en general solo se encargan de temas administrativos, corrección de errores y tutoría a los demás miembros del equipo. Tienen más experiencia, pero menos tiempo.

Este equipo de trabajo, por ende, no puede siempre tener un período de trabajo consistente, donde todos los miembros puedan dedicarse de lleno al proyecto Intranet. El período de verano ve mayor productividad gracias a los practicantes, pero el entrenamiento que requieren y la menor disponibilidad de los ayudantes y funcionarios (debido a otras labores) no permiten un ambiente que se parezca al de un equipo de desarrollo de software profesional. Y en el resto del año, con otros labores y deberes que se agregan, se da un ambiente con alta rotativa y horarios flexibles.

2.4. Aspectos técnicos de Intranet

Es necesario saber qué *hardware* y software se está utilizando actualmente para desarrollar y desplegar Intranet, y así tener una idea de la envergadura del proyecto. La implementación de nuevas metodologías de trabajo o nuevas actividades podrían requerir la utilización de nuevos recursos, o modificar los actuales.

2.4.1. Servidor de producción

El servidor de producción de Intranet es una máquina virtual ejecutando Red Hat Enterprise Linux (RHEL) 7.3 en 64 bits, a la cual se le ha asignado 2 GB de RAM y 12 GB de espacio en disco duro. Esta máquina virtual corre sobre un servidor Dell, el cual cuenta con un procesador Intel Xeon E5-2640 v2 (Ivy Bridge) y se ha configurado para tener acceso a 2 núcleos de CPU.

Intranet sirve sus conexiones mediante Nginx, el cual es un software de servidor web que será visible a conexiones remotas. Éste se conecta con Unicorn, el cual es la gema (*plugin* del lenguaje de programación Ruby) que se encarga de manejar las peticiones web entregadas por Nginx y respondidas por la Intranet, junto con las peticiones de Intranet hacia otros servidores.

2.4.2. Conexiones a otros sistemas

Respecto a este último punto, el servidor Intranet requiere estar conectado a los siguientes servicios externos:

- Base de Datos PostgreSQL, la cual se ubica en *db.inf.utfsm.cl* y comparte hardware con otras bases de datos de uso interno del Departamento de Informática. Usar esta base de datos remota en vez de una local permite aprovechar los beneficios de este servidor, como son respaldos automáticos, discos duros SSD (discos de estado sólido, más rápidos que los discos duros convencionales), y suficiente RAM y CPU para atender una gran cantidad de peticiones por unidad de tiempo.
- API del sistema LDAP (*Lightweight Directory Access Protocol*), el cual se ubica en *ldapapi.inf.utfsm.cl* y permite el intercambio de información con el directorio de personas del Departamento de Informática.
- API del sistema Ruckus, el cual se ubica en *ruckuscontroller.inf.utfsm.cl* y permite la creación de Pases de Wi-Fi para invitados del DI. En un futuro, esta API podría otorgar más funcionalidades.
- Sistema SAML (*Security Assertion Markup Language*) para autenticación con SSO (*Single Sign On*). Se ubica en *saml.inf.utfsm.cl* y sirve para la autenticación al entrar a Intranet, utilizando las credenciales del directorio de personas del Departamento de Informática.

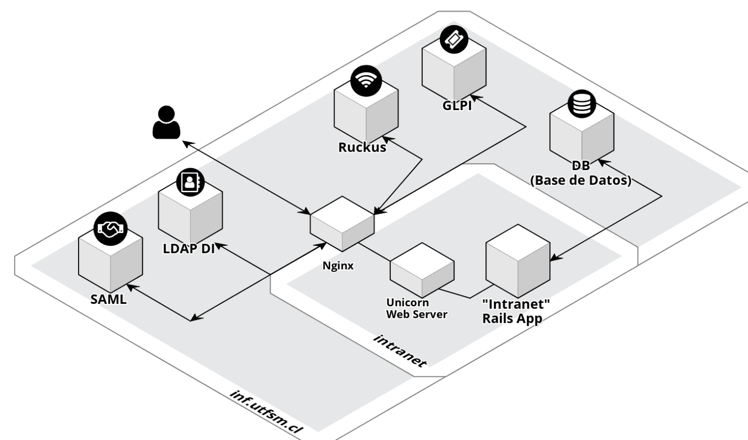


Figura 2.1 *– Diagrama de componentes de alto nivel de Intranet, donde se aprecia el uso de servicios externos que son parte de la red del Departamento de Informática. Fuente: Elaboración propia, utilizando la herramienta Cloudcraft.*

2.4.3. Ambiente de desarrollo

El trabajo en general se desarrolla en el Data Center del Departamento de Informática, donde se encuentran computadores de variadas configuraciones, pero que en general permiten el desarrollo de aplicaciones web (gracias a procesadores modernos, suficiente memoria y conexiones a la red por cable).

Vale mencionar que varias de las conexiones usadas por Intranet (como las lecturas al directorio de personas LDAP, o la conexión al VCS) están restringidas a la subred de I&T. Esto hizo que, antes del desarrollo de esta memoria, los miembros del equipo estén obligados a utilizar un computador del Data Center o llevar uno propio a este lugar para poder trabajar y efectuar pruebas sin problemas de conexión.

2.4.4. Recursos disponibles

La filosofía de trabajo de los laboratorios del Departamento de Informática favorece el despliegue y uso de servicios mantenidos dentro del mismo departamento, o sea, se evita la dependencia de sistemas externos. Esto con tal de permitir a los ayudantes y estudiantes en práctica a aprender administración de sistemas, y poder mantener bajos los costos de los servicios.

Para el desarrollo y mejora del proyecto Intranet, I&T entrega los siguientes recursos:

- Acceso a la **suite Atlassian** contratada por I&T: Jira, Confluence y BitBucket. Posible acceso a otros servicios de Atlassian si se requieren (según el precio).
- Otros servicios del DI ajenos a I&T, si se requieren (por ejemplo, GitLab, cliente webmail Zimbra, sistema de *tickets* GLPI, etc.)
- **Servidores**: Acceso a máquinas virtuales a pedido, con características similares o superiores a las del servidor actual de Intranet (Intel Xeon, 2 núcleos, 2GB de RAM, 12GB de espacio en disco duro). En ellas también se da acceso de superusuario con todo lo que se necesite instalar, mientras no sobrecargue el hardware. Se da preferencia a instalar software gratuito y *self-hosted*, y a acoplar los servicios lo menos posible (para evitar que fallos de un sistema afecten a otros).

La red donde se ejecuta Intranet es la misma donde se encuentran varios de los servicios a los que se conecta, y también es la misma red a la cual los computadores de desarrollo tienen acceso. Para la implementación del plan de calidad, es posible desplegar nuevos servicios *self-hosted* mediante máquinas virtuales, y pedir más recursos para las máquinas actuales.

2.5. Metodología de desarrollo actual

Como se mencionó anteriormente, el desarrollo del sistema Intranet es bastante particular, ya que los desarrolladores son ayudantes administrativos y estudiantes en práctica, con el apoyo de funcionarios de I&T. Esto conlleva a que exista rotación de personal, ya que los estudiantes en práctica solo permanecen por un par de meses, y los ayudantes administrativos vienen y van, dependiendo de su elección propia y la carga académica que les corresponde. En general, en un momento dado, Intranet va a tener de uno a tres desarrolladores.

A través del trabajo efectuado a lo largo del año 2017, se ha descubierto que no existe una metodología de trabajo bien acabada en este proyecto, y por lo tanto, no se puede asegurar la calidad de los despliegues de Intranet. Este trabajo, en general, comenzaba con las siguientes actividades para los nuevos miembros:

1. El nuevo desarrollador instala un ambiente de desarrollo en alguna de las estaciones de trabajo del Data Center del DI. Vale mencionar que en algunos casos se instalan dependencias y *middleware* incorrecto. Por ejemplo, MariaDB como motor de base de datos, en vez de PostgreSQL, el cual es el que se utiliza actualmente en el servidor de producción.

2. Se agrega la llave SSH de la máquina de desarrollo al repositorio Stash, el cual es un servidor de control de versiones (VCS) compatible con Git, parte de la suite de Atlassian. Esta suite es utilizada por miembros de I&T, pero por límites de la licencia, solo permite su uso por diez personas. Por lo tanto, la llave SSH es añadida en la cuenta de algún funcionario de I&T, y no necesariamente en una cuenta propia.

Luego, para el desarrollo de cada funcionalidad, se usaba el siguiente flujo de trabajo.

1. Cada desarrollador debe crear en el programa de control de versiones Git una rama nueva, representando la nueva funcionalidad o *feature* que se va a trabajar.

2. El desarrollador construye la nueva funcionalidad o los cambios requeridos. A lo largo del desarrollo, se va probando lo que se va creando con un navegador conectado a un servidor local en modo "desarrollo" (usando la base de datos local, y no la del Intranet en producción).

3. Luego de terminar su trabajo, el desarrollador efectúa pruebas manuales en su servidor local mediante el navegador. Se arreglan los problemas encontrados.

4. Una vez lista la sección anterior, se suben los cambios al servidor *Stash*, para así poder aplicarlos en otras máquinas. Aquí, el encargado del despliegue de la

plataforma efectúa pruebas en su servidor local, y si no se encuentra error alguno, se suben los cambios al servidor en producción.

5. En el mismo servidor de producción, se hacen otras pruebas, a veces con las personas interesadas en la nueva funcionalidad. Si se encuentran errores en ésta o la etapa anterior, se deben arreglar, para luego repetir el proceso de pruebas y despliegue.

2.6. Problemas específicos en la metodología actual

Ante el análisis del flujo de trabajo actual y las consecuencias de éste, se presentan los cuatro problemas que han llevado a los defectos de software y la reducción de la productividad:

- Muchas etapas de **pruebas manuales**, las cuales están sujetas a error humano y gastan mucho tiempo en actividades del navegador que en ciertos casos no solo modifican bases de datos locales, sino que al probar módulos que se conectan con APIs externas, se impone carga en éstas también (por ejemplo, creando pases de Wi-Fi).
- **Dependencia del jefe de proyecto** o de quién sea el encargado del despliegue. El conocimiento de las herramientas de desarrollo, el acceso al servidor y otros temas relacionados a la operación y mantenimiento tienen **poca documentación**, y en general se enseñan a otros miembros del equipo en forma presencial.
- Exceso de **dependencias de software y de red** en el ambiente de desarrollo. Esto obliga a tener que trabajar en la red de I&T (no se permite trabajo remoto), y ha producido problemas cuando la conexión a alguna de estas dependencias deja de funcionar. Estos problemas son notables cuando se unen nuevos miembros al equipo de trabajo, y el tiempo de preparación del ambiente de trabajo se hace excesivo.
- Uso de muchas ramas de desarrollo con el sistema de control de versiones, pero sin "versionado" y con **mala organización**. Cada desarrollador va creando ramas, pero al tener que resolver problemas en medio del desarrollo de una funcionalidad, se van efectuando cambios en otros módulos y se pierde la idea de que cada rama es auto-contenida.

2.7. Conclusiones

Estos problemas del proceso de desarrollo nos dan la idea de que debería haber una metodología de trabajo acorde a los desarrolladores, los cuales tienen alta rotativa, disponibilidad limitada y poca experiencia. El hecho de que estén pocas personas trabajando en Intranet en un momento dado hace que los problemas específicos encontrados (mala organización, documentación y falta de pruebas) lleven a defectos en el software. Se deben tomar en cuenta las preferencias del departamento a utilizar soluciones *self-hosted* y gratuitas, y se deben aprovechar los recursos dados por I&T (como las máquinas virtuales) para la implementación de herramientas que ayuden a resolver los problemas encontrados.

Capítulo 3

Estado del Arte

En esta sección se hará una reseña sobre la tecnología detrás de Intranet (Ruby on Rails), y las definiciones de calidad de software que permiten comenzar a buscar metodologías y prácticas que se adecúen a resolver los problemas del proyecto. Así, se podrán tomar las actividades y herramientas técnicas utilizadas en otros proyectos Rails, como son las pruebas automatizadas y la integración continua, comparar sus alternativas de implementación y adaptarlas a una metodología de trabajo basada en las tendencias del desarrollo ágil y del desarrollo de código abierto (*open-source*).

3.1. La tecnología detrás de la Intranet: Ruby on Rails

Ruby on Rails (RoR, o simplemente Rails) es lo que se conoce como un *framework* para aplicaciones web, es decir, es un conjunto de bibliotecas y programas que permiten facilitar el desarrollo de un software, al tener componentes reusables y funcionalidades que siguen ciertos patrones de diseño y/o arquitectura. Este *framework* fue lanzado en el año 2005 por David Heinemeier Hansson, desarrollador de la empresa Basecamp, y al ser de código abierto (bajo la licencia *MIT License*) admite cambios y mejoras propuestas y desarrolladas por la comunidad. El lenguaje de programación usado por Rails y sus aplicaciones es Ruby, un lenguaje orientado a objetos desarrollado en los años 90 por Yukihiro “Matz” Matsumoto (Bächle y Kirchberg, 2007).

3.1.1. Características y ventajas

Las características más importantes de Rails, que lo llevarían a su gran “boom” de popularidad, son las siguientes:

- Inclusión de todas las herramientas necesarias para una aplicación web moderna. Es lo que algunos denominan informalmente *with batteries included*, o “pilas incluidas” en inglés. Este framework incluye por defecto un motor de base de datos, bibliotecas de *javascript* para programar eventos a nivel de vistas, bibliotecas para manejo de traducciones y localización, un servidor web para desplegar la aplicación, scripts para la mantención de dependencias, etc.

- Arquitectura basada en Modelo Vista Controlador (MVC, Model View Controller en inglés), con tal de poder separar las responsabilidades de varias partes del código y las bibliotecas necesarias para su funcionamiento. Esto ayuda a la “modularidad” del código, y si bien hoy en día muchos *frameworks* de desarrollo web funcionan bajo MVC, antes de Rails no era así.

- Uso extensivo de patrones de diseño y principios estrictos. Ruby on Rails asume varios patrones, que en varias ocasiones son implementados como bibliotecas (como es el caso de ActiveRecord, un patrón para traducir registros de una base de datos relacional a objetos) y se invita al desarrollador a seguir éstos con tal de poder desarrollar la aplicación web de forma más rápida y con menos contratiempos.

- Desarrollo bajo los principios del desarrollo ágil. Rails incluye muchas funcionalidades pensadas en una metodología de desarrollo ágil. Por ejemplo, el uso de “migraciones” (scripts para manejar cambios en el esquema de una base de datos) con tal de adaptar el sistema a cambios o nuevas funcionalidades. También incluye soporte para aplicar Desarrollo Guiado por Pruebas (TDD, *Test Driven Development*) al generar código para escribir pruebas, e incluir *scripts* para ejecutarlas (Bächle y Kirchberg, 2007).

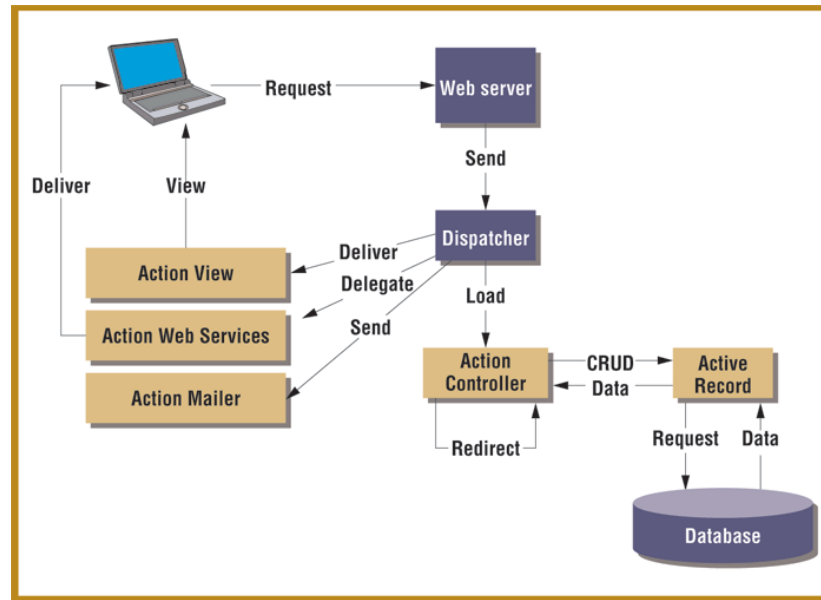


Figura 3.1 – *Componentes de Ruby on Rails, mostrando en forma resumida como los componentes manejan las peticiones de los navegadores web. Fuente: Bächle, M., & Kirchberg, P. (2007). Ruby on rails. IEEE software, 24(6) (Bächle y Kirchberg, 2007).*

Este *framework* también ha sido muy popular en el ámbito educacional, ya que permite el desarrollo rápido, permitiendo que los estudiantes puedan ver los resultados de su trabajo en poco tiempo, y que puedan acostumbrarse a aprender conceptos y patrones de diseño usados por aplicaciones web. Es así como en la Universidad Técnica Federico Santa María, en su carrera de pregrado de Ingeniería Civil Informática, incluye el desarrollo con Rails en el laboratorio de la asignatura de Bases de Datos. Y también es importante destacar que muchos equipos de trabajo escogen Rails como su *framework* de preferencia para proyectos de otras asignaturas, incluyendo el Taller de Desarrollo de Software.

Si bien en el mundo del emprendimiento Ruby on Rails ha sido desplazado por *frameworks* basados en el idioma Javascript y el *runtime* Node.js, no se puede negar que ha dejado un legado importante en la historia del desarrollo de aplicaciones web. Y con una comunidad aún muy activa, Rails es considerado como un *framework* maduro y confiable por miles de desarrolladores alrededor del mundo, y ha sido la elección para el sistema Intranet del Departamento de Informática (DI) de la Universidad Técnica Federico Santa María.

3.2. Calidad de software

Para poder juzgar y decidir sobre los cambios a realizar en la metodología de trabajo del proyecto Intranet, es necesario saber cómo se define la calidad del software: definiciones, estándares y elementos que lo componen, a partir de la literatura y del conocimiento de la industria.

3.2.1. Qué es calidad: dos perspectivas diferentes

Una de las definiciones de la palabra "calidad" dada por el diccionario de la Real Academia Española (RAE) es "Adecuación de un producto o servicio a las características especificadas"(Real Academia Española, 2014).

Esta es una de las definiciones que podemos tomar en cuenta a la hora de definir calidad de software, desde la perspectiva de los requerimientos. Si un producto de software cumple los requerimientos del proyecto, se puede decir que es de calidad.

Sin embargo, esta definición es vista desde la perspectiva del equipo de desarrollo, el cual medirá el cumplimiento de los requerimientos sin tomar en cuenta otros elementos importantes que irán a ser tomados en cuenta por el cliente.

Por ello, es que se tiene otra definición de calidad, también dada por la RAE, que es "Propiedad o conjunto de propiedades inherentes a algo, que permiten juzgar su valor" (Real Academia Española, 2014). Desde la perspectiva del cliente, existen varios elementos que juzgarán el valor del producto de software una vez que se encuentra en su poder: el cumplimiento de lo que éste espera, la facilidad de uso, los tiempos de respuesta, la falta de defectos, etc. Un software puede cumplir los requerimientos establecidos por el equipo de desarrollo, pero si al cliente no le sirve, lo juzgará como uno de "mala calidad".

Con tal de poder asegurar la calidad de un software, debemos tomar en cuenta ambas perspectivas, y proponer tareas que atienden ambas visiones.

3.2.2. Aseguramiento de la calidad del software (SQA)

También llamada SQA (por su sigla en inglés, *Software Quality Assurance*), la literatura la define como "el conjunto sistemático de actividades que provee evidencia de la idoneidad de uso del producto total de software". Esto va a requerir el uso de directrices y criterios de control de calidad, junto con instrucciones que se deben seguir para asegurar la calidad del producto, las cuales estarán en un documento conocido como el "Plan de calidad de software", que será parte de la documentación de éste (Lewis,

2000).

En general, cada organización va a definir su propia manera de implementar un SQA según sus necesidades, lo cual puede ir desde el uso de buenas prácticas y actividades, hasta el cumplimiento de estándares y certificaciones. En general, van a existir varias actividades comunes que se utilizan para el aseguramiento de calidad del software, las cuales se detallan a continuación.

Pruebas de software o *testing*

Se trata de verificar el cumplimiento de los requerimientos funcionales del producto. Es una de las actividades más importantes, ya que constituyen una estrategia de gestión de riesgos: podemos evitar que defectos del software y problemas imprevistos lleguen a ocurrir luego de la entrega al cliente. Bajo las dos perspectivas de calidad, el *testing* puede clasificarse en dos grandes categorías:

Verificación: Probar que el producto cumple con los requerimientos establecidos a través del proceso de desarrollo. Busca que el software realice todas las funciones planificadas, y que el software no realice funciones que puedan afectar o poner en riesgo la integridad del sistema. Este es un proceso de "mejora continua" y debe ser realizado a través de todo el ciclo de vida del producto, lo cual aumenta el costo inicial del *testing*, pero a largo plazo llega a ser una buena inversión por todos los defectos de software que pueden ser encontrados a tiempo.

Validación: Probar que el producto cumple con los requerimientos del cliente al final del ciclo de desarrollo. Los procesos de *testing* de validación, en general, son más fáciles de realizar, pero se debe tomar en cuenta el hecho de que encontrar y arreglar problemas es mucho más costoso a medida que se avanza en el ciclo de desarrollo (Lewis, 2000).

Hoy en día las tendencias establecen que se deben tener ambos tipos de *testing* en el proyecto, e intentar automatizarlos lo más posible. Más adelante se verán los tipos de pruebas más conocidos, y se elegirán los más adecuados para el proyecto de este trabajo.

Control de calidad

Son los procesos que se usan para monitorear el desarrollo y revisar si los requerimientos se están cumpliendo. Se trata de inspecciones donde se realizan *reviews* o revisiones del trabajo realizado, y la eliminación de defectos de software antes de la entrega del producto.

El control de calidad puede ser efectuado por un equipo especializado, o por los mismos desarrolladores, y en general se realizan al final de un hito del ciclo de desarrollo. Se

revisa el código del programa, la documentación, los resultados de las pruebas o *tests* y otros documentos especificados en el Plan de Calidad de Software.

El rol del control de calidad es encontrar y corregir defectos en el software, mientras que el aseguramiento de calidad busca prevenir que los defectos aparezcan en primer lugar. El control de calidad puede ayudar a encontrar defectos en los procesos de SQA, ya que en teoría no deberían haberse encontrado errores (Lewis, 2000).

Gestión de configuración de software (SCM)

También llamado SCM (por su sigla en inglés, *Software Configuration Management*) trata sobre la clasificación, seguimiento y control de los cambios en los elementos de software de un sistema. Se basa en controlar la "evolución" del sistema al manejar las versiones de sus componentes de software a través del ciclo de vida de éste. Es muy útil para organizar el proceso de desarrollo, y evitar defectos o errores debido a cambios repentinos o "de último minuto".

El SCM incluye la documentación del proyecto (planificación, análisis y diseño), código fuente, código ejecutable, bibliotecas externas, planes de pruebas, *scripts* de pruebas, *scripts* de despliegue, y reportes. Los procesos de SCM se pueden dividir en cuatro:

1. **Identificación de componentes**: identificar y nombrar la "línea base" de un componente, sus configuraciones y sus cambios. Esto incluye el "versionado" de los componentes (que en general utiliza números, cómo 1.0 o 1.1.1).
2. **Control de versiones**: organizar e identificar cambios en los componentes de software a medida que avanza el proyecto, indicando un historial de qué, cuándo y por qué se hicieron los cambios. Permite la trazabilidad del proyecto, el uso de "copias de seguridad" para el código, y es esencial para el trabajo colaborativo.
3. **Construcción de configuraciones**: identificar y construir/instalar las versiones correctas de los componentes del software, o las dependencias de éste. Esto puede ser manejado de varias maneras, desde el uso de una lista con los componentes y sus versiones, hasta la construcción automática y puesta a prueba de nuevas versiones.
4. **Control de cambios**: proposición, evaluación, programación de fechas, y seguimiento de las modificaciones de los componentes del software. Los cambios en el software deben ser aprobados y planificados, y se debe evaluar las consecuencias de estos cambios, los cuales generalmente producen cambios en otros componentes. La idea es tener un control sobre los cambios de estado del proyecto (Lewis, 2000).

El uso de SCM es estándar en la industria del software a gran escala o *enterprise*, desde que la trazabilidad del software y nuevas versiones dejó de significar la creación de medios físicos (como discos). Lamentablemente, este control de los cambios del software no es utilizada por estudiantes de pregrado que entran a proyectos como Intranet, donde en general solo se confía en el uso limitado de VCS (Version Control System, o sistemas de control de versiones) sin el uso de políticas o flujos de trabajo, y sin dar importancia a los cambios en ambientes de producción. Un proyecto serio podría no necesitar un software especializado para SCM, pero sí una metodología de trabajo que tome en cuenta los cambios en dependencias, componentes y configuraciones del sistema, y documente las tareas que ésto implica.

3.3. Plan de Calidad de Software

Un plan de calidad de software o *SQA Plan* será un documento con las directrices que nos permitirán evaluar si el desarrollo del software cumple con los niveles de calidad esperados, junto con las actividades que se deben seguir para llegar a estos niveles, y qué hacer cuando no se llega a la calidad esperada. La calidad del software, en este caso, se basará en código entendible y mantenible a través del ciclo de vida del proyecto.

Una plantilla o esquema de ejemplo para un *SQA Plan* se entrega en la literatura (Lewis, 2000), y éste incluye las siguientes secciones:

1. Objetivos y alcances
2. Documentos del proyecto citados en este plan
3. Administración: Estructura organizacional esperada, actividades y responsabilidades
4. Documentación: Qué documentos debe tener el proyecto, y cómo los documentos deben mantener un estándar (junto con quienes deberán encargarse de asegurar esos estándares).
5. Estándares, prácticas, convenciones y métricas.
6. Revisiones e Inspecciones
7. Gestión de Configuración de Software (SCM)
8. Reporte de problemas y acciones a realizar
9. Herramientas, técnicas y metodologías
10. Control del código

11. Control de los medios y acceso a éstos
12. Control de software externo
13. Recolección, mantenimiento y retención de registros
14. Metodología de las pruebas o *testing*

El Plan de Calidad a realizar en este trabajo se basará en esta estructura, realizando cambios según las características del proyecto.

En general, estos planes de calidad son orientados a proyectos de gran envergadura, citando documentos de estándares de madurez organizacional (como las normas ISO y CMM), pero para proyectos pequeños se pueden adaptar para ser válidos por sí mismos.

