Facultad de Ingeniería de Sistemas y Electrónica
Carrera profesional de Ingeniería de Sistemas

Informe del proyecto final:

Detección de Bad Smells y Refactoring Aplicación Web de Matrículas

Autor:
Canaza Tito, Eddy Wilmer

Asesor(a):
Ing. Paola Ana Zevallos Oporto

Arequipa, abril de 2017

Índice general

1. Introducción

Refactoring es realizar una transformación al software preservando su comportamiento, modificando sólo su estructura interna para mejorarlo. El término es de Opdyke, quien lo introdujo por primera vez en 1992, en su tesis doctoral. Fowler menciona que eran cambios realizados en el software para hacerlo más fácil de modificar y comprender, por lo que no son una optimización del código, ya que esto en ocasiones lo hace menos comprensible, ni solucionar errores o mejorar algoritmos. Las refactorizaciones pueden verse como un tipo de mantenimiento preventivo, cuyo objetivo es disminuir la complejidad del software en anticipación a los incrementos de complejidad que los cambios pudieran traer.

Este proyecto de investigación, refactorizará un proyecto desarrollado en CSharp que necesita optimizar su código fuente; además, se resolverá algunos problemas que presentó durante la puesta en producción.

La metodología utilizada en la proyecto, es sencilla de entender y puede servir como guía para aplicarse en proyectos que presenten los mismos bad smells.

1.1. Planteamiento del Problema

El software Matriculas, es un sistema que colabora con la gestión de matrículas para instituciones de educación básica. El sistema fue desarrollado en el año 2016 y cumple con los requisitos especificados por el cliente. sin embargo, presenta problemas de rendimiento y el nivel de mantenibilidad que presenta es inferior de lo esperado.

El cliente requiere que se optimice la arquitectura utilizada en el sistema, para escalarlo sin ningún problema y no se utilice esfuerzos de más en las tareas de implementación.

Los problemas que presenta el sistema, pueden provocar errores que el sistema no pueda controlar, y se requiera parar las operaciones de la institución para resolver los problemas.

1.2. Objetivos

1.2.1. General

Incrementar el nivel de mantenibilidad de la Aplicación Web de Matrículas a través de Refactoring

1.2.2. Específicos

- Investigar técnicas de Refactoring
- Estudiar la Aplicación Web de Matrículas
- Aplicar técnicas de Refactoring
- Validar la solución

1.3. Justificación

Es muy importante crear software que sea eficiente y que sea fácil de entender; por lo que es recomendable mantener el código con las mejores prácticas e desarrollo de software.

El refactoring es una técnica de ingeniería de software que permite restructurar el código fuente sin modificar el comportamiento. Esta técnica nos ayudará a incrementar le nivel de mantenibilidad de la aplicación y tener un código más limpio y óptimo.

2. Marco teórico

2.1. Refactoring

Desde su creación los sistemas de software han sido utilizados, probados e incluso modificados. A lo largo de este período el software y los seres humanos han interactuado entre sí en forma cotidiana, no pudiéndose concebir la vida actual del hombre sin la presencia de los sistemas informáticos (comunicaciones, medicina, transporte, etc.). A tal punto que paulatinamente los procesos de negocio consumen cada vez más información procesada por los sistemas de software, incluso hasta a ser controlados y guiados por software. Por ende, las especificaciones y el diseño de estos sistemas requieren de supuestos acerca de la aplicación dada, del dominio de la aplicación y de su ámbito operativo, hecho que a su vez se refleja en el software.

2.2. La decadencia del Código Fuente

Lehman [1] [2] propone una serie de leyes que guían la evolución de los sistemas de software:

1. **Cambio continuo:** Los sistemas deben adaptarse continuamente de lo contrario se hacen progresivamente menos satisfactorios. Estas adaptaciones son el resultado del cambio en la operación del entorno en el cual la aplicación cumple una función.

