**Universidad de los Andes**

**Facultad de Ingeniería**

**Departamento de Ingeniería de Sistemas**

**Grupo JJACS**

**Santiago Bacaro**

**Camilo Forero**

**Andrea Navas**

**Jose A. Riaño**

**Juan Sebastián Ruiz**

**Reporte de Modificabilidad**

**Entrega parcial del experimento 3**

**Arquitectura y diseño de Software**

**Bogotá, Colombia**

**Noviembre, 2015**

Contenido

[1. Introducción 3](#_Toc435439804)

[2. Arquitectura MVC en Ruby on Rails 3](#_Toc435439805)

[3. Aspectos considerados para la modificabilidad 4](#_Toc435439806)

[3.1 Manejo de la base de datos: 4](#_Toc435439807)

[3.2 No repetir código 6](#_Toc435439808)

[3.3 No sobrecargar archivos 7](#_Toc435439809)

[5. Inspección de código 8](#_Toc435439810)

[5. Conclusiones 11](#_Toc435439811)

# Introducción

El experimento 3 es la última iteración del proyecto desarrollado durante el proyecto. Éste al igual que los dos anterior se ha dividido en dos fases a saber, una entrega parcial y una final. Para la primera se hace énfasis en la necesidad de garantizar los atributos de calidad de modificabilidad y disponibilidad. El siguiente reporte pretende dar cuenta de lo realizado para lograr que el software realizado sea modificable. En primera instancia, se presenta la relación en el tipo de arquitectura y framework utilizado y el atributo mencionado, para luego exhibir los aspectos que se tuvieron en cuenta a la hora de garantizar modificabilidad (teniendo en cuenta siempre las características del framework y de la arquitectura).Posteriormente se incluyen las inspecciones realizadas a través del framework donde se realizó el proyecto así como las acciones tomadas teniendo en cuenta lo observado en tales análisis. Finalmente se presentan conclusiones respecto al proceso en general y a la modificabilidad que se garantiza.

# Arquitectura MVC en Ruby on Rails

El patrón MVC se caracteriza por la separación que se hace entre los datos, el control y las vistas. Las ventajas que ofrece este tipo de arquitectura son múltiples entre ellas: la facilidad para su comprensión, su modularidad y el débil acoplamiento entre sus componentes. Los elementos anteriormente mencionados hacen que el patrón como tal contribuya a la modificabilidad. Por este motivo, el primer paso para asegurar la existencia de dicho atributo de calidad es respetar en todo momento los lineamientos del patrón.

*Ruby on Rails,* el *framework* utilizado para desarrollar el proyecto, se caracteriza por un uso fuerte de convenciones. Si se siguen cuidadosamente las convenciones definidas se logra que los proyectos de *Rails* tengan una estructura estándar. Esto, facilita la ubicación de archivos y la comprensión del proyecto para integrantes nuevos del equipo. En este sentido, el *framework* como tal también contribuye a darle modificabilidad al proyecto.

Entre las convenciones que se manejan en Ruby on Rails se encuentra la estructura de archivos estándar que se corresponde con las características del patrón MVC. Para TBC se respetó dicha estructura con el objetivo de saber en qué lugar se encontraba cada archivo lo que contribuye a la modificabilidad. La figura 1 muestra la estructura de archivos del proyecto.

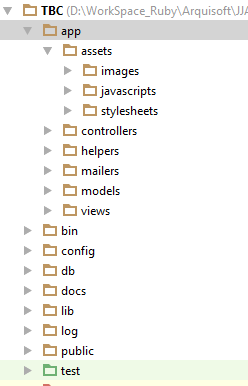


Fig. 1 Estructura de archivos de TBC

# Aspectos considerados para la modificabilidad

A continuación se describen las medidas más importantes que se tuvieron en cuenta con el objetivo de facilitar la modificabilidad del proyecto. Los aspectos considerados deben relacionarse con el patrón de arquitectura y el framework específico utilizado para que el manejo de los mismos se haga de manera adecuada.