Finalmente se debe mencionar que un *SQA Plan* debe ser realizado, revisado y aceptado por todas las partes de la organización involucradas, incluyendo la administración y los desarrolladores, y se debe planificar su implementación según los costos y el tiempo requerido.

Un Plan de Calidad que implemente medidas modernas de calidad debe incluir algún tipo de *testing* automatizado. A continuación se hará una reseña sobre los distintos tipos de pruebas automatizadas, y algunas de las prácticas más utilizadas en la industria que se sostienen en el uso de tales pruebas, con tal de poder lograr un aseguramiento de la calidad del software en forma continua a través del desarrollo del proyecto.

3.4. Pruebas de software automatizadas

Un elemento esencial para demostrar que un software cumple los requerimientos del proyecto, y que no existan errores o problemas que impidan su funcionamiento correcto, es probar el software antes de su despliegue. El caso más simple de pruebas de software es la realización de pruebas manuales mediante un equipo especializado en *Software Quality Assurance* (SQA), y si bien aún es bueno que exista esta práctica, la complejidad de los sistemas actuales llevan a que estas pruebas sean muy costosas, sean propensas a error humano y a no cubrir todos los casos de uso posibles (Lewis, 2000). Ante esto, se han preferido las pruebas automatizadas.

Existen muchos lenguajes de programación y *frameworks* que permiten escribir y ejecutar pruebas, las cuales se basan en probar que al ingresar ciertos parámetros de entrada en una parte del software, éste entregue una salida esperada y consistente. Los tipos de pruebas más utilizados se mencionarán a continuación.

3.4.1. Tipos de pruebas automatizadas

Pruebas unitarias, o *Unit Testing*

Las pruebas unitarias corresponden a pruebas de verificación y buscan probar secciones pequeñas de código (unidades), y en general analizan que la salida de este código bajo ciertos parámetros de entrada sea lo esperado. Son un tipo de pruebas de estilo "caja blanca", donde se debe conocer el funcionamiento de lo que se desea probar, y busca que se puedan hacer pruebas sobre las unidades más pequeñas posibles. Así es como por ejemplo, bajo la programación orientada a objetos, se pueden preparar pruebas para cada uno de los métodos de una clase.

Las pruebas unitarias deben estar hechas para probar solo el código afín y evitar el acoplamiento con otro código, ya que para funcionalidades que requieran acoplamiento pueden haber comportamientos no esperados, y para ello deben hacerse otro tipo de pruebas. Además, cada prueba unitaria debe abarcar un contexto específico, con tal de poder identificar rápidamente qué es lo que no está cumpliendo el código a probar, y poder corregirlo para que pase la prueba (Martin, 2002).

Las pruebas unitarias van a ser la base de la práctica de "Desarrollo guiado por pruebas" o *Test Driven Development*, que se verá más adelante.

Pruebas de aceptación

Las pruebas de aceptación, a veces llamados simplemente pruebas de sistema, son pruebas de tipo "caja negra", o sea, donde no se deben conocer o modificar los mecanismos internos del software. La idea es que estas pruebas abarquen el sistema completo al poder probar los requerimientos del sistema y los casos de uso asociados a cada funcionalidad. Por tanto, son pruebas de validación.

Esto equivale a lo que son las pruebas "manuales" que un equipo de SQA realiza a un sistema funcional, pero lo recomendado es no solo automatizarlas, sino que diseñar el sistema de tal forma de que las pruebas de aceptación no estén acopladas a cambios en los mecanismos internos del software. Por ejemplo, se debe evitar que las pruebas sean acopladas a una interfaz gráfica que puede cambiar constantemente. De esta forma, las pruebas de aceptación podrían ser realizadas por equipos de trabajo externos, que sean más cercanos al cliente y sus necesidades.

Finalmente, se debe mencionar que un buen uso de pruebas de aceptación puede servir como documentación de los casos de uso del sistema: varias metodologías de desarrollo de software proponen crearlas en forma de narración antes del desarrollo (como una forma de capturar requerimientos del cliente), para luego ser traspasadas a código de pruebas. Luego, todo este código de pruebas de aceptación debe ser acumulado a través

del desarrollo del proyecto, con tal de que en todo momento se pueda ejecutar y probar que nuevo código no interfiera con alguna funcionalidad ya implementada (como una forma de evitar regresiones) (Martin, 2002).

Pruebas de regresión

Estas pruebas buscan asegurar que las funcionalidades del software que cumplían su cometido anteriormente (por ejemplo, en una iteración anterior de desarrollo) sigan funcionando, y que los nuevos cambios no produzcan consecuencias imprevistas en estas funcionalidades existentes.

En general, las pruebas de regresión no son pruebas nuevas, sino que se trata de integrar la idea de evitar regresiones en la metodología de *testing*. Por ejemplo, si a lo largo del desarrollo del proyecto se va guardando un banco de pruebas con todas las funcionalidades del sistema, es posible utilizar estas mismas como pruebas de regresión. Si alguna funcionalidad anterior pasaba todas las pruebas, pero luego de ciertos cambios deja de ser así, se puede identificar el error de regresión y arreglarlo.

Pruebas de integración

Estas pruebas buscan probar que ciertos subprocesos del software que utilizan varias unidades de código o software interconectado funcionen como un conjunto, y que no existan conflictos entre partes acopladas en el código. Su uso principal es en funcionalidades que utilizan sistemas distribuidos o arquitecturas cliente/servidor. En general son más costosas que las pruebas unitarias, y existen varias formas de realizarlas: con pruebas de tipo *Bing-Bang* donde se prueba el sistema completo, *Bottom-up* donde se van realizando pruebas desde funcionalidades más acotadas y de bajo nivel (más cercano a funciones y métodos) hasta las funcionalidades de alto nivel (más cercano a los casos de uso). *Top-down* que hace el proceso inverso (Lewis, 2000).

Pruebas estáticas

Finalmente, se deben mencionar las pruebas estáticas, las cuales no dependen de la ejecución del software y que en general, son inspecciones del código o de la configuración. Por ejemplo, se puede analizar el código para encontrar posibles malas prácticas, uso de funciones inseguras, código no utilizado, y faltas a estándares de codificación.

3.4.2. Formas de automatizar las pruebas

A través de la historia del desarrollo de software, la forma de automatizar las pruebas de software ha cambiado. Las primeras herramientas capturaban la entrada del usuario en la interfaz gráfica, para luego poder repetir éstas en forma automática y asegurar que el software tenga la salida esperada. En sus inicios, estas herramientas requerían el ingreso manual de valores de entrada. Sin embargo, esto es muy costoso si es que se quieren revisar distintos casos para cada prueba.

Esto conlleva a la aparición de herramientas de captura/repetición basadas en lenguajes de *scripting*, donde personas que tengan el conocimiento técnico pueden programar distintas pruebas, las cuales simulan los casos de uso del sistema (como pruebas de aceptación), o ir a niveles más bajos y probar los módulos y funciones del código (como pruebas unitarias). Datos de entrada pueden ser definidos en archivos de texto, o incluso generados en forma aleatoria.

Con el surgimiento de las metodologías ágiles (detalladas más adelante) y la aceptación del *testing* como herramienta esencial para el software de calidad, es que muchos *frameworks* y conjuntos de herramientas de desarrollo de software han integrado el uso de pruebas de una u otra forma. Múltiples lenguajes de programación soportan *testing* mediante bibliotecas especializadas, y los entornos de desarrollo integrado (también llamados IDEs, o *Integrated Development Enviroments*) buscan facilitar las pruebas al agregarlas a la interfaz de desarrollo.

3.4.3. Herramientas para su implementación en aplicaciones RoR

Las herramientas necesarias para poder desarrollar y ejecutar pruebas para un proyecto de software van a variar según la "profundidad" de lo que se quiere probar: pruebas para código de "bajo nivel" van a requerir herramientas específicas para el lenguaje de programación elegido para el proyecto, mientras que las implementaciones de requerimientos y el código de "alto nivel" no requiere herramientas tan específicas, si no que se acerca más a simular interfaces de usuario. A continuación se hará una reseña y comparación entre las herramientas de *testing* compatibles con aplicaciones hechas con Ruby on Rails (RoR).

Bibliotecas de *testing*

Como estándares de la industria en bibliotecas para pruebas en sistemas Ruby on Rails, se tienen tres opciones:

- **Test::Unit**, la biblioteca de pruebas unitarias por defecto de Ruby. Se puede utilizar para seguir prácticas modernas como TDD (*Test Driven Development*, la cual se definirá más adelante). No incluye un DSL (*Domain Specific Language*, o Lenguaje de Dominio Específico). Busca ser simple, de poco costo computacional, y para funcionalidad adicional requiere el uso de otras bibliotecas. En general es usada en proyectos pequeños.
- **RSpec**, la biblioteca de pruebas preferida para la práctica de BDD (*Behavior Driven Development*). Tiene las mismas funcionalidades de Test::Unit, y se le agrega: el uso de un DSL, uso de *mocks* y *stubs* (entidades y funciones simuladas), e integración directa con la biblioteca Capybara para pruebas de aceptación con un navegador. Busca ser suficientemente completo para todo tipo de proyectos, pero el costo computacional de la ejecución de las pruebas es mayor.
- **Minitest**, una biblioteca que intenta extender las funcionalidades de Test::Unit para agregar nuevas funcionalidades, como BDD, *mocking* y *benchmarking* (pruebas de tiempo y costo computacional de las mismas pruebas). Así, requiere menor uso de bibliotecas extras respecto a Test::Unit, pero no busca utilizar un DSL como RSpec. Si bien es la biblioteca que utilizan los proyectos Rails por defecto, no es tan popular como Rspec.

Según artículos y anécdotas de desarrolladores alrededor del mundo, RSpec es en general más utilizado en la industria, se adapta mejor a proyectos grandes y permite una lectura más natural (en lenguaje inglés), pero en general, no existen posibles pérdidas de funcionalidad al escoger una biblioteca por sobre otra, ya que existen otras bibliotecas adicionales que pueden equiparar las funcionalidades, por lo que se debe decidir su uso según la preferencia de usar un DSL o no, y la madurez del desarrollo y uso de la biblioteca (Múltiples autores, 2017) (Bradburne, 2016).

Bibliotecas de *mocking* y de captura/repetición

Los *mocks* son entidades simuladas de uso expedito y desechable, las cuales se comportan igual a las entidades reales a nivel de pruebas, pero sin el *overhead* de memoria y tiempo de procesamiento de éstos últimos (los que, por ejemplo, muchas veces deben pasar por una base de datos). Por defecto, los proyectos de Rails permiten hacer *mocking* mediante entidades en archivos YML llamados *fixtures*. Sin embargo, el uso masivo de Rspec y las limitaciones de los *fixtures* han llevado a que una biblioteca alternativa llamada FactoryBot (antes llamada FactoryGirl) sea el estándar de la industria para la creación de estas entidades. Vale mencionar que el uso de *mocking* se aplica en general a las pruebas de bajo nivel (como las pruebas unitarias), y en pruebas de más alto nivel se prefiere usar datos reales, en bases de datos de prueba.

Además, existe un caso muy común en el desarrollo de pruebas para aplicaciones web: el

requerimiento de probar conexiones con sistemas externos mediante APIs (*Application Programming Interfaces*). Se da el problema de que la comunicación a sistemas externos no acepta enviar y recibir peticiones de prueba en todo momento: por ejemplo, ciertas APIs cobran por cada petición, o el uso de una petición va a implicar reglas de negocio que no permitirán hacer esta petición varias veces con la misma respuesta. Esto evita que las pruebas puedan ser ejecutadas en cualquier momento y que siempre respondan de la misma forma, por lo que existen dos alternativas:

- Utilizar una biblioteca o utilidad de captura/repetición, la cual guarda las peticiones y respuestas a sistemas externos que se efectúan en una prueba, y luego las reproducen cada vez que ésta se vuelva a ejecutar. Se tiene la ventaja de contar con una respuesta 100 % real, pero existe la desventaja de que no hay control en caso de que hayan cambios en la API del sistema externo, o si exista una petición inesperada y no capturada anteriormente. Para proyectos de Rails, se utiliza la biblioteca VCR, que captura las peticiones y respuestas en archivos YAML.
- Simular las respuestas del sistema externo como funciones, lo que es conocido como *stubbing*, de forma parecida a como se construyen las entidades simuladas (*mocks*) de las pruebas unitarias. Su ventaja es que se tiene un control total de las respuestas, pero como desventaja existe un mayor esfuerzo por tener que escribir respuestas para cada tipo de petición. Para proyectos de Rails, se utiliza la biblioteca WebMock.

Cobertura de pruebas

Una métrica utilizada para medir la eficacia de las pruebas automatizadas es la cobertura o "coverage". Existen varios tipos, pero el más utilizado es el *statement coverage* o cobertura de sentencias, el cual es el porcentaje de las líneas de código del proyecto que son ejecutadas por la *suite* de todas las pruebas de un proyecto. Si bien una cobertura del 100 % no implica que el software esté libre de defectos (debido a lo complejo de éste), al menos se puede tener la certeza de que se escribieron pruebas suficiente exhaustivas, y que no se añadió código nuevo al proyecto sin incluir pruebas que lo acompañen. En lenguajes de tipificado débil, como Ruby, pruebas exhaustivas con alta cobertura ayudan a saber que, como mínimo, no existen sentencias mal escritas en el código.

El porcentaje de cobertura que se va a considerar satisfactorio va a depender del proyecto y sus requerimientos. En sistemas críticos, una cobertura del 100 % será necesario, pero en otros proyectos se podría aceptar desde un 70 % a 90 % como mínimo, mientras se pueda evitar bajar el porcentaje al agregar nuevos cambios, y mientras haya una razón estratégica para no estar en el 100 %.

En proyectos Ruby se utiliza la biblioteca *simplecov*, la cual utiliza funciones de *coverage* dadas por la biblioteca estándar del lenguaje, y es compatible con todas las bibliotecas

principales de *testing.*

Herramientas para las pruebas de alto nivel

En general, las pruebas de aceptación de aplicaciones web utilizan, a nivel de software, un navegador web "*headless*" o "sin cabeza", el cual será capaz de realizar peticiones web y simular un navegador web moderno, incluyendo la ejecución de Javascript en el cliente. Esto permitirá que las pruebas automatizadas de aceptación puedan ser ejecutadas en un servidor sin interfaz gráfica. Las alternativas de navegación *headless* para proyectos en Rails son:

- **Navegador web convencional** como Firefox o Chrome (mediante Selenium Webdriver 2.0): Esta es la opción que entrega la mayor cercanía con una simulación real de navegación, ya que se utiliza un navegador actualizado y de uso real, al usar éstos en modo "*headless*". El problema es que la instalación de Selenium Webdriver y de uno de estos navegadores va a requerir muchas dependencias en el servidor, y el rendimiento no va a ser el mejor.
- **WebKit** (mediante QtWebKit): Esta opción utiliza un navegador WebKit integrado en las bibliotecas Qt, y si bien éste no tiene todas las funcionalidades de un navegador como Firefox o Chrome, es más que suficiente para realizar pruebas de navegación con Javascript, y su instalación es simple, con pocas dependencias.
- **PhantomJS** (mediante el *plugin* Poltergeist): Este es un *fork* de WebKit (o sea, una nueva versión desarrollada en forma aparte) hecha específicamente para ambientes CI y desarrollada por una comunidad de código abierto. Lamentablemente, el 3 de Marzo de 2018 se anunció que el proyecto no continuará su desarrollo debido a una falta de contribuyentes, por lo que no se recomienda su adopción en nuevos proyectos.

Para poder ejecutar pruebas de alto nivel con ejecución de Javascript, a final de cuentas se tendrá que utilizar un navegador web. Y no solo será necesario tomar en cuenta la instalación y las dependencias requeridas por éstos, si no también el costo que añaden a la ejecución de las pruebas: uso mayor de CPU, RAM y mayor tiempo de ejecución. Un costo excesivo hará que las pruebas se tarden demasiado en ejecutarse, lo que le quita el sentido a que el banco de pruebas o la *suite* pueda ejecutarse en cualquier momento.

3.4.4. Resumen comparativo de los tipos de pruebas

En un proyecto como Intranet, donde ya existe funcionalidad anterior que no es cubierta por pruebas automatizadas, se debe analizar qué tipos de pruebas formarán la *suite* de

pruebas permanente para el proyecto, entre las pruebas unitarias, pruebas de aceptación y pruebas de integración. Su alcance y requerimientos se detallan en la siguiente tabla.

	Pruebas unitarias	**Pruebas de aceptación**	**Pruebas de integración**
Acciones a realizar	Ejecutar funciones y código del sistema con ciertos parámetros de entrada	Simular casos de uso a nivel de interacciones en la interfaz	Simular funcionalidades a nivel de peticiones y respuestas del sistema
Cobertura	Cada elemento del sistema por sí solo (funciones, modelos, métodos, controladores, etc.)	Los requerimientos del proyecto (dados por el cliente)	Varios elementos del sistema que actúan en conjunto
Métodos preferidos de implementación	Frameworks y bibliotecas de pruebas unitarias.	Sistemas de captura-repetición, simulación de la interfaz.	Frameworks y bibliotecas de pruebas unitarias, con soporte de simulación de peticiones.
Métodos preferidos de implementación para aplicaciones Ruby on Rails	Bibliotecas RSpec (BDD) o Minitest (TDD) con herramientas de *mocking* como FactoryBot	Bibliotecas RSpec (BDD) o Minitest (TDD) con herramientas: Capybara y algún navegador web *headless*.	Bibliotecas RSpec (BDD) o Minitest (TDD) con herramientas de *mocking* como FactoryBot y captura/repetición como VCR
Costo de desarrollo	Bajo-Medio, según elemento a probar	Medio	Medio
Costo de ejecución	Bajo	Alto	Medio

Tabla 3.1 *– Comparativa de tipos de pruebas de software*

Finalmente, dado un proyecto RoR ya existente, se deben tomar en cuenta qué tipos de pruebas se utilizarán a la hora de implementar una *suite* que cubra todas las funcionalidades existentes. Se debe hacer un buen balance entre el costo de desarrollo (gasto de tiempo excesivo en escribir pruebas) y una cobertura que asegure la calidad del sistema.

3.5. Prácticas modernas en el desarrollo de software

Los distintos *plugins*, bibliotecas y frameworks de *testing* han adaptado el uso de distintas metodologías y prácticas, las cuales han permitido integrar las pruebas como un elemento esencial en el aseguramiento de la calidad del software. A continuación se presentan las prácticas más conocidas para este fin.

3.5.1. Desarrollo guiado por pruebas (TDD)

El Desarrollo Guiado por Pruebas (o TDD, *Test Driven Development*) es una de las prácticas más conocidas e importantes que han surgido en las últimas décadas. Se refiere a que antes de programar una función o método, se debe escribir una prueba unitaria sobre lo que se espera de ésta. Inicialmente la prueba va a ser reprobada, debido a que aún no existe un código de la función que cumpla lo escrito. Luego, se busca que al programar esta funcionalidad, se logre pasar esta prueba.

Existen varios beneficios en esta práctica. La más obvia es que a medida que se va desarrollando el sistema, se van acumulando muchas pruebas para cada función. Esto permite que todo el código del sistema sea "auto probable" o *self-testable* en todo momento, y se podrá automatizar la prueba de gran parte de código (logrando una amplia "cobertura" en las pruebas). Además, esto ayuda a que se descubran problemas con código acoplado (dependiente de otro) en el mismo momento en que se desarrolla la función.

Otro beneficio no tan obvio es que el desarrollador va a poder tener claro los objetivos de la función a programar desde antes, y va a forzar a que la función pueda interactuar con todo tipo de pruebas fácilmente (Lewis, 2000).

3.5.2. Desarrollo guiado por comportamiento (BDD)

El Desarrollo Guiado por Comportamiento (o BDD, *Behavior Driven Development*) expande lo indicado por TDD para ajustarse mejor a los requerimientos de negocio y a lograr una mejor claridad e interpretación en las pruebas.

Esta práctica indica que las pruebas a escribir antes del desarrollo deben especificar el comportamiento de la unidad a probar, utilizando un lenguaje específico del dominio (DSL, *Domain Specific Language*) que sea similar al lenguaje natural y que pueda ser entendido por todo el equipo de desarrollo. En el caso de utilizar esta práctica en pruebas unitarias, se busca que las pruebas utilicen un lenguaje de tipo "*if* X *then* Y" (si X, entonces Y), o sea, que las pruebas reflejen las especificaciones de la unidad de manera funcional. Y en el caso de usar BDD con pruebas de aceptación, se busca que

las pruebas sean iguales o muy similares a las pruebas de aceptación que se tienen en las Historias de Usuario (evitando posibles problemas de interpretación de parte del equipo de pruebas y/o de desarrollo).

En general, se deben utilizar herramientas específicas (por ejemplo, bibliotecas para el lenguaje de programación o *framework* en uso) que permitan leer e interpretar el DSL utilizado en la escritura de las pruebas.

3.5.3. Integración Continua (CI)

Uno de los elementos mencionados anteriormente como beneficios del desarrollo guiado por pruebas (TDD) es que el código es “auto probable”. Sin embargo, las pruebas unitarias podrían no ser suficientes con tal de probar cómo funcionan varias partes del software a la vez. Para remediar esto, aparece la práctica de Integración Continua (o CI, *Continuous Integration*).

La integración continua implica que los miembros del equipo de trabajo van a trabajar y enviar sus cambios en el código a un mismo repositorio compartido (mediante un sistema de control de versiones). Si otros miembros han efectuado cambios, deberán efectuar un *merge* o unión de código y correr todas las pruebas automáticas con tal de asegurar que su trabajo no produce problemas con el código existente. Si existen problemas, se deben resolver en el momento, y solo una vez que los cambios produzcan una versión funcional del software, se aprobarán los cambios (Martin, 2002).

La idea es que, idealmente, se vayan añadiendo nuevos cambios varias veces al día, y que cada uno de esos cambios esté libre de errores y permitan compilar o construir una nueva versión del software. Así, los problemas de integración de software son arreglados rápidamente y en el momento, y se tiene siempre un software funcional. Por lo tanto, la integración continua va a requerir las siguientes herramientas:

- Sistema de control de versiones. Este va a ser un software que permite el trabajo colaborativo con código y documentos de texto plano, al poder ver cambios a través del tiempo, organizar cambios en “ramas” o *branches* y unir el trabajo de varios desarrolladores mediante *merges*. Algunos de los sistemas más famosos son Git, Subversion, Mercurial y Perforce Helix (este último siendo de pago).

- Código auto probable o *self-testable*. Como se mencionó anteriormente, se busca que las pruebas automáticas tengan la mayor cobertura posible, con tal de que en cada integración los desarrolladores puedan encontrar posibles errores de forma inmediata, y poder asegurar un mínimo estándar de calidad en los entregables del software.

- Construcción automática o *automated build*. Deben existir *scripts* o código que permita que, una vez que se han aprobado todas las pruebas automáticas, se

pueda compilar o construir una nueva versión funcional del software. Esto va a depender de las tecnologías usadas en el proyecto: si es un software ejecutable para un computador de escritorio, puede que sea necesario el enlace de bibliotecas y la compilación de este ejecutable. Si es una aplicación web, va a implicar un despliegue automático en un servidor (que puede ser un servidor de pruebas para el equipo de SQA).

Siguiendo esta práctica, los desarrolladores puedan unir cambios, ejecutar pruebas, resolver conflictos y generar entregables de software en forma ordenada y sin acumular defectos de software imprevistos (lo que es conocido como deuda técnica o *technical debt*), y así, se puede asegurar un estándar mínimo de calidad en cada cambio efectuado en el código (Fowler y Foemmel, 2012).

En la industria existen muchos programas de tipo CI/CD, los cuales integran Integración Continua y Entrega Continua (o CD, por *Continuous Deployment*), y éstos se pueden clasificar en dos tipos:

- CI/CD como SaaS (*Software as a service*), que son programas que funcionan en servidores remotos gestionados por los mismos desarrolladores de la herramienta, y en general mantienen un modelo de negocio basado en suscripciones, a veces teniendo también versiones gratuitas para proyectos de código abierto. Los más populares son Travis CI, Circle CI, Codeship, Semaphore CI, y las versiones SaaS de GitLab CI.
- CI/CD *self-hosted*, los cuales son programas de CI/CD que cualquier persona u organización puede instalar y desplegar en sus propios servidores. Muchas de éstas son gratuitas o tienen versiones gratuitas con restricciones, pero también existen algunas que requieren suscripciones. Los más populares son Jenkins, TeamCity y GitLab CI.

3.5.4. DevOps y tendencias actuales

Finalmente, es necesario mencionar lo que es DevOps. Éste es un concepto relativamente nuevo, el cual es un diminutivo en inglés de *Development Operations*, u Operaciones de Desarrollo, y se trata de unir el trabajo de la fase de desarrollo de un software, con la fase de despliegue y la fase de mantención (a veces llamada *operations* o *live operations*). Esto implica un cambio organizacional, donde se busca que los desarrolladores no ignoren los problemas de despliegue y mantención, y que los administradores de sistemas no ignoren los problemas del desarrollo. Esto se puede lograr con mejor comunicación, intercambio de conocimientos y el uso de herramientas de automatización (Ebert, Gallardo, Hernantes, y Serrano, 2016).

La industria está poco a poco adoptando nuevas tendencias en el desarrollo de software basadas en el uso de procesos de desarrollo definidos o *pipelines*, las cuales han hecho surgir la necesidad de tener profesionales adaptados a la idea de DevOps. Varias de estas tendencias son las mencionadas anteriormente, como el uso de ciclos cortos de desarrollo, integración continua y entrega continua, pero además se suman otras:

- El uso de arquitecturas de software basadas en **Microservicios**, donde se busca separar lo más posible las partes de un software en servicios más pequeños que se comunican entre sí y que sean más fáciles de desplegar y mantener a escala. Esto contrasta los proyectos de software de tipo Monolítico, donde todo el sistema está altamente acoplado y se despliega de una sola vez.
- El uso de **herramientas de orquestación** o configuración de sistemas distribuidos. Usando sistemas en Microservicios, éstos se pueden distribuir en múltiples máquinas y distintas localizaciones geográficas, con tal de poder escalar ante la llegada de muchas nuevas peticiones, o de cargas de trabajo pesadas. Ante ello, han aparecido herramientas SCM como Puppet, Chef o Ansible, y nuevas formas de desplegar aplicaciones distribuídas, como son las máquinas virtuales o VMs, los contenedores (como Docker) y herramientas de orquestación para éstas (como Kubernetes).
- El uso de **herramientas de monitoreo** y de **registro de eventos** o ***logging***. Se va más allá de dejar la tarea de monitoreo a los administradores de sistemas, y se busca un monitoreo integral que no solo incluye el estado de las máquinas de operaciones, sino también de los eventos del software y otra información relevante para los desarrolladores del proyecto (Ebert y cols., 2016).
- Finalmente, se debe mencionar el uso de **herramientas de comunicación**: en los últimos años se han vuelto populares los servicios de mensajería instantánea profesional, como son HipChat, Slack y Microsoft Teams. Éstos no solo permiten una comunicación sin límites geofráficos, sino que también se pueden conectar a otros servicios mediante APIs. Por ejemplo, se pueden configurar para que se notifique a todos cuando hayan cambios en un repositorio SCM, e incluso que se puedan programar tareas al hablar con "*chat bots*" (usuarios falsos del sistema de mensajería, que funcionan como programas de línea de comandos).