2. **Complejidad creciente:** A medida que evoluciona un programa, su complejidad se incrementa, a menos que se trabaje para mantenerla o reducirla. Esta ley implica un tipo de degradación o entropía en la estructura del programa. Esto a su vez conlleva a un aumento progresivo del esfuerzo de mantenimiento, a menos que se realice algún tipo de mantenimiento perfectivo a este respecto.

3. **Autoregulación:** El proceso de evolución el programa se autoregula con una distribución de medidas de atributos de producto y procesos cercana a la normal.

4. **Conservación de la Estabilidad Organizativa:** La velocidad de actividad global efectiva media en un sistema en evolución es invariante a lo largo del ciclo de vida del producto.

5. **Conservación de la Familiaridad**: Durante la vida activa de un programa en evolución, el contenido de las versiones sucesivas es estadísticamente invariante.

6. **Crecimiento Continuo:** El contenido funcional de un programa debe incrementarse continuamente para mantener la satisfacción del usuario durante su ciclo de vida.

7. **Calidad Decreciente:** Los sistemas serán percibidos como de calidad decreciente a menos que se mantengan de manera rigurosa y se adapten al entorno operativo cambiante.

2.3. Mal olor en el código fuente

¿Cuándo un sistema de software comienza a tomar mal olor? Un programador experimentado puede intuir que su programa va camino a oler mal cuando hay [3]:

1. **Código duplicado:** Se deben eliminar las líneas de código que son exactamente iguales en varios sitios, o bien eliminar líneas de código muy parecidas o con estructura similar en varios sitios.

2. **Métodos muy largos:** En este caso hay que particionar el código fuente en trozos y extraerlos para crear métodos más pequeños, que sean más fáciles de mantener y de reusar.

3. **Clases muy grandes:** En estos casos se debería tratar de identificar qué cosas hace esa clase, ver si realmente todas esas cosas tienen algo que ver la una con la otra y si no es así, hacer clases más pequeñas, de forma que cada una trate una de esas cosas. Por ejemplo, si una clase es una ventana, además realiza cálculos y escribe los resultados en una base de datos, ya está haciendo demasiadas cosas. Debería haber una clase que sea la ventana, otra que realice los cálculos y otra que sepa escribir en base de datos.

4. **Métodos que necesitan muchos parámetros:** Suele ser buena solución hacer una clase que contenga esos parámetros y pasar la clase en vez de todos los parámetros. Especialmente si esos parámetros suelen tener que ver unos con otros y suelen ir juntos siempre.

5. **Instrucciones Switch-Case:** Normalmente un switch-case se tiene que repetir en el código en varios sitios, aunque en cada sitio sea para hacer cosas distintas. Existen formas, usando el polimorfismo, que evitan

   tener que realizar esta repetición a lo largo del código fuente, o incluso evitan tener que ponerlo en algún lado.

Estas y otras pautas dan al programador la idea de que su código fuente puede ser susceptible a producir mal olor en un corto lapso de tiempo.

Dentro de este panorama casi desolador surge el concepto de refactorización de código fuente como una técnica que permite mejorar la comprensibilidad, la claridad, el diseño, la legibilidad y a su vez reducir la cantidad de errores. En definitiva una técnica para mejorar la calidad del software.

En un principio asociada a la programación orientada a objetos, el concepto de refactorización fue extendiéndose hacia otros paradigmas de programación, teniendo en cuenta la gran cantidad de código fuente escrito en los últimos 50 años de existencia del campo de la informática.

2.4. Refactoring

El primer uso conocido del término refactorización en la literatura se encuentra publicado en el artículo Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems por William F. Opdyke y Ralph E. Johnson [4]. La refactorización surge como un intento de mejorar la producción de software reusable. El desarrollo de software es un proceso complejo y continuo en el cual un producto transita, a lo largo de su construcción, por un proceso iterativo (desarrollo espiral, desarrollo por prototipos, desarrollo de tipo iterativo incremental, otros). El desarrollo de software re-usable es un proceso aun más complejo pues éste es el

resultado de varias iteraciones de diseño, incluso cuando éste ya ha sido reusado, por ende los cambios a los que está sujeto, no lo afectan únicamente a él sino también al software que lo utiliza [5]. En base a ello la refactorización surge como "el proceso en el cual se aplican cambios en un sistema de software de forma tal que no altere el comportamiento externo del código, mejorando su estructura interna" [3].