* 1. Manejo de la base de datos:

*Ruby on* *Rails* está pensado para ser prácticamente independiente de la base de datos que se esté utilizando. Toda modificación que se deba hacer a las tablas se realiza a través unos archivos escritos en *Ruby* llamados migraciones. *Rails* lee estas migraciones y automáticamente busca y ejecuta los controladores necesarios para interactuar con la base de datos actual, de esta manera el desarrollo no se ve afectado al cambiar bases de datos.

Una característica muy importante de las migraciones es su nombre. *Rails* tiene como convención llamar a las migraciones según su fecha de creación. Si se sigue está convención, el *framework* es capaz de identificar el orden en que se deben ejecutar los cambios en la base de datos y de hacerlo de forma automática si hace falta. Adicionalmente, conocer este orden de ejecución facilita regresar a estados anteriores de la base de datos usando herramientas que ofrece el *framework*. La figura 2 ilustra algunas de las migraciones realizadas durante el proyecto. Durante todo el proceso se siguió la convención de nombramiento que recomienda Ruby on Rails (cada archivo comienza con la fecha en que fue creado). Adicionalmente, los nombres son significativos respecto al tipo de cambio que se realizó en la base de datos.

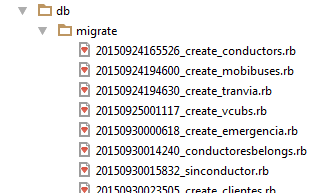


Fig 2. Migraciones de TBC

.

La figura 3 muestra cómo se realiza una migración como tal. En este caso corresponde a una eliminación de una columna a saber la de "id\_conductor" de las tablas "mobibuses" y "tranvías".

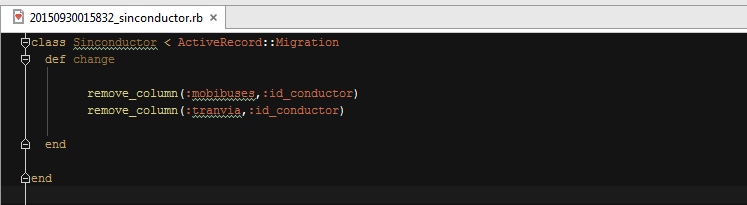


Fig 3. Migración sinConductor.

* 1. No repetir código

Las convenciones de Rails fomentan la no repetición de código. Un ejemplo de este concepto que se aplicó en el proyecto se puede observar en la implementación de las vistas. En vez de escribir páginas HTML para cada una de las vistas se utiliza un archivo llamado “application.html.erb”. El código escrito en este archivo corresponde a aquel que se comparte en todas las vistas de la aplicación, por lo que se utiliza para guardar las partes estáticas de la página web (cómo el *header*, el *footer* y el *layout*). De esta forma, no es necesario reescribir código y el tamaño de cada vista se reduce considerablemente. Asimismo esto permite que en caso de tener que realizar un cambio este se realice únicamente en este archivo lo que contribuye de manera importante a la modificabilidad que se busca. La figura 4 exhibe dónde se encuentra el archivo mencionado.

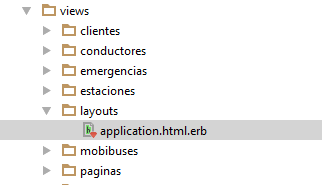


Fig 4. Ubicación del "application layout" del proyecto.

La navegación es otro elemento que se comparte en todas las vistas, pero tiene una connotación especial debido a que no es estática (la navegación cambia según el rol del usuario conectado) y se debe modificar con frecuencia. En lugar de colocarla dentro del “application.html.erb”, se creó un nuevo archivo con el propósito de manejar el código de la navegación. Este tipo de archivos se conoce como *partials* y pueden ser llamados en cualquiera de las vistas. En la figura 5 se puede observar cómo se realiza el llamado a un *partial* dentro de una vista.