3.5.5. Herramientas para su implementación en aplicaciones RoR

Como una de las herramientas principales para el aseguramiento de calidad del proyecto, y como un complemento importante a las herramientas de pruebas, es necesario comparar los software de Integración Continua y Entrega Continua (CI/CD, por sus siglas en inglés). Éstos fueron hechos para atender la necesidad de equipos que desean

utilizar metodologías ágiles, pero su eficacia en el aseguramiento de calidad es tal, que se han convertido en un estándar para todo tipo de proyectos de software, sin importar su metodología (Meyer, 2014).

Servidores de Integración Continua

Dada las políticas y la forma de trabajo de los proyectos en I&T del Departamento de Informática, no se considerarán las herramientas de CI/CD tipo SaaS. Se busca que los servidores sean alojados y mantenidos en el mismo DI, y se desea evitar el pago de una subscripción. Por lo tanto, se comparan solo las herramientas CI/CD self-hosted.

Jenkins Esta es la herramienta de CI/CD más desplegada en el mundo (Ebert y cols., 2016), y es considerada de las más maduras. Jenkins comenzó como un *fork* o un nuevo desarrollo sobre la herramienta Hudson, que se utilizaba mucho en la década del 2000-2010 en equipos de desarrollo en Java.

Jenkins es gratuito, de codigo abierto, continuamente en desarrollo y tiene un ecosistema de *plugins* muy maduro, dando muchas posibilidades de configuración según el tipo de proyecto y las necesidades del equipo. Por ejemplo, permite conectarse a muchos sistemas de VCS usando *plugins*, y las ejecuciones de pruebas y *builds* puede ser de varias formas: en el mismo servidor de Jenkins, con servidores "réplica", con contenedores Docker, etc.

Vale decir que para algunos esto puede significar una gran curva de aprendizaje debido a la complejidad del sistema. Sin embargo, en las versiones más recientes, se han hecho varios avances en la facilidad de configuración y uso, como son: el lanzamiento de una versión de Jenkins como contenedor de Docker, el uso de archivos *Jenkinsfile* para configurar proyectos en forma declarativa y el *plugin* BlueOcean, que permite ver el estado de los *pipelines* de los proyectos asociados a Jenkins en una interfaz simple.

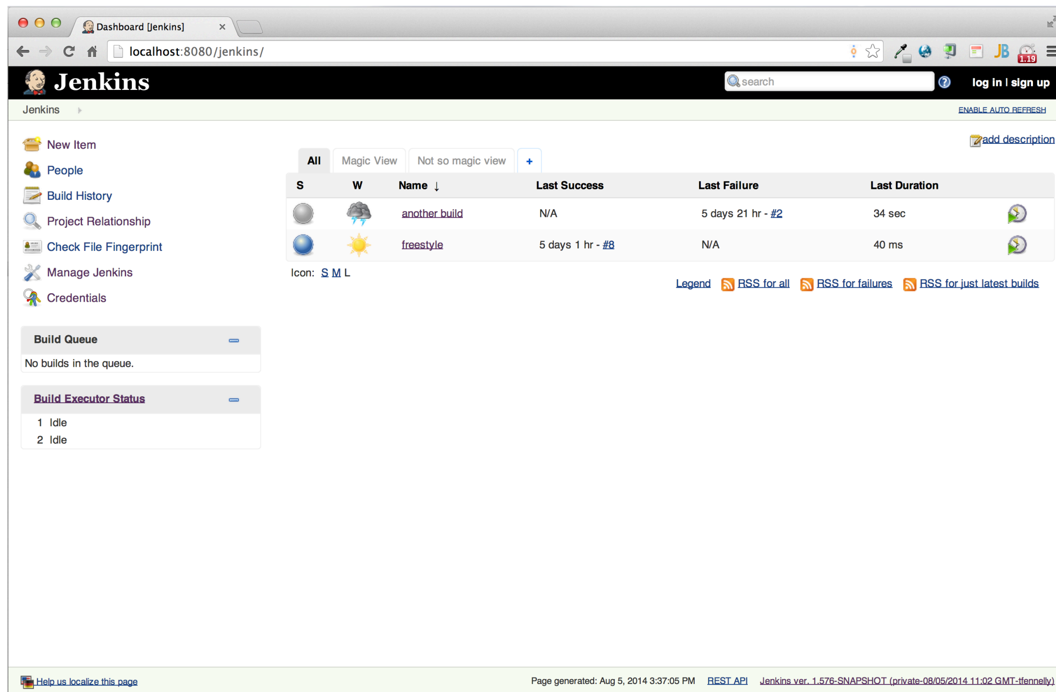


***Figura 3.2** – Interfaz de usuario convencional de Jenkins. Fuente: Blog de Jenkins.io*

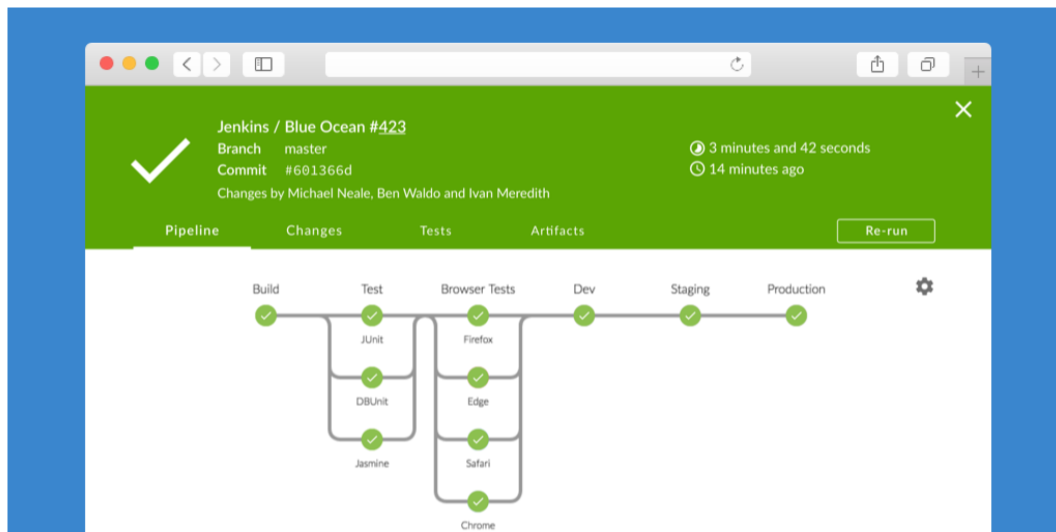


***Figura 3.3** – Arte conceptual del proyecto BlueOcean, plugin que busca dar mejor interfaz a Jenkins. Fuente: Blog de Jenkins.io*

TeamCity TeamCity es también un sistema de CI/CD gratuito, pero no de código abierto. Es mantenida por la empresa JetBrains (conocida por sus IDEs de pago IntelliJ Idea) y se destaca por ser una alternativa más simple pero casi tan potente como Jenkins. Sin embargo, es mucho más reciente y por ende, menos maduro.

El gran problema que presenta TeamCity, al igual que muchos otros programas gratuitos con código cerrado, es que el mantenimiento y el modelo de negocio queda sujeto a lo que dice su desarrollador (en este caso, JetBrains) y puede cambiar en cualquier momento. Ya ocurrió que desde las nuevas versiones, comenzando por las del segundo semestre de 2017, JetBrains ha añadido limitaciones a la versión gratuita de TeamCity, las cuales solo se pueden quitar al suscribirse a la versión de pago, o al utilizar una versión anterior sin soporte. Este tipo de modelo entrega desconfianza sobre la versión gratuita, y su uso a futuro por parte del DI (JetBrains, 2018).

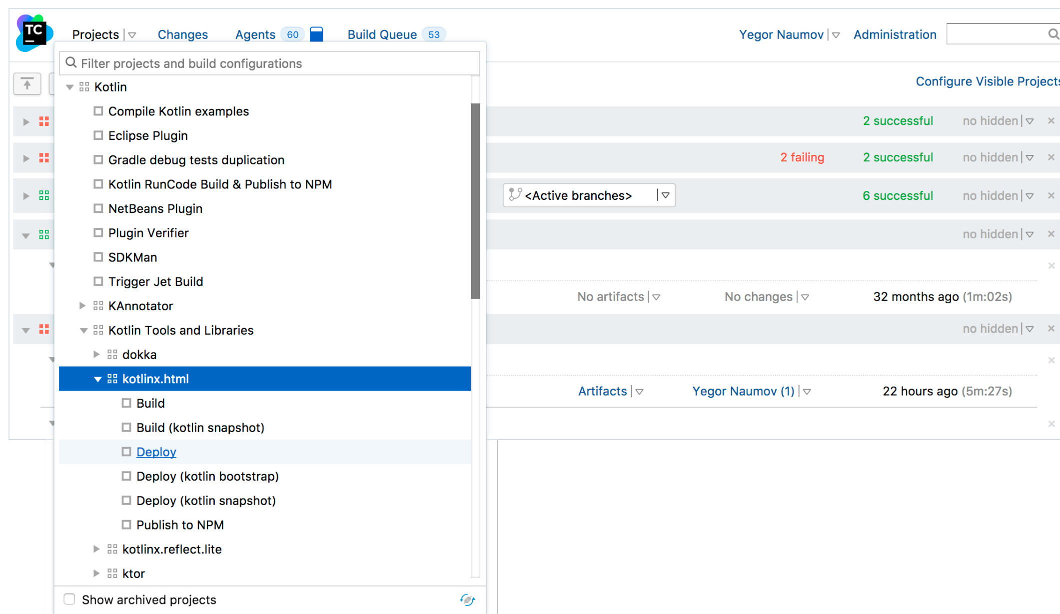


Figura 3.4 *– Arte promocional de TeamCity. Fuente: Sitio oficial de TeamCity en Jetbrains.com*

GitLab CI GitLab CI es una de las herramientas más nuevas e interesantes para CI/CD, la cual está ganando bastante uso en la industria. Este es un módulo del sistema de control de versiones GitLab, el cual es gratuito y de código abierto, y permite la activación de *pipelines* de CI/CD configurables con un archivo YAML (en forma parecida a los Jenkinsfile de Jenkins) los cuales se pueden ejecutar en agentes que tengan instalado GitLab Runner. Es una alternativa simple de usar y de configurar si es que ya se utiliza GitLab como repositorio VCS, y está siendo desarrollada continuamente. (GitLab, 2018a)

Los problemas principales de GitLab CI son el requerimiento de tener el código en GitLab (o de hacer un repositorio "espejo", lo cual solo está disponible en la versión de pago GitLab Enterprise), y el hecho de que este software intenta ser una solución SCM todo en uno: es principalmente un repositorio VCS, al cual se le han desarrollado módulos para CI, *Issue Tracking* y documentación, pero estos últimos no son siempre prioritarios para los desarrolladores de GitLab, y aún están sufriendo cambios

importantes en cada nueva versión. Finalmente, vale decir que existen limitaciones en la versión gratuita, pero que es posible acceder a GitLab Enterprise sin pago por parte de instituciones educacionales, lo cual es una opción para equipos de trabajo de la universidad. (GitLab, 2018b)

Figura 3.5 *– Arte promocional de GitLab. Fuente: Documentación de GitLab en docs.gitlab.com*

Resumen comparativo de las herramientas de CI/CD Con el objetivo de una comparación a nivel técnico, se presentan las diferencias entre los productos de CI/CD *self-hosted* en la siguiente tabla.

	Jenkins	**TeamCity**	**GitLab CI**
Funcionalidad y robustez	Muy robusto y configurable, tiene *plugins* para todo tipo de proyecto	Robusto, con un ecosistema de plugins emergente	Abarca los casos de uso comunes, pero no más allá
Trayectoria	Activamente en desarrollo por más de 10 años, uso masivo y gran aceptación por parte de la industria	Activamente en desarrollo por más de 10 años, medianamente popular	Relativamente nuevo (2012-), activamente en desarrollo, medianamente popular
Licencia y precio	Código abierto, gratuito	Código cerrado, gratuito con limitaciones	Código abierto, gratuito con limitaciones
Dificultad de configuración y uso	Alta	Media	Media
Compatibilidad con VCSs	Alta, mediante plugins y *webhooks*	Alta, mediante plugins y *webhooks*	Baja, requiere uso de repositorio en el GitLab mismo, o crear un repositorio "espejo" que copie del otro (no disponible en versión gratuita)
Agentes de construcción	Se puede usar el mismo servidor, o conectar a otros remotamente (SSH, agente instalado)	Se puede usar el mismo servidor, o conectar a otros remotamente (SSH, agente instalado). Limitación de 3 agentes en versión gratuita	Se puede usar el mismo servidor, o conectar a otros remotamente (SSH, agente instalado)
Configuración de proyectos	Interfaz del programa, Archivo Groovy (Jenkinsfile)	Interfaz del programa, Archivo XML o Kotlin	Interfaz del programa, Archivo YML (.gitlab-ci.yml)
Compatibilidad con proyectos RoR	Total	Total	Total, en versiones recientes

Tabla 3.2 *– Comparativa de herramientas CI/CD self-hosted*

En general, todas estas herramientas tienen funcionalidades y formas de trabajo similares, por lo que a la hora de tomar una decisión, otros factores deben ser tomados en cuenta. La falta de madurez y las limitaciones de sus versiones gratuitas pueden ser limitantes ante la búsqueda de tener un sistema CI/CD que pueda ser usado y mantenido para el futuro.

Herramientas de comunicación

Las tendencias de la industria han puesto pie en la utilización de herramientas de mensajería instantánea profesional para su uso en equipos de desarrollo, gracias a que permiten integraciones con otras herramientas de desarrollo como programas de CI/CD, y programas de planificación como Jira. El lanzamiento de HipChat y Slack han llevado a la popularidad de éstos en los últimos años, pero así también han aparecido detractores y conclusiones importantes respecto a su mal uso (LaFrance, 2002).

A partir de lo entregado por veteranos de la industria, el uso de correo electrónico siempre debe ser el canal de comunicación primario, gracias a su trazabilidad y la inmutabilidad de los mensajes, o sea, siempre existirá un registro de todo lo escrito. Y si bien la herramienta de mensajería instantánea **Slack** es la más popular, existen varias propiedades de éste que no le permiten ser un canal primario para el proyecto Intranet:

- Funciona como SaaS (Software as a service), con servidores alojados fuera del DI, y propensa a posibles cambios imprevistos al programa y al modelo de negocio por parte de los desarrolladores.
- Versión gratuita con limitaciones: luego de cierta cantidad de mensajes, los mensajes más antiguos se empiezan a eliminar.
- Al no ser *self-hosted* o de código abierto, no se adhiere a los principios de los servicios del DI.

Sin embargo, han aparecido alternativas *self-hosted* y de código abierto a Slack, pero que también tienen restricciones en sus versiones gratuitas (que no son tan limitantes como las de Slack). Las dos más conocidas son las siguientes:

- **Rocket.chat**, el cual comenzó su desarrollo como un módulo de chat para un sistema CMS. y que de a poco está ganando fama en la industria. Incluye aplicaciones móviles para Windows, Mac, Linux, Android y iOS, junto con la interfaz web. El único gran problema es su falta de madurez, con aplicaciones móviles incompletas, altos requerimientos de hardware en servidor y clientes, y una instalación más complicada. Vale mencionar que aún está en continuo desarrollo, y estas limitaciones podrían no ser válidas en el futuro.
- **Mattermost**, una alternativa más madura que Rocket.chat, es también popular. Permite el acceso a servidores self-hosted de éste mediante aplicaciones en varias plataformas móviles y de escritorio, al igual que Rocket.chat. La versión gratuita no incluye ciertas funcionalidades para el segmento *enterprise*, como acceso a SSO (Single Sign On) o acceso con cuentas LDAP, lo cual limita su uso por parte de

los usuarios generales del DI, pero que no será necesario para equipos pequeños de trabajo.

Para la elección de un sistema de mensajería instantánea, la madurez del proyecto y la confianza en las aplicaciones de cliente es esencial: se busca que el sistema no quede abandonado por el equipo de desarrollo, y que no haya problemas que desmotiven su uso (Nichols, 2018).

3.6. Desarrollo Ágil

La gran importancia del *testing* y de varias de las prácticas antes mencionadas son fruto de la adopción de nuevas metodologías de desarrollo de software, particularmente las metodologías de "Desarrollo Ágil". Poder integrar nuevas actividades en una metodología acorde al problema de esta memoria requiere hacer una revisión de las metodologías ágiles más importantes, y así poder comparar sus requerimientos, lo que logran y poder aplicar alguna (o una mezcla de varias) en el proyecto Intranet.

3.6.1. Historia y definición

El desarrollo ágil o "agilismo" aparece a fines de los años 90 e inicios de la década del 2000 como una alternativa a los modelos de desarrollo de software clásicos, basados en procesos lineales y planificación estricta (Martin, 2002). Un ejemplo común de una metodología clásica de desarrollo de software es el modelo *waterfall* o "en cascada", donde se sigue un proceso específico que consta de:

- Captura de requerimientos
- Análisis del sistema
- Diseño del sistema
- Programación del sistema
- Pruebas del sistema
- Despliegue del sistema

Si bien este modelo es bastante antiguo, aún es usado en muchos proyectos de software debido a que logra cumplir los objetivos básicos de la administración o *management* del desarrollo de software, efectuando una división del trabajo en equipos especializados

y permitiendo una administración de los tiempos y recursos del proyecto en forma bastante simple. Sin embargo, al tomar al desarrollo de software como un proceso lineal se ignoran los elementos que separan los proyectos de software de los proyectos en otros rubros. El desarrollo de software requiere un nivel alto de conocimientos técnicos, y la comunicación entre clientes y desarrolladores es difícil. En el proyecto pueden aparecer problemas que van a atrasar la planificación, y agregar más personal o más recursos no va a ayudar a que el proyecto avance más rápido. Y los cambios en requerimientos del cliente ocurren frecuentemente (más aún en proyectos que duran un año o más), implicando costos muy altos para un proceso de desarrollo tan rígido (Sharma y Hasteer, 2016).

A partir de esto es que aparecen varias metodologías de desarrollo que se basan en el desarrollo ágil, el cual se basa en cuatro valores, los cuales son:

- Individuos e interacciones por sobre procesos y herramientas. Se señala que es muy importante tener buenos desarrolladores que saben comunicarse y trabajar como un equipo. Y si bien tener herramientas de trabajo en equipo es importante, estas no pueden salvar un proyecto donde los individuos no pueden trabajar en pos de un objetivo común.

- Software funcional por sobre documentación extensiva. Es importante tener documentación en el desarrollo del software, pero es mejor tener poco a tener mucho. Demasiada documentación puede va ser muy costosa de mantener, y mantenerla sincronizada al código va a terminar retrasando las cosas. Si es posible, el mismo código debería ser la primera fuente de información. Una regla basada en este punto es "No producir documentación a menos que su necesidad sea inmediata y significativa".

- Colaboración con el cliente por sobre re-negociación de contratos. Este punto se relaciona con lo explicado anteriormente sobre cómo tratar proyectos de software. Éstos no pueden ser vistos con una planificación y presupuesto rígidos, y tampoco pueden tener éxito si el cliente no interactúa a lo largo del proceso. El contrato, si el caso lo amerita, debe tomar en cuenta los cambios que ocurrirán a lo largo del proyecto.

- Responder ante los cambios por sobre seguir un plan. Este es el punto más importante, donde se define la base del desarrollo ágil. La planificación del proyecto debe ser flexible y no debe extenderse a un futuro incierto. Las organizaciones y los clientes van a sufrir cambios, y estos cambios deben ser tomados en cuenta a lo largo del proyecto. Una planificación de tipo Gantt o PERT no es efectiva si los procesos granulares en el desarrollo de software van a ser cambiantes.

Así, se puede concluir que el desarrollo ágil busca como objetivo maximizar la satisfacción del cliente mediante procesos que se ajustan a cambios y períodos de tiempo

cortos, equipos de trabajo auto-organizados con buenas habilidades de comunicación (entre sí y con el cliente), y una motivación por entregar nuevas versiones funcionales de software en poco tiempo, con tal de obtener retroalimentación.

3.6.2. Metodologías de Desarrollo Ágil

A partir de estos valores y principios es que han aparecido varias metodologías de desarrollo de software. Dos de los más famosos y más importantes son Extreme Programming (XP) y Scrum, pero es necesario tomar en cuenta además a las metodologías Crystal y Kanban, las cuales también han influenciado a los procesos de desarrollo de software de la industria.

Extreme Programming (XP)

Esta metodología se basa en tomar varias de las buenas prácticas del desarrollo de software, como el desarrollo de pruebas, la comunicación con el cliente y las entregas frecuentes del producto, y las intenta llevar al extremo. XP favorece un desarrollo en oficinas de espacios abiertos, con una alta comunicación entre miembros del equipo, comunicación directa con el cliente (mediante un *customer* que lo representa), iteraciones de software de no más de dos semanas, programación de a pares (dos desarrolladores en una misma estación de trabajo, sirviendo como revisión continua), captura de requerimientos del cliente en una reunión llamada *planning game* o el juego de planificación (mediante "historias de usuario"), desarrollo guiado por pruebas automatizadas, integración continua y pruebas de aceptación.

Para maximizar la calidad de los entregables del software, también se habla de la práctica de refactorización o *refactoring*, la cual es reestructurar el sistema al eliminar duplicaciones, simplificar la lógica y añadir flexibilidad al código, con el objetivo de mejorar su mantenibilidad. En general XP busca cumplir los principios del desarrollo ágil construyendo un software lo más simple posible, al favorecer la comunicación del equipo de desarrollo y la programación de funcionalidades en poco tiempo, pero con el uso de pruebas y entregas continuas para validar el trabajo realizado (Martin, 2002).

Historias de usuario Las historias de usuario son una forma de capturar requerimientos de software, que busca ser una alternativa más simple a los casos de uso y a los documentos de especificaciones. En XP, las historias de usuario son escritas en conjunto con cliente, y se asume que su simplicidad puede ser complementada con la comunicación directa con éste en medio del desarrollo. En varias metodologías ágiles se propone su uso mediante tarjetas o papeles autoadhesivos, y van a servir también para estimar el esfuerzo y la duración de las iteraciones asignadas a éstas.

La estructura general de estas historias de usuario es:

- Número identificador y título
- Descripción, la cual lleva la forma “Como [tipo de usuario] necesito que [descripción rápida del requerimiento]”
- Estimación de esfuerzo (como número o categoría)
- Prioridad (como número o categoría)
- En el reverso del papel o en la parte de abajo, las Pruebas de Aceptación, con una narración corta de cada una. Aquí debe describirse que ocurre ante casos límite.

Scrum

La metodología Scrum se centra en organizar el equipo en lo que se llama un *Scrum Team* o Equipo Scrum, donde se tendrán varios roles y subdivisiones:

- *Product owner* o dueño del producto, que será el encargado de la comunicación con el cliente, la captura de requerimientos mediante *historias de usuario*, la división del trabajo y la formulación de los planes
- Scrum Master, encargado de liderar a los desarrolladores y preparar reuniones diarias
- Los desarrolladores, que serán divididos en equipos.

El proyecto será dividido en varios *sprints*, que serán iteraciones del proyecto de software donde los equipos podrán escoger tareas a desarrollar (las cuales serán elegidas por el *Product owner* según prioridades, en lo que se llama una Pila del Producto o *product backlog*). Existen reuniones diarias para saber cómo avanza el proyecto, y reuniones entre cada *sprint* con tal de obtener retroalimentación y definir tareas para la siguiente iteración. En estas reuniones se pueden tomar decisiones según los problemas o cambios que han aparecido, cambiando prioridades y tareas de la Pila del Producto (Sharma y Hasteer, 2016)

Scrum busca aplicar ideas basadas en la teoría de control de procesos industriales, y lograr que el equipo de trabajo sea suficientemente flexible como para atender cualquier cambio en el proceso de desarrollo. Se asume que éste es poco predecible y complejo, y que las constantes reuniones permitan afinar los procesos de desarrollo, y encontrar errores rápidamente (Abrahamsson, Salo, Ronkainen, y Warsta, 2017). Un resumen del proceso Scrum se puede ver en el diagrama a continuación.