Esta definición, que no deja de ser muy abierta, plantea una serie de cuestiones: ¿En qué consiste el comportamiento externo del software? ¿El área de aplicabilidad de la refactorización está restringida únicamente a software reusable? ¿Puede extenderse este concepto a otros productos del proceso de

desarrollo, o es solamente aplicable a código fuente? ¿En qué consiste la estructura interna? ¿Cuándo, dónde, por qué refactorizar?

2.4.1. El comportamiento externo

Dentro de la definición de refactorización, existe un concepto en el cual hay que detenerse un segundo: la idea de la preservación del comportamiento externo del software. Por definición el proceso de refactorización de código fuente debe preservar el comportamiento del software. Pero ¿qué se entiende por comportamiento? En la bibliografáa no se encuentra una definición exacta sobre qué es el comportamiento externo del software. En su tesis Opdyke [5], sostiene que el conjunto de entrada y el conjunto de salida deben ser los mismos antes y después de aplicar el proceso de refactorización. Si bien en la definición de Opdyke los conjuntos de entrada y salida se conservan, Mens y Tourwé [6] agregan que ver la conservación del comportamiento sólo desde el punto de vista de las entradas/salidas es exiguo, pues existen otros aspectos dependiendo del dominio que hay que tener en cuenta. Este último enfoque permite ampliar la definición de comportamiento que debe ser o no preservado, dependiendo del dominio e incluso a veces dependiendo de las necesidades del usuario. Mens y Tourwé proponen tipos distintos de dominios:

- Software de tiempo real, en el cual los procesos de refactoring deben preservar las restricciones temporales. El tiempo es un factor que forma parte del comportamiento externo en este caso.

- Software embebido, el consumo de memoria en este caso forma parte del comportamiento a preservar en aplicaciones pertenecientes a este dominio.

Existen además otros dominios en los cuales se pueden definir otros aspectos que hacen al comportamiento externo. En las aplicaciones web, el contenido puede pasar a formar parte del comportamiento externo o como se especifica en el conjunto de nodos y los links de navegación entre los nodos y el conjunto de operaciones disponibles para el usuario y la semántica de cada operación, pueden ser considerados comportamiento externo, desde el punto de vista de la refactorización. A pesar de los intentos de formalización, no existe una definición formal de lo que es considerado comportamiento externo del software.

2.4.2. Características internas y externas del software

Otro punto destacable aportado por el trabajo de [6] es el efecto de los procesos de refactorización sobre la calidad del software. El software posee características que se manifiestan en forma externa y otras que son propias de la estructura interna del mismo, definidas como características internas. En las primeras encontramos conceptos como robustez, extensibilidad, performance, reusabilidad, etc. Entre las características internas nos encontramos con los conceptos de comprensibilidad, legibilidad, correctitud, redundancia, etc. Los procesos de refactorización pueden afectar a las características internas: al aplicar reducción de código redundante, al aplicar cambios de nombres de métodos o variables. Pero también pueden afectar a factores o características externas que hacen a la calidad del software, por ejemplo la performance. Si bien se cree que la refactorización de código fuente afecta negativamente en cuanto a la performance [3], existen estudios que demuestran lo contrario. Refactorizaciones tendientes a reemplazar instrucciones if con polimorfismo mejoran la performance de la aplicación gracias a las optimizaciones que hacen los compiladores actuales [6].

2.5. Beneficios

Al refactorizar un sistema no se pueden solucionar todos los problemas del mismo. Sin embargo, es una herramienta valiosa, si se aplica regularmente. El refactoring es una herramienta que puede, y debe, ser utilizada para diversos fines. Las motivaciones para realizar un refactoring pueden ser varias. A continuación, [7], discute algunas de ellas.

