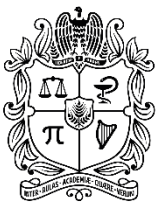


UNIVERSIDAD
NACIONAL
DE COLOMBIA

Factores de mantenibilidad en el desarrollo de aplicaciones web

Juan David Meza González

Universidad Nacional de Colombia

Facultad de Minas, Departamento de Ciencias de la Computación y la Decisión

Medellín, Colombia

2017

Factores de mantenibilidad en el desarrollo de aplicaciones web

Juan David Meza González

Trabajo presentado como requisito parcial para optar al título de:

Magister en Ingeniería: Ingeniería de Sistemas e Informática

Director:

Ph.D Fernando Arango Isaza

Línea de Investigación:

Mantenibilidad en el desarrollo de software

Universidad Nacional de Colombia

Facultad de Minas, Departamento de Ciencias de la Computación y la Decisión

Medellín, Colombia

2017

Este documento debe estar dedicado a un número de personas que ni siquiera puedo contar. Han sido muchos, quienes de una u otra forma han estado conmigo durante este largo proceso, apoyándome en los momentos de debilidad y cuando estuve a punto de desfallecer. Este trabajo es el resultado de un proceso que ha pasado por diferentes etapas y donde cada una de éstas ha tenido un impacto significativo en la forma de percibir las cosas y, por ende, de finalmente definir este documento. Sin el apoyo, consejo y ayuda de muchos esto no habría sido posible.

A Dios por darme la vida, la fortaleza y el espíritu necesarios para llegar hasta el final.

A mi familia, por apoyar y ayudarme a elegir siempre lo mejor para salir adelante. Especialmente mis padres, que, aunque no comprenden mucho de lo que hago, siempre estuvieron para escuchar mis conversaciones y lecturas, darme su respaldo y ánimo para continuar y finalmente terminar.

A mi pareja, por ser un gran soporte a pesar de la distancia, por apoyar mis decisiones, incluso cuando parecían ir en contra de todo y estar ahí para escucharme.

A todos los educadores y tutores que formaron parte de este proceso en cualquiera de sus etapas. Por sus enseñanzas, formación y seguimiento.

:)

Resumen

La mantenibilidad es un concepto que se utiliza en múltiples contextos, haciendo alusión a la cantidad de trabajo o esfuerzo necesario para que un sistema cualquiera conserve su funcionamiento frente a determinados cambios que se introduzcan o a errores que se presenten. La mantenibilidad se puede aplicar en el contexto del desarrollo de software y puntualmente en el desarrollo de aplicaciones web. Para este último, mantenibilidad se define como la facilidad o dificultad para modificar la aplicación web, ya sea a causa de un arreglo que se debe realizar o de una mejora que se desea implementar. La mantenibilidad es un atributo que tiene un impacto significativo en la calidad general de una aplicación web y, de hecho, de no ser tenida en cuenta durante el diseño y la implementación del proyecto, puede acarrear a sobrecostos durante el tiempo de vida de la misma. En este trabajo muestran los factores que impactan directamente la mantenibilidad de una aplicación, especialmente durante el proceso de implementación de la misma.

Palabras clave: mantenibilidad, aplicaciones web, desarrollo de software

Abstract

The maintainability is a concept used in multiple contexts, alluding to the amount of work or effort necessary for any system to keep its operation against certain changes introduced or errors that arise. The maintainability can be applied in the context of software development and in that way in the development of web applications. For the latter, maintainability is defined as the ease or difficulty of modifying the web application, either because of an adjustment that must be made or an improvement to be implemented. The maintainability is an attribute within a significant impact on the overall quality of a web application and, in fact, if it is not taken into account, during project design and implementation, can lead to over costs during the lifetime of the project. This document, show the factors directly influences the maintainability of an application, especially during the implementation process.

Keywords: maintainability, web applications, software development

Contenido

Pág.

Lista de figuras

Pág.

Lista de tablas

Pág.

Introducción

La mantenibilidad es un concepto que se utiliza en múltiples contextos haciendo alusión a la cantidad de trabajo o esfuerzo necesario para que un sistema cualquiera conserve su funcionamiento frente a determinados cambios que se introduzcan o a errores que se presenten (Gu, 2016; Till, Ceolin, & Visser, 2016). Se entiende que un sistema tiene alta mantenibilidad si el esfuerzo requerido para conservar su estado es considerablemente bajo, pues en caso contrario su nivel de mantenibilidad se reduce.

La mantenibilidad es un tributo de calidad esencial que puede ser medido para diferentes tipos de sistemas y en diferentes contextos, entre los cuales se encuentra el desarrollo de software (Abilio, Teles, Costa, & Figueiredo, 2012; Devi, Sharma, & Kesswani, 2016; Kumar, 2012; Rodríguez & Fernández, 2015).

Ahora bien, el desarrollo de software abarca un amplio rango de sistemas de software y como es de esperarse cada uno de estos tipos posee a la mantenibilidad como un atributo importante de calidad (Kundu & Tyagi, 2016) incluyendo, por supuesto, el desarrollo de aplicaciones web como un tipo particular de sistemas de software (Kundu & Tyagi, 2016; Maurya & Shankar, 2012; Misra & Egoeze, 2014; Vern & Dubey, 2014).

En el contexto del desarrollo de software, la definición de mantenibilidad no varía mucho. Se le define como la facilidad o dificultad para modificar un sistema software, ya sea a causa de un fallo que debe ser reparado o de una mejora que se desea implementar (Braude & Bernstein, 2016; Kumar, 2012). Esta definición se puede puntualizar para el desarrollo de aplicaciones web, puesto que esta actividad es netamente afín al desarrollo de software, permitiendo definir la mantenibilidad en el desarrollo de aplicaciones web como la facilidad o dificultad (esfuerzo requerido), para realizar un cambio en dicha aplicación ya sea en busca de mejoras o de reparar el funcionamiento de la misma, conservando un buen funcionamiento (Chae, Kim, Jung, & Lee, 2007; Martínez, Cachero, & Meliá, 2014; Misra & Egoeze, 2014).

Una vez establecido lo anterior, es necesario puntualizar que este documento se enfoca principalmente en los diferentes factores de la mantenibilidad en el desarrollo de aplicaciones web. Por tal motivo, a partir de este punto cada vez que en el documento se mencione el concepto de mantenibilidad, se estará haciendo referencia a la definición del mismo en el contexto del desarrollo de aplicaciones web.

Ahora bien, la mantenibilidad es un atributo que tiene un impacto significativo en la calidad general de una aplicación web (Offutt, 2002; Rodríguez & Fernández, 2015) y, de hecho, de no ser tenida en cuenta durante el diseño y la implementación del proyecto, puede acarrear a sobrecostos durante el tiempo de vida de la misma (E Ghosheh, Qaddour, Kuofie, & Black, 2006; Kundu & Tyagi, 2016; Martínez et al., 2014). Es importante resaltar que, a lo largo del ciclo de vida de un software, un alto porcentaje del tiempo de desarrollo es consumido por la mantenibilidad y si el software en cuestión no es mantenible, entonces el consumo de tiempo aumentará aún más (Kumar, 2012; Kumar Pandey & Agrawal, 2016).

Por otro lado, la mantenibilidad es un atributo de calidad difícil de medir ya sea de manera cualitativa o cuantitativa (E Ghosheh et al., 2006; Till et al., 2016) puesto que su aplicación, tanto en la instancia de diseño como en la instancia de desarrollo de la aplicación web, dependen en gran medida de la experiencia y destreza de los involucrados en tales tareas y además puede ser interpretada con diferentes resultados según el enfoque considerado y la característica a evaluar al momento de determinar la mantenibilidad (Kumar, 2012; Till et al., 2016). Por ejemplo, la mantenibilidad de una aplicación web puede ser de alto nivel al momento de realizar algún tipo de cambio en los módulos o componentes existentes de la aplicación, pero podría ser de bajo nivel al momento de introducir un nuevo módulo o componente que interaccione con los ya existentes (Abilio et al., 2012). Esto hace a la mantenibilidad de una misma aplicación dependiente de la característica evaluada. Adicionalmente, una misma característica de una misma aplicación web puede presentar diferentes factores de mantenibilidad que deben ser considerados, y de ser posible, priorizados según los requerimientos de la aplicación como tal (Gosheh, Black, & Qaddour, 2008; Martínez et al., 2014).

Dicho lo anterior, se hace necesario definir un factor específico a considerar al momento de determinar la mantenibilidad de una aplicación web para así evitar ambigüedades en el

proceso o que éste se realice de manera subjetiva (Misra & Egoeze, 2014). De este modo, se puede afirmar que una misma aplicación web puede presentar diferentes valores de mantenibilidad según el factor y características analizadas (E Ghosheh et al., 2006), llevando al hecho de que es también necesario definir la característica del proyecto que tenga el mayor impacto o mayor importancia para el análisis de la mantenibilidad de la misma y haciendo uso de un factor específico. En definitiva, para realizar un análisis puntual de la mantenibilidad de una aplicación web, se debe especificar la característica cuya mantenibilidad sea de mayor importancia en el proyecto y el factor a utilizar para estudiar la mantenibilidad de esa característica puntual.

En consideración de lo anterior, en este trabajo se busca entonces estudiar los factores de mantenibilidad en el desarrollo de aplicaciones web bajo un enfoque específico que se definirá más adelante en el documento y haciendo énfasis en las prácticas de desarrollo habituales que permitan al desarrollador determinar en alguna medida la mantenibilidad del código durante y/o después del proceso de desarrollo. En definitiva, se ha optado por un enfoque más práctico y apegado a la implementación tangible de código fuente en una aplicación web, que un enfoque teórico y meramente informativo.

Objetivos del trabajo

El objetivo general de este trabajo es implementar un caso de estudio de una aplicación web haciendo uso de las mejores técnicas que maximizan la mantenibilidad. Esto permitirá determinar los factores más cercanos al proceso de implementación de una aplicación web que favorecen la mantenibilidad de la misma.

Los objetivos específicos que surgen, a partir del objetivo general, son:

1. Identificar los elementos y prácticas que influyen en la mantenibilidad de una aplicación web. De este modo se podrán conocer los elementos más cercanos al proceso de desarrollo e implementación.
2. Identificar los parámetros o métricas que permitan determinar y medir la mantenibilidad de una aplicación web. Por medio de esto, es posible realizar una medición que permita aproximar la mantenibilidad de una aplicación web.

3. Desarrollar un prototipo de una aplicación web que demuestre la aplicación de los elementos y prácticas de la mantenibilidad. Así se podrán evidenciar los diferentes factores identificados, junto con su respectiva implementación.
4. Validar, mediante el uso de los parámetros identificados, la mantenibilidad del prototipo propuesto, con el fin de constatar la adecuada implementación de los factores de mantenibilidad identificados.

Organización del documento

Este documento está organizado de la siguiente forma:

El capítulo 1, proporciona una visión general de la calidad en el software, definiendo los componentes principales que de una u otra forma aportan a la calidad final del software como un todo. En este capítulo se introduce también, la mantenibilidad como uno de los componentes de la calidad en el software, junto con los factores que impactan tal componente.

El capítulo 2, se enfoca en un dominio particular que son las aplicaciones web, mostrando así la definición de mantenibilidad ya no desde el punto de vista del software en general, sino desde el punto de vista de las aplicaciones web, algunas de las variaciones que este dominio presenta y los escenarios particulares en los que se presentan los diferentes factores de la mantenibilidad en las aplicaciones web.

El capítulo 3, muestra la forma particular en la que se pueden implementar y evidenciar cada uno de los diferentes factores de mantenibilidad identificados, desde el dominio de las aplicaciones web, indicando asó las características a las que se debe apuntar con la finalidad de constatar la presencia de cada uno de estos factores, en las aplicaciones web.

El capítulo 4, presenta el prototipo desarrollado y el modo en el que se implementaron y, por supuesto, se evidenciaron las diferentes implementaciones para cada uno de los factores de mantenibilidad identificados.

El capítulo 5, resume los conocimientos alcanzados con el desarrollo del trabajo, junto con las conclusiones y recomendaciones resultantes.

1. Calidad en el software

La ingeniería de software es una actividad compleja para la resolución de problemas con propósitos de calidad (Kumar, 2012). El estándar de calidad ISO 9126 (ISO 9126 International Standard ISO/IEC 9126: Software Engineering - Product Quality, 2001) detalla un modelo de calidad para los productos de software.

El modelo anteriormente mencionado define un conjunto de seis características con sus respectivas sub-características (Ver Tabla 1-1).

Tabla 1-1: Características de Calidad en el software (ISO 9126 International Standard ISO/IEC 9126: Software Engineering - Product Quality, 2001)

Característica	Sub-características
Funcionalidad	– Idoneidad – Precisión – Interoperabilidad – Seguridad
Confiabilidad	– Disponibilidad – Tolerancia a las fallas – Madurez – Facilidad de recuperación
Usabilidad	– Facilidad de comprensión – Facilidad de aprendizaje – Facilidad de operación
Eficiencia	– Comportamiento en el tiempo

	– Uso de recursos – Rendimiento
Mantenibilidad	– Facilidad de análisis – Facilidad de modificación – Estabilidad – Facilidad de pruebas
Portabilidad	– Facilidad de adaptación – Facilidad de sustitución – Co-existencia – Facilidad de instalación

El estándar de calidad ISO 9126, proporciona así, un conjunto de características con sus respectivas sub-características, que permitan definir la totalidad de requerimientos que un producto de software debe satisfacer en sus componentes, para ser considerado como un software de calidad (Hashim & Key, 1996). Por supuesto, entre mayor sea el número de características a considerar, más compleja será la estimación de la calidad del software (Till et al., 2016).

En concordancia con lo anterior, este trabajo se enfoca en una de las características de calidad del software, que es la mantenibilidad. De hecho, aunque la mantenibilidad no es la única característica para la definición de la calidad del software, es de interés para diversas fuentes (Abilio et al., 2012; Chae et al., 2007; Kumar Pandey & Agrawal, 2016) y posee una mayor importancia con respecto a otras (Hashim & Key, 1996; Till et al., 2016).

1.1 La mantenibilidad en el software

Como se puede determinar en la Tabla 1-1, la mantenibilidad es una característica de la calidad del software y se puede dividir en cuatro diferentes sub-características (facilidad de análisis, facilidad de modificación, estabilidad y facilidad de pruebas), por supuesto, existen diversos factores que impactan de un modo u otro en alguna de estas sub-características y de manera general a la mantenibilidad como tal.

Como el énfasis de este trabajo, está en la mantenibilidad, a continuación, se detallan estas cuatro sub-características.

1.1.1 Facilidad de análisis

La facilidad de análisis, se refiere a qué tan fácil o difícil es diagnosticar el sistema para identificar deficiencias o identificar los componentes que necesitan ser modificados para una propósito establecido (Misra & Egoeze, 2014). La facilidad de análisis puede encontrarse en múltiples etapas de un proyecto de software, desde la documentación y diseño, hasta el desarrollo e implementación de la misma (Kundu & Tyagi, 2016). En términos generales, la facilidad de análisis se puede ver como la facilidad para diagnosticar fallos en el software (Meananeatra, 2012).