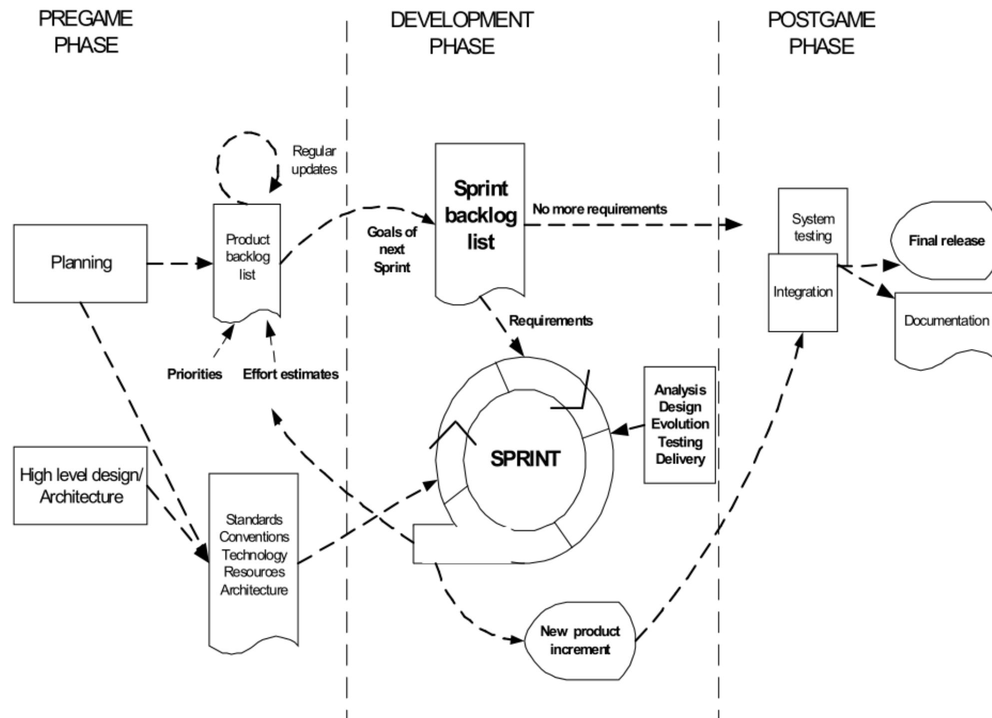


Figura 3.6 *– Las tres fases de un proceso de desarrollo Scrum, junto con sus actividades. Fuente: Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J. (2017). Agile software development methods: Review and analysis. (Abrahamsson y cols., 2017)*

Crystal y Crystal Clear

Un grupo de metodologías ágiles menos conocidas son las metodologías Crystal, las cuales fueron formuladas por Alistair Cockburn, quien trabajó en IBM en los años 90 para encontrar una buena metodología de desarrollo de software basada en orientación a objetos. En forma específica, Crystal es un “generador de metodologías” que indica ciertas herramientas ágiles y prácticas a utilizar, pero deja libertad de elección (lo que permite utilizar elementos de XP o Scrum) y se centra más que nada en una serie de principios a seguir para llevar a cabo proyectos de software exitosos según dos variables: el tamaño del proyecto en sí, y lo crítico que es éste (o sea, qué cosas están en riesgo)(Abrahamsson y cols., 2017).

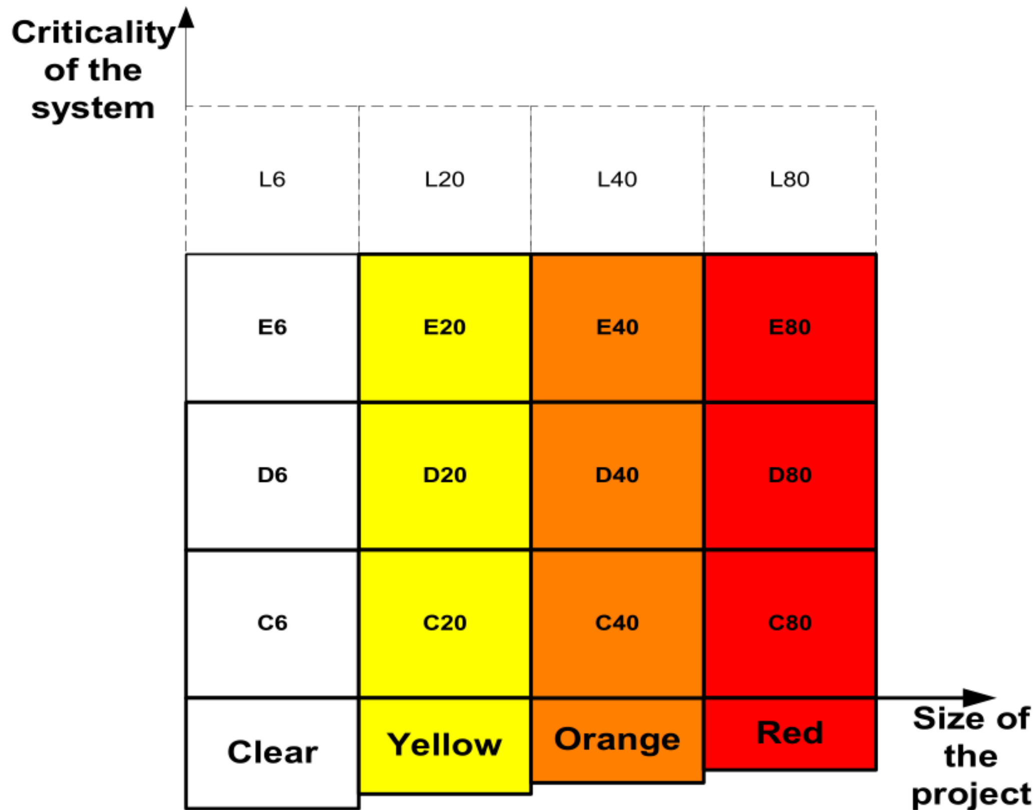


Figura 3.7 *– Las dos dimensiones de las metodologías Crystal. Fuente: Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J. (2017). Agile software development methods: Review and analysis (Abrahamsson y cols., 2017).*

Las tres reglas principales de Crystal indican lo mínimo que debe tener una metodología de desarrollo de software, con tal de que los proyectos tengan éxito. Éstas son:

- Entregas frecuentes
- Mejora reflexiva o mejora continua
- Estrecha comunicación

Las metodologías Crystal Clear serán el enfoque de esta reseña, ya que se centran en proyectos de bajo riesgo de hasta 8 desarrolladores, lo cual se acerca al proyecto Intranet. En Crystal Clear, se asume que este equipo está localizado en un mismo lugar, y al ser un equipo pequeño, puede superar la "comunicación estrecha" y llegar a la fase de "comunicación osmótica": gracias a tener nulas barreras de comunicación, las conversaciones espontáneas son escuchadas por todos, y la resolución de dudas es rápida y eficiente. Esto se apoya en la responsabilidad compartida del proyecto (todos

deben conocer sobre todos los detalles del proyecto), el concepto de “seguridad personal” (permitir que los miembros del equipo opinen sobre lo que les molesta sin miedo a represalias, con tal de llegar a la plena confianza) y con el uso de “radiadores de información“: pizarras, pantallas y otros elementos visuales que sirven como documentación, las cuales pueden ser discutidas y modificadas por el equipo.

Según sean las características del proyecto, cada metodología de la matriz Crystal nos indicará: políticas, entregables, “asuntos locales” y herramientas. En algunos casos, también se indican estándares y roles. A continuación se detallan los elementos indicados para Crystal Clear:

Políticas: Entregas de software incrementales en forma regular, integración frecuente del código, seguimiento del progreso basado en hitos y decisiones importantes, participación directa del cliente, pruebas de regresión automatizadas, dos pruebas por parte del cliente para cada entregable, y talleres para ajustar la metodología de trabajo al inicio y a la mitad de cada iteración.

Entregables: Deben tener un identificador de secuencia de entrega, modelos para los objetos/clases del sistema, manual de usuario, descripciones para los casos de uso o funcionalidades, casos de prueba y código para migraciones.

Asuntos locales: Se indica que los prototipos, documentación, codificación, pruebas de regresión y los estándares de interfaces de usuario deben ser propuestos y mantenidos por el equipo de desarrollo, y la elección de cómo se realizarán será a elección propia de éste.

Herramientas: Compilador, versionado, SCM y el uso de *printing whiteboards* o pizarrones imprimibles (o algún equivalente, como papelógrafos o pizarrones normales a los que se les pueda sacar una fotografía). Estos pizarrones servirán para presentar y almacenar lo que se discute en cada reunión.

Actividades: Como se mencionó anteriormente, las metodologías Crystal se basan en trabajo iterativo. El siguiente diagrama representa una iteración o “incremento” en un proyecto. Cabe destacar que las metodologías Crystal Clear omiten algunos de estos elementos (Abrahamsson y cols., 2017).

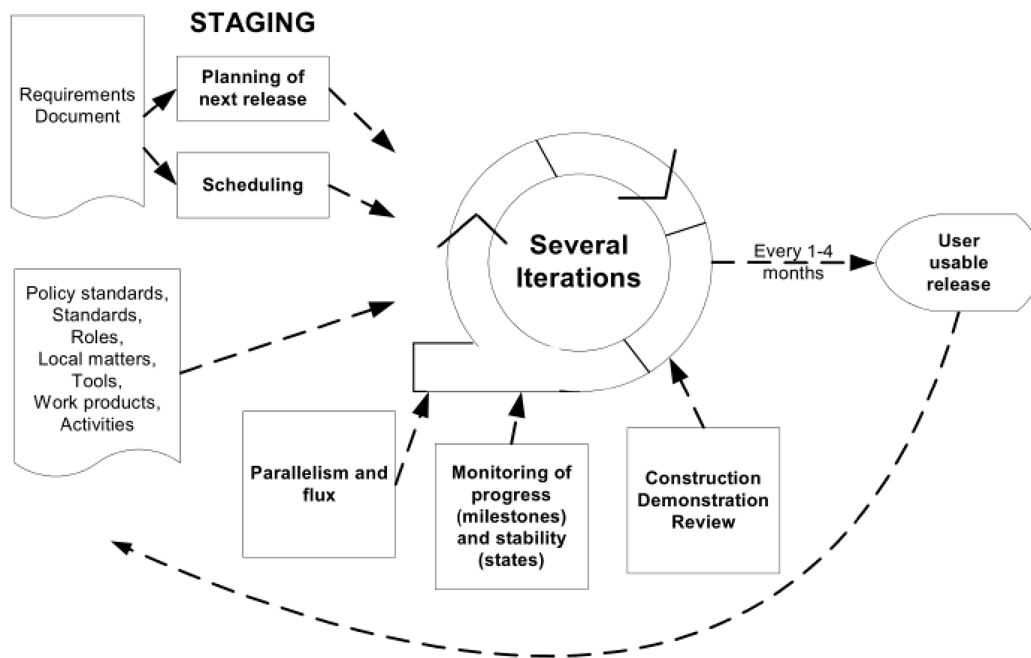


Figura 3.8 – Una iteración o "incremento" en las metodologías Crystal. Fuente: Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J. (2017). Agile software development methods: Review and analysis (Abrahamsson y cols., 2017).

Kanban

Kanban es una metodología de desarrollo basada en las tendencias *Lean*. *Lean* viene del uso de esta palabra en el inglés para representar carne magra (sin grasa), y se puede definir como un "estado mental" para trabajar, que se centra en varios principios:

- **Eliminar desperdicios**: eliminar todas las actividades que no llevan a crear software con valor.
- **Amplificar el aprendizaje**: usar la retroalimentación de proyectos anteriores para construir mejor software.
- **Decidir lo más tarde posible**: efectuar decisiones cuando se tenga la mayor información posible, o sea, lo más tarde posible.
- **Entregar lo más pronto posible**: entender el costo de los retrasos, y minimizarlo.

- **Darle poder al equipo**: tener un ambiente de trabajo con enfoque y efectivo, y tener un equipo con gente “energizada”.
- **Construir con integridad**: construir software que sea intuitivo para los usuarios, y que forme una entidad única y coherente
- **Ver el “todo”**: entender el trabajo que se lleva a cabo en el proyecto, en forma clara y sin barreras

Kanban trata de crear una metodología de trabajo que utiliza el estado mental *Lean* o *Lean mindset*, y su objetivo principal es la mejora de los procesos, mediante la búsqueda y eliminación de los “desperdicios”, y el entendimiento de cómo funciona el software por parte de todo el equipo de trabajo. Esta metodología, por lo tanto, trata sobre una forma de trabajo que va cambiando con el tiempo.

Los principios de Kanban que se deben aceptar para comenzar la implementación son: comenzar con el estilo de trabajo actual, luego comprometerse a efectuar cambios iterativos y evolutivos, y en un principio, mantener los roles y responsabilidades actuales.

Luego, se deben adoptar las siguientes prácticas:

1. **Visualizar el flujo de trabajo**: se dice que al expresar el flujo de trabajo en forma explícita, el equipo podrá tomarlo más en serio, y tener una mejor idea sobre cómo trabajan los demás. Esto generalmente se logra mediante un *kanban board*, los cuales son pizarrones donde se escriben (o se pegan con papel autoadhesivo) las historias de usuario a trabajar por un mismo equipo. Estas historias de usuario pueden ir en alguna de varias columnas, que en general dividen si la historia está en cola, es trabajo en curso, o está lista (y dependiendo del equipo de trabajo, se pueden ajustar o agregar más columnas, por ejemplo, una columna de “en pruebas”). Gracias al uso de un *kanban board*, no solo se puede tener una idea clara del estado de trabajo del equipo, si no que también se pueden encontrar problemas y cuellos de botella (que son parte del “desperdicio” que Kanban busca eliminar).

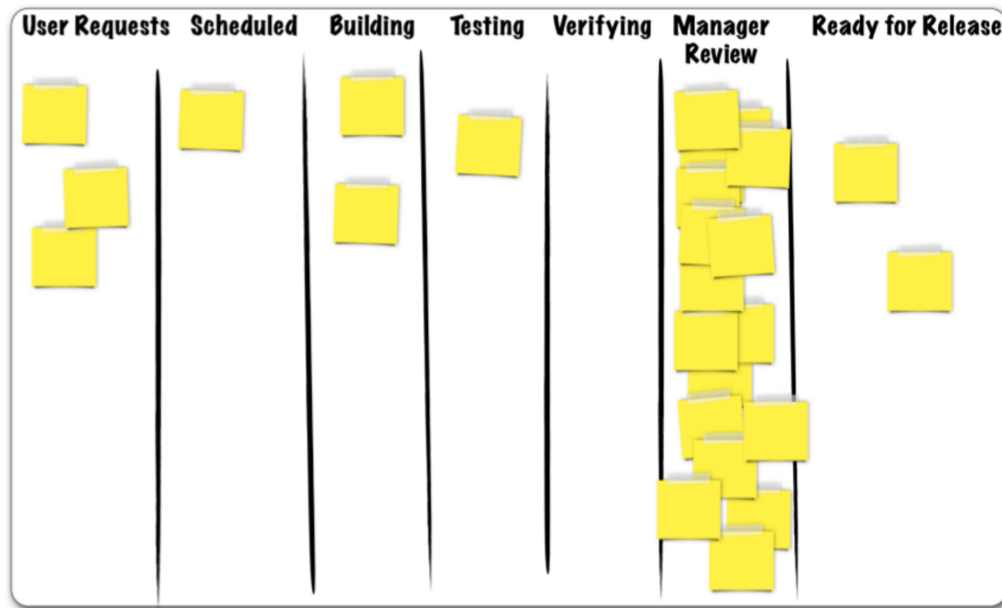


Figura 3.9 *– Un ejemplo de kanban board, donde el proceso de trabajo de un equipo de visualiza en columnas, y papeles autoadhesivos representan las historias de usuario. Se puede ver que "cuellos de botella" son fáciles de identificar. Fuente: Stellman, A., & Greene, J. (2014). Learning agile: Understanding scrum, XP, lean, and kanban (Stellman y Greene, 2014).*

1. **Limitar el WIP (*Work in Progress*, o trabajo en curso)**. Es poder ajustar un límite del trabajo en curso para evitar cuellos de botella y desbalances en la asignación de trabajo. Pensando en la linealidad de ciertos flujos de desarrollo (una actividad o columna luego de otra) y sabiendo las capacidades del equipo, el trabajo debería ordenarse en colas.

2. **Administrar el "flujo"**. Un objetivo principal de Kanban es hacer que el trabajo siga un flujo lo más rápido posible, sin miembros esperando del trabajo de otros. Se pueden aplicar fórmulas e ideas del campo de la Teoría de Colas para optimizar el flujo, y gráficos de tareas *vs.* días para verificar la eficiencia.

3. **Hacer explícitas las políticas que gobiernan los procesos**: Esto no tiene por qué ser documentación compleja, puede ser números límite para el WIP escrito en las columnas del *kanban board*, o criterios y definiciones escritas al pie de ésta.

4. **Implementar *Feedback Loops* o bucles de retroalimentación**. Por ejemplo, ajustes de las políticas de trabajo cada cierto tiempo, todo con tal de mantener el flujo.

5. **Mejorar colaborativamente y evolucionar experimentalmente**. Se basa en usar el método científico, o sea, tomar mediciones del desempeño y la efectividad

del flujo de trabajo, y aplicar cambios en los bucles de retroalimentación para poder mejorar el desarrollo en forma continua.

Así, Kanban no presenta cómo administrar el desarrollo de software, si no que es una metodología para lograr que los productos de software generados tengan valor mediante la mencionada eliminación de desperdicios, el ajuste del trabajo como un flujo eficiente, y la mejora continua. Por lo tanto, se puede aplicar como complemento a otras metodologías (Stellman y Greene, 2014).

3.6.3. Resumen comparativo de metodologías ágiles

Como resumen de la información sobre cada metodología ágil, y con tal de demostrar una base para las decisiones tomadas en este trabajo, se adjunta la siguiente tabla comparativa.

	Extreme Programming	Scrum	Crystal Clear	Kanban
Diseño del sistema	Simplicidad por sobre todo	No se indica	Simple, acordado bajo reuniones (con pizarrón).	Simple y visual.
Roles	Sin roles	Scrum Master, Product Owner, Equipo de desarrollo	Patrocinador, Diseñador-Programador *Senior,* Diseñador-Programador, Usuario.	No se indica
Responsabilidad	Compartida por todo el equipo de desarrollo	Scrum Master es responsable del desarrollo. *Product Owner* es responsable del producto.	Acordada bajo reuniones	Todo el equipo debe conocer y responsabilizarse del “todo”
Colaboración	El equipo se organiza por sí solo	Múltiples equipos se organizan por sí solos y colaboran entre sí	El equipo se organiza por sí solo	Equipos con miembros especializados (según las partes del proceso)
Flujo de trabajo	Iterativo, con iteraciones cortas.	Iterativo (*sprints*)	Iterativo (incrementos)	Continuo. Eliminación de desechos y reducción del WIP (*Work in Progress*).
Captura de Requerimientos	Con Historias de Usuario	Con el *Sprint Backlog* y *Product Backlog*	Con Casos de Uso o Descripciones de Funcionalidades	Con Historias de Usuario
Entregables de Software	Con Entrega Continua	Al final de cada *Sprint*	Al final de cada incremento/iteración	No se indica.
Pruebas de software	Desarrollo guiado por pruebas (TDD). Pruebas de aceptación.	No se indica.	Pruebas automatizadas de regresión.	No se indica.
Otras prácticas	El Juego de la Planificación. Refactorización. Integración continua. Programación a pares.	Reuniones diarias y cortas para coordinación.	Dos revisiones por parte de los usuarios antes de cada entrega. Talleres de reflexión y auto-aprendizaje. Flexibilidad a cambios en metodología según el proyecto en sí.	Bucles de autoaprendizaje. Uso del “flujo” y maximización de éste. Ajuste de parámetros para asegurar mejora continua.

Tabla 3.3 *– Comparativa de metodologías ágiles (Abrahamsson y cols., 2017) (Matharu y cols., 2015)*

3.7. Desarrollo de código abierto

Antes de concluir con las distintas metodologías, se debe tomar en cuenta cierto tipo de proyectos que comparte características con Intranet: los proyectos de código abierto o *open-source*.

Actualmente gran parte del software utilizado incluye programas, bibliotecas y otro software hecho bajo el esquema de "código abierto", donde el código que genera a este programa está alojado en un repositorio VCS del que cualquiera puede leer. Esto permite que cualquier desarrollador pueda revisar este código, o poder compilar y/o ejecutar su propia instancia del programa en forma gratuita. Y al mismo tiempo, el desarrollo de este programa puede continuar con cualquier persona que se presente ante los "mantenedores" y proponga añadir cierta funcionalidad o resolver algún defecto. Todo esto a través de Internet.

Si bien existen desventajas respecto al modelo de negocio de este tipo de software y la complejidad de organizar equipos de trabajo distribuidos, muchos afirman que las ventajas de este modelo de desarrollo supera al del software de código cerrado. Estas ventajas incluyen el fácil descubrimiento y resolución de errores por parte de cualquier persona (ya que todos pueden ver el código), el constante mantenimiento de estos programas (dado a que cualquier organización o persona interesada en éstos puede continuar su desarrollo) y la mayor confianza por parte de los usuarios (ya que asumen que este software ha sido probado y mejorado por muchos anteriormente). Por supuesto, todas estas ventajas solo se aplican si este software aún tiene interesados alrededor del mundo, y no ha sido abandonado.

Con tal de contrarrestar los problemas que puede ocasionar un software desarrollado por muchas personas alrededor del mundo (con distintos intereses, niveles de experiencia y horarios flexibles), hay varias actividades y medidas que se aplican en estos proyectos:

- La existencia de uno o más "mantenedores", que serán los jefes del proyecto y que analizarán si los cambios propuestos por otras personas serán aceptados en el código base.
- El desarrollo utilizando arquitecturas modularizadas, para asegurar que haya poca interdependencias entre partes del código y evitar conflictos entre desarrollos en paralelo.
- En general, utilizan un proceso de desarrollo basado en proponer cambios al código mediante *pull requests*, donde una persona presenta los cambios que hará y por qué los hará, y los mantenedores discutirán con la comunidad si se aceptará o no.
- El uso de herramientas tipo "*issue tracker*" para recibir retroalimentación, *pull requests*, reportes de errores y recomendaciones para nuevas funcionalidades. A

veces, se utiliza además *kanban boards* para presentar el avance del proyecto, y listas de correo para la comunicación entre desarrolladores.

Además, se siguen políticas y reglas definidas por los mismos mantenedores y desarrolladores del proyecto, que incluyen:

- Uso obligatorio de pruebas automatizadas, sistemas de integración continua y métricas de cobertura de pruebas, para asegurar la calidad del software ante los cambios entregados por distintas personas alrededor del mundo, y evitar que aparezcan nuevos defectos.
- Uso obligatorio de documentación dentro del código, generadores de documentación y otros tipos de documentación (como *wikis*), con tal de guiar a nuevos desarrolladores.
- Códigos de conducta y otros reglamentos de comunicación.

Figura 3.10 – *Captura de pantalla de builds del software de código abierto Dolphin. Se ve como un software de CI/CD fue capaz de publicar nuevos builds a partir de las contribuciones de los desarrolladores, de los cuales varios se asocian a pull requests con el formato PR #XXXX. Fuente: Sitio oficial de Dolphin en dolphin-emu.org*

Finalmente vale mencionar que los desarrolladores de estos proyectos en general son voluntarios o miembros de organizaciones que tienen intereses en éste, y en su mayo-

ría tienen mucha experiencia y disciplina en proyectos similares. (Mockus, Fielding, y Herbsleb, 2002)

El desarrollo de software de código abierto, por tanto, logra cumplir muy bien sus objetivos a pesar de la distribución de desarrolladores existentes gracias a procesos bien definidos, reglas estrictas, y el uso de herramientas de comunicación, control de versiones y de aseguramiento de calidad automático (en este caso, *testing*, CI/CD y métricas automáticas) (Mockus y cols., 2002).

3.8. Conclusiones

Los problemas del proyecto son algo que han lidiado muchos otros proyectos anteriormente, y afortunadamente existen varias metodologías y herramientas que pueden ayudar a resolverlas. Si bien varias de estas metodologías pueden funcionar en equipos pequeños, éstas también asumen que el equipo de desarrollo es presencial y de tiempo completo (Crystal), o que tiene un fácil acceso a clientes (Extreme Programming), a jerarquías organizaciones (SCRUM) y a flujos de trabajo optimizados (Kanban). Por otro lado, el desarrollo de código abierto logra asumir y resolver el problema de trabajo remoto y rotativo, pero asume alta motivación y experiencia de los desarrolladores. Por lo tanto, lo mejor para el proyecto Intranet es tomar elementos de las metodologías XP, Crystal, Kanban y de código abierto, y poder generar una metodología basada en comunicación casi-osmótica utilizando documentación y herramientas de comunicación, que al mismo tiempo permita el aseguramiento de calidad del proyecto utilizando pruebas automatizadas, integración continua, despliegue continuo, métricas e integraciones de tipo DevOps.

Capítulo 4

Propuesta de Solución

En esta sección se presenta el análisis y elección de las actividades, metodologías y herramientas de desarrollo de software que van a ser aplicados en el proyecto Intranet, y que luego formarán parte del Plan de Calidad. Se comenzará con la metodología de desarrollo, junto con las actividades que conformarán el flujo de trabajo a seguir y las razones para la elección de éstas. Luego, se presentará la metodología de pruebas, con los tipos de pruebas que se harán y su forma de implementación. Y después, a partir de estas metodologías, se procede a la comparativa y selección de herramientas de apoyo. Estas decisiones serán realizadas tomando en cuenta las características particulares del proyecto y su equipo de trabajo, junto con los recursos disponibles y las políticas de I&T.

4.1. Metodología de desarrollo

La metodología a aplicar será una mezcla entre varias de las vistas anteriormente, con un énfasis en un flujo iterativo que utiliza actividades basadas en las metodologías ágiles y el desarrollo de software de código abierto. Se tendrá un énfasis en automatización, documentación y comunicación, y se apoyará en herramientas autogestionadas (*self-hosted*) para desarrollo, pruebas, integración continua, comunicación y gestión de configuración.

Las razones detrás de la decisión de aplicar esta metodología al proyecto se verán a continuación, comparando las metodologías de desarrollo vistas en la sección Estado del Arte y contrastando con lo que podría o no funcionar con un equipo de ayudantes y estudiantes en práctica.

4.1.1. Análisis de las metodologías existentes

Las metodologías de desarrollo modernas proponen diversas actividades a partir de las suposiciones que tiene cada una. Se establecen razones para realizar cada una, pero es necesario mencionar cuales son las que tienen más sentido respecto al proyecto Intranet.

Extreme Programming (XP)

Extreme Programming es una alternativa interesante. El estilo de desarrollo (iterativo, y con integración y entrega continuas) se adapta bastante bien a estudiantes con horarios flexibles, con tal de asegurar la calidad del sistema y evitar conflictos de código. Sin embargo, existen algunas prácticas que podrían dan problemas:

- El uso de Desarrollo guiado por pruebas (TDD) va a requerir de disciplina y un posible período de capacitación, pensando en que los estudiantes en general no tienen experiencia en desarrollar pruebas automatizadas (dado a que no son requeridas en tareas y proyectos pequeños de asignaturas de pregrado).
- Además, el uso de Historias de Usuario como forma de capturar requerimientos se adecúa muy bien a la realización de pruebas de aceptación, pero hay que tener el cuenta el riesgo de interpretaciones confusas. Las frases cortas pueden ser interpretadas de forma distinta por cada miembro del equipo, y tomando en cuenta la rotativa de desarrolladores, se requiere que exista claridad respecto a qué quiere el cliente y cómo se va a implementar. Por lo tanto, las Historias de Usuario deben ser acordadas en conjunto y documentadas, si es posible, en reuniones.
- Finalmente, la programación a pares es una idea muy buena, ya que permite hacer una revisión continua de la calidad del código y una oportunidad de compartir conocimiento (por ejemplo, entre un miembro con experiencia y uno sin experiencia), pero podría no ser posible en todo momento (por las diferencias de horarios).