1. **Mejora el diseño de software:**

   Sin refactorización, el diseño del programa decaerá. Los programadores desarrollan nuevos requerimientos con la finalidad de alcanzar objetivos a corto plazo o cambios en el sistema realizados sin una comprensión completa del diseño del sistema. Este pierde su estructura con el transcurso del tiempo y se hace más difícil ver el diseño mediante la lectura del código.

   El refactoring es en parte poner orden en el código. La pérdida de la estructura de un sistema tiene un efecto acumulativo. Cuanto más difícil es ver el diseño en el código del sistema, más difícil es mantenerlo. Un código mal diseñado generalmente toma más líneas para que funcione. Reducir la cantidad de código no hará que el sistema funcione más rápido, sin embargo, puede hacer una gran diferencia en la modificabilidad del código. Refactorizar regularmente ayuda a conservar la estructura del sistema.

2. **Hace al software fácil de entender**

   El programador escribe código que indica a la computadora qué hacer, y esta, responde haciendo exactamente lo que le dice. Hay una brecha entre lo que el programador quiere que haga y lo que el código realmente hace [3].

   A la hora de programar, es una mala práctica que no se piense en que otro programador o el mismo en un futuro tenga que modificar el código. Este es un problema que impacta cuando un programador necesita una semana para hacer un cambio que habría tomado sólo una hora si el código fuera más entendible.

   El refactoring ayuda a hacer el código más legible. Cuando existe código que funciona, pero no está muy bien estructurado, un poco de tiempo dedicado a

refactorizar puede hacer que sea más entendible y rápido de modificar.

Se pueden empezar realizando pequeños refactorings corrigiendo detalles. Como el código va quedando más claro, se pueden ver aspectos del diseño que no se podían ver antes.

3. **Ayuda a encontrar errores**

   Al ayudar a comprender el código también ayuda a localizar errores puntuales. Al refactorizar código se trabaja profundamente en la comprensión de lo que hace el código, y se puede volcar ese nuevo entendimiento de nuevo en el código. Tomar como un hábito la refactorización permite escribir código robusto de forma más eficiente.

2.6. Momentos para refactorizar

Se recomienda refactorizar todo el tiempo en pequeñas ráfagas. El desarrollador no decide refactorizar, refactoriza porque quiere hacer algo más, y refactorizando ayuda a preparar el sistema para el cambio. A continuación,, se discuten algunas guías de cuándo es recomendable refactorizar.

1. **Al agregar una funcionalidad**

   El momento más común para refactorizar es cuando se quiere agregar una nueva funcionalidad a un sistema. El código puede haber sido escrito por el mismo programador al que le toca implementarla o por otro. Al momento de agregarla, el programador ve que si se hubiera diseñado el código de otra manera, la implementación sería más fácil. Se debe refactorizar el diseño para que el sistema se más fácil de modificar en el futuro y por la tanto el proceso de agregar una nueva funcionalidad sea más rápido.

2. **Al corregir un error**

   Cuando se mira el código al tratar de entenderlo, se refactoriza para ayudar a mejorar la comprensión. Este proceso activo de trabajar con el código ayuda a encontrar errores. Si se reporta un error en el sistema, es una clara señal de

que se necesita una refactorización, ya que el código no era lo suficientemente claro para que el desarrollador vea que había un error en el área en la que trabajó.

3. **Al revisar el código**

   Las revisiones de código ayudan a difundir el conocimiento a través de un equipo de desarrollo. Los desarrolladores más experimentados pasan conocimiento a otros con menos experiencia. También son muy importantes para la escritura de código claro. El código puede parecer claro para el desarrollador que lo escribió, pero no para el resto del equipo. Las revisiones también dan la oportunidad para que más personas sugieren ideas útiles.