Ahora bien, a pesar de que la facilidad de análisis es mencionada en diferentes trabajos como una sub-característica de la mantenibilidad (E Ghosheh et al., 2006; Emad Ghosheh, Systems, Black, Kapetanios, & Baldwin, 2010; Gosheh et al., 2008; Kundu & Tyagi, 2016; Martínez et al., 2014; Meananeatra, 2012; Misra & Egoeze, 2014; Till et al., 2016), muchos de estos no la toman como un factor medible o simplemente de fácil consideración para la determinación de la mantenibilidad de un proyecto de software, pues se centran en la medición de la mantenibilidad desde el código fuente, dado que es una tarea más sencilla, y la facilidad de análisis no parece ser fácilmente medible o perceptible en esta instancia (el desarrollo o implementación del código fuente) (E Ghosheh et al., 2006).

1.1.2 Facilidad de modificación

La facilidad de modificación se refiere simplemente a eso, a qué tan fácil o difícil es realizar algún cambio o adaptación en el software (Misra & Egoeze, 2014). En cierto modo, la facilidad de modificación va de la mano con la escalabilidad, pues se entiende que una aplicación de software es de fácil modificación si no se deben llevar a cabo muchos cambios en la estructura existente de tal aplicación, con la finalidad, ya sea de cambiar el funcionamiento actual o de introducir nuevas características (Stella, Jarzabek, & Wadhwa, 2008).

Profundizando en este contexto, existen cuatro tipos principales de actividades, directamente relacionados con la facilidad de modificación -correctiva, preventiva, perfectiva y adaptativa- (Hashim & Key, 1996; Kumar, 2012; Malhotra & Chug, 2016). Es razonable que surja la necesidad de aclarar la diferencia entre facilidad de modificación y mantenibilidad, pues parecen tener objetivos similares. Pues bien, aunque comparten propósitos, la mantenibilidad va más allá que la facilidad de modificación, pues tiene en consideración la corrección de fallas, junto con la introducción o actualización de funcionalidades mientras que la facilidad de modificación no se ocupa de la corrección de fallas (Malhotra & Chug, 2016).

En vista de lo anterior, se podría decir que la facilidad de uso es un subconjunto de la mantenibilidad, lo cual permite indicar que la mantenibilidad, al menos en el contexto de la facilidad de uso, posee cuatro actividades principales que son las actividades correctivas, preventivas, perfectivas y adaptativas, y de hecho, diferentes fuentes definen estas cuatro actividades como directamente relacionadas con la mantenibilidad (E Ghosheh et al., 2006; Kumar, 2012), por supuesto, a estas cuatro actividades se podrían agregar las relacionadas con la corrección de fallas.

1.1.3 Estabilidad

La estabilidad se refiere a la capacidad del software, para evitar efectos inesperados al momento de introducir cambios en el mismo (Martínez et al., 2014). Básicamente, se entiende que si un software sigue funcionando de forma correcta después de realizar un cambio, entonces es estable (Meananeatra, 2012). Qué tan fácil o difícil es mantener el software en un estado consistente, al introducir cambios (Misra & Egoeze, 2014). Si al momento de realizar un cambio en un software, existe el riesgo o se tiene el temor de que algo deje de funcionar correctamente, es una señal de poca estabilidad en el sistema (Kumar, 2012). Cabe resaltar, que estabilidad está más relacionada con modificaciones a componentes existentes del software, sin embargo, si la introducción de nuevos componentes o funcionalidades, también agrega una incertidumbre sobre el correcto funcionamiento de los componentes previamente existentes, entonces también se

evidencia baja estabilidad en el software (Di Lucca, Fasolino, Tramontana, & Visaggio, 2004; Martínez, Cachero, Matera, Abrahao, & Luján, 2011).

Continuando con lo anterior, la estabilidad, está relacionada con la madurez del producto de software (Di Lucca et al., 2004). Un software maduro, requiere poco o ningún cambio a lo largo de su ciclo de vida, tanto porque no es necesario cambiar componentes existentes para introducir otros o simplemente porque ya cumple con su funcionamiento y no es propenso a errores (Di Lucca et al., 2004), lo cual coincide con la definición de estabilidad. Sin embargo, la estabilidad está netamente relacionada con la mantenibilidad, mientras que la madurez no (Kumar, 2012).

1.1.4 Facilidad de pruebas

La facilidad de pruebas se refiere, principalmente, a qué tan fácil o difícil es validar o corroborar el correcto funcionamiento del software, a partir de criterios de aceptación definidos (Martínez et al., 2014; Meananeatra, 2012; Misra & Egoeze, 2014; Stella et al., 2008). Diferentes fuentes se refieren a la facilidad de prueba como un mecanismo de validación del correcto funcionamiento del software en cualquier momento, pero especialmente después de realizar algún cambio (Kumar, 2012; Martínez et al., 2011; Misra & Egoeze, 2014). Es de esperarse entonces que un software supere las pruebas de correcto funcionamiento si en éste no se ha realizado ninguna modificación, lo cual da pie a considerar que la facilidad de pruebas es un buen mecanismo para corroborar la estabilidad del software (Kumar, 2012).

Ahora bien, es necesario aclarar que la facilidad de pruebas considera la facilidad que se tenga en el software para crear o implementar dichas pruebas (Kumar, 2012), sin embargo, diferentes fuentes concuerdan en lo que se mencionó anteriormente, en que se refieren de manera especial a la facilidad de corroborar el funcionamiento adecuado del software después de implementar algún cambio.

2. La mantenibilidad de una aplicación web

La mantenibilidad de una aplicación web cualquiera, está sujeta a múltiples parámetros, enfoques y contemplaciones (E Ghosheh et al., 2006; Till et al., 2016). Como se ha mencionado, el proceso de diseño e implementación de la aplicación web introduce cierta incertidumbre y dependencia de las destrezas y experticia de los involucrados en tales procesos, junto con la subjetividad que puede darse al realizar la medición de la mantenibilidad según el enfoque que se utilice y que el nivel de mantenibilidad, incluso para un mismo enfoque, puede variar según la característica a evaluar.

A continuación, se describen los principales enfoques mencionados en la literatura que impactan la mantenibilidad de una aplicación web.

Enfoque adaptativo: Se centra en los casos en los que la aplicación requiera adaptarse a nuevos entornos, tales como un nuevo hardware, un nuevo sistema operativo, una nueva base de datos, una nueva ubicación, entre otros (Kumar, 2012; Malhotra & Chug, 2016; Stella et al., 2008).

Enfoque correctivo: Se enmarca en los momento que se requiere corregir alguna falla que se presente en cualquier instancia del ciclo de vida de la aplicación (análisis, diseño, implementación o documentación) (E Ghosheh et al., 2006; Kumar, 2012; Kundu & Tyagi, 2016; Malhotra & Chug, 2016).

Enfoque preventivo: Se concentra en el momento en que se requiera realizar modificaciones en busca de prevenir futuros errores para así mantener el correcto funcionamiento y extender el tiempo de vida de la aplicación (E Ghosheh et al., 2006; Kumar, 2012)

Enfoque perfectivo: Se centra en las mejoras de la aplicación, ya sea al momento de mejorar la eficiencia aumentando su velocidad, al agregar o modificar funcionalidades o al mejorar el rendimiento (Kumar, 2012; Kundu & Tyagi, 2016).

Cada uno de los enfoques presentados anteriormente, está netamente relacionado con la mantenibilidad y, de hecho, a todos estos se les puede dar una interpretación desde el código fuente de la aplicación, es decir, desde el desarrollo e implementación de la aplicación web. Por supuesto, en las etapas de análisis y diseño estos enfoques también son relevantes, pero son más fácilmente evidenciables desde el código fuente (Di Lucca et al., 2004; Kataoka, Imai, Andou, & Fukaya, 2002).

Ahora bien, para el propósito de este trabajo se busca establecer los factores de mantenibilidad relacionados con estos diferentes enfoques, pero desde la interpretación del código fuente de la aplicación web. De este modo, se puede determinar el conjunto de prácticas o y/o características que se deben tener bajo consideración durante el desarrollo e implementación de una aplicación web y, por supuesto, a lo largo de su ciclo de vida.

2.1 Factores de mantenibilidad para la implementación de aplicaciones web

En la literatura se encuentran múltiples categorías para los factores que afectan la mantenibilidad (Kumar, 2012), relacionados con diferentes etapas del ciclo de vida de la aplicación web, sin embargo, muchos de estos centran sus esfuerzos específicamente en estos factores durante el proceso de desarrollo e implementación de la misma, a lo largo de su ciclo de vida (Rodríguez & Fernández, 2015).

A continuación, se presentan algunos de los factores que están relacionados con la mantenibilidad de una aplicación web desde la interpretación del código fuente y que centran su definición en las prácticas de desarrollo habituales recomendadas.

Modularidad: Se refiere a la separación lógica de la aplicación en diferentes partes, componentes y módulos. La aplicación podría ser fácilmente comprendida y modificada cuando esté compuesta por diferentes módulos (Anda, 2007; Hashim & Key, 1996; Peercy, 1981). También se hace referencia a este concepto como facilidad de modificación (ISO 9126 International Standard ISO/IEC 9126: Software Engineering - Product Quality, 2001)

Consistencia: Se refiere a la uniformidad en la notación, terminología y simbología. El adecuado uso de convenciones de nombres, manejo de excepciones, acoplamiento de módulos (Anda, 2007; Peercy, 1981), uso adecuado de un patrón definido. También se le conoce como estandarización (Hashim & Key, 1996).

Simplicidad: Se refiere a la inexistencia de complejidad, ya sea en el funcionamiento o en la comprensión de la organización, lenguaje y técnicas de implementación. Se refleja en el uso de conceptos singulares y estructuras básica (Anda, 2007; Hashim & Key, 1996; Peercy, 1981).

Facilidad de lectura: El grado con el cual un "lector "pueda entender el código fuente (Hashim & Key, 1996) y la lógica del funcionamiento de la aplicación (o módulo).

Promedio de "variables vivas": Se refiere a las variables que son usadas con frecuencia tanto por sentencias anteriores o posteriores. Se obtiene a partir de la cantidad de "variables vivas" dividido por la cantidad de sentencias ejecutables. Entre mayor sea este promedio, más complejo será el desarrollo y la mantenibilidad de la aplicación (Aggarwal, Singh, Chandra, & Puri, 2005).

Proporción de comentarios: Es la proporción entre la cantidad de líneas de código y las líneas de comentarios. Entre menor sea esta proporción, se entiende que mayor será la facilidad de lectura del código y por ende será mejor la mantenibilidad (Aggarwal et al., 2005).

Comentarios apropiados: Se refiere a que los comentarios sean expresivos y significativos, de modo que el código se pueda leer e interpretar con mayor facilidad (Anda, 2007). Cabe resaltar que este factor no busca medir la cantidad de comentarios, sino la idoneidad y eficacia de cada uno.

Objetos y clases: La trazabilidad entre las clases de los requerimientos de la aplicación y el código debería ser simple (Anda, 2007; Jabangwe, Borstler, Smite, & Wohlin, 2015).

Como es de esperar, todos estos factores mencionados son fácilmente identificables en el código fuente de la aplicación web (McIntosh, Kamei, Adams, & Hassan, 2016), y algunos de ellos, son más simples de evidenciar e incluso medir que otros. Incluso, en la literatura misma, se dan casos en los que las definiciones tienden a solaparse (Kundu & Tyagi, 2016; McIntosh et al., 2016; Till et al., 2016), sin embargo, pueden finalmente agruparse en estos factores mencionados.

2.2 Escenarios concretos para los factores de mantenibilidad en una aplicación web

Puesto que el enfoque de este trabajo es más práctico que investigativo, se hace necesario, una vez identificados los factores que impactan la mantenibilidad, brindar casos o escenario específicos que permitan identificar de manera sencilla la presencia de algún nivel de mantenibilidad den una aplicación web, desde el punto de vista del código fuente.

En concordancia con lo anterior, uno de los escenarios más habituales, en los cuales se puede evidenciar la existencia de mantenibilidad es, por supuesto, cuando la aplicación web, sufre algún cambio requerido, ya sea en busca de mejorar un componente actual o agregar uno nuevo. En ambos casos, la necesidad de identificar componentes con facilidad es inherente, para de ese modo, conocer los componentes que deberán ser modificados y agregados (modularidad) (Kumar, 2012), sin embargo la modificación o actualización de componentes ya existentes, implica además la interacción con el código fuente existente y por ende, permitirá evidenciar fácilmente los demás factores de mantenibilidad como consistencia, simplicidad, facilidad de lectura, proporción de comentarios y demás (Aggarwal et al., 2005; Jabangwe et al., 2015; Malhotra & Chug, 2016; McIntosh et al., 2016).

A partir de lo anterior, se plantea entonces a modo de ilustración, la necesidad de modificar una funcionalidad actual de la aplicación web. De momento, no se especificarán detalles respecto a determinados componentes o funcionalidades, pues esto se profundiza en una sección posterior (ver sección 4.9).

Ahora bien, cuando surge la necesidad de modificar una funcionalidad existente y previamente diseñada, lo primero que se debe hacer es identificar el o los componentes a intervenir, pues de este modo, el desarrollador podrá llegar fácilmente a tales componentes a partir de un diseño o modelo original. Para este propósito, los factores de mantenibilidad más relevantes son la modularidad, los objetos y clases y la consistencia (Aggarwal et al., 2005; Jabangwe et al., 2015; Kumar, 2012; Mkaouer et al., 2015). Pues por medio de la modularidad el desarrollador podrá clasificar fácilmente los componentes relevantes de los que no lo son, por medio del uso de objetos y clases, tendrá certeza que el diseño original planteado que sigue en ese momento, tiene una trazabilidad precisa con el código fuente, permitiéndole identificar el componente exacto a intervenir y finalmente, por medio de la consistencia, tendrá la certeza de que la manera en la que se ha implementado el código y los componentes a modificar, son el mecanismo a seguir para las sentencias de código a introducir y que éstos se mantienen de forma consistente a lo largo de toda la aplicación web (Kumar, 2012).

Finalmente, por medio de la facilidad de lectura, simplicidad, relación de comentarios y comentarios adecuados, el desarrollador podrá comprender con facilidad y en poco tiempo el funcionamiento de los componentes actuales e identificar los que finalmente tendrán que ser modificados o no, y por supuesto, el modo en el que tendrá que hacerlo, acorde a los nuevos requerimientos (Aggarwal et al., 2005; Kumar, 2012).

3. Implementación de los factores de mantenibilidad en una aplicación web

Para poder definir que una aplicación web es mantenible, a partir de su código fuente, es necesario especificar el modo en el que cada uno de estos factores puede ser evidenciado allí. La definición de los diferentes factores de mantenibilidad es clara en la literatura, sin embargo, es imperativo determinar el modo de llevar cada una de estas definiciones a la práctica de modo que se pueda evidenciar la presencia de tal definición en el código fuente como tal de una manera “tangible”.