Prácticas a aplicar: Entrega e Integración Continua, Historias de Usuario, Programación a Pares

Prácticas en conflicto: TDD

Scrum

Scrum tiende a ser una metodología más orientada a la organización de las personas en proyectos de software de gran envergadura, con múltiples equipos de trabajo. A pesar

de ser una metodología muy popular en empresas de desarrollo de software, bajo los requerimientos del proyecto (que no es grande o crítico) y las características del equipo de trabajo (pequeño y flexible), su implementación sería muy costosa, dado el énfasis en la organización de las tareas y metas de trabajo. Se van a requerir más personas para roles que no harían mucho trabajo, o se tendrían que combinar roles. Las reuniones diarias serían un gasto importante de tiempo y podrían no poder realizarse según la disponibilidad de los desarrolladores, y en general, el costo de organización terminaría siendo perjudicial para el proyecto.

Prácticas a aplicar: Reuniones de retroalimentación en medio del desarrollo

Prácticas en conflicto: Reglas de organización y jerarquización.

Crystal Clear

Crystal Clear comparte algunos elementos de Extreme Programming, pero añadiendo la importancia de documentación, autoevaluación y comunicaciones del equipo.

El uso de radiadores de información y pizarrones para apoyar la documentación es una idea bastante llamativa, ya que miembros del equipo que trabajen en horarios distintos pueden revisar lo acordado anteriormente, o dejar mensajes a los demás en éste sin que todos estén allí a la vez. Por supuesto, este pizarrón podría ser acompañado por otras herramientas de trabajo asincrónicas, como software de mensajería en grupo, documentación compartida, o incluso sistemas de gestión de proyectos e *issue trackers*.

Estas mismas herramientas podrían ayudar a suplir algunos de los problemas que se pueden tener al tratar de llegar a una "comunicación osmótica" que propone Crystal Clear (por la flexibilidad de horarios). Además, se busca que no hayan interrupciones en el trabajo debido a falta de información, dudas o conflictos, por lo tanto, se debe hacer un buen balance de reuniones presenciales y comunicación remota.

Además, Crystal Clear indica que se deben implementar principios, políticas y estándares que se pueden tomar como base, lo cual puede ser beneficioso para los proyectos de software realizados en la universidad, los cuales en general sufren de desorden que no ocurre en ambientes de desarrollo de software modernos. El mismo equipo de trabajo debe ser capaz de seguir y ajustar las políticas según sus características.

Finalmente, Crystal Clear propone el uso de pruebas de integración y entrega frecuente (CI/CD), junto con pruebas de integración para complementar a éstas. Si bien estas prácticas son muy útiles, el uso de pruebas de integración debe analizarse con cuidado, ya que en general los estudiantes no tienen experiencia creando pruebas, y es posible que exista una mejor metodología de pruebas para el proyecto en cuestión.

Prácticas a aplicar: Diseño visual, radiadores de información, estándares y prácticas

definidas, integración y entrega frecuente.

Prácticas en conflicto: Comunicación osmótica, Pruebas de integración.

Kanban

En paralelo se debe tomar en cuenta a Kanban, la cual es una metodología que puede ser implementada como complemento a otras metodologías. Si bien el uso de un *kanban board* para la visualización del flujo de trabajo es interesante y podría ser útil, la perspectiva de "optimizar" este flujo de trabajo con tal de maximizar la productividad no es algo que sea prioritario en este proyecto, donde podrían entrar a trabajar estudiantes y programadores con poca experiencia y/o con horarios flexibles. Por lo tanto, el uso de métricas de desempeño y ajuste de variables/limitaciones no debería implementarse, ya que podría no ser válido ante cambios repentinos de horarios y de cargas de trabajo.

Por otro lado, otros elementos de Lean y Kanban sí pueden ser tomados en cuenta: la visión de que todos los miembros del equipo deban conocer y responsabilizarse de todo el proyecto completo es válida, teniendo en cuenta que teniendo a estos miembros con disponibilidad limitada, en caso de alguien necesitase ayuda para el desarrollo, o algún usuario necesite soporte, cualquiera puede ser de utilidad. O sea, que ante eventualidades, cualquiera pueda responder, lo cual también se relaciona con el principio de crear software que funcione como una entidad única (el hecho de que todos deban responsabilizarse por el software conlleva al uso de estándares e ideas compartidas). Y además, la búsqueda de retroalimentación es algo que se comparte con otras metodologías ágiles, y podría implementarse sin tener que usar la optimización de métricas.

Prácticas a aplicar: Flujo de desarrollo visual, responsabilidad compartida.

Prácticas en conflicto: Software como entidad única, optimización del flujo de trabajo.

Desarrollo de código abierto

Finalmente el desarrollo de software de código abierto comparte varias de las prácticas usadas por otras metodologías, como son las pruebas automatizadas, herramientas CI/CD, el uso de documentación y el uso de herramientas *issue tracker*. Sin embargo, existe una organización y una estructura bien definida con tal de evitar el caos y el desorden ante desarrolladores de lugares e intereses tan distintos, y por lo tanto, se asume que los desarrolladores tienen alta disciplina y alta experiencia.

Se ha descubierto que en los estudiantes hay un grado de disciplina importante, en especial los estudiantes en práctica, ya que si bien no existe una gran remuneración,

ellos esperan ser bien calificados por su jefe para poder aprobar su práctica. El uso de reglas, políticas y procesos de desarrollo bien definidos deberán ser bien recibidos por los nuevos miembros del equipo, pero siempre se debe tomar en cuenta que su nivel de experiencia es bajo. Afortunadamente, el uso de mentores y buena documentación puede mitigar este último problema.

El proceso de desarrollo mediante *pull requests* no se aplica al proyecto Intranet dado a que este proyecto es de código cerrado, y solo los miembros del equipo (que de por sí tienen acceso al SCM) harán cambios. A menos que exista un cambio en la apertura del código, esta práctica no tendrá sentido.

Como punto importante, el uso de métricas por parte de proyectos de código abierto, como son el porcentaje de cobertura de pruebas, el porcentaje de pruebas aprobadas y el número de *builds* exitosos se asimila muy bien al uso de CI/CD, y puede servir para el aseguramiento de calidad que se busca en Intranet.

Prácticas a aplicar: Estándares y prácticas definidas, integración y entrega frecuente, herramientas de comunicación, métricas de calidad.

Prácticas en conflicto: Proceso de desarrollo basado en *pull requests*.

4.1.2. Actividades escogidas desde metodologías existentes

Comenzando con las metodologías ágiles, desde Extreme Programming (XP) y Crystal Clear se usarán las actividades de **Integración Continua** y creación de requerimientos como **Historias de Usuario** como base obligatoria. La idea de Programación a pares de XP será recomendada como actividad opcional.

Desde Crystal Clear no se puede llegar a la "comunicación osmótica" plena dada las características del equipo de trabajo, pero al menos se buscará un acercamiento utilizando Radiadores de Información (**documentación visual**), apoyado por **herramientas de trabajo colaborativo remoto** y la búsqueda de una **responsabilidad compartida**. Además, se utilizará la **Entrega Frecuente** (apoyada en la Integración Continua y la comunicación con el cliente), el uso de pruebas automatizadas de integración y las reuniones de retroalimentación.

Desde Kanban se adaptará el uso del ***kanban board*** para el seguimiento del trabajo (Historias de Usuario o tareas de mantenimiento/corrección) en forma visual, como apoyo a los radiadores de información, y se va a rehacer el flujo de trabajo (pero sin intentar optimizar tiempo y recursos, si no que se buscará maximizar la calidad del producto final).

Finalmente del desarrollo de código abierto se utilizarán las **métricas de cobertura de pruebas** como parte del flujo de integración continua, y al igual que Crystal Clear

y Kanban, se dará una gran importancia a la **comunicación**, la **documentación** y el **seguimiento de las reglas**, con tal de que no se pierda la "cultura" del aseguramiento de calidad ante desarrolladores de horario flexible.

4.1.3. Resumen de la metodología

Ante el análisis anterior y las características del proyecto, se propone una metodología híbrida basada principalmente en las prácticas del software de código abierto, con elementos de las metodologías Crystal Clear, Extreme Programming (XP) y Kanban. Será una metodología basada en trabajo iterativo incremental, donde se definirá cada iteración según un requerimiento o tarea de mantenimiento/corrección.

Cada una de estas iteraciones y sus subactividades serán asignadas mediante un *kanban board*, donde cada una será una "tarea". El flujo de trabajo definirá cada iteración con una fase de análisis y diseño que implica comunicación con el cliente (mediante reuniones si es posible, si éste es parte del Departamento de Informática) y comunicación entre el mismo equipo (presencial, y si no es posible, remota). Luego, se tendrá una fase de desarrollo que tendrá el desarrollo de pruebas, y la integración con entrega continua en un ambiente de puesta en escena o *staging*. Luego del desarrollo, se podrá mostrar el avance al cliente y a posibles usuarios de prueba utilizando este mismo ambiente de *staging*, y poder hacer otra iteración (si se requieren cambios) o desplegar en producción.

En paralelo al trabajo iterativo, se tendrá un énfasis importante en la documentación y la responsabilidad compartida del proyecto. Cada persona debe ser responsable de asegurar que sus cambios no produzcan conflictos con el trabajo de otros, o defectos en el software inesperados, y se debe tener como responsabilidad el de dejar información para el resto del equipo (y posibles nuevos miembros) sobre nuevos cambios, funcionalidades, entidades, dependencias, etc. Como infraestructura para sostener las nuevas actividades, la gestión de configuración y la documentación, es que se escogerán herramientas que puedan ser instaladas y mantenidas por el equipo dentro del mismo Data Center donde se trabaja.

4.2. Metodología para pruebas de software

La búsqueda por un nivel deseable de calidad para Intranet, y el uso de actividades como Integración Continua amerita una metodología de pruebas para el proyecto que incluya pruebas automatizadas que verifiquen que no hayan problemas de regresión o integración cada vez que se agrega nuevo código, y que al mismo tiempo, los requerimientos de los usuarios siempre estén cubiertos. Además, se debe tomar en cuenta la poca experiencia de los estudiantes en práctica y posibles ayudantes en la realización de pruebas automáticas.

4.2.1. Análisis de las metodologías existentes

El **desarrollo guiado por pruebas (TDD)** y el uso de **pruebas unitarias** son algunas de las bases más importantes para el aseguramiento de calidad en la metodología Extreme Programming. Pero hay que tener en cuenta que ésta no indica que las pruebas unitarias deban tener una cobertura del 100 %, si no que las pruebas deben abarcar "todo lo que se pueda estropear", dejando las pruebas de aceptación como la serie de pruebas principales para demostrar que el sistema cumple los requerimientos del cliente.

En la metodología Crystal Clear se habla de que deben haber **pruebas automatizadas de integración**. Estas pruebas deben asegurar que la integración de código de distintos desarrolladores no produzca efectos inesperados. Sin embargo, la cobertura de las pruebas de integración se puede ver como un subconjunto de las pruebas de aceptación: si se puede probar que todas las funcionalidades del sistema cumplen los requerimientos, y que los casos de uso alternativos y de excepciones llegan a salidas esperadas, entonces puede haber confianza de que la integración de código fue exitosa, y por tanto, no se deberían escribir pruebas de integración. Al tener estos todos estos casos, se espera que pruebas alcancen un alto porcentaje de cobertura de código (de al menos 70 %).

Para el caso de las **pruebas de aceptación**, puede ser muy beneficioso si se aplica la práctica de desarrollo guiado por comportamiento (BDD) en vez de TDD, ya que si bien se requerirá aprender el DSL (Lenguaje específico del dominio) de las herramientas de BDD, se podrán escribir directamente como si fuesen los casos de prueba de las Historias de Usuario discutidas en la captura de requerimientos, evitando posibles problemas de interpretación.

Finalmente, para los casos en que las pruebas de aceptación no sean suficientes, se harán pruebas unitarias. Por ejemplo, si existen casos de excepción muy difíciles de reproducir, o si existe código fuera de las historias de usuario (por ejemplo, código que se usa solo por parte de los desarrolladores, para *debugging*).

4.2.2. Metodología escogida

En la búsqueda del aseguramiento de la calidad del proyecto Intranet, se aplicará como metodología base una versión modificada del **Desarrollo Guiado por Comportamiento** (BDD, *Behaviour Driven Development*), el cual indicará la escritura de **pruebas de aceptación** antes del desarrollo, las cuales serán **basadas en las Historias de Usuario**. Sin embargo, el desarrollo de las pruebas a nivel de código no será antes del desarrollo en sí, sino en paralelo a ésta. Y para casos que las pruebas de aceptación no puedan cubrir, se utilizarán **pruebas unitarias**. Se utilizará la aprobación de todas estas pruebas y la cobertura de código como métricas de efectividad.

4.3. Comparativa y selección de herramientas

Las ideas presentadas anteriormente necesitan una base tecnológica para que su implementación no deba requerir tiempo y esfuerzo excesivos. Ante esto, se debe identificar todo lo que puede ser automatizado y todo lo que puede ser apoyado por herramientas ya existentes, con tal de poder configurar el proyecto y desplegar tales herramientas.

4.3.1. Herramientas de gestión de configuración del software

Sistema de Control de Versiones (VCS)

Tomando en cuenta que a través de la malla curricular de los estudiantes del Departamento de Informática se enseña y requiere el uso de **Git**, y dado a que la mayor parte de las empresas jóvenes de desarrollo web (según anécdotas de ex-alumnos de la universidad) utilizan Git o algún repositorio compatible con Git, es que se escogerá a éste como el sistema de control de versiones de preferencia.

Sin embargo, si es que el repositorio permite utilizar otro software sin obstaculizar el trabajo colaborativo, se podrá permitir el uso de una herramienta compatible. Por ejemplo, existen *scripts* para utilizar ciertas parte de la funcionalidad de Mercurial o SVN en repositorios Git, y viceversa.

Repositorio para el Sistema de Control de Versiones

Con tal de permitir el trabajo colaborativo utilizando un sistema de control de versiones, es que debe hacer un servidor que actúa como "repositorio" del código, desde donde los desarrolladores subirán sus propios cambios y recogerán los avances de otros

desarrolladores. Actualmente en el Departamento de Informática existen dos repositorios de uso interno, y se debe evaluar el uso de alguno de éstos, y las opciones de nuevos repositorios.

- **Stash** de I&T, DI. Este es un repositorio BitBucket de la suite Atlassian. Está alojado en uno de los servidores del Data Center y tiene todas las funciones permitidas por su versión actual (5.7.1, de Enero de 2018). Hasta ahora Intranet se ha alojado en este repositorio, pero éste tiene un problema: La licencia de la suite Atlassian solo permite su uso por una cantidad limitada de usuarios, por lo tanto, no todos los desarrolladores tienen acceso por defecto por cuenta a éste. En muchas ocasiones, es necesario hacer ciertos trucos o *workarounds*, como alojar llaves SSH de los practicantes en cuentas compartidas, con tal de que éstos tengan permisos de subida y descarga de código.

- **GitLab** del LabComp, DI. Este es un repositorio alojado en los servidores del Laboratorio de Computación del DI. Éste tiene gran funcionalidad, ya que funciona como una suite completa de trabajo colaborativo, incluyendo *Issue Tracker*, asignación de tareas mediante un Kanban, espacio para documentación, etc. En general, logra ser una buena opción ante la suite Atlassian, con la ventaja de ser gratuito. El problema principal de su adopción es el hecho de que sea mantenido por el Laboratorio de Computación, el cual es conformado por ayudantes externos al equipo de I&T que maneja el proyecto Intranet, y no hay confianza plena en una buena mantención.

- **Contratación de repositorio externo**. Existen servicios externos de repositorios de código. El más popular a nivel mundial es GitHub, seguido por sus competidores BitBucket y GitLab. Si bien estos servicios tienen muy buena reputación por parte de la industria, éstos requieren un pago mensual o anual por el acceso a repositorios privados que se puedan usar por muchas personas, lo cual podría no valer la pena. Se pueden obtener repositorios gratuitos si es que el código se mantiene abierto (*open source*) o utilizando promociones especiales para estudiantes, pero éstas últimas son limitadas por la duración de la carrera de quien cree el perfil del proyecto. Además, existe el inconveniente de que se estará confiando en este servicio externo para el manejo de cuentas del repositorio y para el alojamiento del código, lo cual va en contra de la idea de la Unidad de I&T, que promueve el uso de herramientas *self-hosted* y la autosuficiencia.

- Despliegue de un **nuevo repositorio** self-hosted. Finalmente, se debe considerar montar un nuevo repositorio (por ejemplo, un nuevo GitLab) que pueda servir para el proyecto Intranet, o para reemplazar la suite Atlassian de I&T. Esto permitiría modernizar y mejorar los problemas que tiene la suite Atlassian, dado a que no tendría más limitaciones de usuarios y podría ser actualizado periódicamente. Sin embargo, la migración desde la suite Atlassian no es trivial, y requiere un esfuerzo para traspasar la documentación y los proyectos actuales, y hacer una

capacitación a los miembros de I&T, sin contar el esfuerzo de desplegar este sistema en primer lugar. Lamentablemente, todo esto terminaría siendo muy costoso e innecesario, y debe ser formulado como un proyecto por separado.

Finalmente, para el caso del proyecto Intranet, se ha decidido mantener el uso del repositorio **Stash** de I&T, pero además planteando los problemas de éste, y una posible solución o reemplazo para el futuro. En caso de otros proyectos del DI, se debe considerar el uso del GitLab del LabComp, si es que el proyecto es cercano a las labores de éste y no se trata de un proyecto crítico. Si se tiene un proyecto crítico, se debe considerar el uso de proveedores externos.

Alojamiento de documentación

Se ha decidido que la documentación del sistema será alojado como archivos dentro del repositorio del sistema de control de versiones (VCS). De esta forma, ésta puede ser vista y modificada directamente por los miembros del equipo, sin tener que entrar a un sitio web o herramienta aparte.

Sin embargo, existen casos donde la documentación incluirá pasos o indicaciones que son previas a descargar el código desde el VCS (por ejemplo, indicaciones para configurar el ambiente de trabajo). Esta documentación adjunta será alojada en el sistema **Confluence** de I&T.

Alojamiento de información sensible

Como se mencionó anteriormente, se utilizará **Confluence** para almacenar archivos de carácter sensible, como son: contraseñas, llaves de APIs, credenciales, dominios y puertos de servidores externos. Ya que éste ya incluye una capa de autenticación y autorización, existe confianza en que estos archivos no podrán ser vistos por otras personas u otros miembros del Departamento de Informática.

Identificación y construcción de configuración

La configuración de los componentes del sistema Intranet ya vienen dados por el framework Ruby on Rails, el cual en cada proyecto utiliza un archivo *Gemfile* con las dependencias a utilizar y un archivo *Gemfile.lock* para indicar las versiones específicas que se necesitan. El mantenimiento y actualización de estas componentes será compartido por el equipo, al utilizar Integración Continua y el VCS para mantener un conjunto uniforme de versiones. Y para el caso de componentes a nivel de sistema operativo (como la distribución del lenguaje Ruby), la identificación de ésta será parte de

la documentación, donde los miembros del equipo indicarán lo que se debe instalar en cada servidor y estación de trabajo.

4.3.2. Herramientas de desarrollo

Editor de código e IDEs

Hasta ahora se ha mantenido la libertad de elegir el editor de código fuente a criterio de los miembros del equipo, y se piensa mantener así. Sin embargo, si existe compatibilidad con el stack del proyecto y las otras herramientas a utilizar, se recomienda su uso.

Por ejemplo, dado a que el proyecto utiliza un lenguaje dinámico y sin compilación, se va a recomendar el uso de editores de texto con soporte de *linters*, los cuales son programas o plugins para el editor que revisan el código por posibles errores de formato (como operadores mal escritos) o para dar advertencias (por ejemplo, de variables no usadas). Un editor con un *linter* de Ruby puede ser capaz de advertir inmediatamente si es que hay errores simples en el código, sin tener que ejecutar éste.

En cuanto a la elección específica, mientras el editor sea compatible con el código de Intranet (utilizando saltos de línea de tipo Unix e identación consistente, por ejemplo) y que no produzca efectos adversos (como la creación de archivos basura o de copia de seguridad que se copien al control de versiones) debería ser permitido.

En general, todas las herramientas de desarrollo que no produzcan efectos secundarios en el trabajo colaborativo del proyecto, serán dejados a criterio de cada desarrollador. Si existe soporte para *linters* y conexión con las otras herramientas a usar, se debe motivar el uso de tal herramienta.

4.3.3. Herramientas de pruebas

Bibliotecas

A nivel de código, se escribirán las pruebas en Ruby utilizando la biblioteca RSpec, la cual definirá el DSL y su interpretación, en forma de especificaciones (*specs*), las cuales estarán en el mismo repositorio del código principal, y podrán ser ejecutadas en cualquier momento.

Las pruebas de aceptación utilizarán la biblioteca Capybara para expandir el DSL de RSpec. Éstas utilizarán un navegador *headless* (para poder ejecutar *javascript*), el cual será Webkit dado a que utiliza menos recursos que las otras alternativas, y está disponible para su instalación en repositorios oficiales de sistemas operativos para

servidores (como CentOS, el cual se utiliza en el servidor de Staging). Para utilizar este navegador se usará la biblioteca *capybara-webkit*, y para poder ejecutar pruebas que utilizan APIs externas (como la de Ruckus y la del LDAP del DI) se utilizará la biblioteca VCR. Las peticiones y respuestas capturadas por VCR se guardarán en el mismo VCS del código.

Ambiente de pruebas

Se tendrán cuatro ambientes donde se puede ejecutar la aplicación: ambiente de **desarrollo**, de **pruebas**, de **puesta en escena** y de **producción**. Las pruebas se ejecutarán en el ambiente de pruebas, y las posibles pruebas manuales y presentaciones al cliente serán en el ambiente de puesta en escena.

Hasta ahora, casi todas las pruebas del proyecto Intranet se realizan en los mismos computadores de los desarrolladores, utilizando el mismo ambiente de desarrollo (que en Rails, significa utilizar el ambiente "*development*"), el cual va a tener varias herramientas para el desarrollo y la resolución de problemas (como mensajes de error muy específicos) pero que no son un reflejo fiel de cómo la aplicación se comportará en el ambiente de producción. En otras ocasiones, se utiliza el mismo ambiente de producción para hacer pruebas, lo cual puede ser muy peligroso al manipular datos que se podrían estar utilizando por parte de los usuarios.

Para la resolución de estos problemas, se seguirá una práctica usada por varios proyectos web en Ruby on Rails alrededor del mundo (Mornini y Loy, 2006), el de tener cuatro ambientes o *environments* donde se puede ejecutar la aplicación web, los cuales tendrán bases de datos y configuraciones separadas, y se detallan a continuación:

- Ambiente de desarrollo o *development*, el cual es el usado por los desarrolladores mientras trabajan. Este, por ende, estará en su misma estación de trabajo, y será distinto para cada persona.
- Ambiente de pruebas o *test*, el cual será usado por los desarrolladores para ejecutar las pruebas automatizadas en su misma estación de trabajo.
- Ambiente de puesta en escena o *staging*, el cual también será usado para pruebas automatizadas, integrando el código de los desarrolladores en la rama *development* del VCS, y estará en un servidor web especial, el cual debe ser muy similar al de despliegue. Además, permitirá pruebas manuales por parte de los desarrolladores y de parte de clientes y usuarios.
- Ambiente de producción o *production*, donde se ejecuta Intranet como producto final, y se atenderá las solicitudes de los clientes y usuarios.

4.3.4. Herramientas para Integración y Entrega Continua

Se elegirá **Jenkins** como herramienta de CI/CD debido a que no tiene limitaciones en su versión gratuita *self-hosted*, es continuamente mantenido, y existen suficientes *plugins* para cubrir las necesidades del proyecto Intranet. La dificultad de configuración será mitigada utilizando documentación y haciendo un despliegue inicial de Jenkins que ya venga configurado.

Como agente de Jenkins se utilizará el servidor de *staging*, ya que éste ya tendrá todas las dependencias de Ruby, Rails y las bibliotecas utilizadas por el proyecto. Se configurará Jenkins para que encuentre cambios en el repositorio VCS y notifique al servidor de staging para que realice el proceso de ejecutar pruebas y desplegar la nueva versión.

4.3.5. Herramientas de comunicación

Como medio de comunicación primario se utilizará el correo electrónico. Por defecto se asumirá el uso del correo de alumnos de informática **Zimbra** (mantenido por el LabComp) gracias a su integración con el LDAP del departamento, pero se dará la opción de utilizar el correo institucional (sansano.cl o usm.cl).

Se utilizará la versión piloto de **Mattermost** montada por I&T como medio de comunicación secundario, debido a la madurez en su desarrollo. El registro de usuarios será manual, y se va a configurar para enviar notificaciones en caso de cambios en el código, resultados de pruebas y despliegues.

Se utilizarán las plataformas ya existentes **Confluence** y **Jira** de I&T para complementar la organización y comunicación. Confluence se usará para agregar documentación secundaria e información de carácter sensible (como contraseñas y claves de API), y Jira para el reporte interno de errores y la asignación de tareas en un *kanban board*.

4.3.6. Herramientas de comunicación con el cliente

En el caso de Intranet, los clientes serán los mismos usuarios del DI: Profesores, funcionarios y estudiantes. Problemas, bugs y recomendaciones se recibirán mediante el sistema de Tickets del DI (basado en la plataforma de código abierto **GLPI**), **correo electrónico** y, como última instancia no deseada, **llamadas telefónicas a I&T**. Reuniones presenciales, que ya de por sí se utilizan en el departamento, serán las utilizadas para comunicar progreso o buscar retroalimentación con las autoridades.

4.4. Conclusiones

Se aplicará en el proyecto Intranet una metodología híbrida de desarrollo basada en varias metodologías ágiles y en los procesos de desarrollo de código abierto. Para ésta, se han escogido una serie de prácticas de desarrollo de software que se ajustan a los requerimientos del proyecto, las características del equipo de trabajo y la experiencia encontrada en la literatura. Estas serán la captura de requerimientos como Historias de Usuario, la importancia de documentación visual de fácil acceso, responsabilidad compartida, y flujos de trabajo con integración y entregas continua.