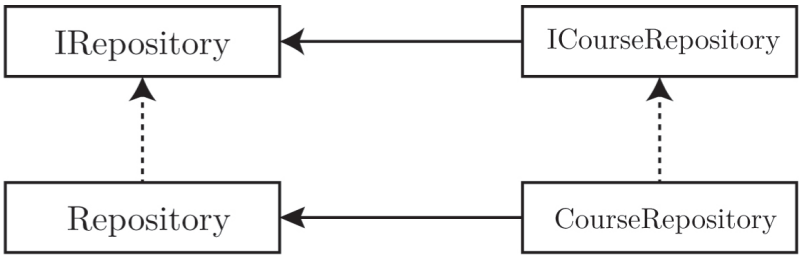
3. Metodología

Para esta etapa del proyecto se detectarán los bad smells del código existente del proyecto y se aplicarán técnicas de refactoring e ingeniería para resolverlos.

3.1. Arquitectura

3.1.1. Patrón repository mal implementado

- **Síntoma:** La arquitectura de la aplicación no cumple con los fundamentos el patrón repository.
- **Solución:** Restructurar el código fuente para que se adapte al patrón mencionado.
- **Procedimiento:** Se investigó como trabaja el patrón repository y se traslado cada método a la clase que corresponde.



3.2. Vistas

3.2.1. Librerías de terceros innecesarias

- **Síntoma:** Existen librerías de terceros procedentes de la plantilla que no se utilizan en ningún momento.
- **Solución:** Revisar que librerías se están usando dentro del proyecto.

- **Procedimiento:** Se eliminó la carpeta que contenía los elementos de la plantilla, se migraron los paquetes a bower y se reciclo las hojas de estilos. Además, se escribió comentarios indicando la función que cumple cada uno.

```
@* CSS *@
<!-- Fuentes -->
<!-- Plugins -->
<!-- Bootstrap -->
<link href="~/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
 <!-- Mensajes emergentes -->
<link href="~/lib/toastr/toastr.css" rel="stylesheet" />
<!-- Iconos -->
<!-- Iconos principales -->
<link href="~/lib/fontawesome/css/font-awesome.css" rel="stylesheet" />
<!-- Estilos de la aplicacion -->
<!-- Normalizacion de elementos -->
<link href="~/css/components.css" rel="stylesheet" />

@* JavaScript *@
<!-- jQuery -->
<script src="~/lib/jquery/dist/jquery.min.js"></script>
<!-- Bootstrap -->
<script src="~/lib/bootstrap/dist/js/bootstrap.min.js"></script>
<!-- Waypoints: Trigger para iniciar funcion -->
<script src="~/lib/waypoints/lib/jquery.waypoints.min.js"></script>
```

3.2.2. Error en la carga de las vistas

- **Síntoma:** Las vistas de la aplicación presentan un ligero retardo al cargar la data.
- **Solución:** Revisar las directivas de Angular.
- **Procedimiento:** Se agregó la directiva ng-cloak

```
<div class="row" ng-app="app" ng-controller="alumnosController as vm">
  <wait-cursor display-when="vm.isBusy"></wait-cursor>
  <!-- Tabla de visualizacion -->
```

```
  <div ... ng-cloak>
    ...
  </div>
</div>
```

3.2.3. Mensajes de validación permanentes

- **Síntoma:** Los mensajes de validación no desaparecen cuando se cancela la operación o se cierra el formulario.
- **Solución:** Implementar un evento que reinicie los atributos de validación.
- **Procedimiento:** Se agregó un método en javascript para que realice la tarea.

```
$('.cancel').on('click', function(){
  var $form = $("form");
  var $validator = $form.validate();
  var $errors = $form.find(".field-validation-error span");
  $errors.each(function () { $validator.settings.success($(this)); })
  $validator.resetForm();
});
```