Dicho lo anterior, en este capítulo se muestran las posibles características que son evidenciables en el código fuente de una aplicación web y que se ajustan a la definición de los diferentes factores de mantenibilidad de éstas.

3.1 Modularidad

Modularidad en el código puede ser evidenciada de diferentes formas dependiendo del nivel de abstracción con el que se analice el código. Si se analiza desde el punto de vista del patrón o arquitectura seguida durante la implementación, se encontrará modularidad al nivel de los componentes básicos esenciales de la aplicación web, es decir, si una aplicación web separa de manera adecuada los componentes implementados, por ejemplo, usando un diseño por capas (MVC, MVVM, MVP, entre otros) (Riaz, Breaux, & Williams, 2015; Zanoni, Perin, Fontana, & Viscusi, 2014) brindando así modularidad suficiente como para reconocer los componentes principales de la aplicación web e inclusive, cómo interaccionan entre ellos.

Por supuesto, y en continuación con lo anterior, si se reduce un poco el nivel de abstracción, se puede evidenciar modularidad también, para una aplicación web mantenible. Si se va más allá de los componentes o capas que conformen el patrón o arquitectura elegidos, se puede encontrar separación en módulos a nivel de clases o recursos (Martínez et al., 2014; Stella et al., 2008), es decir, cada capa o componente se podrá encontrar separado o modularizado por las diferentes clases o recursos comunes que se presentan. Una misma capa podría estar separada en múltiples módulos según la cantidad de recursos que existan y los componentes que se puedan agrupar de forma común a ellos (Riaz et al., 2015).

Ahora bien, si se va un poco más allá, al interior de cada una de esas clases y recursos que conforman las diferentes agrupaciones identificadas, también se podrá encontrar modularidad, pues cada clase y recurso, en una aplicación mantenible, tendría que estar compuesto por diferentes módulos como paquetes, métodos y atributos (Al-Jamimi, Alshayeb, & Elish, 2011; Riaz et al., 2015; Till et al., 2016).

3.2 Consistencia

La consistencia a nivel del código se puede evidenciar en el correcto seguimiento de un estándar a lo largo de la aplicación web, es decir que elementos como el estilo del código, la forma de nombrar variables, clases, archivos, definir comentarios, entre otros, se conserven en todas las estructuras y componentes del proyecto de software.

Por otro lado, si cada desarrollador define su propio “estándar” en cada proyecto, esto podría acarrear a problemas de compatibilidad entre aplicaciones desarrolladas por diferentes desarrolladores, incluso aunque hayan sido implementadas en un mismo lenguaje de programación, y por ello la consistencia debe ir más allá de esto, e interpretarse de acuerdo al correcto seguimiento de los estándares de programación propios del lenguaje o tecnología usados, de modo que cada desarrollador siga las mismas recomendaciones.

Ahora bien, cuando se habla de un mismo lenguaje o tecnología común entre proyectos, es de esperar que se haya seguido un mismo estándar, sin embargo, cuando el lenguaje

o tecnología cambian, las recomendaciones a seguir pueden variar pues cada lenguaje define sus propios estándares a seguir al momento de escribir código fuente (ESA Board for Software Control Standardisation and (BSSC), 2000; PHP Framework Interop Group, 2016a; van Rossum, Warsaw, & Coghlan, 2001; Whyte, 2014), por fortuna, muchos de estos estándares coinciden en varios aspectos, aunque no todos son aspectos comunes.

Continuando con lo anterior, los estándares de programación definidos por los lenguajes de programación definen varios aspectos que comparten. Entre los más importantes se pueden encontrar las convenciones para nombrar las variables, los métodos, las clases, los paquetes, los archivos, entre otros. De manera general, el nivel de detalle de estos estándares es extenso, indicando inclusive la manera de organizar el código, definir los espacios y escritura en general de las estructuras comunes de código, pero es allí donde comienzan a aparecer las diferencias más importantes, pues algunos de estos estándares definen algunos elementos, mientras que otros no, por supuesto, como se mencionó, varios de estos elementos se comparten entre estándares, pues se trata de características comunes entre estos.

Ahora bien, es importante aclarar que, aunque estas recomendaciones de estilo de código tienen elementos en común, cada uno define a su manera la forma de llevar a cabo tal elemento. Por ejemplo, para declarar el nombre de las variables el lenguaje de programación Java define que las variables se nombren iniciando cada palabra que conforme el nombre de la variable con mayúscula excepto la primera (ESA Board for Software Control Standardisation and (BSSC), 2000), mientras que el lenguaje de programación Python define que las palabras usadas en los nombres de las variables inician todas con mayúscula, incluyendo la primera (van Rossum et al., 2001) y finalmente, el lenguaje de programación PHP, no da claridad en ello, indicando que el desarrollador debe elegir una notación y apegarse a ella en todo el proyecto (PHP Framework Interop Group, 2016b).

En concordancia con lo anterior, se puede esperar entonces que la consistencia en el código pueda ser definida libremente por el desarrollador asegurándose de seguirla a cabalidad en todo el proyecto, sin embargo, es recomendable que el desarrollador conozca las recomendaciones propias del lenguaje y tecnología que esté usando para asegurar consistencia entre diferentes proyectos e incluso entre diferentes desarrolladores, de ese

modo, la consistencia se podrá determinar según el lenguaje y tecnología usados, pues lo que se recomienda en un lenguaje no necesariamente aplica en otro.

3.3 Simplicidad:

La simplicidad puede tener una definición un tanto vaga y que se presta a interpretaciones múltiples. Por suerte, en diferentes trabajos se muestra cómo evidenciar la simplicidad de una aplicación web a partir de diferentes características que se pueden evidenciar (Anda, 2007). Aunque suene un tanto evidente, la simplicidad es la falta de complejidad, lo cual implica que si un componente cualquiera de una aplicación web, se lo considera complejo, es porque la simplicidad del mismo es baja o inexistente y por ende se estaría afectando la mantenibilidad de la aplicación.

Ahora bien, en múltiples fuentes se pueden encontrar diferentes parámetros que permiten determinar si un componente de una aplicación web o de una pieza de software en general es complejo (o simple). Una vez más, como se ha visto en otros factores de la mantenibilidad, la simplicidad (o ausencia de complejidad) se puede apreciar desde diferentes instancias o niveles de abstracción sobre el proceso de desarrollo e implementación de una aplicación web, pues la complejidad se puede evidenciar al nivel de las líneas de código o estructuras usadas, a nivel de la separación de componentes y capas de la aplicación, a nivel de las técnicas de implementación, e incluso en el lenguaje de programación utilizado (Anda, 2007; Kumar, 2012; Peercy, 1981).

Continuando con lo anterior, la simplicidad a nivel de código, se centra en la definición de métodos para generalizar el código para evitar repetir sentencias de manera innecesaria y por ende disminuir la cantidad resultante de sentencias a ejecutar, junto con la eliminación de métodos y variables sin uso (Anda, 2007). A nivel de capas y estructuras, se puede hablar de la existencia de clases, con contenidos adecuadamente estructurados, es decir, con métodos y atributos definidos y de utilidad; por supuesto, es de esperar que una misma clase no contenga una gran cantidad de métodos o atributos, pues se reduciría la simplicidad, lo adecuado sería entonces separar la cantidad de métodos y atributos requeridos en diferentes clases y componentes. Lo opuesto a esto también es inadecuado, es decir, la existencia de clases vacías o casi sin funcionalidad alguna, implica que esta

clase puede ser removida y suplir sus funcionalidades en una estructura más completa (Anda, 2007). A nivel del lenguaje de programación, se espera que un lenguaje de programación de alto nivel facilite la implementación de código y componentes simples, pues muchas funcionalidades y componentes ya están incluidos, evitando tener que crearlos desde cero.

En consecuencia, con lo anterior, la simplicidad consiste en el uso de la menor cantidad de componentes (al nivel de capas y estructuras), junto con la menor cantidad de sentencias (al nivel del código fuente) para conseguir la mayor cantidad de funcionalidades posible, maximizando así la funcionalidad a cambio de una estructura y código más simple. Por supuesto, el uso de un lenguaje de programación de alto nivel, para la implementación de la aplicación web, facilitará la implementación de código y componentes simples, a partir de las facilidades brindadas por el lenguaje mismo (Peercy, 1981).

3.4 Facilidad de lectura

La facilidad de lectura está inherentemente relacionada con el código fuente de la aplicación web (Aggarwal et al., 2005). El nombre de este factor de mantenibilidad sugiere su intención, pues se espera que un código sea fácil de leer para así aumentar la facilidad de comprensión y por ende permitir que ese código sea fácilmente modificado y consecuentemente mantenido (Aggarwal et al., 2005; Kumar, 2012).

Sobre la base de las consideraciones anteriores, resulta necesario definir cómo es una sentencia de código de fácil lectura. Pues bien, existen diferentes casos en los que una misma línea de código puede presentar mejores grados de facilidad de lectura, dependiendo de su composición y longitud. Una línea de código larga, llena de operadores, llamadas a métodos y símbolos, sería un claro ejemplo de una línea difícil de leer (Hashim & Key, 1996; Silva, Tsantalis, & Valente, 2016).

En concordancia con lo anterior, cuando el desarrollador/lector del código fuente presenta dificultades para comprender en pocos segundos la finalidad o utilidad de una sentencia, posiblemente se deba a que ésta no posee una adecuada facilidad de lectura (Silva et al., 2016).

Por otro lado, la facilidad de lectura se ve afectada por otros factores de mantenibilidad, que, de no haber sido considerados, empeorarán gravemente el desempeño de este factor. El uso de buenas prácticas para la correcta definición para los nombres de las variables, métodos y demás componentes, junto con el uso de comentarios descriptivos y concisos, ayudarán a mejorar la facilidad de lectura del código fuente (Hashim & Key, 1996).

Por lo tanto, la facilidad de lectura busca que el código sea fácilmente leído, a través del uso de nombres descriptivos para los diferentes elementos de la aplicación web, junto con la existencia de comentarios descriptivos que permitan hacerse una idea del propósito de una sentencia o conjunto de sentencias y finalmente, por medio de sentencias cortas y efectivas (Hashim & Key, 1996; Kumar, 2012). Por supuesto, existen sentencias de código que requieren una longitud considerable, por lo que se recomienda que éstas sean separadas en varias sentencias, por medio de saltos de línea, para así facilitar la lectura y la comprensión (Di Lucca et al., 2004; Silva et al., 2016).

3.5 Promedio de “variables vivas”

Las variables vivas están plenamente relacionadas con el código fuente (Aggarwal et al., 2005; Kundu & Tyagi, 2016). Se entiende que entre mayor sea el promedio de variables vivas en una aplicación web, más compleja será su mantenibilidad (Kumar, 2012).

Ahora bien, se entiende que una variable, es una variable viva, si en una sentencia de código específica, tal variable ha sido referenciada en otras sentencias de código, tanto antes como después (Aggarwal et al., 2005). Por supuesto, entre mayor sea el número de variables vivas presentes en una aplicación web, más compleja será su mantenibilidad, puesto que tales variables representan puntos críticos que de ser modificados podrían introducir resultados inesperados en diferentes partes de la aplicación web.

Continuando con lo anterior, una variable viva puede incluso ser más crítica que otra, pues entre mayor cantidad de sentencias dependan de ella, más compleja será la mantenibilidad de esos componentes involucrados. De igual manera, independientemente de la cantidad de sentencias que dependan de una misma variable viva, tales variables deben evitarse

en la medida de lo posible, aunque, por supuesto, habrá circunstancias en las que sean necesarias (Aggarwal et al., 2005; Kumar, 2012).

En definitiva y de acuerdo con todo lo anterior, las variables vivas pueden ser identificadas directamente en el código fuente, y aunque unas pueden ser más críticas que otras, es ideal evitar su uso frecuente, haciéndolo únicamente cuando sea estrictamente necesario (Aggarwal et al., 2005; Kumar, 2012). Finalmente, el promedio de variables vivas puede ser calculado tanto a nivel del código fuente al dividir la cantidad de variables vivas entre la cantidad de sentencias ejecutables, como a nivel de los módulos que compongan la aplicación web, dividendo la cantidad de variables vivas presentes en todos los módulos, sobre la cantidad de módulos (Aggarwal et al., 2005).

3.6 Proporción de comentarios

La proporción de comentarios es un factor evidenciable en el código fuente de la aplicación web y se refiere a la cantidad de comentarios presentes en el código, respecto a la cantidad de sentencias ejecutables (Aggarwal et al., 2005; Kumar, 2012). Contrario a lo que se podría esperar, entre menor la proporción de comentarios, más alta será la mantenibilidad del código. Esto responde a la estrecha relación entre la facilidad de lectura y la proporción de comentarios, pues se espera que el código pueda ser fácilmente comprendido y leído sin necesidad de comentarios y los comentarios sean simples mecanismos de apoyo a la interpretación del código fuente (Kumar, 2012).

En concordancia con lo anterior, se espera que el código fuente se dé a entender por sí solo por medio de nombres de variables y métodos descriptivos, líneas de código cortas y concretas, mientras que los comentarios se encargan de documentar estructuras esenciales como métodos y clases, más no de indicar el objetivo o funcionalidad de cada sentencia de código (Hashim & Key, 1996; Kumar, 2012). De este modo, entre menor sea la cantidad de líneas de comentarios respecto a la cantidad de sentencias ejecutables, mejor será la mantenibilidad respecto a este factor (Aggarwal et al., 2005; Hashim & Key, 1996).

Finalmente, de acuerdo con todo lo anterior, la proporción de comentarios va de la mano con la facilidad de lectura del código, entre menor cantidad de comentarios sean necesarios, mayor tendría que ser la facilidad de lectura del código fuente, resultando en una fácil comprensión del mismo y finalmente en una fácil mantenibilidad del código y de la aplicación web en general (Aggarwal et al., 2005; Kumar, 2012).

3.7 Comentarios apropiados

En la sección anterior, se detallaba la proporción de comentarios, sin embargo, la existencia de los comentarios también tiene un propósito concreto y por ende deben ser apropiados. Los comentarios que estén presentes en el código fuente deben ser comentarios apropiados, es decir, que los expresen con claridad su propósito y de manera concreta, pues de no ser así el comentario es innecesario y debería ser removido (Di Lucca et al., 2004; Hashim & Key, 1996; Kumar, 2012; McIntosh et al., 2016).

En consecuencia con lo anterior, los comentarios son apreciables directamente en el código y se usan como mecanismo para mejorar la facilidad de lectura del código fuente, lo cual finalmente permite mejorar la mantenibilidad (Kumar, 2012).

Ahora bien, los comentarios se usan para describir la funcionalidad de una pieza de código específica, por supuesto, sin ser redundante, pues si el código por sí mismo ya insinúa su función, el comentario es innecesario (Kumar, 2012; Silva et al., 2016). Se sugiere que los comentarios describan detalles de la aplicación web, tales como objetivos, sugerencias, valores de entrada y de salida, básicamente cualquier cosa que soporte la fácil comprensión del código fuente (Peercy, 1981).