Como herramientas que den soporte a esta metodología se han escogido herramientas y bibliotecas que son estándares de la industria, que son gratuitas y en algunos casos, ya existentes en la infraestructura del departamento. Para la planificación se usará Jira y sus *kanban boards*, para mensajería se usará Mattermost y el correo electrónico Zimbra, y la documentación se alojará en el mismo SCM (Stash) o en Confluence. Se hará integración continua y despliegue continuo con Jenkins y el recogimiento de feedback se hará mediante el sistema de Tickets del Departamento de Informática.

Capítulo 5

Implementación de la Solución

En esta sección se presenta el trabajo realizado a lo largo del 2018 para implementar progresivamente los puntos del Plan de Calidad en el proyecto Intranet, lo cual ha implicado cambios importantes en el flujo de trabajo, la instalación y el uso de nuevas herramientas. Se ha establecido un nuevo flujo de trabajo que implementa lo indicado en el plan, y se han preparado pruebas, código, configuraciones y *scripts* para integrar las nuevas herramientas con cada nuevo *commit* o cambio en el código. Todo esto con tal de preparar la nueva metodología para los miembros actuales del equipo, junto con los nuevos miembros que se unan en el futuro.

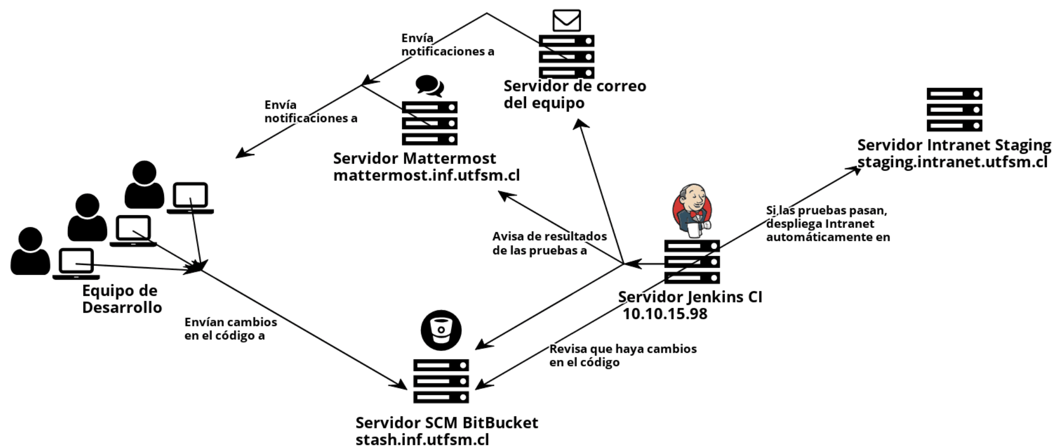


Figura 5.1 *– Diagrama de alto nivel que muestra el nuevo flujo de trabajo, junto con el deber de las herramientas instaladas. Fuente: Elaboración propia utilizando la herramienta Cloudcraft*

5.1. Trabajo realizado

Para la implementación de la nueva metodología, se agregaron nuevos servicios:

- Servidor de Staging
- Servidor de CI/CD

Y además, ciertos servicios existentes se comienzan a utilizar y conectar al flujo de trabajo de Intranet:

- Servidor de pruebas de GLPI, montado por I&T
- Servidor Mattermost, montado por I&T

En cuanto a código y desarrollo, se hizo lo siguiente:

- Se desarrollaron pruebas de aceptación para todas las funcionalidades existentes en el sistema, junto con algunas pruebas unitarias para validar otras partes del código.
- Se hacen cambios en el código para evitar la dependencia de APIs externas y de repositorios de código privados, para así permitir el trabajo remoto.
- Se hacen mejoras generales en la semántica de las vistas HTML para hacer el *testing* más fácil, y se hacen cambios en el código para quitar lógica desde ciertas vistas, la cual es difícil de probar en forma automatizada.
- Se añaden nuevas funcionalidades de monitoreo, *logging* y reporte de errores en Intranet, los cuales son accesibles por los administradores mediante la interfaz web, las cuales son:
  - Acceso a los *logs* existentes de la aplicación.
  - Nuevo *log* “intranet_audit”, que muestra eventos del sistema y errores específicos.
  - Menú para ver el estado de conexión con las APIs externas.

En cuanto a la configuración del proyecto, se hizo lo siguiente:

- Se instalan varios *plugins* nuevos en la aplicación, para hacer revisiones de seguridad.

- Se hace una actualización general de las dependencias y *plugins* del proyecto.
  - Se migra el proyecto desde Rails 5.0 a 5.1
  - Se actualizan gemas de uso general hasta su última versión
  - Se quitan gemas que no se están usando y gemas con problemas de seguridad
  - Se actualizan paquetes de RHEL y CentOS en servidores
- Se crean nuevas configuraciones para migrar la ejecución del sistema desde un servicio init.d de RHEL a un servicio *system.d*, resolviendo problemas de disponibilidad ante caídas.
- Se escribe un pequeño *script* de *self-test* o auto-prueba, para identificar rápidamente posibles errores de configuración antes de cada despliegue.

5.2. Desarrollo de *suite* de pruebas

La mayor apuesta de la metodología para asegurar la calidad del software es la construcción de pruebas en paralelo al desarrollo. Pero con tal de que estas pruebas sean efectivas para detectar problemas de regresión, es necesario crear pruebas para cubrir todo el desarrollo anterior de Intranet.

Para ello, se ha creado una estructura para las pruebas, y una *suite* que cubre todas las funcionalidades existentes del sistema, mediante pruebas de aceptación y pruebas unitarias.

En el directorio *spec/* de la aplicación, se incluyen archivos de configuración y la estructura por defecto dada por RSpec para ir agregando pruebas. Las pruebas de aceptación irán en *spec/features/* y las pruebas unitarias quedarán en el directorio que corresponda a su tipo (por ejemplo, */spec/models/* para pruebas de modelos.

Los archivos de configuración indican bibliotecas a cargar y comandos a ejecutar antes de las pruebas. Aquí se especifica el navegador web headless a utilizar y ciertas tareas de mantenimiento como son la de limpiar la base de datos de pruebas, limpiar cachés de archivos y cargar el revisor de cobertura de código (*simplecov*).

La estructura de las pruebas de aceptación se basará en la que es dada por el DSL de Capybara, con tres niveles de jerarquía:

- *Feature* (funcionalidad), donde se indica qué funcionalidad se probará con las pruebas escritas bajo sus contextos.
- *Context* (contexto), donde se indica qué tipo de usuario se simulará, o qué tipo de interacciones se harán en relación al objeto de la funcionalidad.

- *Scenario* (escenario), donde se describe la prueba como Historia de Usuario, con un sujeto y lo que se espera o no se espera que pueda hacer.

Cada archivo de extensión .rb en */spec/features/* contendrá un *Feature*, el cual tendrá un conjunto de *Contexts*, y de donde cada uno tendrá varias pruebas que comienzan con *Scenario*.

```
feature "Módulo de Documentos" do
  context "Listado y acceso a Documentos", js: true do
    initialize_unit_and_users

    scenario "Usuarios no verán secciones de documentos a los que no tengan acceso de lectura" do
      switch_to_user(student[:uid])
      expect(page).not_to have_content("Administración")
      expect(page).not_to have_link(href: users_path)
      visit users_path
      expect(page).to have_content("usted no tiene acceso a esta página")
    end
  end
end
```

Figura 5.2 – *Un ejemplo de la estructura de las pruebas en RSpec con Capybara. Se muestra el DSL de RSpec con Feature, Context y Scenario, y el DLC de Capybara con funciones especiales para las pruebas de aceptación. Fuente: Elaboración propia utilizando el editor Visual Studio Code*

Luego, las pruebas pueden ser ejecutadas mediante el comando *rspec*, el cual acepta argumentos de línea de comandos para entregar sus resultados en distintos formatos, y permite mostrar el tiempo que tardan las pruebas más lentas de la *suite* elegida.

5.2.1. Uso de VCR como herramienta de captura/repetición

Como se mencionó anteriormente, existen conexiones con APIs externas, que se encuentran en otros servidores del departamento: la API de LDAP, la API de Ruckus y la API de GLPI para el sistema de tickets. Para evitar que la *suite* de pruebas deba enviar peticiones a estas APIs en cada ejecución, se utiliza VCR para guardar un *caché* de todas las peticiones y respuestas efectuadas la primera vez que se ejecuta una prueba. Así, en las siguientes ejecuciones, se simularán las conexiones a APIs usando este caché, el cual quedará en el directorio ***spec/fixtures/vcr_casettes*** del VCS.

Los beneficios del uso de este *caché*, como permitir el trabajo fuera de los segmentos de red aceptados por las APIs, y el menor tiempo de ejecución de las pruebas, también viene con responsabilidades: en caso de haber cambios en el código, cambios en las pruebas o cambios en las APIs, se deberán eliminar los archivos del *caché* correspondientes, volver a generarlos y considerarlos en los cambios a subir en el VCS.

Finalmente, debe haber cuidado al enviar información sensible en éstas peticiones a APIs. En la configuración de las pruebas, se han añadido máscaras para ocultar esta información, y poder obtenerla desde otra fuente más segura, como es el archivo de secretos del proyecto (*secrets.yml*).

rails_helper.rb

```ruby
33 Capybara.server = :puma, {Silent: true}
34 Capybara.server_host = "localhost"
35 Capybara.server_port = 3001
36 Capybara.configure do |config|
37   config.always_include_port = true
38 end
39
40 Capybara::Webkit.configure do |config|
41   config.debug = false # True para mostrar de todo
42   config.timeout = 30
43   config.ignore_ssl_errors = true
44   config.allow_url "saml.inf.utfsm.cl"
45 end
46
47 VCR.configure do |config|
48   config.cassette_library_dir = "spec/fixtures/vcr_cassettes"
49   config.hook_into :webmock # or :fakeweb
50   config.allow_http_connections_when_no_cassette = true
51   config.ignore_localhost = true
52   config.filter_sensitive_data("<PWC>") { Rails.application.secret
53   config.filter_sensitive_data("<PW1>") { Rails.application.secret
54   config.filter_sensitive_data("<PW2>") { Rails.application.secret
55   config.filter_sensitive_data("<PW3>") { Rails.application.secret
56   config.filter_sensitive_data("<PW4>") { Rails.application.secret
57   config.filter_sensitive_data("<PW5>") { Rails.application.secret
58   config.filter_sensitive_data("<PWI1>") { Rails.application.secre
59   config.filter_sensitive_data("<PWI2>") { Rails.application.secre
60   config.filter_sensitive_data("<PWI3>") { Rails.application.secre
61   config.filter_sensitive_data("<PWI4>") { Rails.application.secre
62   config.filter_sensitive_data("<GLPIP>") { Rails.application.sec
63   config.filter_sensitive_data("<GLPIAT>") { Rails.application.sec
64   config.filter_sensitive_data("<GLPIH>") { Rails.application.sec
65   config.filter_sensitive_data("<RUCKUSU>") { Rails.application.se
66   config.filter_sensitive_data("<RUCKUSP>") { Rails.application.se
67 end
68
```

ticket_05_creation_teacher.yml

```yaml
1  ---
2  http_interactions:
3  - request:
4      method: get
5      uri: https://<GLPIH>/apirest.php/initSession
6      body:
7        encoding: US-ASCII
8        string: ''
9      headers:
10       App-Token:
11       - "<GLPIAT>"
12       Content-Type:
13       - application/json
14       Authorization:
15       - Basic aW50cmFuZXQ6Z3I0bmQ1dHIzNG0qIQ==
16   response:
17     status:
18       code: 200
19       message: OK
20     headers:
21       Date:
22       - Mon, 08 Oct 2018 19:41:25 GMT
23       Server:
24       - Apache/2.4.6 (Red Hat Enterprise Linux)
         OpenSSL/1.0.2k-fips PHP/7.2.10
25       X-Powered-By:
26       - PHP/7.2.10
27       Access-Control-Expose-Headers:
28       - content-type, content-range, accept-range
29       Expires:
30       - Mon, 26 Jul 1997 05:00:00 GMT
31       Cache-Control:
32       - no-store, no-cache, must-revalidate
33       Pragma:
34       - no-cache
35       Content-Length:
```

Figura 5.3 *– Captura de pantalla de configuración de VCR, junto con parte de un archivo de caché de VCR. Este caché incluye todas las peticiones y respuestas a APIs, y puede enmascarar contenido sensible*

5.3. Despliegue de servidor de pruebas: Servidor de puesta en escena o *staging*

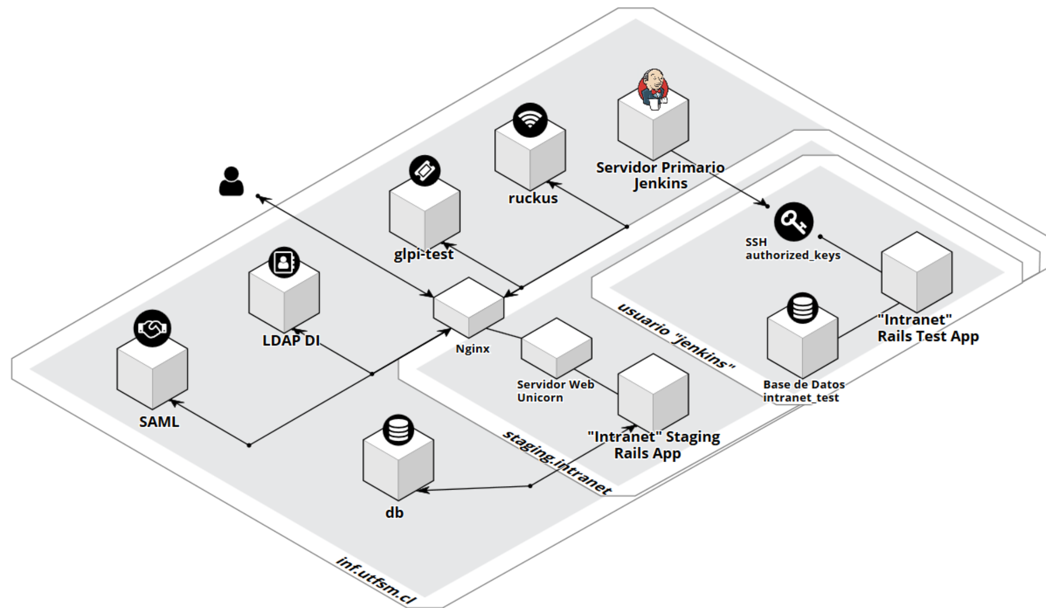


Figura 5.4 *– Diagrama de alto nivel mostrando cómo se configuró el servidor staging. Éste incluye una versión funcional de Intranet, y además ejecutará las órdenes de Jenkins. Fuente: Elaboración propia utilizando la herramienta Cloudcraft*

Existen varios factores que llevan a la decisión de desplegar un nuevo servidor para las pruebas:

- Realizar pruebas en las estaciones de trabajo de los desarrolladores, bajo un ambiente de desarrollo, no se apega totalmente a cómo se comporta la aplicación web en la realidad. Es necesario, por lo tanto, hacer pruebas en un ambiente lo más parecido posible al ambiente de producción.
- Mediante la instalación de un software de Integración Continua, se puede configurar para que este servidor pueda descargar automáticamente los cambios de código desde el repositorio del VCS, ejecutar las pruebas automatizadas y desplegar una nueva versión del proyecto en un ambiente de pruebas cercano al de producción, al que se llamará "ambiente de *staging*".
- Se han dado casos donde es muy útil poder realizar pruebas con los clientes o los *stakeholders* de ciertos módulos del proyecto Intranet, con tal de obtener retroalimentación y encontrar posibles problemas antes del despliegue final del módulo. Al tener un servidor de *staging* con los últimos cambios del código, se

puede obtener información gracias al uso directo del sistema por parte de los usuarios.

Ante esto, se ha realizado el despliegue de un servidor como ambiente de *staging* de Intranet, el cual se encuentra actualmente en el dominio **staging.intranet.inf.utfsm.cl**. Por el lado de los desarrolladores, permitirá la ejecución de pruebas automatizadas, el uso de Integración Continua y Despliegue Continuo, y la realización de pruebas manuales. Por el lado de los usuarios, se podrán probar las nuevas funcionalidades del sistema antes de su lanzamiento oficial.

Este servidor está alojado en el Data Center del Departamento de Informática, y tiene características muy similares al servidor de producción de Intranet, con la diferencia de que el sistema operativo es CentOS 7 en vez de RHEL 6.

Figura 5.5 – *Captura de pantalla de la aplicación web Intranet en el servidor staging. Es la misma aplicación, pero con una base de datos diferente, y mensajes para indicar a los usuarios que esta no es la plataforma principal. Además, se muestra información del último cambio desplegado. Fuente: Elaboración propia*

El servidor, además de poder correr pruebas y alojar un Intranet provisorio, ha permitido experimentar con distintos cambios en las configuraciones, establecer nuevas conexiones y validar el uso de las herramientas elegidas.

5.4. Despliegue de servidor de CI/CD: Jenkins

En una nueva máquina virtual del Data Center de I&T se ha instalado un servidor de CI/CD, en el cual se configuró un servidor primario de Jenkins. Se decidió utilizar Docker y la imagen oficial de Jenkins de Docker para desplegar este servicio, ya que la imagen no tiene limitaciones ni problemas de configuración respecto a una instalación estándar (usando un gestor de paquetes de GNU/Linux) como ocurre con otras imágenes de Docker de servicios *self-hosted*.

Por ahora, dado a su uso interno en el DI, se ha configurado para servir la interfaz web de Jenkins en HTTPS utilizando un certificado SSL autofirmado. Esto asegura la encriptación de datos sensibles, pero genera mensajes de error en el navegador. Queda como trabajo futuro analizar si se le dará una IP pública y un dominio inf.utfsm.cl, para así poder optar a un certificado de confianza que no produzca este problema

Figura 5.6 – *Captura de pantalla de Jenkins instalado, donde el proyecto Intranet se ha configurado con un Multibranch Pipeline. Se puede ver información sobre los últimos builds y el estado de los agentes. Fuente: Elaboración propia*

El proyecto Intranet se ha configurado para revisar cambios en el código utilizando *polling*, o sea, cada cierto tiempo se revisa el repositorio BitBucket donde se aloja del código Intranet, y se analiza si hubo cambios. Se intentó utilizar un WebHook a BitBucket, pero el uso de un certificado SSL autofirmado hace que BitBucket rechace la conexión.

La configuración de la *pipeline* de pruebas y despliegue del proyecto se define en el archivo "*Jenkinsfile*", el cual utiliza la sintaxis del lenguaje de programación Groovy, y en la cual se puede agregar de forma declarativa los pasos a seguir para la contrucción o *build*.

Algo que se debe tomar en cuenta es que la descarga del código desde el SCM (el cual es un paso de la *pipeline*) no es suficiente para poder obtener una versión funcional de Intranet. Es necesario también agregar al directorio del proyecto una serie de archivos de carácter sensible: credenciales, contraseñas y llaves de API que, por seguridad, no deben guardarse en el repositorio. Afortunadamente Jenkins tiene un plugin para manejar credenciales, y luego de agregar éstas, se puede configurar en el *Jenkinsfile* a que se utilicen las credenciales para ciertos comandos.

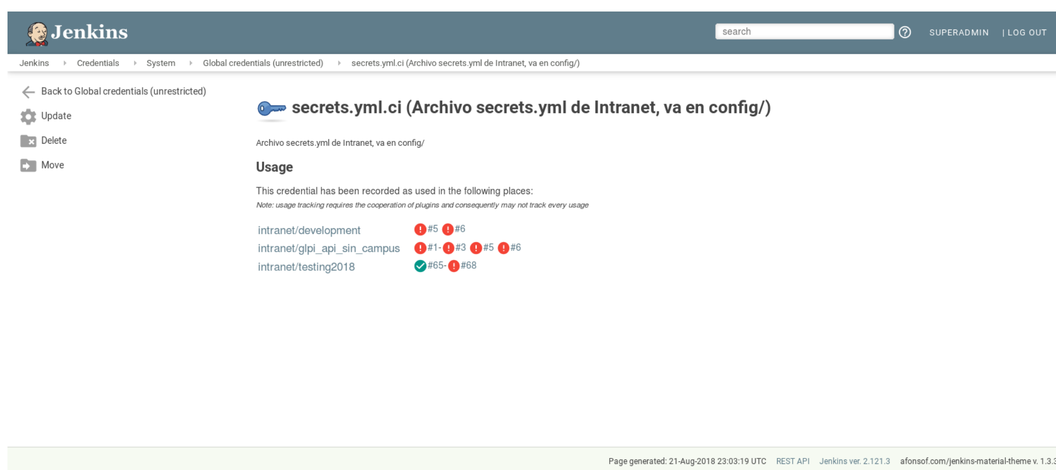


Figura 5.7 – *Captura de pantalla de Jenkins, donde se muestra el uso del plugin de credenciales. Fuente: Elaboración propia*

El servidor de Staging se ha configurado para ser además el servidor "réplica" de Jenkins, el cual se conectará mediante una sesión SSH a ejecutar los comandos indicados en el *Jenkinsfile*, bajo el usuario "jenkins". Se eligió así debido a que de esta forma, no es necesario instalar de nuevo todas las dependencias del proyecto Intranet, asumiendo que la versión actual de Intranet en Staging ya va a tener todas o la mayorías de las dependencias satisfechas. Además, usando SSH no se requerirá instalar un agente de Jenkins, ya que el servidor "primario" se encarga de todo.

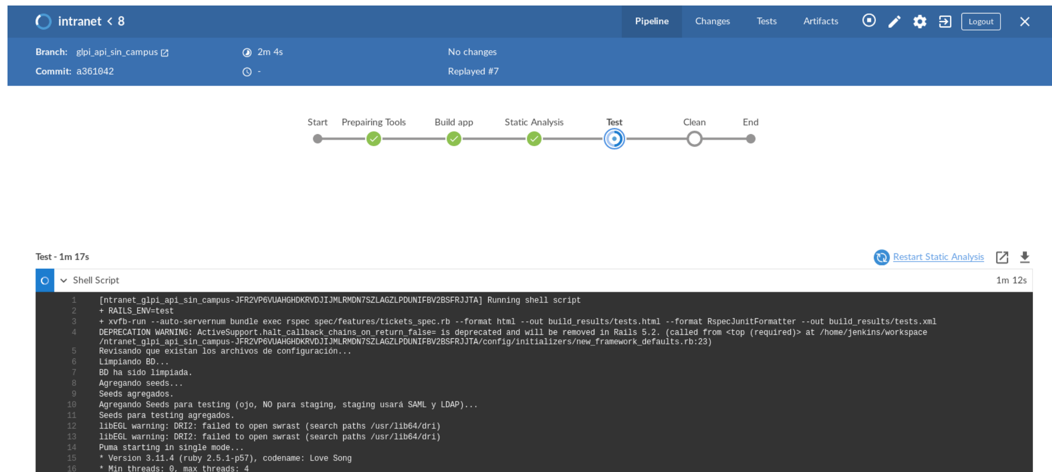


Figura 5.8 *– Captura de pantalla de la interfaz BlueOcean de Jenkins, donde se está ejecutando una pipeline. Fuente: Elaboración propia*

En la interfaz BlueOcean se pueden ver las ejecuciones de *builds*, que en este caso, se activan al detectar nuevas versiones del código en BitBucket. Ya que el proyecto Rails se basa en código interpretado y no compilado, este *build* no va a construir ningún archivo ejecutable o paquete, pero sí entregará resultados de las pruebas y ejecutará los comandos de despliegue.

Si bien Jenkins está basado en Hudson y es primordialmente como servidor CI/CD para proyectos en Java, la biblioteca RSpec usado para ejecutar pruebas de Ruby en el proyecto permite exportar los resultados en el formato de resultados de Junit, el cual puede ser interpretado por Jenkins y otros sistemas de CI/CD.

5.5. Integraciones y configuraciones

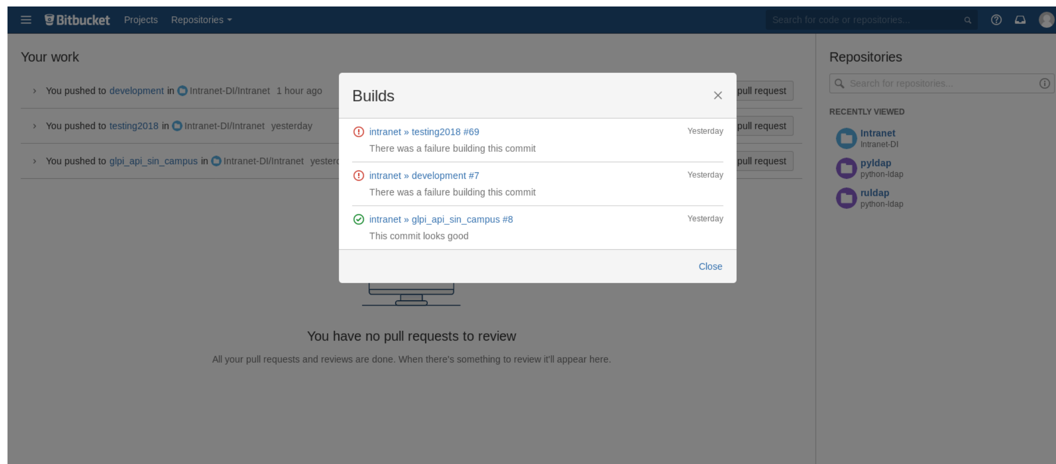


Figura 5.9 – *Captura de pantalla de VCS BitBucket, el cual se ha configurado para recibir información de Jenkins. Se muestra un aviso por cada ejecución de la pipeline de CI, y su resultado. Fuente: Elaboración propia*

Se han realizado las siguientes configuraciones para integrar los sistemas implementados:

- Jenkins revisa cada 15 minutos si existe un cambio en ciertas ramas del proyecto Intranet. Si es así, se activa la *pipeline* y se muestra el resultado en la página principal de BitBucket.
- Se ha configurado Jenkins para enviar información del resultado de cada *build* a BitBucket.
- Se configura para que luego de cada *build* el resultado se envíe como mensaje al canal Intranet de Mattermost.
- Se configura el proyecto Intranet en Jira, para que se puedan asociar tareas a *commits* específicos del SCM (BitBucket) y para poder aprovechar el *kanban board* de éste.