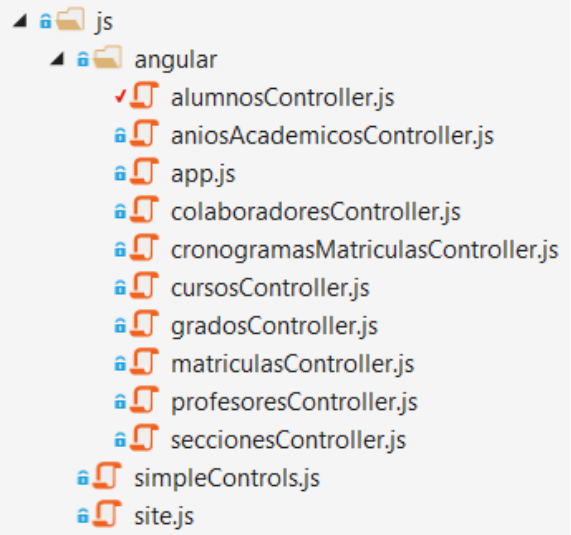
3.2.4. Agregar eventos tecla escape

- **Síntoma:** El usuario no puede utilizar la tecla escape para cerrar el formulario emergente.
- **Solución:** Crear un método que cierre el formulario y reinicie los atributos de validación.
- **Procedimiento:** Se agregó un método en javascript para que realice la tarea.

```
$(document).keyup(function (e) {
  if (e.keyCode === 27)
  $('.cancel').click();
});
```

3.2.5. Mala distribución de los archivos Javascript

- **Síntoma:** Los archivos javascript de la aplicación están mezclados con los controladores angular.
- **Solución:** Colocar todos los controladores dentro de una carpeta.
- **Procedimiento:** Se creó la carpeta "angular"para trasladar los controladores y se cambió la ruta de los paquetes en las vistas.



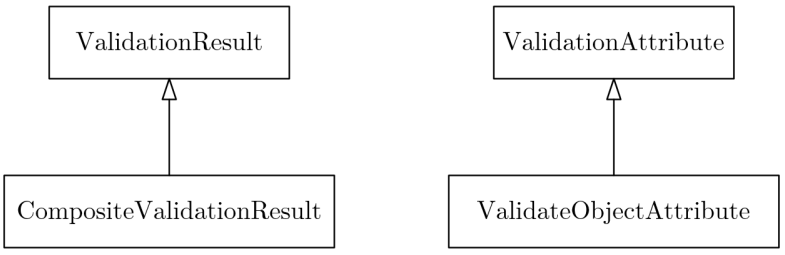

```
@section scripts {
  ...
  <script src="~/js/angular/app.js"></script>
  <script src="~/js/angular/alumnosController.js"></script>
  ...
}
```

3.3. Consistencia de información

3.3.1. Validación de los nested objects

- **Síntoma:** El controlador no valida los nested objects y se valida los objetos manualmente.
- **Bad Smell:** Duplicated code.

- **Solución:** Crear una clase que realice este proceso iterativamente.
- **Procedimiento:** Se creó dos clases, una llamada "CompositeValidationResultz la otra "ValidateObjectAttribute"que agregan un atributo a los decoradores de validación del modelo para que puedan usarse en todos los DTO.




```
public class AlumnoViewModel
{
  ...
  [Required, ValidateObject]
  public virtual ApoderadoViewModel Apoderado { get; set; }
  ...
}
```

3.3.2. Validaciones asíncronas

- **Síntoma:** Las validaciones que se hacen el la vista modelo no consultan con el servidor para revisar información duplicada y otros.
- **Solución:** Crear un controlador que realice la validación a través de los repositorios.
- **Procedimiento:** Se creó un controlador para validar todas las transacciones y se agregó el decorador remote en la vista-modelo.

```
public class ValidatorController : Controller
{
  private IAppRepository _repository;

  public ValidatorController(IAppRepository repository)
  {
```

```
    _repository = repository;
  }

  [AcceptVerbs("Get", "Post")]
  public IActionResult IsDniColaboradorUnique([Bind(Prefix = "Id")] int
     id, string dni)
  {
    if (_repository.Colaboradores.HasDniUnique(new Colaborador { Id =
    id, Dni = dni}))
      return Json(data: " ");
    return Json(data: "Dni no válido");
  }
}
```

```
[Remote("IsDniColaboradorUnique","Validator",AdditionalFields="Id")]
[Required(ErrorMessage = "Este campo es obligatorio.")]
[RegularExpression("([0-9]{8})", ErrorMessage = "Este campo ...")]
public string Dni { get; set; }
```