3.8 Objetos y clases

Este factor de mantenibilidad, presenta una especie de relación entre el diseño original de la aplicación web y el código fuente –trazabilidad- (Hashim & Key, 1996), pues se enfoca en cómo los objetos y por ende las clases que componen a la aplicación de software, pueden ser fácilmente relacionadas desde el diseño original hacia el código fuente, es decir, que las clases que se indicaron originalmente en el diseño de la aplicación web,

puedan ser identificadas fácilmente en el código fuente y estructura de la aplicación web (Kumar, 2012).

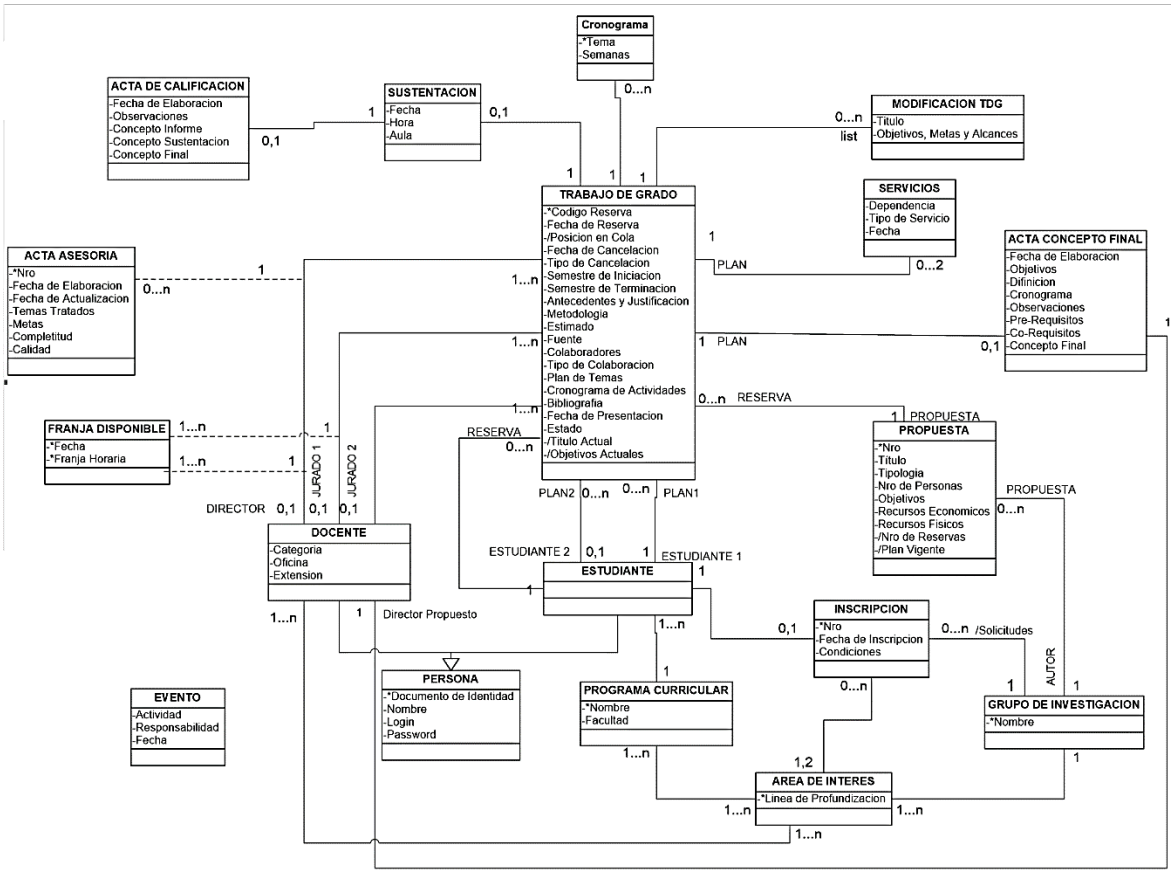
Ahora bien, la trazabilidad puede ser evaluada directamente desde el código fuente, pero requiere de un diseño base original para validar su existencia y, por supuesto, su correctitud. Se podría pensar, que esta trazabilidad es un factor que no afecta directamente la mantenibilidad desde el punto de vista del código fuente, sin embargo, si no es fácil para un desarrollador cualquiera determinar con facilidad los componentes a intervenir a partir de un requerimiento nuevo o modificado desde el diseño original de la aplicación web entonces la mantenibilidad del código fuente, se verá finalmente afectada también (Hashim & Key, 1996; McIntosh et al., 2016).

En definitiva, este factor de mantenibilidad es importante para facilitar las intervenciones futuras que sean requeridas en el código fuente de la aplicación web a partir del diseño establecido de la misma; de modo que los cambios o adiciones de características y componentes puedan realizarse sin mayor dificultad por cualquier desarrollador involucrado, teniendo o no conocimientos previos de la estructura del proyecto, lo cual refuerza y , de hecho, coincide, con la definición de mantenibilidad (Hashim & Key, 1996; Meananeatra, 2012; Silva et al., 2016).

4. Prototipo desarrollado y sus factores de mantenibilidad

Con el propósito de evidenciar la implementación de los diferentes factores de mantenibilidad de una aplicación web, es necesario construir una aplicación web funcional y lo suficientemente completa, de modo que permita mostrar el detalle de cada uno de los factores de mantenibilidad identificados y su respectiva implementación en un proyecto real y funcional.

Figura 4-1: Diagrama de clases completo del sistema de gestión de trabajos dirigidos de grado (Acevedo Orrego, 2008)



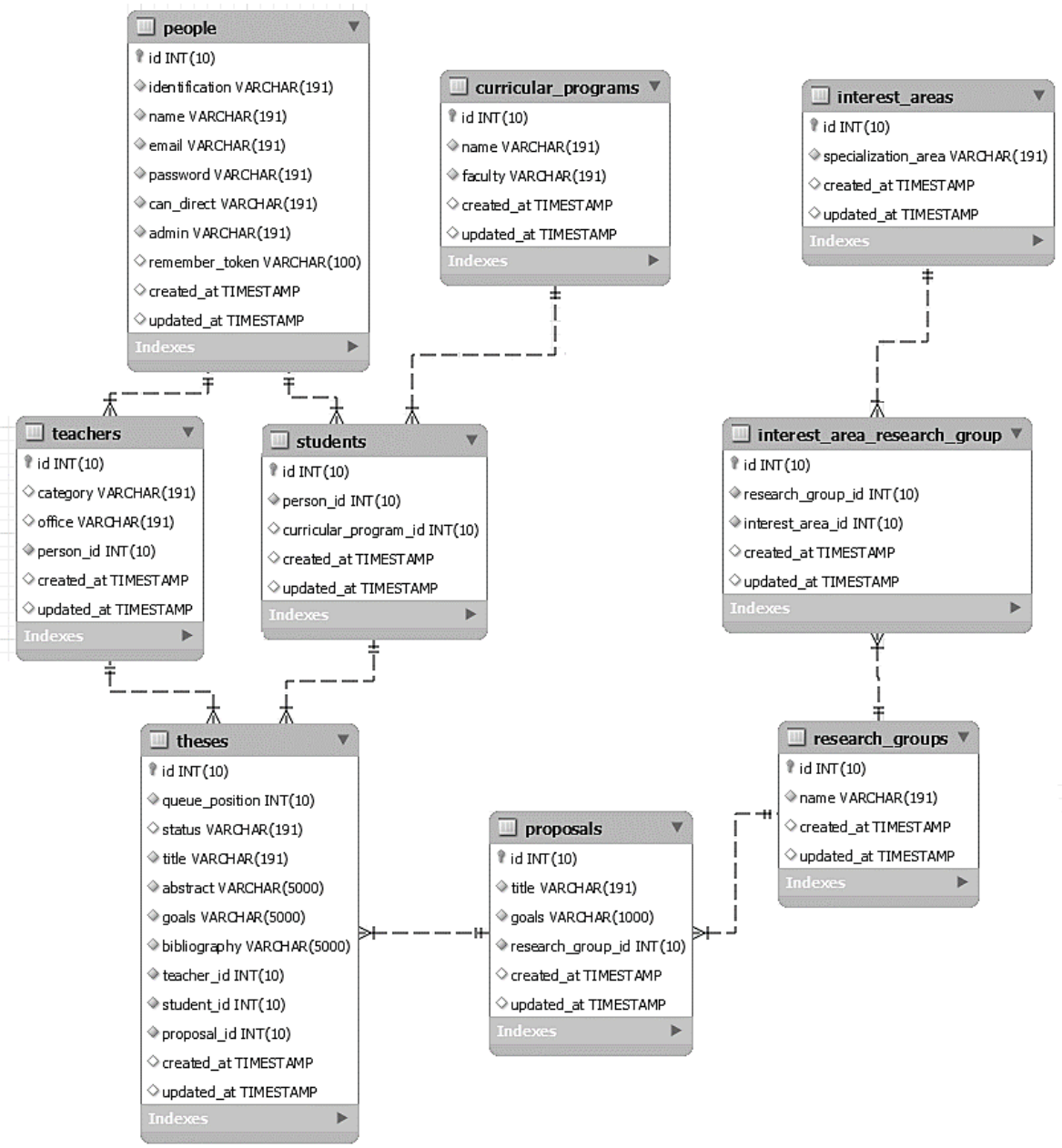
Para el propósito anteriormente mencionado, se ha implementado un prototipo de un sistema para la gestión de trabajos dirigidos de grado [1]. Este es un sistema complejo (ver Figura 4-1) que involucra una cantidad considerable de componentes y actores (profesores, estudiantes, grupos de investigación, jurados, entre otros). Por fines de simplicidad, en este prototipo, se han implementado los componentes que están directamente relacionados con los estudiantes como actor. Por supuesto, tales componentes requieren la existencia de otros recursos como profesores, grupos de investigación, entre otros, sin embargo, hay que aclarar que estos recursos no poseen componentes netamente dedicados a ellos, sino que cuentan con lo estrictamente necesario para su adecuado funcionamiento en relación con los componentes requeridos para los estudiantes.

En concordancia con lo anterior, el prototipo implementado presenta una cantidad inferior de características, pues no se encarga de operaciones que actúen sobre los otros recursos (creación, eliminación actualización), sino únicamente de operaciones relacionadas con los estudiantes como usuarios del sistema, lo cual resulta en diferentes entidades que ya no son necesarias, junto con atributos que pueden ser removidos de aquellas que sí deban estar presentes en la aplicación web.

Continuando con la idea anterior, el prototipo sólo cuenta con nueve de los dieciocho recursos propuestos en el sistema original y de estos nueve recursos, por lo menos tres sufrieron modificaciones en sus atributos, tanto porque no fue necesario conservarlos como porque fue necesario agregar algunos nuevos. En la Figura 4-2 se pueden apreciar los nueve recursos que conforman el prototipo, junto con sus respectivas relaciones. Adicionalmente, se pueden apreciar los diferentes atributos para cada uno de esos recursos y determinar cuáles de estos fueron removidos o agregados. Por ejemplo, en el caso de los recursos profesor y estudiante, para facilitar su relación con persona, puesto que ambos son personas, se agregaron atributos que permitan determinar a qué persona está ligado cada estudiante y profesor y viceversa.

[1] El código fuente de la aplicación web implementada se puede encontrar en un repositorio público en la siguiente dirección: https://github.com/JuanDMeGon/trabajo-final

Figura 4-2: Diagrama entidad-relación simplificado para el prototipo

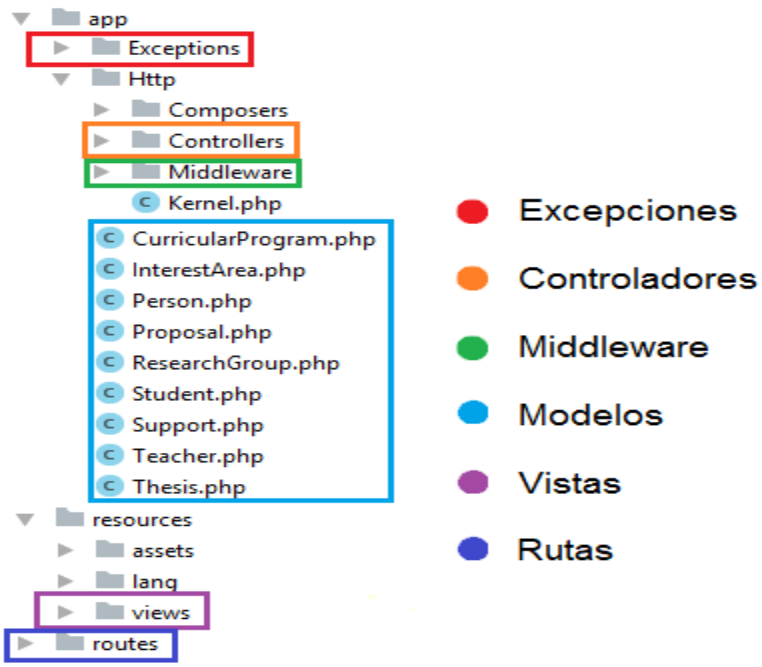


A continuación, se detalla el modo en el que, cada uno de los factores de mantenibilidad identificados y especificados en las secciones anterior, fue implementado en el código fuente de la aplicación web prototipo. Es importante aclarar, que cada uno de los componentes analizados y mostrados a continuación, son propiamente implementados en el prototipo y no son estructuras pertenecientes a los componentes internos del framework utilizado.