El archivo *Jenkinsfile* que define el pipeline de Intranet funciona bajo los siguientes pasos:

1. **Start:** Se obtiene la última versión de la rama SCM con cambios.
2. **Checking tools:** Se revisa que el *runtime* de Ruby esté instalado correctamente.
3. **Build app:**

  - Se obtiene las credenciales desde Jenkins y se añaden al directorio de trabajo.
  - Se revisa que estén todas las gemas (bibliotecas de Ruby), y se instalan las faltantes.
  - Se prepara la base de datos de prueba.

4. **Static analysis:**
   - Se revisa el código por posibles vulnerabilidades de seguridad, y se genera un reporte (como un artefacto del *build* de Jenkins).
   - Se revisa que no hayan gemas con vulnerabilidades de seguridad, y se genera un reporte.
5. **Test:** Se ejecutan todas las pruebas del directorio *spec/*
6. **Deploy:** Si no hubo problemas hasta este paso, se hace el despliegue al servidor de *staging* si y solo si la rama de SCM actual es la rama *deployment*.
7. **Post:** Se envían notificaciones del resultado de este *build* por Mattermost (y por ende, por correo).

5.6. Conclusiones

Se han implementado las herramientas, configuraciones y código necesarios para poder trabajar utilizando el Plan de Calidad confeccionado, con un flujo de trabajo que integra DevOps. Se ha descubierto que la mayor dificultad a la hora de implementar esta nueva forma de trabajo es la configuración inicial de todas las partes del proceso: herramientas como Jenkins tienen multitudes de plugins y posibles configuraciones, pero las que requiere el proyecto en sí va a ser un conjunto reducido.

Lo mismo ocurre con las pruebas, que van a requerir nuevas dependencias en el sistema (como el navegador Webkit) y el esfuerzo de entender los DSLs asociados. Pero una vez que todo esté configurado y que se tengan algunas pruebas realizadas, continuar con el trabajo se hace más fácil. Las únicas excepciones han sido el inicio de sesión por SAML y ciertos casos de excepción del sistema, donde se ha tenido que mantener código sin probar debido a dificultades para probar éstos sin tener que conectarse a sitios externos en cada ejecución.

Una observación importante es que, a medida que se realizaba la implementación, han surgido nuevas necesidades o obligaciones para quienes mantengan esos sistemas, en particular, mantener las dependencias y plugins actualizados, y estar al tanto de vulnerabilidades de seguridad. A lo largo del trabajo de esta memoria, nuevas versiones de Jenkins y de varios *plugins* han aparecido, y afortunadamente, actualizar éstos no es difícil, pero siempre existe el riesgo de que ciertas actualizaciones dejen incompatibles a

otras partes del proceso (por ejemplo, las conexiones con el repositorio VCS o la lectura del archivo *Jenkinsfile*). Este mantenimiento de la infraestructura DevOps debe ser agregado al Plan de Calidad, y ser parte de la responsabilidad del equipo.

Capítulo 6

Validación de la Solución

En esta sección se presentan los resultados de la implementación del Plan de Calidad. La metodología y las herramientas configuradas fueron puestas a prueba en la construcción de un nuevo módulo de *tickets* de soporte para Intranet. Los beneficios en la calidad del software y la satisfacción de los clientes fue evidente, y los resultados del trabajo con los nuevos estudiantes en práctica van a indicar cómo afinar el Plan de Calidad para su uso futuro.

6.1. Caso de evaluación: Sistema de *tickets* en Intranet

Para la validación de la solución implementada, se fueron aplicando progresivamente la metodología de trabajo en el período de Enero de 2018 hasta Julio de 2018. El equipo de trabajo de Intranet fue conformado por un ayudante administrativo y dos estudiantes en práctica, y el trabajo principal fue la implementación de un nuevo módulo de Intranet: una interfaz para el sistema de *tickets* de soporte del Departamento de Informática.

Hasta antes de la implementación, el equipo de I&T entregaba soporte a la comunidad del DI principalmente mediante llamadas telefónicas, visitas presenciales y correo electrónico, pero esto no tendría que haber sido así: existe un sistema de *tickets* de soporte en el dominio *tickets.inf.utfsm.cl*, el cual se basa en GLPI, una plataforma web de código abierto. Lamentablemente, muy pocos miembros del D.I. la utilizan debido a lo complicado que es su uso, y a la confusión que produce el hecho de que GLPI tenga tantas funcionalidades.

Por lo tanto, se decidió que como tarea para dos de los estudiantes en práctica, se debe realizar un mantenimiento de GLPI, que incluye actualizar sus componentes, limpiar

plugins, limpiar código heredado o *legacy*, y luego utilizar la API REST de ésta para desarrollar una nueva interfaz de creación de *tickets*. La idea es que los profesores y funcionarios del departamento puedan entrar a Intranet y solicitar ayuda a I&T creando *tickets* de soporte, con una interfaz mucho más simple y fácil de usar, y que Intranet almacene y lea éstos desde GLPI. Así, los funcionarios de I&T pueden seguir usando las funcionalidades completas de GLPI, y el resto de los usuarios puede obtener más motivación a crear *tickets* en vez de utilizar otros medios.

6.2. Resultados y métricas del despliegue

El equipo de Intranet logró terminar el módulo de Tickets, y éste ya está en producción con la versión 1.0.0 de Intranet. Los registros indican que se crearon 26 Tickets en el mes de Agosto de 2018 y 19 Tickets en el mes de Septiembre (número es menor debido a la semana de vacaciones de "fiestas patrias" de Chile) sin encontrarse errores por parte de los usuarios. Entrevistas presenciales a funcionarios indican que el sistema funciona perfectamente, y que la retroalimentación recibida a lo largo del año dió frutos con tal de generar un sistema que sea bien usado por los miembros del departamento.

Como se ha indicado en los alcances de la memoria y en el plan de calidad, se utilizarán los resultados de los *builds* como métricas iniciales. La *build* que presenta el módulo de Tickets para su uso en la comunidad es el marcado como la versión 1.0.0.

En el paso a producción de esta versión se encontró inmediatamente un defecto en el software, el cual ocurrió debido a un error de configuración que no ocurrió en el servidor de puesta en escena. Se creó el script de *self-test* para descubrir y evitar que defectos de configuración como éstos vuelvan a afectar la experiencia de los usuarios. Aparte de este defecto, ningún otro error ha aparecido, y se considera el lanzamiento como un éxito.

Luego del lanzamiento de la versión 1.0.0, ciertos cambios produjeron una regresión de un desarrollo anterior, afectando en forma menor la vista de Tickets para un usuario. Se utilizaron los medios de comunicación mencionados en el plan, se arregló el problema y se creó una prueba unitaria para evitar que esta regresión vuelva a ocurrir.

Intranet termina finalmente con **123 pruebas o *specs***, las cuales todas están aprobadas en la última versión (1.1.0) que va a desplegarse en producción en un futuro cercano. La cobertura de las pruebas es de **76.91 %** según *simplecov*, y la ejecución de todas las pruebas tarda 4 minutos 45 segundos en una estación de trabajo, y 7 minutos 24 segundos en el servidor de *staging* (medido desde Jenkins).

6.3. Resultados de la aplicación de la metodología

La implementación progresiva de la metodología ha resultado en varios beneficios para el proyecto, los cuales se mencionan a continuación.

Descubrimiento inmediato de defectos triviales de regresión

El uso de una *suite* de pruebas que abarca la mayor parte del sistema ha permitido que se descubran rápidamente errores de regresión, producidos por cambios en código usado por otros módulos que anteriormente funcionaban. Anteriormente estos errores podrían haber pasado desapercibidos debido a que hubiesen requerido pruebas manuales, las cuales en general no abarcan todas las funcionalidades.

Mejora sustancial en velocidad para arreglar defectos descubiertos

Errores pequeños pero de gravedad han sido descubiertos y resueltos antes del paso al servidor de producción, ahorrando tiempo considerable que anteriormente se hubiese gastado en un despliegue de software con errores, y evitando pruebas manuales que podrían o no haber llegado a encontrar el error.

Por ejemplo, la falta de ciertos archivos de configuración normalmente no hace que un despliegue falle, y no se encuentran los errores hasta que se llega a una página que depende de esa configuración. Gracias a los *scripts* de *self-test* y la suite de pruebas, se pueden encontrar estos errores rápidamente y resolverlos.

Descubrimiento de código no utilizado y deprecado

El uso de herramientas de cobertura de código como *simplecov* han revelado gran parte de código que no se utiliza, o que se utilizó en algún momento y que luego quedó obsoleto. La eliminación de este código mejora el orden y la calidad del código, junto con mejorar las métricas de cobertura.

Benchmarking de tiempos de respuesta y corrección de cuellos de botella

Un caso de uso inesperado para las pruebas es el de *benchmarking* para ciertas operaciones del sistema. Hubo una ocasión en especial donde se consideró que cierta historia de usuario relacionada a los Tickets tardaba demasiado.

Gracias a que esta historia ya estaba automatizada con una prueba de aceptación, se pudieron hacer distintas pruebas y experimentos para ver qué operaciones y funciones son las que hacían que ésta se tardara tanto. Finalmente, se pudieron encontrar

los cuellos de botella, corregirlos y mejorar el tiempo de respuesta de las operaciones considerablemente.

De la misma forma, otros cuellos de botella menores se han descubierto, y se han podido mitigar o arreglar para mejorar la experiencia de uso del sistema.

Baja sustancial de tiempo requerido para pruebas

El hecho de poder ejecutar pruebas automatizadas entrega mucha más tranquilidad a la hora de probar Historias de Usuario basadas en casos de uso alternativos y de error. Así, antes de publicar nuevos cambios al servidor de producción, las pruebas manuales no requieren hacer una revisión exhaustiva. Así, se ha generado mucha más confianza en que no existan errores antes de un despliegue.

Mejor comunicación con los usuarios finales y retroalimentación

La implementación de un servidor de puesta en escena o *staging* ha permitido tener más interacciones con los clientes. En específico, varias versiones del sistema de Tickets han podido ser probadas por profesores y ayudantes del Departamento de Informática antes de su despliegue en producción, obteniendo retroalimentación de gran valor para lograr cumplir los requerimientos del módulo en forma exitosa.

6.4. Observaciones en el equipo de trabajo

Si bien los beneficios del plan de calidad en los resultados del proyecto son notables, existen varias conductas y observaciones respecto a la aplicación de éste y su metodología con los nuevos miembros del equipo (practicantes y ayudantes).

La primera observación importante es que los nuevos miembros van a requerir un período de entrenamiento si se observa una falta en los conocimientos mínimos para aplicar el plan, como son el uso de VCS, la línea de comandos, el trabajo con APIs y la construcción de pruebas. Muchos de los estudiantes en práctica son estudiantes de tercer o cuatro año (en una carrera de 5 años y medio, o 6 años) con conocimientos sólidos de programación, pero con poca experiencia en proyectos reales o que tienen un uso por parte de usuarios no técnicos.

De la misma forma, los estudiantes en práctica y ayudantes administrativos en general no conocen las buenas prácticas y las consideraciones que se deben tener al trabajar en proyectos con otras personas y con código que ha sido heredado de otros estudiantes. La mayoría de las experiencias de estos estudiantes son proyectos personales o proyectos realizados desde cero, donde hay una libre elección sobre las políticas y estilos de

desarrollo.

Otra observación importante es que los estudiantes que se unan al equipo de programación están dispuestos a aprender y a seguir reglamentos y políticas, pero por su falta de experiencia se requiere que existan ejemplos, documentación o una infraestructura que guíe su trabajo. Esto es especialmente importante para las elaboración de pruebas: la mayor barrera es entender cómo realizarlas, pero cuando se tenían todas las dependencias instaladas, los sistemas de CI/CD configurados y una *suite* de pruebas ya preparada, crear nuevas pruebas y utilizar las herramientas se hace mucho más fácil.

Finalmente, se debe mencionar que los nuevos miembros del equipo siempre tienen gran motivación por programar cosas nuevas, pero el poder entender otras partes existentes del sistema se les hace difícil y menos motivante. La búsqueda de que “todos conozcan todo sobre el sistema” y una “responsabilidad compartida” se hace difícil, y solo se logra luego de mucho tiempo de trabajo en el proyecto. Para estudiantes en práctica que no continúan como ayudantes, esto podría nunca ocurrir.

6.5. Conclusiones de la validación de la solución

Se ha realizado una primera validación del Plan de Calidad a lo largo del desarrollo de una nueva funcionalidad de Intranet. Si bien el alcance de este caso de evaluación es limitado, se ha podido obtener información importante sobre los beneficios y limitaciones de éste.

Se ha logrado aumentar el compromiso y la motivación ante el uso de actividades para el aseguramiento de calidad: Pruebas automatizadas e Integración Continua para reducir defectos de software, y medios de comunicación, documentación y despliegue de versiones de prueba en servidor *staging* para la validación de requerimientos. Se comienzan a utilizar herramientas modernas para la organización y comunicación de tareas, y se ha reducido el tiempo requerido para la identificación e implementación de parches y nuevas funcionalidades.

Pero como la mayoría de los planes aplicados a proyectos de software, existen problemas inesperados relacionados a las personas. El entrenamiento de los miembros del equipo es simple de resolver, pero el compromiso de tomar responsabilidad, seguir reglamentos y no intentar “saltarse pasos” en los procesos es un tema que entra en temas de autoridad y resolución de conflictos, y va a depender mucho de las motivaciones de las personas en sí. Un Plan de Calidad puede dar directrices de cómo hacer las cosas, pero hacer que todos sigan este plan es un desafío permanente, en especial en un equipo de trabajo de poca experiencia que trabaja con sistemas poco críticos.

Capítulo 7

Conclusiones

7.1. Conclusiones generales

Esta memoria ha presentado el planteamiento e implementación de un Plan de Calidad para el software Intranet, el cual **ha generado un impacto positivo en el proceso de desarrollo**, al instaurar un flujo de trabajo que adopta la Integración y Entrega Continua (CI/CD) con tal de obtener un aseguramiento de la calidad en cada nueva versión. Esto se logra mediante el uso de pruebas automatizadas, métricas de coberturas de pruebas, un servidor CI/CD y un ambiente de puesta en escena para ofrecer oportunidades de retroalimentación. Además, nuevas herramientas de diagnóstico para antes y después del despliegue han ayudado a encontrar (y corregir) errores triviales y de mala configuración.

Además, el uso de este flujo de trabajo ha llevado a que los miembros del equipo **adopten una cultura de colaboración** mediante documentación, integraciones rápidas de código y un proceso de desarrollo más ordenado. El solo hecho de tener procesos de desarrollo definidos y una infraestructura para automatizarlos, invita a los nuevos miembros del equipo a trabajar bajo una perspectiva de mantener la calidad del sistema en todo momento. Así, se resuelve el problema de motivación que normalmente evita que equipos de desarrolladores jóvenes adopten prácticas modernas de desarrollo basadas en *testing* e Integración Continua.

7.2. Cumplimiento de objetivos

A continuación se presentan los objetivos de esta memoria, cómo se cumplieron y conclusiones respecto a su realización.

1. Conocer las características del equipo de desarrollo de la Intranet, y las circunstancias de su trabajo

   Se ha logrado hacer una reseña de las características del proyecto, de las cuales se destacan: el uso de un equipo de trabajo pequeño y conformado por estudiantes de carreras de Informática con alta rotativa, una desarrollo iterativo e incremental basado en módulos con distintas funcionalidades, y un ambiente de trabajo que favorece el uso de herramientas gratuitas y auto-gestionadas en servidores GNU/Linux.

   A partir de esta reseña, se descubre que **la característica que más produce carencias en la calidad de Intranet es la falta de una metodología de desarrollo bien definida**, y por ende, un uso insuficiente de actividades y herramientas que permitan evitar defectos de software a través del desarrollo.

2. Comparar metodologías de desarrollo y herramientas técnicas que puedan formar un plan de calidad para el software Intranet

   Luego de un análisis de qué implica la calidad de un software en la literatura, se plantea la creación de un Plan de Calidad, y se exploran distintas metodologías, actividades y herramientas de desarrollo que puedan formar este plan y adaptarse al proyecto.

   A partir de ello, se concluye que **se debe crear una metodología y un flujo de trabajo basados en el agilismo, el desarrollo de código abierto y DevOps**, con un énfasis en mantener una buena comunicación, organización, y documentación del proyecto. Además, se debe tener código que permita el trabajo remoto, y que siempre pueda ser probable o *testable* mediante una *suite* de pruebas de aceptación y de pruebas unitarias. El proceso de desarrollo de Intranet, por ende, debe integrar herramientas de comunicación y colaboración (correo electrónico, suite de software Atlassian, mensajero instantáneo Mattermost) y una infraestructura de Integración Continua y Despliegue Continuo (Jenkins CI, y servidor Staging).

3. Implementar el Plan de Calidad en el ambiente de desarrollo y despliegue del software Intranet

   Se ha logrado la implementación del plan utilizando las herramientas dadas por la Unidad de I&T del departamento, junto con el desarrollo de pruebas y la configuración de los componentes del flujo de trabajo.

   A lo largo de esta implementación se descubre que **existe una gran cantidad de bibliotecas, *plugins* y software**, con bastante documentación y madurez, **que permiten la creación de flujos de trabajo con un aseguramiento de calidad continuo**. Si bien muchas de las herramientas son específicas para ciertos lenguajes de programación, la mayoría de éstas tienen equivalentes para todo tipo de proyectos, o incluso son agnósticas ante lenguajes. Así, se tiene que **el mayor reto para implementar estos flujos no es la viabilidad, sino la instalación, configuración y adaptación inicial**, junto con la **curva de**

aprendizaje de uso y mantenimiento por parte del equipo de desarrollo, lo cual se puede mitigar con buena documentación y con una infraestructura preparada a priori.

4. Proponer recomendaciones para otros proyectos de software del Departamento de Informática

   El plan ha logrado una implementación exitosa con pocos contratiempos, y **se puede concluir que éste puede ser aplicado en otros proyectos de I&T que tengan característica similares**. Se ha preparado en los anexos de este documento una versión genérica del plan, pensada en ser una guía para nuevos miembros de I&T, y con consideraciones especiales para el proyecto Intranet.

   Respecto a otros proyectos que ya estén en desarrollo, es importante destacar la opción de aplicar el Plan de Calidad propuesto por sobre el de aplicar re-ingeniería o re-construír el sistema, con tal de permitir el aseguramiento de calidad. Si bien es necesario hacer un análisis según sea el proyecto, Intranet logra adaptarse muy bien al flujo de trabajo propuesto, gracias a que como base utiliza un *framework* suficientemente modular, maduro y compatible con todo tipo de *plugins* y herramientas externas para pruebas automáticas e Integración Continua.

   Si bien se tuvo que realizar un esfuerzo importante para crear las pruebas, se puede concluir que **es posible aplicar el Plan de Calidad presentado en esta memoria no solo en proyectos nuevos, si no que también en proyectos de I&T que pueden estar en desarrollo, mientras utilicen lenguajes de programación y *frameworks* de suficiente madurez**. Mientras antes se puedan tomar medidas para evitar defectos de software, menor será el costo de su implementación, y se podrán aprovechar beneficios similares a los que tiene Intranet en este momento.

7.3. Sobre el desarrollo de las pruebas

- Las pruebas fueron realizadas desde una perspectiva *top-down*, donde se hicieron pruebas de aceptación a partir de las historias de usuario, y luego se hicieron pruebas unitarias para vistas y funciones que no fueron cubiertas por las pruebas de aceptación. **Esta metodología de pruebas logra adaptarse exitosamente al proyecto** dado a que no es difícil de entender para los miembros del equipo, permitiendo crear pruebas que son básicamente "simulaciones" de las interacciones del usuario, las cuales son similares a las pruebas manuales que acostumbran los estudiantes en sus proyectos de la universidad. Y además, **permite cubrir todas las partes del código que se utilizan en una interacción**, lo que les permite servir también como pruebas de integración, mientras éstas sean suficientemente exhaustivas respecto a los procesos del sistema.

  Sin embargo, se debe tomar en cuenta que **el costo de ejecutar las pruebas de**

aceptación es bastante alto respecto a las pruebas unitarias, debido a que se utiliza un navegador (que a pesar de ser *headless*, consume bastantes recursos) y una base de datos de prueba. Según el tamaño del proyecto, **es necesario tomar en cuenta el costo de tener todos los casos de excepción y los casos alternativos como pruebas de aceptación**, ya que en el caso de Intranet, tener esta estructura hace que ejecutar la *suite* de pruebas completa tarde varios minutos, lo cual aumenta si los recursos de la máquina son más bajos, y pruebas lentas pueden hacer que el servidor de Integración Continua se tarde demasiado en aprobar y desplegar nuevos cambios. Una opción que se propone es **reemplazar algunas pruebas de aceptación por pruebas unitarias equivalentes**, lo cual se tendrá como trabajo a futuro.

- **Las pruebas de aceptación**, al estar basadas en RSpec y Capybara, utilizan un DSL para que su lectura sea simple y fácil de adaptar a las historias de usuario del sistema. Sin embargo, el hecho de tener que interactuar con un navegador y tener páginas dinámicas (con Javascript, por ejemplo, para autocompletar formularios o hacer aparecer y desaparecer elementos) trae como consecuencia **tener que tratar con casos especiales de ciertos *plugins* de Javascript, y con las limitaciones del navegador *headless*.**

  Así, el desarrollo de las pruebas comienza con una alta dificultad, pero a medida que se van creando más pruebas, se va aprendiendo más como funciona el DSL, y las técnicas para abarcar casos especiales se pueden reutilizar. Por ello, **el hecho de tener ejemplos, funciones de ayuda y pruebas completas ya desarrolladas anteriormente ayuda mucho a nuevos desarrolladores a adaptarse a la construcción de pruebas.**

- Como otro punto importante, el elemento que no permitió a las pruebas alcanzar un porcentaje de cobertura total o cercano a 100 % fue la funcionalidad de entrar al sistema usando *Single Sign On* con SAML. Como uno de los cambios en el código para permitir el trabajo remoto, se definió un "Modo no SAML" para los ambientes de desarrollo y de pruebas de Intranet, donde los miembros del equipo podrán entrar al sistema como cualquier usuario local, sin tener que ingresar contraseñas. Esto permite que las pruebas manuales y las pruebas automatizadas que impliquen varios tipos de usuario sean más simples. Sin embargo, esto hace que las pruebas no incluyan el código que maneja las sesiones con SAML (que se utilizan en los ambientes de puesta en escena y producción).

  Se probaron varias configuraciones y plugins para poder ejecutar ciertas pruebas con SAML activado, pero finalmente **no se pudo adaptar el sistema para que se puedan probar los inicios de sesión con SAML sin tener que conectarse al servidor *saml.inf.utfsm.cl* en cada ocasión**. Este servidor solo permite conexiones desde direcciones IP específicas, y lo ideal es que se pudiese tener un caché de peticiones y respuestas de éste, o una forma de simularlas, **con tal de poder mantener el código de Intranet siempre auto-probable y**

sin dependencias de red. Por lo tanto, se dejó el código relacionado fuera de las pruebas, y por ende, el porcentaje de cobertura es cercano a 75 %.

7.4. Trabajo a futuro

Si bien se considera la adopción del Plan de Calidad como un éxito, existen varios temas pendientes que deben ser mencionados.

El trabajo a futuro más importante es **analizar y modificar el Plan de Calidad a través del tiempo**, por ejemplo, en base a la aparición de nuevos miembros del equipo, nuevos requerimientos, y nuevos proyectos de I&T. El alcance de la validación de este plan se ve limitado por las circunstancias de Intranet: se ha adaptado la metodología en forma progresiva para la implementación de una funcionalidad no muy compleja, con un equipo de trabajo pequeño, un proyecto de baja envergadura y, como característica importante, una fecha tope flexible. Podrían haber circunstancias donde nuevos miembros del equipo tienen aún menos experiencia que los actuales, y que se deba desarrollar bajo mayor presión de tiempo. En este caso, sería necesario hacer un seguimiento de la efectividad del plan. Otro caso podría ser el de implementar una funcionalidad compleja, lo cual puede significar mayor trabajo en las fases de análisis y diseño de las iteraciones, y se debe modificar el plan para cubrir estos casos. Por lo tanto, así como el plan indica que la documentación del sistema debe ser parte del proceso continuo de desarrollo, **el Plan de Calidad en sí será parte de la documentación, y por ende, debe ser desarrollado continuamente.**

El otro tema importante es el mantenimiento y mejora de la infraestructura tecnológica que mantiene al flujo de trabajo actual. **Se debe hacer el mantenimiento de los servidores y herramientas (como Jenkins CI y el servidor de Staging)** mediante actualizaciones de seguridad, actualización de *plugins*, renovación de certificados de seguridad, y otras tareas de administración de sistemas. En el Plan de Calidad se define que esto será una responsabilidad compartida del equipo de trabajo.

Finalmente, **se debe hacer mantenimiento y mejoras en la *suite* de pruebas automatizadas**, siendo éstas el elemento principal para el aseguramiento de calidad del sistema. Como se mencionó anteriormente, las pruebas aún no alcanzan el 100 % de cobertura, y se deben analizar formas de probar los casos de excepción no cubiertos (como los errores de conexión) y el inicio de sesión con Single Sign On por SAML. Además, el tiempo de ejecución es alto y se debe analizar el poder separar ciertas pruebas de aceptación por un conjunto de pruebas unitarias. Este mantenimiento podría ayudar a fortalecer la motivación de los miembros del equipo a ejecutar la *suite* de pruebas en cualquier momento, y así identificar errores rápidamente.

Referencias

Abrahamsson, P., Salo, O., Ronkainen, J., y Warsta, J. (2017). Agile software development methods: Review and analysis. *arXiv preprint arXiv:1709.08439*.

Bächle, M., y Kirchberg, P. (2007). Ruby on rails. *IEEE software*, *24*(6).

Bradburne, A. (2016). Testing. En *Rails 5 revealed* (pp. 35–42). Springer.

Ebert, C., Gallardo, G., Hernantes, J., y Serrano, N. (2016). Devops. *IEEE Software*, *33*(3), 94–100.

Fowler, M., y Foemmel, M. (2012). Continuous integration, 2006. *URL http://martinfowler. com/articles/continuousIntegration. html*.

GitLab. (2018a). *Gitlab continuous integration and delivery | gitlab.* `https://about.gitlab.com/product/continuous-integration/`. (Recuperado el 9 de Noviembre de 2018)

GitLab. (2018b). *Gitlab pricing | gitlab.* `https://about.gitlab.com/pricing/#self-managed`. (Recuperado el 9 de Noviembre de 2018)

JetBrains. (2018). *Licensing policy - teamcity 10.x and 2017.x documentation - confluence.* `https://confluence.jetbrains.com/display/TCD10/Licensing+Policy`. (Recuperado el 9 de Noviembre de 2018)

LaFrance, A. (2002). *Here comes the slack backlash.* `https://web.archive.org/web/20181026023510/https://www.theatlantic.com/technology/archive/2016/04/i-love-slack-i-hate-slack-i-love-slack/479145/`. (Recuperado el 23 de Octubre de 2018)

Lewis, W. E. (2000). *Software testing and continuous quality improvement.* CRC Press.