3.4. Mejoras de rendimiento

3.4.1. Duplicidad en el mapeo

- **Síntoma:** Se está mapeando el objeto dos veces, una para validar algún atributo y otro para transaccionar.
- **Bad Smell:** Duplicated code.

```
public class AlumnoViewModel
[HttpPost()]
public async Task<IActionResult> PostAlumno([FromBody] AlumnoViewModel
    alumnoDetails)
{
  ...
  if (!_repository.IsDniValido(Mapper.Map<Alumno>(alumnoDetails)))
    ModelState.AddModelError("otros", "Este DNI ya fue registrado.");

```

```
  if (ModelState.IsValid)
  {
    var _newAlumno = Mapper.Map<Alumno>(alumnoDetails);
    ...
  }
  ...
}
```

- **Solución:** Extraer las sentencias duplicadas y colocarlo en una variable.
- **Técnica:** Extract variable.

```
[HttpPost()]
public async Task<IActionResult> PostAlumno([FromBody] AlumnoViewModel
    alumnoDetails)
{
  ...
  var alumno = Mapper.Map<Alumno>(alumnoDetails);

  if (!_repository.IsDniValido(alumno))
    ModelState.AddModelError("otros", "Este DNI ya fue registrado.");

  if (ModelState.IsValid)
  {
    _repository.AddAlumno(alumno);
    ...
  }
  ...
}
```

3.4.2. Sobrecarga de Entity Framework

- **Síntoma:** Se está accediendo muchas veces a un atributo del objeto

```
public void ToggleEstado(int id)
{
  var colaborador = Get(id);
```

```
4   _context.Colaboradores.Attach(colaborador).State = EntityState.
      Modified;
5
6   if (colaborador.Estado == "1")
7   colaborador.Estado = "3";
8   else if (colaborador.Estado == "3")
9   colaborador.Estado = "1";
10
11  _context.Update(colaborador);
12 }
```

- **Técnica:** Extract variable.
- **Solución:** Extraer la operación en una variable temporal.

```
1 public void ToggleEstado(int id)
2 {
3   var colaborador = Get(id);
4   _context.Colaboradores.Attach(colaborador).State = EntityState.
      Modified;
5
6   var estado = colaborador.Estado;
7
8   colaborador.Estado = estado == "1" ?  "3" : estado == "3" ? "1" :
      estado;
9 }
```

3.4.3. Actualización de campos

- **Síntoma:** Se está pasando cada uno de los atributos para la actualización del objeto.

```
1 public void UpdateAlumno(Alumno entity)
2 {
3   var thisAlumno = GetAlumnoById(entity.Id);
4   thisAlumno.ApellidoPaterno = alumnoToUpdate.ApellidoPaterno;
5   thisAlumno.ApellidoMaterno = alumnoToUpdate.ApellidoMaterno;
```

```
  thisAlumno.Nombres = alumnoToUpdate.Nombres;
  thisAlumno.Dni = alumnoToUpdate.Dni;
  thisAlumno.Sexo = alumnoToUpdate.Sexo;
  thisAlumno.Direccion = alumnoToUpdate.Direccion;
  thisAlumno.FechaNacimiento = alumnoToUpdate.FechaNacimiento;

  _context.Update(thisAlumno);
}
```

- **Solución:** Utilizar Entries de Entity Framework para determinar los comportamientos de las entidades y no alterar los Nested Objects.

```
public Alumno UpdateAlumno(Alumno entity)
{
  _context.Entry(entity).State = EntityState.Modified;
}
```

3.4.4. Uso inadecuado de procedimientos almacenados

- **Síntoma:** El método llama a un procedimiento en la base de datos.

```
public void Update(Profesor entity)
{
  ...
  _context.Database.ExecuteSqlCommand(String.Format("EXEC
    SP_DeleteCursos @idProfesor='{0}'", profesor.Id));
  foreach (Curso curso in cursos)
  {
    _context.Database.ExecuteSqlCommand(
    String.Format("EXEC SP_AddProfesorCurso @idProfesor='{0}', @idCurso
    ='{1}'",
    profesor.Id, curso.Id));
  }
  ...
}
```

- **Técnica:** Extract Method.