4.1 Modularidad

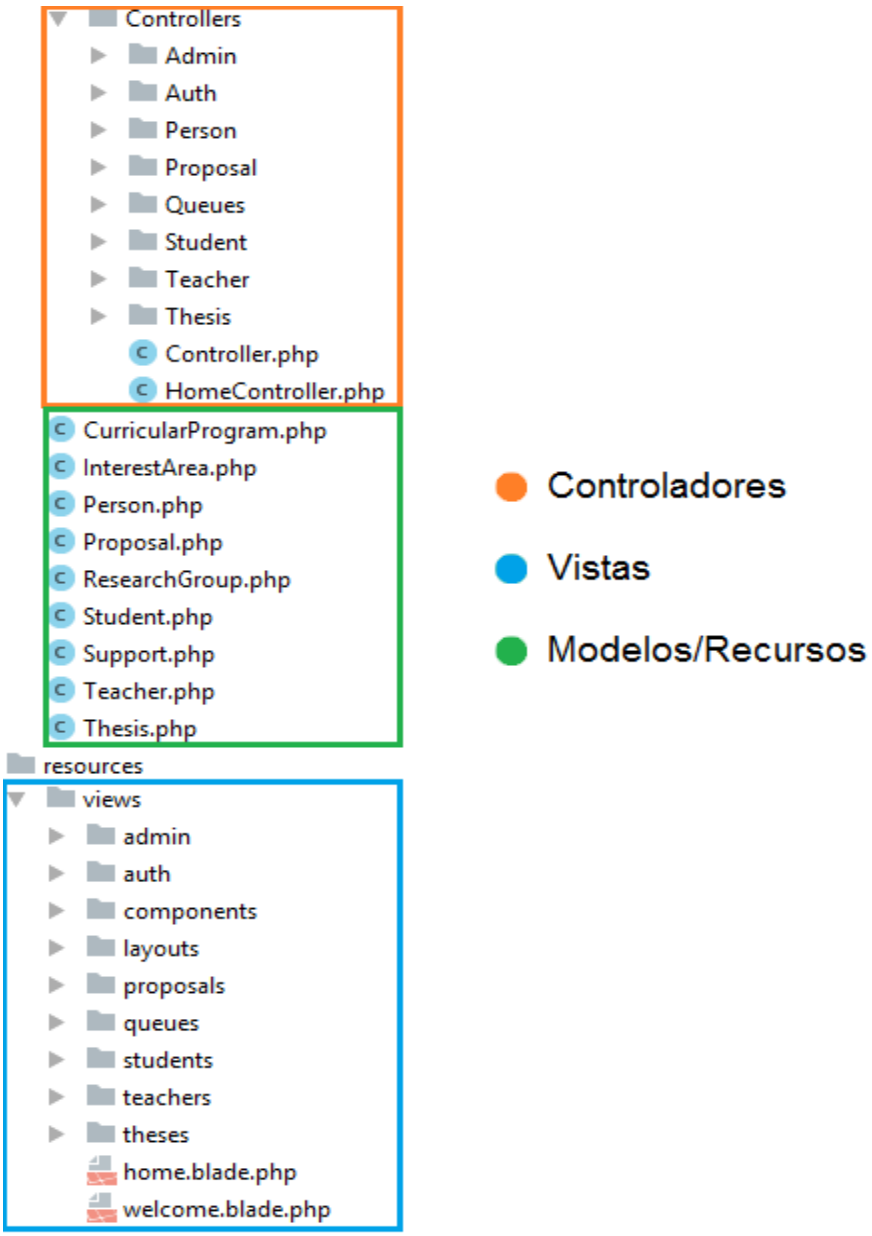
Tal y como se mencionó en la sección anterior, la modularidad en el prototipo implementado, se puede apreciar en diferentes niveles. De manera inicial, la aplicación web se encuentra dividida en capas fundamentales como son los modelos, las vistas y los controladores, siguiendo un patrón de diseño habitual MVC (Stella et al., 2008). Cabe resaltar, que la aplicación web implementada posee capas adicionales a los estipulados por MVC, tales como rutas para enlazar los destinos de las peticiones (URL) con sus respectivas acciones en los controladores, eventos para centralizar determinadas acciones a ejecutar a partir de ciertas acciones (creación, actualización o eliminación), middleware para controlar determinadas condiciones sobre las peticiones y las respuestas, excepciones para manejar los diferentes errores y fallas en un punto central de la aplicación web, entre otros. En la Figura 4-3 se pueden apreciar estos componentes junto con otros más que hacen parte de la estructura fundamental del framework utilizado.

Figura 4-3: Capas principales de la aplicación web



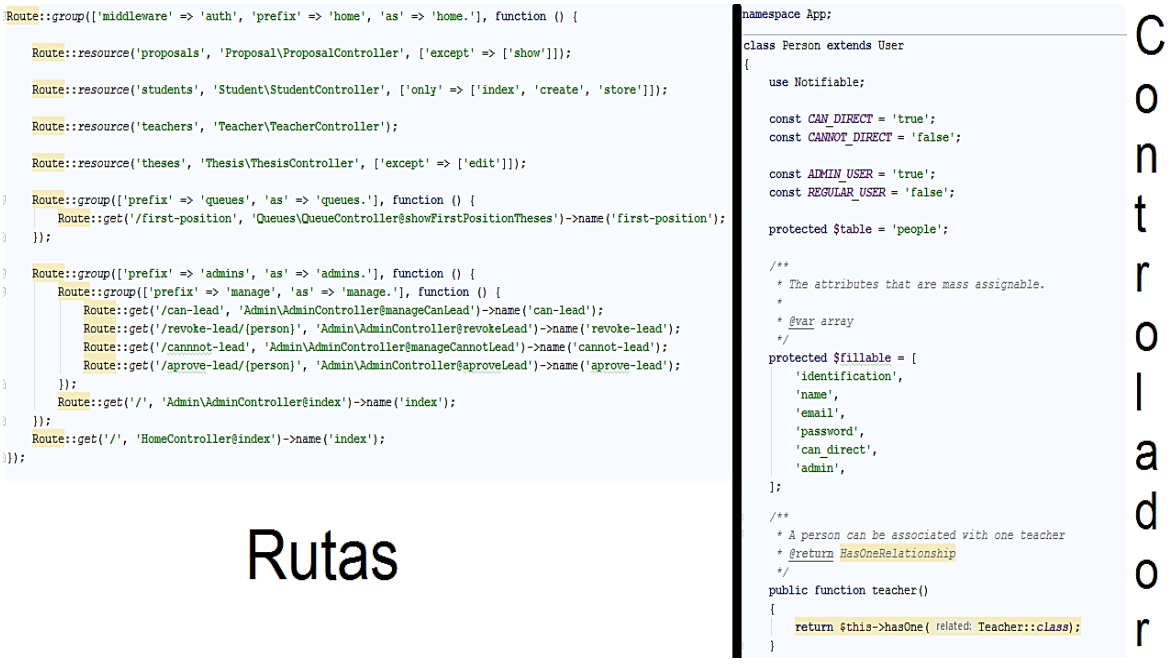
Ahora bien, a nivel de cada una de estas capas, también se puede evidenciar modularidad, pues los diferentes controladores, vistas, modelos, y demás (según sea necesario), han sido separados según el recurso, es decir, modularizando tales capas según el recurso con el cual están relacionados (profesor, estudiante, grupo de investigación, etc.). De este modo, todos los controladores, vistas, y demás, relacionados con los estudiantes estarán en una carpeta adecuada para este componente. En la Figura 4-4 se puede apreciar cómo están modularizadas las vistas y los controladores, acorde al recurso relacionado.

Figura 4-4: Modularidad al interior de las capas



Finalmente, a nivel de los elementos más fundamentales como las clases, la modularidad también se puede apreciar al estar cada una de estas correctamente organizada en un paquete respectivo con un conjunto de atributos y métodos fácilmente identificables y debidamente separados. Cabe resaltar, que este mismo nivel de modularidad se aplica de manera idéntica en cada clase presente en la aplicación web, lo cual involucra a todos los modelos, controladores, middleware, y demás. Adicionalmente, en los componentes que es posible se han modularizado de manera adecuada las diferentes estructuras y componentes.

Figura 4-5: Modularidad a nivel del código fuente



Tal como se puede apreciar la Figura 4-5 las rutas de la aplicación web están debidamente separadas en grupos acorde al recurso al que corresponden, al nivel del código fuente, al igual que las clases que componen esta aplicación.

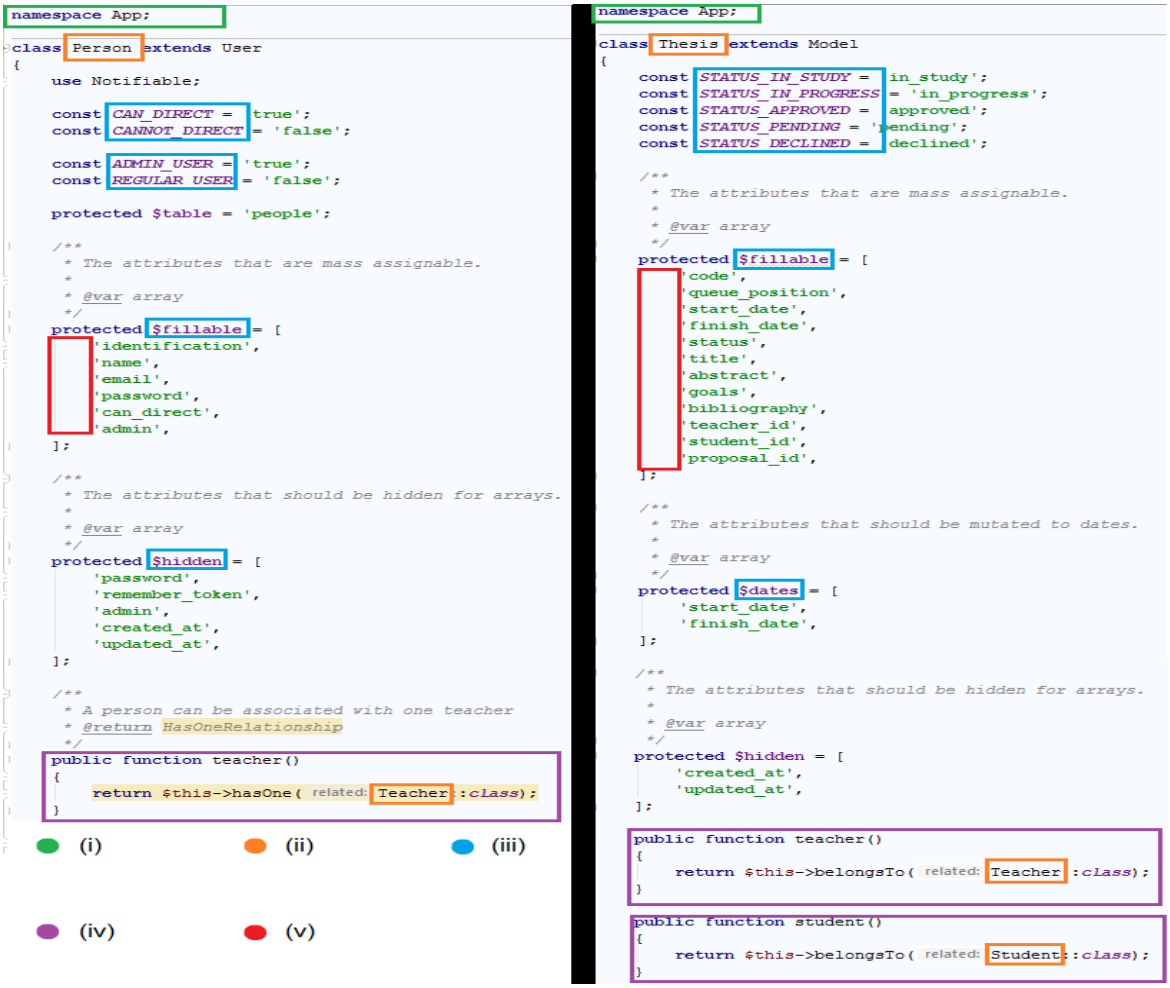
4.2 Consistencia

El prototipo implementado se construyó haciendo uso del framework web, Laravel, el cual funciona con PHP. Como se mencionó en el capítulo anterior, PHP define diferentes prácticas a seguir al momento de escribir el código fuente de la aplicación web (PHP

Framework Interop Group, 2016a, 2016b), sin embargo no proporciona un nivel de detalle específico para determinados aspectos como la manera de nombrar las variables, método, y clases, entre otros aspectos (PHP Framework Interop Group, 2016b). Afortunadamente, Laravel como tal se apega a un estándar, basado a partir de las recomendaciones brindadas por el lenguaje PHP, pero llenando esos vacíos mencionados (Laravel, 2016, 2017).

En consecuencia, con lo mencionado anteriormente, el código fuente de la aplicación web implementada para este prototipo sigue de forma estricta el estándar sugerido por Laravel en todos sus componentes. En la Figura 4-6 se puede apreciar cómo cada clase, lo cual involucra modelos, controladores, excepciones y demás, siguen una misma estructura y prácticas de manera consistente.

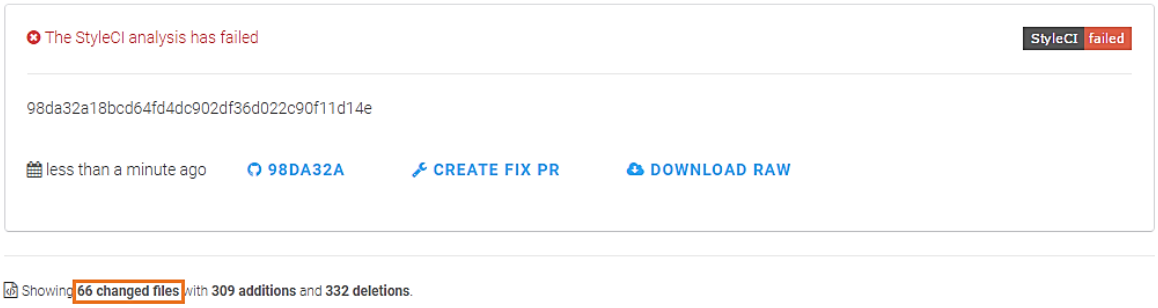
Figura 4-6: Consistencia a nivel del código fuente de las clases



En la Figura 4-6 se puede apreciar el uso de espacios de nombre de manera consistente en cada componente (i), al igual que la manera de nombrar las clases (ii), las variables (iii), los métodos (iv), y el correcto espaciado del código e indentación (v). Por supuesto, no es posible mostrar de forma sencilla todos los componentes de la aplicación de modo que se puede mostrar, sin lugar a dudas, la consistencia entre estos, por suerte existen algunas herramientas que permiten analizar y estudiar, no tanto la consistencia en los diferentes componentes, sino el estilo del código fuente en cada uno de ellos. Por supuesto, estas herramientas aplican las mismas normas de estilo sobre todos los componentes de la aplicación web, lo cual permite concluir que si todos estos las cumplen, entonces el código fuente en general de la aplicación es consistente.

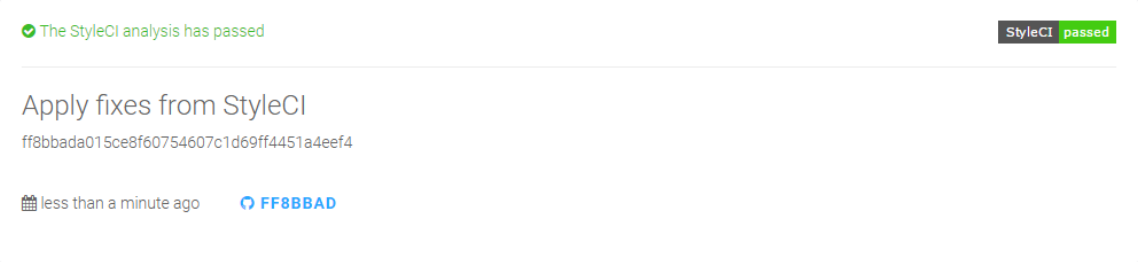
Para finalidades de lo mencionado anteriormente, se ha hecho uso de una herramienta llamada StyleCI (Alt Three, 2017), la cual es usada también por el framework Laravel. En la Figura 4-7 se muestra el resultado de un análisis del código fuente final de la aplicación web.

Figura 4-7: Resultados de análisis de estilo y consistencia del prototipo



Como se puede apreciar en la figura, se analizaron todos los archivos que componen la aplicación web, y se realizaron modificaciones en 66 de ellos, obteniendo un resultado que indica las correcciones a implementar, siendo en su mayoría asuntos de estilo relacionados con saltos de línea, comentarios y uso de espacios de nombre innecesarios.