Martin, R. C. (2002). *Agile software development: principles, patterns, and practices.* Prentice Hall.

Matharu, G. S., Mishra, A., Singh, H., y Upadhyay, P. (2015). Empirical study of agile software development methodologies: A comparative analysis. *ACM SIGSOFT Software Engineering Notes*, *40*(1), 1–6.

Meyer, M. (2014). Continuous integration and its tools. *IEEE software*, *31*(3), 14–16.

Mockus, A., Fielding, R. T., y Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *11*(3), 309–346.

Mornini, T., y Loy, M. (2006). *Capistrano and the rails application lifecycle.* O'Reilly Media, Inc.

Múltiples autores. (2017). *What do you use and why 'rspec' vs 'minitest' ?* `https://web.archive.org/web/20181026015759/https://www.reddit.com/r/rails/comments/5v5tuy/what_do_you_use_and_why_rspec_vs_minitest/`. (Recuperado el 23 de Octubre de 2018)

Nichols, G. (2018). *Remote working 101: Professional's guide to the tools of the trade.* `https://web.archive.org/web/20181026023906/https://www.zdnet.com/article/remote-working-101-professionals-guide-to-the-tools-of-the-trade/`. (Recuperado el 25 de Octubre de 2018)

Real Academia Española. (2014). *Calidad.* Descargado de `http://dle.rae.es/?id=6nVpk8P|6nXVL1Z`

Sharma, S., y Hasteer, N. (2016). A comprehensive study on state of scrum development. En *Computing, communication and automation (iccca), 2016 international conference on* (pp. 867–872).

Stellman, A., y Greene, J. (2014). *Learning agile: Understanding scrum, xp, lean, and kanban.* O'Reilly Media, Inc.

Apéndice A

Plan de Calidad de Software para Intranet

A.1. Objetivo

El objetivo de este plan es entregar una guía para los desarrolladores nuevos y existentes del proyecto Intranet, con el cual se puedan conocer fácilmente las actividades, estándares y formas de trabajo que puedan asegurar el éxito del proyecto, mediante un aseguramiento mínimo de la calidad.

Este plan ha sido elaborado luego de varios semestres de trabajo en Intranet, y toma en cuenta la variedad de tipos de desarrollador que puede tener el equipo. Se espera que, además, se puedan discutir y escribir nuevas versiones de este plan si es que lo que se indica no se ajusta a los requerimientos del proyecto en el futuro.

A.2. Documentos adjuntos

La documentación entregada por este plan de calidad se complementa con la documentación general de Intranet, la cual está en la misma carpeta del proyecto en Git, en el directorio doc/ (requiere cuenta del DI con permisos asociados).

https://stash.inf.utfsm.cl/projects/INTDI/repos/intranet/browse/doc

Además, cierta documentación para la configuración del entorno de trabajo inicial y del despliegue de los servidores se encuentran en Confluence del Departamento de Informática, en el espacio de Intranet (requiere cuenta del DI con permisos asociados).

https://confluence.inf.utfsm.cl/display/IN/Intranet

A.3. Administración

El trabajo de Intranet es parte de I&T (Unidad de Infraestructura y Tecnología) del Departamento de Informática. Vale mencionar que esta unidad está en un proceso de reestructuración, y pueden ocurrir cambios en su estructura y sus políticas.

El siguiente es un Organigrama de I&T, pero que cambiará en un futuro cercano.

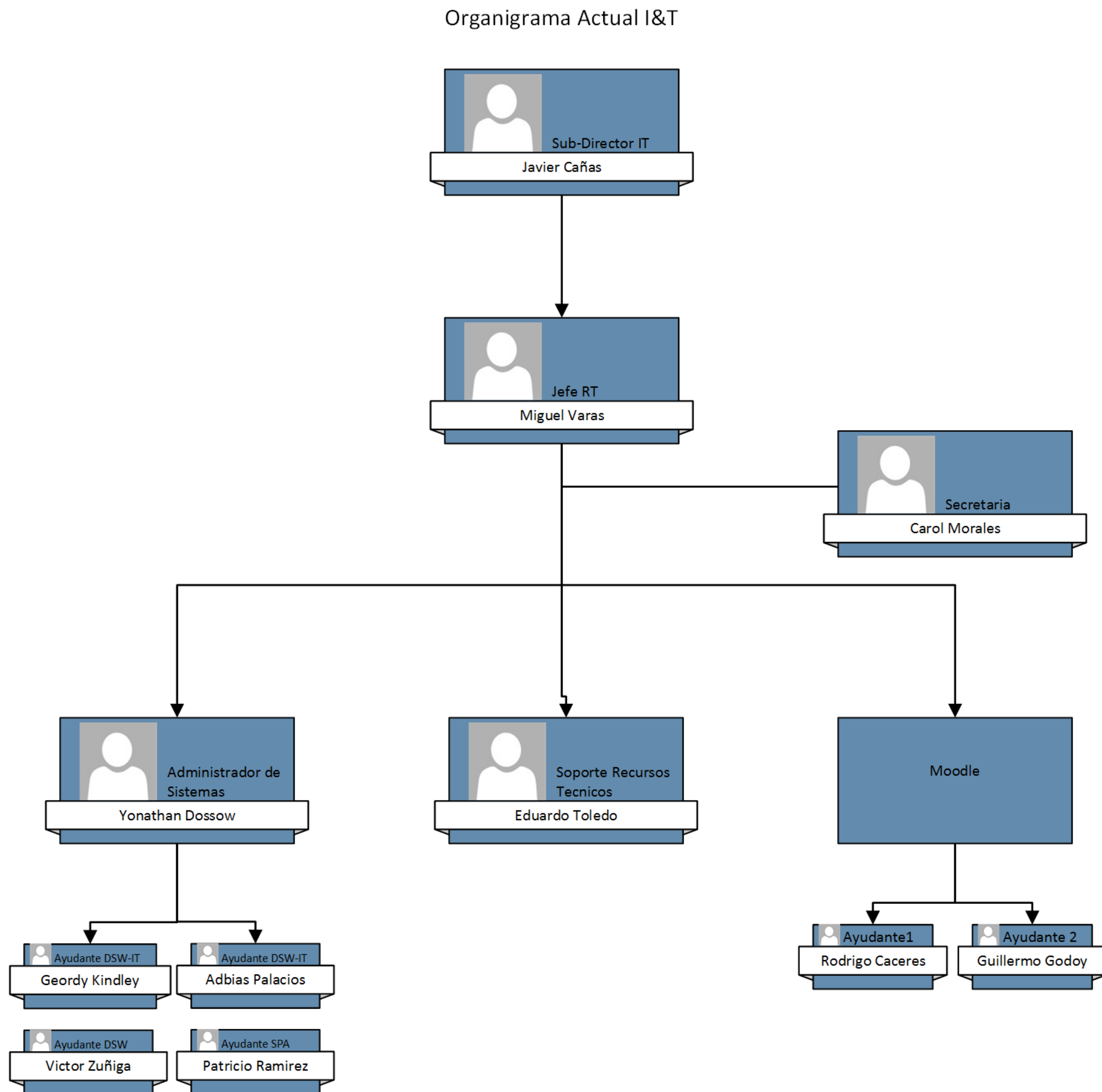


Figura A.1 – *Organigrama de I&T actual. Fuente: Confluence I&T*

Como se puede apreciar, existe una estructura jerárquica, y todo nuevo miembro del equipo de desarrollo de algún proyecto de software de I&T tendrá uno o más encargados: Para los estudiantes en práctica y nuevos ayudantes administrativos, como encargado oficial se tendrá a alguno de los funcionarios de I&T, como el Administrador de Sistemas o el Jefe IT. Además, es posible que como "mentor" o jefe se tenga a algún ayudante u otro funcionario que sea el mantenedor del sistema a trabajar, si es que este sistema ya existe.

Es posible que el equipo del proyecto además incluya a otros estudiantes en práctica o ayudantes. Se espera que también ellos puedan ayudar a nuevos desarrolladores a asimilarse al proyecto.

Ante cualquier duda o ante cambios en la estructura, consultar al funcionario de I&T que lo ingresó al proyecto, y agregar a este documento la información que Ud. considere faltante.

A.4. Documentación

La documentación principal del sistema estará en el repositorio VCS (*Version Control System*) de cada proyecto, donde se aloja todo lo que es relacionado con el código y la lógica de negocio. De esta manera, se pueden añadir y corregir cosas directamente al efectuar *commits* y subirlos al repositorio.

Por ende, es responsabilidad de todos los desarrolladores de mantener la documentación disponible y actualizada. Se recomienda utilizar el formato Markdown para los documentos, dado es simple de leer, simple de editar y no tiene problemas con los VCS como Git.

- En el caso de Intranet, la documentación se encuentra en el directorio ***doc/*** del proyecto

Toda documentación que no pueda o no deba estar en el VCS, como lo que es relacionado a preparar el ambiente de trabajo, o lo que incluya información sensible (como contraseñas, llaves de APIs, etc.) deberá ser registrado en un sistema que solo permita el acceso a los miembros del equipo u otros miembros de I&T, como una *Wiki* privada (como Confluence, Gitlab, Redmine, Phabricator, etc.)

- En el caso de Intranet, se utilizará **Confluence** de I&T, en *confluence.inf.utfsm.cl*

A.5. Estándares, prácticas, convenciones y métricas

A.5.1. Flujo de trabajo

La metodología de trabajo se basará en **desarrollo iterativo incremental** en base a módulos, y las iteraciones serán distintas según sea el caso: si existen funcionalidades pendientes, o si no existen funcionalidades pendientes. Cada iteración será llamada "tarea" a nivel de flujo de trabajo.

Las tareas pueden ser desarrolladas por miembros distintos del equipo (por ejemplo, un miembro puede encargarse de desarrollar una nueva funcionalidad, mientras otro miembro arregla errores o aplica optimización). Además, se puede cambiar la asignación de tareas a criterio del equipo.

Si existen funcionalidades pendientes, se define una iteración para implementar tal funcionalidad. Esta iteración tendrá los siguientes pasos:

1. **Reunión inicial**: Se define qué se desarrollará, cómo se desarrollará, quienes lo harán. Se debe discutir el requerimiento, el nuevo modelo o diseño del sistema para implementarlo, y los pasos que se seguirán. El equipo de desarrollo debe validar y acordar lo que se realizará.
2. **Desarrollo de las especificaciones**: Se escribirá una primera versión de las pruebas de aceptación que indican las especificaciones discutidas en la reunión, y se añadirá a la documentación del sistema todo lo que sea muy complejo, o lo no se pueda expresar en las pruebas de aceptación.

- Los siguientes dos pasos se pueden desarrollar en paralelo:

    4. **Desarrollo de la funcionalidad**: Se programa el código de la nueva funcionalidad, logrando que se cumplan las historias de usuario o casos de uso dadas por las pruebas de aceptación del paso anterior.
    5. **Desarrollo de pruebas**: Se escribirán las pruebas de aceptación en código, junto con pruebas para casos límite, pruebas unitarias de métodos/funciones y otros tipos de pruebas que no se cubren con las pruebas de aceptación. Este paso podría ser omitido en casos extremos (por ejemplo, si hay restricciones de tiempo muy estrictas), lo cual debe ser discutido en la reunión inicial. En este caso, las pruebas deberán ser marcadas como "pendientes" en la documentación, y se deberán desarrollar más adelante.

6. **Integración**: Se integra el código a la rama de desarrollo del repositorio, y se verifica que todas las pruebas del sistema son aprobadas. Si no es así, se deben arreglar los problemas volviendo al paso 3.

- En el caso de Intranet, se utilizará la rama *development*

7. **Despliegue en *staging***: Se hace el despliegue de la nueva versión del sistema en el ambiente de *staging*. De preferencia, el despliegue debe ser automatizado. Para software compilado, se espera tener un ejecutable de *staging*, y si es posible, un dispositivo para las pruebas. Para desarrollo web, se espera tener un servidor de *staging*.
    - En el caso de Intranet, se utilizará el servidor de *staging* en *staging.intranet.inf.utfsm.cl*, y el despliegue será automatizado por Jenkins, o manual mediante comandos por SSH.
8. **Inspección**: Se realizan pruebas manuales pertinentes al nuevo desarrollo en el ambiente de *staging*, y una inspección de los nuevos cambios en general. Se recomienda que sea realizado entre varias personas, y no solo los que estaban encargados de la tarea actual. Si se encuentran problemas o faltas, se debe volver al paso 2.
9. **Despliegue final**: Se hace el despliegue de la nueva versión en el ambiente de producción. De preferencia, el despliegue debe ser manual (o sea, que no sea automatizado), pero se pueden automatizar los pasos en sí.
    - En el caso de Intranet, se hará una conexión por SSH al servidor y se ejecutarán comandos para hacer el despliegue.

Si no existen funcionalidades nuevas, pero se debe efectuar desarrollo (por ejemplo, para corregir un error o para mejorar la calidad del código) se puede hacer lo siguiente:

- El paso 1 puede ser omitido, pero el trabajo a realizar debe ser comunicado (por ejemplo, mediante el mismo *issue tracker* o con el sistema de mensajería de grupo)
- El paso 2 puede ser la inclusión de especificaciones para evitar regresiones.

Notas:

- Las tareas se representarán visualmente en un *kanban board*, donde cada iteración será una tarea o un papel auto-adhesivo y la estructura de las iteraciones definirá las columnas.
    - Para el caso de Intranet se utilizará el *kanban board* de Confluence, donde cada tarea será un Issue, con un encargado, una prioridad y una clasificación (Nueva función, Tarea, Mejora).

- Se utilizará Integración Continua, lo que implica que todos los miembros del equipo añaden su trabajo a un repositorio o rama del VCS única (*development*), con tal de encontrar errores de integración y de aceptación de forma inmediata.
- Opcionalmente, se puede utilizar la programación de a pares. Por ejemplo, en casos donde existan miembros con experiencia y miembros sin experiencia que necesiten ayuda, o si se tiene un problema complejo a resolver.

A.5.2. Comunicación entre el equipo

- El Correo electrónico será el canal de comunicación oficial. En éste se deben comunicar las decisiones y anuncios finales del equipo de trabajo.
- Las discusiones más importantes deben ser mediante reuniones, las cuales pueden ser presenciales o remotas (videoconferencia o llamadas de voz grupal). Como apoyo a las reuniones se utilizarán pizarrones o programas de dibujo/escritura en línea. Al final de cada reunión debe haber un registro de los dibujos/diagramas (como una fotografía) y se debe enviar un correo electrónico a todos los miembros con los puntos tratados y las decisiones tomadas.
- Las discusiones menores deben ser realizadas en un programa de conversaciones grupales en línea (por ejemplo, un canal IRC, Slack, HipChat, etc.) para asegurar la participación de todos. Las decisiones importantes realizadas en discusiones presenciales espontáneas deben ser registradas aquí. El canal de conversaciones deberá tener subdivisiones (por ejemplo, un canal de desarrollo, otro sobre mantenimiento de los servidores, otro sobre resolución de errores, etc.) y la organización de éstas será a criterio del equipo.
- Luego del cierre de una iteración/tarea o un conjunto de tareas simultáneas, se debe hacer una retroalimentación del trabajo realizado y la metodología. Esta discusión no debe ser muy extensa y puede ser hecha en el programa de conversaciones grupales. Se debe utilizar un pizarrón o una herramienta de escritura colaborativa para presentar opiniones. Finalmente, los cambios y decisiones obtenidas deberán ser registradas y comunicadas por correo electrónico.

A.5.3. Comunicación con el cliente

- Antes de cada iteración y de la reunión correspondiente, debe haber una captura de requerimientos con los clientes. Se deja a criterio del equipo cómo se realiza (por ejemplo, entrevistas presenciales).
- Si el cliente tiene conocimiento técnico o buen dominio del negocio, luego del paso de desarrollo de especificaciones, se debe comunicar al cliente la forma de

implementación (por ejemplo, con *mockups* de interfaz), con tal de obtener retroalimentación. Si se requieren cambios drásticos, se puede citar a repetir la reunión inicial de la iteración/tarea. Si no, se deben comunicar los cambios por correo electrónico.

- Luego del despliegue en *staging*, se recomienda hacer una presentación al cliente. Si existe una satisfacción mínima (o sea, se cumplen los requerimientos y la calidad del producto está asegurada), se da por terminada la iteración y se agregan las posibles mejoras o cambios a una nueva iteración (la cual puede ser comenzada inmediatamente, o pospuesta para más adelante, a criterio del equipo).
    - Si no hay una satisfacción mínima del cliente, se debe evaluar un reinicio de la iteración, o volver a un paso anterior. Esto requiere discusión del equipo.

A.6. Reporte de problemas y acciones a realizar

En caso de problemas encontrados **antes** del despliegue a producción

- Utilizar las herramientas de comunicación para crear y asignar una tarea para su resolución. Luego, seguir el flujo normal de una iteración para el desarrollo de la resolución del error.
- Asegurarse que las pruebas automatizadas cubran este problema. Si no, modificarlas.
    - En Intranet, utilizar Jira y agregar una tarea al *kanban board*. El *commit* asociado a este *fix* debe tener en el texto la tarea correspondiente (IN-XX) para poder revisar éste en Confluence.

En caso de problemas encontrados **luego** del despliegue a producción

- Utilizar las herramientas de comunicación para crear y asignar una tarea para su resolución, con prioridad alta y asignada a alguien. Comunicar mediante correo electrónico y/o sistema de mensajería instantánea. Luego, comenzar una nueva iteración o tarea.
- Crear una nueva prueba automatizada para evitar que este problema se presente en el futuro (para evitar una regresión).
- En el VCS, se debe comentar este cambio como un *hotfix*
- Comunicar al resto del equipo cuál fue la razón de este error y cómo se hizo el arreglo, para evitar que ocurra un problema similar a futuro.

- En Intranet, utilizar Jira y agregar una tarea al *kanban board* con prioridad crítica y asignada a alguien. El *commit* asociado a este *hotfix* debe tener en el texto la tarea correspondiente (IN-XX) para poder revisar éste en Confluence.

A.7. Revisiones e inspecciones

Las revisiones e inspecciones se dejarán para el paso 6 del flujo de trabajo.

Vale decir que el trabajo del proyecto será una responsabilidad compartida, por lo que se recomienda siempre discutir los estándares a utilizar en código, pruebas y documentación, y agregar como *tareas* en el *issue tracker* los arreglos y "refactorizaciones" cuando se encuentren faltas.

A.8. Gestión de configuración del software

- Se debe utilizar un sistema de control de versiones (VCS) como Git, SVN o Mercurial, con un servidor funcionando como repositorio remoto.
    - Para Intranet: Se usará Git con el servidor *Stash* de I&T, *stash.inf.utfsm.cl*
- En el VCS se utilizará la siguiente nomenclatura de ramas o *branches*:
    - Rama "master" para el código que está listo para el despliegue en producción (o sea, probado y funcional).
    - Rama "development" para el código que junta iteraciones terminadas (o sea, código funcional). Aquí es donde se aplica la Integración Continua.
    - Ramas de tareas: Se le da la libertad a los miembros del equipo de crear nuevas ramas, para representar el avance en nuevas tareas o funcionalidades asignadas en el *issue tracker*. Para el caso de funcionalidades o módulos grandes, se recomienda utilizar la menor cantidad de ramas posible (si se puede, una sola rama por funcionalidad) para poder aprovechar la Integración Continua en caso de funcionalidades hechas por varias personas a la vez, y para evitar llenar el repositorio de ramas.
    - Ramas personales adicionales: Si un miembro del equipo necesita crear nuevas ramas (por ejemplo, para experimentos), es libre de hacerlo. Luego de su uso, si la rama se subió al repositorio, se debe eliminar.
    - Estas políticas pueden ser cambiadas bajo acuerdo, en caso de haber discrepancias (por ejemplo, si la mayor parte de los miembros del equipo están acostumbrados a otra política, y se va a realizar un proyecto nuevo).

- Se debe tener un servidor de Integración Continua, el cual correrá todas las pruebas automatizadas de software a partir de los nuevos cambios en la rama "*development*" del VCS, y en caso de software compilado, podrá realizar la compilación luego de que se aprueben todas las pruebas. Finalmente, este servidor hará el despliegue en el servidor de puesta en escena, y opcionalmente, en el servidor de producción.
    - En el caso de Intranet, el despliegue a producción no será automático, pero sí el despliegue a *staging*. Se define este despliegue en el Jenkinsfile como la ejecución del comando *bundle exec rake deploy:staging*. Este comando ejecuta el *script* definido en *lib/tasks/deploy.rake*

- Se debe tener un servidor de puesta en escena (*staging*), con tal de realizar pruebas manuales en un ambiente similar al de despliegue. En caso de equipos con uno o pocos proyectos de software, se puede reutilizar el servidor de Integración Continua para este fin.
    - En el caso de Intranet, el despliegue al servidor de puesta en escena (*staging*) será automático.

- Para el nombramiento de las versiones del sistema, se utilizará una versión simplificada del estándar Semantic Versioning 2.0, con las siguientes reglas:
    - Versiones numeradas en la forma X.Y.Z
        - X denota versiones con cambios drásticos que llevan a incompatibilidades respecto a versiones anteriores (por ejemplo, si es una API, incompatibilidad con llamadas anteriores).
        - Y denota cambios importantes pero no drásticos, por ejemplo, nuevas funcionalidades.
        - Z denota cambios menores, como mejoras o corrección de errores.
    - Uso de guión (-) y texto opcional al final de la versión, para denotar versiones funcionales pero no estables (por ejemplo, 1.0.0-beta o 1.1.2-rc1).
    - Uso de signo "más" (+) y texto opcional al final de la versión, para denotar metadatos relacionados al "*build*" o a pasos intermedios de compilación y del VCS (por ejemplo, 1.0.0+b76c30aa).
    - Uso de la versión 0.X.Y para versiones anteriores al uso del cliente en producción, y 1.X.Y para versiones de uso del cliente en producción.
    - Al igual que las políticas de ramas en el VCS, estas reglas pueden ser cambiadas bajo acuerdo del equipo.

- La responsabilidad de las herramientas, incluyendo los servidores, será compartida. Se permite que exista gente especializada, pero todo el equipo debe tener un conocimiento mínimo, credenciales y acceso a las herramientas, con tal de evitar interrupciones o esperas ante eventualidades.

A.9. Herramientas, técnicas y tecnologías

A.9.1. Editores de código y/o IDEs

En general, todas las herramientas de desarrollo que no produzcan efectos secundarios en el trabajo colaborativo del proyecto son permitidos, y serán dejados a criterio de cada desarrollador. Si existe soporte para *linters* y conexión con las otras herramientas a usar, se debe motivar el uso de tal herramienta.

- En el caso de Intranet, recordar que el proyecto tiene muchos archivos y que el lenguaje Ruby es dinámico: se recomienda utilizar editores o IDEs con *linters* de Ruby y otros *plugins* que puedan ayudar, pero evitar que se generen archivos basura en el VCS.

A.10. Metodología de las pruebas de software

- Se deberán tener, en forma obligatoria, pruebas automatizadas de aceptación, las cuales pasarán por todas las funcionalidades de la aplicación, incluyendo casos alternativos y casos límite.
    - Para Intranet: Se usará RSpec como biblioteca de *testing*, con las gemas de complementos: Capybara para pruebas de aceptación, capybara-webkit para ejecutar éstas en un navegador *headless*, FactoryBot para generar *mocks* y VCR para guardar las peticiones y respuestas de APIs en un caché.

      Para entender el formato de las pruebas y ver ejemplos, revisar los archivos existentes en el directorio *spec/*
- Si existe código que no se cubra con las pruebas de aceptación, se deben hacer pruebas unitarias para suplir las de aceptación.
- Cuando sea posible, utilizar bibliotecas de cobertura de código, para asegurar que las pruebas pasen por todo el código del proyecto (100 %) o al menos, por el 70 % de éste, mientras existan razones de peso para estar bajo el 100 %, y mientras este porcentaje no baje a medida que avanza el proyecto.

  - Para Intranet: Se utilizará, mediante la biblioteca *simplecov*. Luego de ejecutar pruebas, se puede visitar el archivo *coverage/index.html* para ver resultados detallados sobre la cobertura de las pruebas ejecutadas.

- Las pruebas nunca se borrarán, y deberán acumularse para hacer que en todo momento el sistema sea probable o *testable*. Se debe fomentar el uso de las pruebas automatizadas a lo largo del desarrollo de funcionalidades, con tal de descubrir problemas a tiempo.

  - En el caso de Intranet, las pruebas se pueden ejecutar con el comando *RAILS_ENV=test bundle exec rspec spec/*.
    Se puede definir un archivo específico a ejecutar, por ejemplo,
    RAILS_ENV=test bundle exec rspec spec/features/paa_form_spec.rb.
    Y se pueden ejecutar pruebas específicas, si éstas tienen un tag de RSpec, por ejemplo,
    RAILS_ENV=test bundle exec rspec spec/features/paa_form_spec.rb –tag problem ejecuta las pruebas con el *tag* “problem”

- El equipo debe asegurarse que las pruebas puedan ejecutarse en cualquier ambiente de *testing*, y siempre obteniendo el mismo resultado. Tener cuidado con las pruebas que pasan en forma aleatoria (*flaky tests*), , indicar si las pruebas requieren ciertos archivos fuera del VCS (como archivos de secretos), y tener cuidado con pruebas que utilicen sistemas externos o APIs.

  - En el caso de Intranet, los archivos extras usados por las pruebas irán en el directorio *spec/samples/* (por ejemplo, archivos PDF para pruebas de subida de archivos). Las pruebas a veces asumen que se tienen datos en el archivo *config/secrets.yml*, por lo que se debe revisar la documentación respecto a qué se espera en éste.

    Finalmente, para las APIs se utiliza la gema VCR, la cual guarda un caché de las peticiones y respuestas de APIs en *spec/fixtures/vcr_cassettes*. **Cada vez que el código que utiliza APIs cambie, o las pruebas que utilizan APIs cambien, o que la API cambie, se deben eliminar los archivos de caché pertinentes**. Estos archivos guardan las peticiones y respuestas efectuadas en un momento dado, y VCR mostrará un error si se hace una petición inesperada o no guardada.