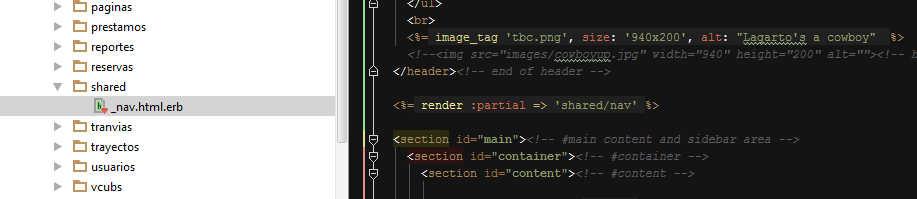


Fig 5. Llamado a un partial.

* 1. No sobrecargar archivos

Una buena práctica para usar *Ruby on Rails* es mantener los métodos de los controladores y las vistas lo más cortos posibles. Para lograrlo anterior, se utilizan los *helpers*, que son simplemente archivos de *Ruby*. Cada controlador tiene un *helper*, y es ahí donde se escriben los métodos personalizados que este o sus vistas van a utilizar. De esta forma, se reduce la cantidad de código que se escribe en los controladores y en las vistas y se favorece la reutilización a través de los métodos. En la figura 6 se puede observar la ubicación de los *helpers*, así como parte del código del *helper* que corresponde al controlador de Trayectos.

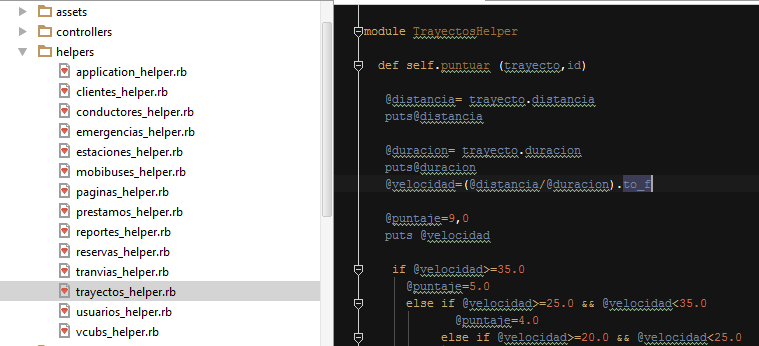


Fig 6. Helpers de TBC.

# Inspección de código

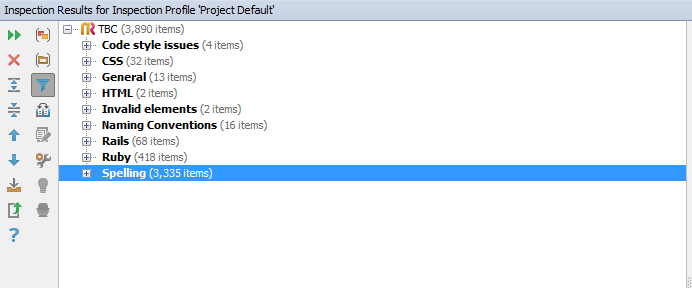
El IDE RubyMine con el que se está desarrollando el proyecto ofrece una buena herramienta de inspección de código. Se ejecutó dicha herramienta en el proyecto. El resultado de cada inspección es una lista de advertencias que indican elementos sospechosos de salirse de las convenciones del patrón MVC, de Ruby, de Ruby on Rails, o de buenas prácticas de programación en general. Aun cuando existen otras alternativas para realizar se consideró que la inspección que provee RubyMine es mucho más completa y permite realizar las correcciones pertinentes. La figura 7 muestra los resultados de la primera inspección realizada.

Fig 7. Primera Inspección

Como se puede observar en la figura anterior el asistente de inspección ofrece sugerencias respecto al estilo del código el *CSS, HTML* y convenciones de nombramiento. Asimismo, señala problemas de gramática y ortografía y específicos de *Ruby* de y de *Rails*. Además señala unos aspectos generales que deben corregirse. En esta primera inspección se cuenta con 3,890 ítems: la mayoría se relacionan con ortografía y gramática. Esto, se debe a que el programa así como los comentarios se realizaron en español y *RubyMine* está en inglés. A continuación se muestran ejemplos puntuales de lo que se sugiere en esta primera inspección.