Figura 4-8: Resultados positivos de consistencia y estilo

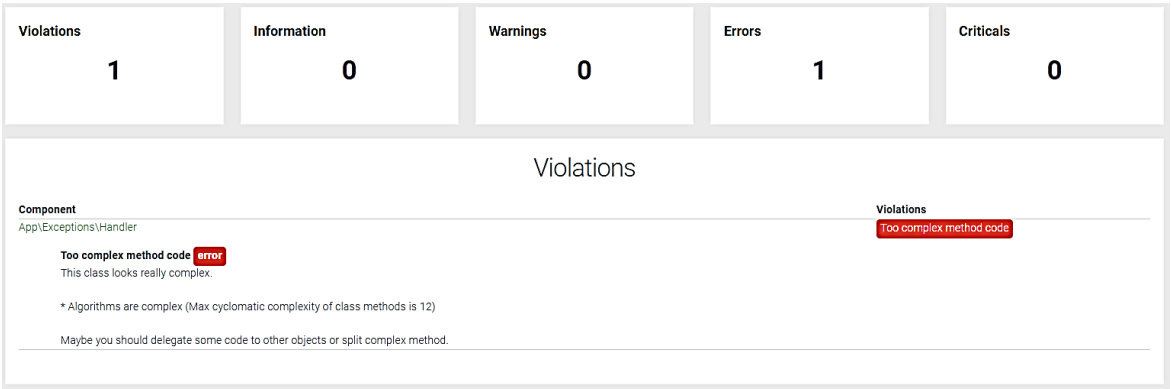


Una vez implementadas esas correcciones, tal como muestra la **Figura 4-8**, se puede evidenciar de manera confiable el correcto seguimiento de los estilos de código sugeridos por Laravel y PHP y por ende la consistencia de las buenas prácticas en el código fuente de la aplicación web.

4.3 Simplicidad

Para evidenciar la simplicidad, se debe evitar cualquier tipo de complejidad en el código fuente. Existen diferentes herramientas que permiten realizar un análisis completo del código fuente de una aplicación en busca de diferentes problemas que pueden afectar la mantenibilidad, entre los cuales se encuentra la existencia de complejidad en cualquiera de los componentes. Para tal fin, se hace uso de la herramienta llamada PhpMetrics (Lépine, 2017) la cual realiza de manera automatizada un análisis completo de la aplicación web.

Figura 4-9: Reporte de error de simplicidad



En la Figura 4-9 se muestran los resultados de un primer análisis, en los cuales se evidencia un problema de simplicidad relacionado directamente con el archivo encargado del control de las diferentes excepciones de la aplicación web. La figura muestra el detalle del problema donde se indica que uno de los métodos implementados en este componente tienen un alto índice de complejidad pues, en términos generales es muy amplio (posee un alto número de líneas de código).

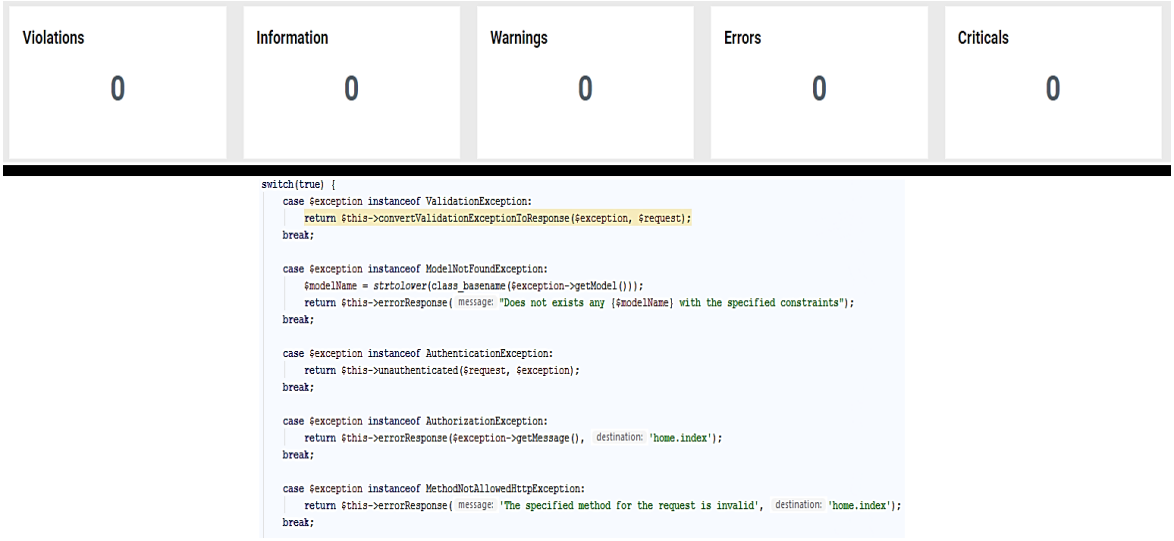
Este análisis es completamente acertado, pues de modo específico y como se puede apreciar en la Figura 4-10 el método “render” de este componente cuenta con 47 líneas de código, las cuales, casi en su totalidad son condicionales aumentando la complejidad del código para este componente específico. Cabe resaltar que el componente analizado es parte propia del prototipo implementado y no es un componente propio del framework Laravel.

Figura 4-10: Secuencia de líneas compleja en el prototipo creado

```
if ($exception instanceof ValidationException) {
    return $this->convertValidationExceptionToResponse($exception, $request);
}

if ($exception instanceof ModelNotFoundException) {
    $modelName = strtolower(class_basename($exception->getModel()));

    return $this->errorResponse( message: "Does not exists any {$modelName} with the specified constraints");
}

if ($exception instanceof AuthenticationException) {
    return $this->unauthenticated($request, $exception);
}

if ($exception instanceof AuthorizationException) {
    return $this->errorResponse($exception->getMessage(), destination: 'home.index');
}

if ($exception instanceof MethodNotAllowedHttpException) {
    return $this->errorResponse( message: 'The specified method for the request is invalid', destination: 'home.index');
}

if ($exception instanceof NotFoundHttpException) {
    return $this->errorResponse( message: 'The specified URL cannot be found', destination: 'home.index');
}

if ($exception instanceof HttpException) {
    return $this->errorResponse($exception->getMessage(), destination: 'home.index');
}

if ($exception instanceof QueryException) {
    $errorCode = $exception->errorInfo[1];
```

Para reducir la complejidad de este método y así resolver el problema de simplicidad de la aplicación web, se han sustituido todos los condicionales por una estructura switch, permitiendo reducir la complejidad como se evidencia en la Figura 4-11

Figura 4-11: Solución del problema de complejidad



En definitiva, se puede mostrar cómo después de la corrección implementada, el código en general ya la aplicación cuenta con un nivel de simplicidad adecuado, pues no existen más errores reportados con relación a este factor de mantenibilidad.

4.4 Facilidad de lectura

En términos generales, la facilidad de lectura, busca que las líneas de código sean cortas, concisas y sin mucha complejidad. Al hablar de complejidad, se hace referencia a la inexistencia de un alto número de operadores o llamadas a métodos dentro de una misma línea de código.

A modo de ejemplo, en la Figura 4-12 se muestra un caso de una línea de código compleja y de difícil lectura, pues está compuesta por llamadas a múltiples métodos encadenados, además de tener una longitud considerable.

Figura 4-12: Línea de código de difícil lectura

```
return view( view: 'theses.index')->with(['asStudent' => $asStudentThesis, 'asTeacher' => $asTeacherThesis]);
```

Por suerte, resolver este tipo de problemas suele ser sencillo si se han seguido algunas prácticas de mantenibilidad como modularidad y en especial simplicidad, pues basta con dividir una misma línea de código en varias de modo que se evidencie con facilidad cada uno de los componentes allí utilizados, con su correspondiente indentación.

Figura 4-13: Línea de código de fácil lectura

```
return view( view: 'theses.index')
        ->with([
            'asStudent' => $asStudentThesis,
            'asTeacher' => $asTeacherThesis
        ]);
```

Como se muestra en la Figura 4-13 la facilidad de lectura de una misma línea de código se pudo mejorar separando, en este caso, las diferentes llamadas a los métodos usados en líneas separadas, al igual que los elementos del arreglo allí presente, reduciendo así la longitud final de la línea y facilitando su lectura.

Finalmente, cabe enfatizar en que la facilidad de lectura va de la mano con las buenas prácticas y la consistencia del código fuente, al hacer uso de variables, métodos y demás componentes con nombres descriptivos y concretos.

4.5 Promedio de “variables vivas”

El promedio de variables vivas es fácilmente calculado al dividir el número de variables vivas encontradas respecto al número de sentencias ejecutables. El problema surge al momento de determinar cuándo una variable es viva o no en una sentencia de código específica.

Ahora bien, no se encuentran herramientas que permitan determinar la cantidad de variables vivas presentes en un componente específico de código. Por supuesto, basándose en la definición de este factor de mantenibilidad, se ha procurado referenciar

variables de manera repetitiva en múltiples sentencias de código. En la Figura 4-14 se muestra uno de los pocos casos en los que una variable fue referenciada en repetidas ocasiones en un mismo componente de la aplicación web, sin embargo, no es un caso común.

Figura 4-14: Variable viva

```
public function render($request, Exception $exception)
{
    if ($exception instanceof ModelNotFoundException) {
        $modelName = strtolower(class_basename($exception->getModel()));

        return $this->errorResponse( message: "Does not exists any {$modelName} with the specified constraints");
    }

    if ($exception instanceof AuthenticationException) {
        return $this->unauthenticated($request, $exception);
    }

    if ($exception instanceof AuthorizationException) {
        return $this->errorResponse($exception->getMessage(), destination: 'home.index');
    }
```

Dada la dificultad de evidenciar este factor de mantenibilidad, se ha optado por mostrar un escenario en el cual fue necesaria la existencia de una variable viva, con el propósito de demostrar que se les ha identificado y que, por supuesto, no son comunes en el código fuente de la aplicación web completa.

4.6 Proporción de comentarios

En el código fuente de la aplicación web, se buscó reducir la cantidad de comentarios al máximo. De hecho, todos los comentarios existentes en el código de la aplicación web son los recomendados en las buenas prácticas para documentar los diferentes métodos de los componentes de la aplicación, junto con descripciones concretas.

Figura 4-15: Bloque de comentarios de un método

```
/**
 * Used when a student accepts a thesis on the first position
 * @param  App\Thesis $thesis
 * @return Http\Response
 */
public function update(Thesis $thesis)
{
```

Como se aprecia en la Figura 4-15 los métodos declarados en la aplicación web, cuentan con un bloque de comentarios que explica de manera concreta su propósito, tipos de datos para los parámetros y tipo de dato de retorno (si aplica).

Para este factor de mantenibilidad, el código fuente se ha apoyado en las buenas prácticas de programación y de consistencia para el lenguaje PHP, lo cual implica el uso de métodos, clases y variables con nombres descriptivos, de modo que no sea necesario agregar comentarios para reforzar la facilidad de comprensión final del código, pues las sentencia se explican por sí solas gracias al uso de estas prácticas a lo largo de toda la aplicación.

Ahora, en la Figura 4-16 se muestra el formato estándar seguido para todos los bloques de comentarios definidos, manteniendo la consistencia en este aspecto con dada uno de estos bloques.

Figura 4-16: Plantilla para bloque de comentarios

```
/**
 * [method description]
 * @param  [type] $paramName
 * @return [type]
 */
```

Tal estándar indica que se debe especificar una descripción breve al comienzo del bloque, seguida por la lista de parámetros que recibe el método (@param), en caso de que aplique, y finalmente el tipo de retorno del método (@return).

4.7 Comentarios apropiados

En términos generales, la existencia de un comentario es netamente informativa y es importante que las pocas líneas que se usen en líneas de comentario tengan un propósito concreto y de valor.

Figura 4-17: Bloque de comentarios apropiados

```
/**
 * Allowing to add a single constrain to obtain people who can direct
 * @param  QueryBuilder
 * @return QueryBuilder
 */
public function scopeCanDirect($query)
```

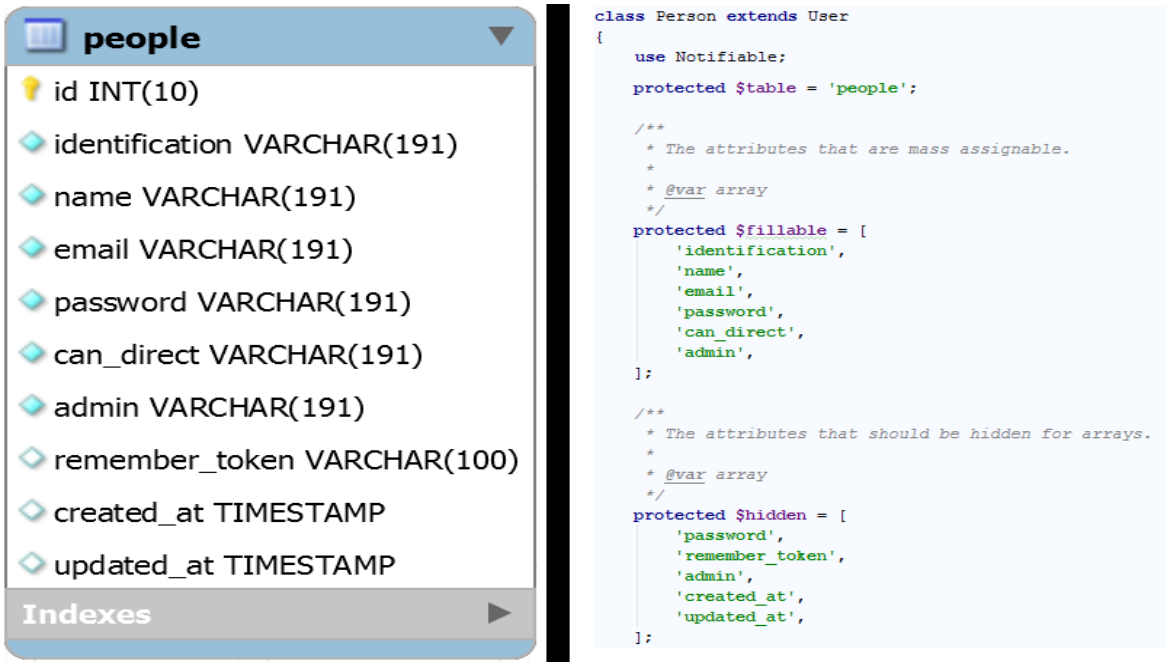
Por fortuna, al seguir el formato estándar mostrado en la Figura 4-16, se consigue evidenciar de forma contundente los parámetros y tipo de retorno de cada método asociado a un bloque de comentarios.

Finalmente, como se muestra en la Figura 4-17 la descripción en la línea inicial del bloque de comentarios, se trata de un texto corto y concreto que apoye sin lugar a dudas lo que ya los parámetros y nombre del método sugieren. En ese caso particular, la descripción muestra el propósito del funcionamiento del método lo cual va en concordancia con lo que sugiere el nombre del mismo. Este mismo pensamiento se aplica de forma consistente en todos los bloques de comentarios presentes en la aplicación web.

4.8 Objetos y clases

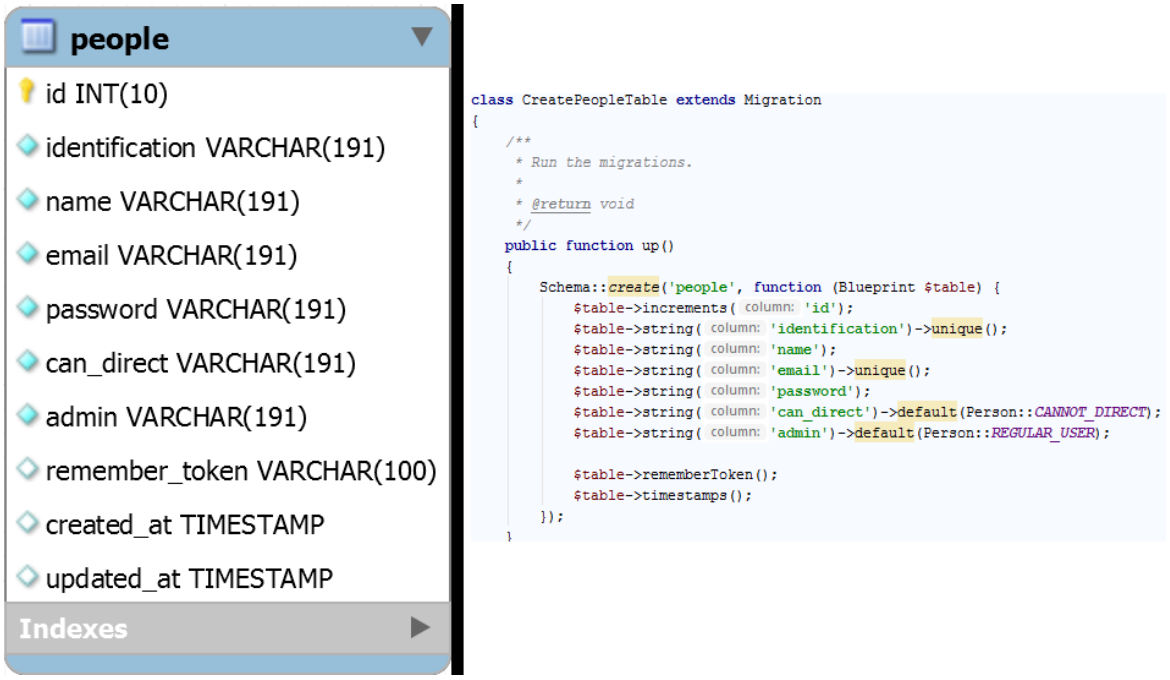
La trazabilidad entre los objetos y clases construidos en el código fuente de la aplicación web y el diseño original de la misma, incrementa la mantenibilidad de la aplicación al facilitar la identificación de los componentes involucrados en determinadas funcionalidades a partir del diseño original.

Figura 4-18: Trazabilidad entre entidad original y el modelo implementado



De este modo el desarrollador/lector de la aplicación web, podrá tener la certeza de que cada entidad y atributo especificados en el diseño original se encuentra adecuadamente implementado en el código fuente. En la Figura 4-18 se puede evidenciar la trazabilidad precisa que existe entre el modelo Person de la aplicación web y la entidad People (originalmente proveniente de la Figura 4-2).

Figura 4-19: Trazabilidad entre entidad original y migración de tabla en el prototipo



En adición a lo anterior, en la **Figura 4-19** se muestra la trazabilidad precisa, no sólo entre los nombres de los atributos especificados en el diseño original, sino también sus tipos y características (claves foráneas, claves únicas y claves primarias).

4.9 Escenario concreto de mantenibilidad en el prototipo implementado

Como se introdujo en la sección 2.2, en el proceso de desarrollo de cualquier aplicación web mantenible, lo cual incluye el prototipo implementado para este trabajo, se introducen diferentes factores que permiten evidenciar la mantenibilidad considerada y por tanto es interesante mostrar un caso concreto de cómo el uso e implementación adecuados de la mantenibilidad, facilitan y sustentan la integración de cambios en la aplicación web. En esta sección se muestra cómo se evidenciaron tales factores para así asegurar una adecuada mantenibilidad general de la aplicación web construida.

Se muestra entonces, cómo a partir de un nuevo requerimiento para modificar una funcionalidad existente, la correcta aplicación de los diferentes factores de mantenibilidad, facilita llevar a cabo tal requerimiento.

Para ilustrar el modo en el que los diferentes parámetros de mantenibilidad y su respectiva implementación en el prototipo creado impactan la facilidad de modificación de una funcionalidad o componente ya existente, se propone al lector hacerse con la siguiente idea: “No es muy relevante el tipo de modificación que es requerida, lo que es de importancia es que se requiere realizar un cambio cualquiera y ello implica una serie de tareas que el desarrollador debe identificar con facilidad, para luego completar su labor en el menor tiempo y con el menor esfuerzo posibles.”

Ahora bien; la correcta implementación de los diferentes factores de mantenibilidad le será de ayuda al desarrollador en el proceso de implementación del requerimiento. De modo específico, se ilustrará, como en el prototipo creado, los diferentes factores de mantenibilidad implementados son de utilidad para este fin.

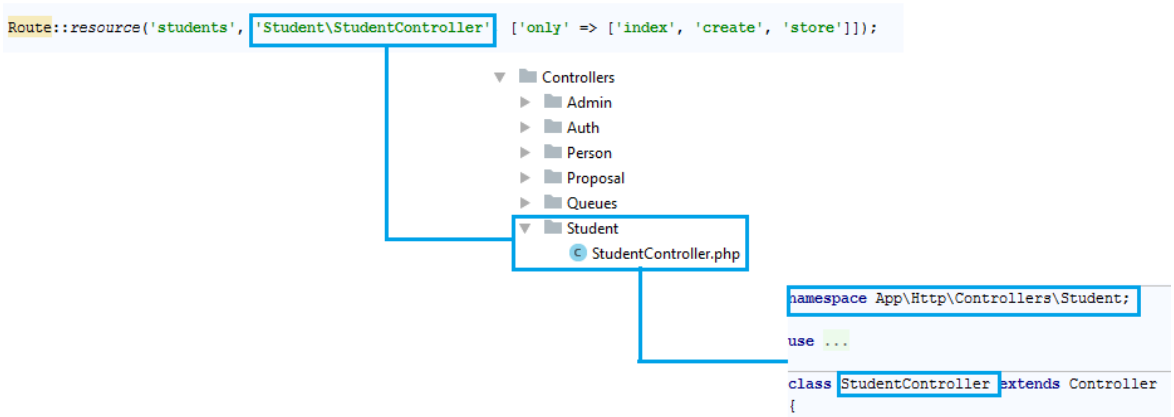
Continuando con lo anterior, bajo la necesidad de modificar una funcionalidad existente, el desarrollador deberá identificar los componentes que se ejecutan al momento de ejecutar una acción específica de la aplicación web prototipo. Para esto, el archivo de rutas es de vital importancia, pues le permitirá saber, a partir de una URL o ruta específica, la acción que se ejecuta y su ubicación. En la Figura 4-20 se aprecian las diferentes rutas que componen la aplicación web, donde se indica el componente (en este caso un controlador) encargado de cada conjunto de acciones, aquí la modularidad entra en acción, permitiendo al desarrollador identificar diferentes grupos de rutas que están separados según el recurso a usar (profesores, estudiantes, etc.). Con el conocimiento previo del recurso que debe intervenir, podrá conocer el controlador (o controladores) a los que debe dirigirse según sea el caso.

Figura 4-20: Rutas del prototipo debidamente modularizadas

```php
Route::group(['middleware' => 'auth', 'prefix' => 'home', 'as' => 'home.'], function () {
    Route::resource('proposals', 'Proposal\ProposalController', ['except' => ['show']]);
    Route::resource('students', 'Student\StudentController', ['only' => ['index', 'create', 'store']]);
    Route::resource('teachers', 'Teacher\TeacherController');
    Route::resource('theses', 'Thesis\ThesisController', ['except' => ['edit']]);
    Route::group(['prefix' => 'queues', 'as' => 'queues.'], function () {
        Route::get('/first-position', 'Queues\QueueController@showFirstPositionTheses')->name('first-position');
    });
    Route::group(['prefix' => 'admins', 'as' => 'admins.'], function () {
        Route::group(['prefix' => 'manage', 'as' => 'manage.'], function () {
            Route::get('/can-lead', 'Admin\AdminController@manageCanLead')->name('can-lead');
            Route::get('/revoke-lead/{person}', 'Admin\AdminController@revokeLead')->name('revoke-lead');
            Route::get('/cannnot-lead', 'Admin\AdminController@manageCannotLead')->name('cannot-lead');
            Route::get('/aprove-lead/{person}', 'Admin\AdminController@aproveLead')->name('aprove-lead');
        });
        Route::get('/', 'Admin\AdminController@index')->name('index');
    });
});
```

En concordancia con lo anterior, a este punto la modularidad y la consistencia del prototipo implementado, le permiten identificar sin lugar a dudas el o los controladores a intervenir, debido a que cada uno de estos, está debidamente separado en carpetas relacionadas con su recurso y manteniendo exactamente el mismo nombre indicado originalmente en las rutas. En la Figura 4-21 se aprecia la consistencia entre el nombre indicado en el grupo de rutas y el nombre y ubicación de un controlador dado, junto con la modularidad presente en la organización de tales controladores.

Figura 4-21: Consistencia entre componentes de la aplicación web



Una vez que el desarrollador ha identificado el componente (en este caso el controlador) a intervenir, debe identificar la acción o método específico que requiere ser modificado. En este punto entra en acción la proporción de comentarios y los comentarios apropiados, pues de ese modo el desarrollador puede identificar el método que realiza la acción que debe ser modificada. En la Figura 4-22 se evidencia cómo el bloque de comentarios de un método al interior de un controlador (y de hecho al interior de los demás componentes) indica tanto el propósito específico de tal método, junto con el tipo de parámetros que recibe y valores que retorna (cuando aplica).

Figura 4-22: Bloque de comentarios descriptivo, en uno de los controlador

```php
/**
 * Display a listing of the proposals (all of them).
 *
 * @return \Illuminate\Http\Response
 */
public function index()
{
    $currentThesis = Auth::user()
        ->student
        ->theses()
        ->with('proposal')
        ->get()
        ->pluck('proposal.id')
        ->unique();

    $proposals = Proposal::all()
        ->except($currentThesis->toArray());

    $teachers = Teacher::all()->except(Auth::id());

    return view( view: 'proposals.index')
        ->with([
            'proposals' => $proposals,
            'teachers' => $teachers
        ]);
}
```

Después de lo anterior, el desarrollador tendrá que comprender el funcionamiento actual del método a modificar para determinar los segmentos de código a intervenir o los lugares dónde agregar otros. Para tal propósito entra a ser de ayuda una vez más la consistencia permitiendo tener la certeza de que el código está estructurado de manera consistente en todo el proyecto y así conocer qué estilo de código y prácticas adoptar, junto con la

simplicidad y facilidad de lectura, las cuales le permitirán al desarrollador comprender con facilidad el funcionamiento de cada sentencia de código ejecutable para consecuentemente determinar las acciones a llevar a cabo.

Tal como se puede observar en la Figura 4-23 es habitual que las diferentes acciones de un controlador involucren al menos un modelo y una vista. En este punto, la modularidad y la consistencia entran en juego, para que el desarrollador tenga la certeza de que los diferentes modelos que requiera estén debidamente organizados, además de que sigan las prácticas recomendadas e implementadas en la aplicación web.

Figura 4-23: Controlador haciendo uso de un modelo y una vista

```
public function manageCanLead()
{
    $canLead = Person::canDirect()->get();

    return view( view: 'admin.manage.can-lead')->with(['canLead' => $canLead]);
}
```

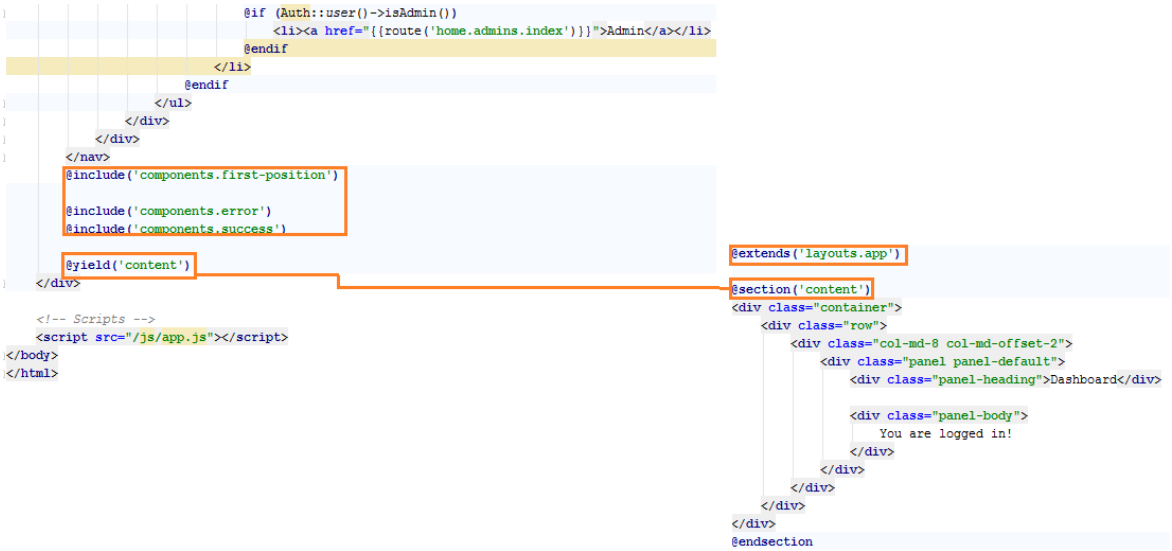
- Modelo
- Vista

Ahora bien, la Figura 4-23 muestra cómo una acción de un controlador hace uso de un modelo en su interior, junto con una vista. Desde el aspecto del modelo, cabe resaltar que la importancia de este, radica en el modo en el que las diferentes responsabilidades o acciones que un recurso puede realizar (profesor, estudiante, curso, etc.) se centralizan desde allí, evitando que tales acciones deban ser implementadas de forma repetitiva e innecesaria, además de proporcionar un punto central en el cual intervenir para modificar alguno de esos aspectos y así, propagar tal modificación en toda la aplicación web. La **Figura 4-24** muestra cómo desde el controlador se hace uso directo de una se las diferentes responsabilidades o funciones del modelo y cómo está ésta definida en tal modelo.