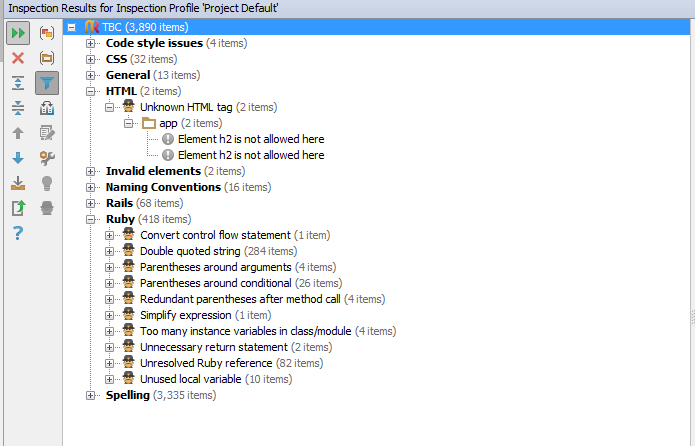


Fig. 8 Sugerencias de la primera inspección.

En la figura 8 se despliegan las sugerencias en cuanto a *HTML* y Ruby. Se observa que en el primer caso corresponde a un *tag* no identificado en HTML que se ubica en la carpeta de APP. En el caso de *Ruby* se señalan cuestiones relacionadas con un exceso de paréntesis, variables no utilizadas y *returns* innecesarios. Así mismo, se sugiere simplificar algunas partes del código lo que contribuye a una mayor modificabilidad.

Teniendo en cuenta la inspección realizada se hicieron las modificaciones que se consideraron pertinentes. Las figuras 9 y 10 muestran un ejemplo de un cambio realizado siguiendo lo indicado en la inspección. En este caso se cambia un condicional con negación por un *unless, pues* resulta más conveniente evaluarlo de esa manera.

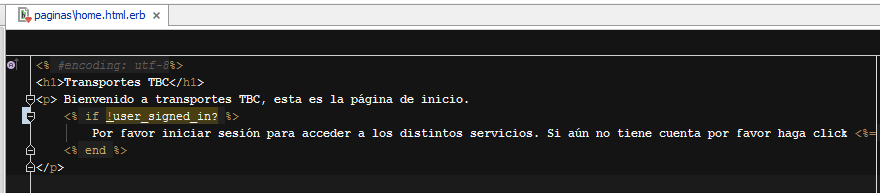


Fig 9. Código antes del cambio

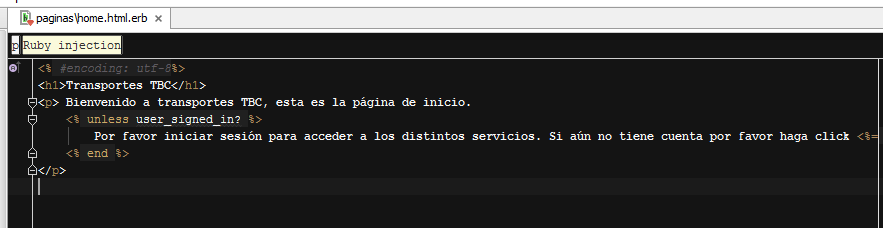


Fig 10. Código modificado

Con mayor detalle, los cambios realizados fueron de los siguientes tipos:

* **Correcciones en código HTML:** cerrar adecuadamente cada elemento y eliminar expresiones redundantes.
* **Optimización de CSS:** se reescribieron varias propiedades que podían expresarse de una forma más corta. También se hicieron cambios para que las propiedades no se sobrepusieran entre sí.
* **Eliminación de variables que nunca eran utilizadas.**
* **Corrección de nombres:** Varios métodos y variables fueron renombradas debido a que no tenían en cuenta las convenciones de ruby. Por ejemplo una variable llamada puntajeConductor fue renombrada a puntaje\_conductor.
* **Aplicación de convenciones de *Rails*:** 
  + Algunas vistas llamaban directamente métodos de los modelos violando convenciones del *framework*. Se movió el código correspondiente a los controladores para respetar la convención.