Figura 4-24: Acciones centralizadas en el modelo, usadas desde el controlador

Por otro lado, un controlador también hace uso de una vista, de hecho, el controlador simplemente indica la vista a retornar o usar y envía los datos que ésta puede requerir. Manteniendo la modularidad en mente, al nivel de las vistas, éstas están debidamente separadas según el recurso, tal como se muestra en la Figura 4-4, pero además se ha incluido modularidad al nivel de las vistas mismas, separándolas en componentes que se pueden reutilizar en toda la aplicación web. De ese modo, los componentes que son comunes a todas las vistas, como encabezados, pie de página, estilos, scripts, entre otros, estarán definidos en un único componente central y podrán ser fácilmente modificados reflejando así los cambios en toda la aplicación web, sin mayor dificultad.

Figura 4-25: Separación en módulos y componentes en las vistas



Como se puede apreciar en la parte izquierda de la Figura 4-25, se define un componente principal que establece la estructura común de las vistas, junto con el uso de otros componentes que pueden estar separados (@include) y finalmente permite implementar de manera personalizada los contenidos variables al interior de la aplicación web (@yield). Una vez que el componente define esto, cualquier vista puede extender ese componente (@extends) implementando la sección correspondiente (@section), tal y como se aprecia en la parte derecha de la misma figura. Al poder dividir las vistas en módulos e inclusive centralizar las estructuras repetitivas, se aumenta la modularidad de la aplicación web y por ende la mantenibilidad de la misma en estos componentes.

Con todo lo anterior, se puede evidenciar cómo la adecuada implementación de los diferentes factores de mantenibilidad facilita la inevitable modificación de las diferentes funcionalidades actuales, llevando al desarrollador secuencialmente a cada uno de los componentes que debe identificar hasta llegar a las sentencias ejecutables que debe modificar o agregar en el código fuente.

5. Conclusiones y recomendaciones

5.1 Conclusiones

Dar prioridad al código fuente para la medición y determinación de la mantenibilidad de una aplicación web, es más sencillo que con otras instancias y brinda elementos evidenciables de esto en las diferentes unidades y líneas de código relevantes.

Debido a que el código fuente de una aplicación web es el resultado de un proceso de planeación y diseño, es posible que justo al momento de implementar la aplicación se descubran problemas relacionados con la mantenibilidad que provenían desde el diseño de la misma, y presentando estas problemáticas en una etapa muy avanzada del ciclo de vida de la aplicación web.

Determinados factores de mantenibilidad pueden depender directamente de las intenciones y probabilidad de cambio de la aplicación web al mediano y largo plazo. A partir de ese conocimiento, el desarrollador podrá determinar si la generalización del código traería problemas de mantenibilidad a futuro, puesto que esta generalización tiende a aumentar la complejidad del código cuando se necesitan funciones más adaptadas y específicas que genéricas.

Paradójicamente, la consistencia de las aplicaciones web es inconsistente a nivel de lenguajes o herramientas de desarrollo, pues cada lenguaje, framework y tecnología en general, tiene sus propias definiciones para las prácticas recomendadas a seguir durante el desarrollo y son pocos los parámetros en los que concuerdan.

En términos generales, la consistencia de una aplicación web, a nivel del código fuente, queda sujeta a las buenas prácticas del desarrollador (y su equipo de trabajo), esperando a que todos estos se apeguen a un mismo estándar y lo sigan sin excepción alguna a lo largo de toda la aplicación, pues los lenguajes de programación no son claros o concuerdan en tales prácticas.

Por fortuna, en la actualidad se evidencia cómo diferentes desarrolladores se apegan estrictamente a las buenas prácticas de desarrollo en los diferentes lenguajes y herramientas que usan, pues existe una conciencia general de lo importante de ello para la mantenibilidad general de las aplicaciones web y facilitar así también las contribuciones de otros desarrolladores en una misma aplicación web.

Múltiples herramientas facilitan el análisis automatizado del código en poco tiempo, permitiendo al desarrollador centrarse en la funcionalidad del código, sin dejar de lado las buenas prácticas, pero sin preocuparse por los detalles más ínfimos, pues con el uso regular de estas herramientas el desarrollador podrá mantener controlados diferentes factores de mantenibilidad y de calidad de la aplicación web.

5.2 Recomendaciones

Sería interesante profundizar un poco más en la consistencia de las aplicaciones web, a nivel del código fuente, en la búsqueda de un estándar o practicas comunes de modo transversal a los diferentes lenguajes de programación y herramientas de desarrollo. Es evidente que se hace necesaria la existencia de un estándar único universal de buenas prácticas, de modo que se pueda asegurar la consistencia entre el código fuente de diferentes desarrolladores y aplicaciones web, incluso cuando se hayan usado diferentes lenguajes de programación y herramientas de desarrollo.

La literatura es un poco vaga en ciertas definiciones para los factores de mantenibilidad, y de modo particular, el promedio de variables vivas es uno de los factores menos explorados y que presenta una dificultad para ser comprendida y evidenciada. Sería adecuado explorar con más profundidad su definición y especialmente mecanismos que permitan detectar tales variables de modo simple o incluso automático.

Sería atractivo profundizar en la manera en la que determinados factores de mantenibilidad se ven afectados por intenciones y probabilidades de cambio de la aplicación web al mediano y largo plazo. En otras palabras, cómo a partir del conocimiento previo de la necesidad de cambio a lo largo del tiempo permita determinar cómo abordar un factor de mantenibilidad como, por ejemplo, determinar si la generalización del código traería problemas de mantenibilidad a futuro, puesto que tendería a aumentar la complejidad del código a medida que se requieren funcionalidades más especializadas.

Bibliografía

Abilio, R., Teles, P., Costa, H., & Figueiredo, E. (2012). A Systematic Review of Contemporary Metrics for Software Maintainability. *2012 Sixth Brazilian Symposium on Software Components, Architectures and Reuse*, (Dcc), 130–139. https://doi.org/10.1109/SBCARS.2012.15

Acevedo Orrego, M. A. (2008). Gestión de Trabajos Dirigidos de Grado.

Aggarwal, K. K., Singh, Y., Chandra, P., & Puri, M. (2005). Measurement of Software Maintainability Using a Fuzzy Model. *Journal of Computer Science*, *1*(4), 538–542. https://doi.org/10.3844/jcssp.2005.538.542

Al-Jamimi, H. A., Alshayeb, M., & Elish, M. O. (2011). Investigating the effect of aspect-oriented refactoring on software maintainability. *Communications in Computer and Information Science*, *181 CCIS*(PART 3), 611–623. https://doi.org/10.1007/978-3-642-22203-0_52

Alt Three. (2017). StyleCI. Retrieved May 17, 2017, from https://styleci.io/

Anda, B. (2007). Assessing software system maintainability using structural measures and expert assessments. *IEEE International Conference on Software Maintenance, ICSM*, 204–213. https://doi.org/10.1109/ICSM.2007.4362633

Braude, E. J., & Bernstein, M. E. (2016). *Software Engineering: Modern Approaches* (2nd ed.). Waveland Press, Inc.

Chae, H. S., Kim, T. Y., Jung, W. S., & Lee, J. S. (2007). Using metrics for estimating maintainability of web applications: An empirical study. *Proceedings - 6th IEEE/ACIS International Conference on Computer and Information Science, ICIS 2007; 1st IEEE/ACIS International Workshop on E-Activity, IWEA 2007*, (Icis), 1053–1058. https://doi.org/10.1109/ICIS.2007.192

Devi, U., Sharma, A., & Kesswani, N. (2016). A review on quality models to analyse the impact of refactored code on maintainability with reference to software product line, 3705–3708.

Di Lucca, G. A., Fasolino, A. R., Tramontana, P., & Visaggio, C. A. (2004). Towards the definition of a maintainability model for web applications. *Eighth*

European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings., 279–287. https://doi.org/10.1109/CSMR.2004.1281430

ESA Board for Software Control Standardisation and (BSSC). (2000). Java Coding Standards. *Methods*, *2005*(1), 1–33.

Ghosheh, E., Qaddour, J., Kuofie, M., & Black, S. (2006). A comparative analysis of maintainability approaches for web applications. *IEEE International Conference on Computer Systems and Applications, 2006*, *2006*(M), 1155–1158. https://doi.org/10.1109/AICCSA.2006.205235

Ghosheh, E., Systems, S., Black, S., Kapetanios, E., & Baldwin, M. (2010). Exploring the Relationship between UML De- sign Metrics for Web Applications and Main- tainability. *Journal of Object Technology*, *2*(3).

Gosheh, E., Black, S., & Qaddour, J. (2008). Design metrics for web application maintainability measurement. *WestminsterResearch*.

Gu, X. (2016). The impact of maintainability on the manufacturing system architecture. *International Journal of Production Research*, *7543*(November), 1–19. https://doi.org/10.1080/00207543.2016.1254356

Hashim, K., & Key, E. (1996). A software maintainability attributes model. *Malaysian Journal of Computer Science*, *9*(2), 92–97.

ISO 9126 International Standard ISO/IEC 9126: Software Engineering - Product Quality, Pub. L. No. 9126 (2001). Retrieved from https://www.iso.org/standard/22749.html

Jabangwe, R., Borstler, J., Smite, D., & Wohlin, C. (2015). Empirical evidence on the link between object-oriented measures and external quality attributes: A systematic literature review. *Empirical Software Engineering*, *20*(3), 640–693. https://doi.org/10.1007/s10664-013-9291-7

Kataoka, Y., Imai, T., Andou, H., & Fukaya, T. (2002). A quantitative evaluation of maintainability enhancement by refactoring. *Software Maintenance, 2002. Proceedings. International Conference on*, 576–585. https://doi.org/10.1109/ICSM.2002.1167822

Kumar, B. (2012). A survey of key factors affecting software maintainability.

Proceedings: Turing 100 - International Conference on Computing Sciences, ICCS 2012, 261–266. https://doi.org/10.1109/ICCS.2012.5

Kumar Pandey, A., & Agrawal, C. P. (2016). Fuzzy ANP Model to Measure the Maintainability of Desktop Software based on Software Development Factors. *Indian Journal of Science and Technology*, *9*(33). https://doi.org/10.17485/ijst/2016/v9i33/100218

Kundu, S., & Tyagi, K. (2016). Effort Estimation of Software Maintainability Using Soft Computing Techniques : a Critical Literature Survey, (January). https://doi.org/10.5829/idosi.wasj.2016.34.6.47

Laravel. (2016). Laravel Code Style Guide. Retrieved May 17, 2017, from https://github.com/laravel/framework/blob/5.4/.php_cs

Laravel. (2017). Laravel Coding Style. Retrieved May 17, 2017, from https://laravel.com/docs/5.4/contributions#coding-style

Lépine, J.-F. (2017). PhpMetrics. Retrieved May 19, 2017, from http://www.phpmetrics.org/

Malhotra, R., & Chug, A. (2016). An empirical study to assess the effects of refactoring on software maintainability. *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 110–117. https://doi.org/10.1109/ICACCI.2016.7732033

Martínez, Y., Cachero, C., Matera, M., Abrahao, S., & Luján, S. (2011). Impact of MDE approaches on the maintainability of web applications: An experimental evaluation. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *6998 LNCS*, 233–246. https://doi.org/10.1007/978-3-642-24606-7_18

Martínez, Y., Cachero, C., & Meliá, S. (2014). Empirical study on the maintainability of Web applications: Model-driven Engineering vs Code-centric. *Empirical Software Engineering*, *19*(6), 1887–1920. https://doi.org/10.1007/s10664-013-9269-5

Maurya, S. L., & Shankar, G. (2012). Maintainability Assessment of Web Based Application. *Journal of Global Research in Computer Science*, *3*(7), 30–33.

McIntosh, S., Kamei, Y., Adams, B., & Hassan, A. E. (2016). An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, *21*(5), 2146–2189. https://doi.org/10.1007/s10664-015-9381-9

Meananeatra, P. (2012). Identifying refactoring sequences for improving software maintainability. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, 406. https://doi.org/10.1145/2351676.2351760

Misra, S., & Egoeze, F. (2014). Framework for maintainability measurement of web application for efficient knowledge-sharing on campus intranet. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *8583 LNCS*(PART 5), 649–662. https://doi.org/10.1007/978-3-319-09156-3_45

Mkaouer, W., Kessentini, M., Shaout, A., Koligheu, P., Bechikh, S., Deb, K., & Ouni, A. (2015). Many-Objective Software Remodularization Using NSGA-III. *ACM Transactions on Software Engineering and Methodology*, *24*(3), 1–45. https://doi.org/10.1145/2729974

Offutt, J. (2002). Quality Attributes of Web Software Applications. *IEEE Software*, *1*(April), 25–32. https://doi.org/10.1002/1529-7950(200009)1:1<15::AID-SWF3>3.3.CO;2-

Peercy, D. E. (1981). A Software Maintainability Evaluation Methodology. *IEEE Transactions on Software Engineering*, *SE-7*(4), 343–351. https://doi.org/10.1109/TSE.1981.234534

PHP Framework Interop Group. (2016a). PHP Standards Recommendations. Retrieved May 6, 2017, from http://www.php-fig.org/psr/

PHP Framework Interop Group. (2016b). PSR-1: Basic Coding Standard. Retrieved May 6, 2017, from http://www.php-fig.org/psr/psr-1/

Riaz, M., Breaux, T., & Williams, L. (2015). How have we evaluated software pattern application? A systematic mapping study of research design practices. *Information and Software Technology*, *65*, 14–38. https://doi.org/10.1016/j.infsof.2015.04.002

Rodríguez, M., & Fernández, C. M. (2015). Certificación de la Mantenibilidad del Producto Software : Un Caso Práctico, *3*(3), 127–134.

Silva, D., Tsantalis, N., & Valente, M. T. (2016). Why we refactor? confessions of GitHub contributors. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, 858–870. https://doi.org/10.1145/2950290.2950305

Stella, L. F. F., Jarzabek, S., & Wadhwa, B. (2008). A comparative study of maintainability of web applications on J2EE, .NET and ruby on rails. *Proceedings - 10th IEEE International Symposium on Web Site Evolution, WSE 2008*, 93–99. https://doi.org/10.1109/WSE.2008.4655401

Till, D., Ceolin, D., & Visser, J. (2016). Towards a Benchmark for the Maintainability Evolution of Industrial Software Systems, 11–21. https://doi.org/10.1109/IWSM-Mensura.2016.35

van Rossum, G., Warsaw, B., & Coghlan, N. (2001). Style Guide for Python Code. Retrieved May 6, 2017, from https://www.python.org/dev/peps/pep-0008/

Vern, R., & Dubey, S. K. (2014). A review on appraisal techniques for web based maintainability. *Proceedings of the 5th International Conference on Confluence 2014: The Next Generation Information Technology Summit*, (2004), 795–799. https://doi.org/10.1109/CONFLUENCE.2014.6949320

Whyte, A. (2014). Google JavaScript Style Guide.

Zanoni, M., Perin, F., Fontana, F. A., & Viscusi, G. (2014). Pattern detection for conceptual schema recovery in data-intensive systems. *Journal of Software: Evolution and Process*, *26*(12), 1172–1192. https://doi.org/10.1002/smr