* + Se redujo el número de métodos de modelos que cada acción de los controladores utiliza.
  + Se redujo el número de variables de instancia que cada controlador comparte con sus vistas.
* **Aplicación de convenciones del lenguaje Ruby:** 
  + Remplazar dobles comillas por comillas simples: En Ruby las dobles comillas solo se utilizan cuando se declara un *string* que contiene caracteres especiales.
  + Corrección en el uso paréntesis: Se quitaron los paréntesis que rodeaban las sentencias *if* debido a que no son necesarios en este lenguaje de programación. Los paréntesis tampoco se deben colocar alrededor de los parámetros cuando se llama a un método (al menos que haya algún tipo de ambigüedad).
  + Eliminación de comandos *return* innecesarios. Los métodos en Ruby retornan automáticamente la última variable modificada, por lo que muchos comandos *return* eran innecesarios y fueron eliminados.
  + Simplificación de expresiones: Algunas expresiones podían escribirse de otra manera para ser más cortas y claras, cómo el ejemplo de las figuras 9 y 10.

Tras ese proceso se volvió a correr la inspección y se obtuvieron los resultados que se ilustran en la figura 11. Aún hay *items* pero estos no se corrigieron en la medida en la que no se consideró adecuado hacerlo considerando los propósitos del proyecto. Pero la reducción es considerable: de 3,890 a 409.

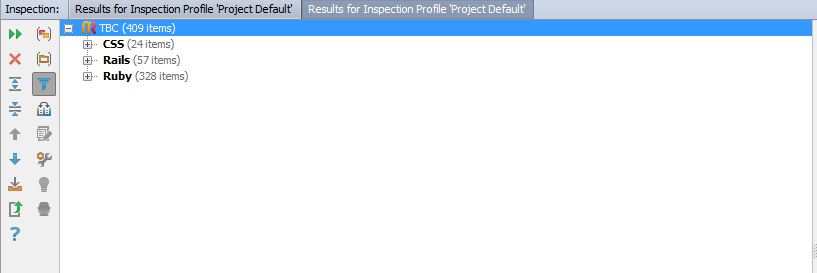


Fig 11. Inspección final

# Conclusiones

* El cambio es inevitable en el desarrollo de software en la medida en la que puede resultar necesario adicionar, modificar o eliminar características. Asimismo, se requiere continuamente solucionar defectos, mejorar la seguridad o adaptarse a nuevas tecnologías. De allí que es imprescindible referirse y garantizar modificabilidad en cualquier software que se realice.
* En el negocio de TBC el cambio también es inevitable por lo que la modificabilidad debe ser un atributo de calidad a tener en cuenta. En este contexto los cambios pueden darse respecto a la necesidad de adaptación a nuevas tecnologías así como en el caso de querer incluir nuevos servicios por nombrar algunos.
* En el caso de este proyecto se tuvieron en cuenta tanto el acoplamiento como la cohesión. El bajo acoplamiento y la alta cohesión que se buscan para hacer el software más modificable se lograron a través de la características de la arquitectura MVC y la plataforma *Ruby on Rails*.
* En cuanto a las tácticas de modificabilidad se aplicaron para incrementar cohesión y reducir acoplamiento. En el primer caso se mantiene la coherencia semántica y en el segundo se utiliza encapsulamiento y se hace uso de intermediarios. Estas tácticas se aplican al utilizar el patrón MVC.
* Aunque el patrón MVC y la arquitectura ya ofrecen las bases para garantizar modificabilidad es necesario utilizarlos de manera adecuada siguiendo los estándares para poder aprovechar las ventajas que ofrecen en este aspecto. Así el proyecto TBC sigue los lineamientos establecidos y garantiza modificabilidad de esta forma.
* Adicionalmente se hizo uso de inspecciones para corregir aspectos específicos del código.