- **Solución:** Remplazar el procedimiento almacenado con sentencias LINQ, y extraer el segmento de código que elimina los cursos actuales del profesor en un nuevo método.

```
public void EliminarCursos(int id)
{
  var cursos = _context.ProfesorCursos
    .Where(t => t.ProfesorId == id)
    .ToList();

  _context.ProfesorCursos.RemoveRange(cursos);
  _context.SaveChanges();
}
```

3.5. Mejoras en los servicios

3.5.1. Ruta de los controlador API

- **Síntoma:** Se está repitiendo en cada acción la ruta del controlador.
- **Bad Smell:** Duplicated code.

```
public class AlumnosController : Controller
{
  ...

  [HttpGet("api/alumnos"))]
  public IActionResult GetAllAlumnos()
  { ... }

  [HttpGet("api/alumnos/{id}")]
  public IActionResult GetAlumno(int id)
  { ... }

  ...
}
```

- **Solución:** Utilizar el decorador Route para indicar la ruta del controlador. También indicar la versión del API y cambiar el nombre de las acciones según convenciones.

```
[Route("api/v2/[controller]")]
public class AlumnosController : Controller
{
  ...

  [HttpGet()]
  public IActionResult GetAllAlumnos()
  { ... }

  [HttpGet("{id}")]
  public IActionResult GetAlumno(int id)
  { ... }

  ...
}
```

3.6. Mejoras en la lógica de negocio

3.6.1. Lógica de negocio en el controlador

- **Síntoma:** El controlador tiene código de la lógica del negocio; las tareas que deberían ejecutarse en el repositorio.
- **Bad Smell:** Intimidad inadecuada.

```
public async Task<IActionResult> PostUpdateProfesorCurso([FromBody]
    CursoProfesorViewModel profesorDetails)
{
  ...
  if (ModelState.IsValid)
  {
    var profesor = Mapper.Map<Profesor>(profesorDetails.Profesor);
    var curso = Mapper.Map<Curso>(profesorDetails.Curso);
```

```
    ...
  }
  ...
}
```

- **Técnica:** Move Method y Extract Method.
- **Solución:** Trasladar el código al repositorio creando un método que realice las tareas de mapeo.

```
public void AssignProfesor(int id, int idProfesor)
{
  var anioAcademico = new AniosAcademicosRepository(_context).
   GetAnioAcademico(DateTime.Now.Year);

  if (anioAcademico != null)
  {
    var cursoAnioAcademico = _context.CursosAnioAcademico
      .Where(t => t.AnioAcademicoId == anioAcademico.Id)
      .Where(t => t.CursoId == id)
      .FirstOrDefault();
    ...
  }
}
```

3.6.2. Método SearchCursos en controlador de cursos

- **Síntoma:** El método SearchCursos se encarga de listar los cursos que dicta un profesor está en un controlador que no le corresponde.
- **Bad Smell:** Intimidad inadecuada.

```
public class CursosController : Controller
{
  ...
  [HttpGet("{id}")]
  public IActionResult GetCursosDisponibles(int id)
  {
```

```
    try
    {
      _logger.LogInformation("Recuperando la lista de cursos que dicta
    el profesor.");
      var results = _repository.GetCursosByIdProfesor(id);
      return Ok(Mapper.Map<IEnumerable<CursoViewModel>>(results));
    }
    catch (Exception ex)
    {
      _logger.LogError($"No se pudo recuperar los cursos: {ex}");
      return BadRequest("No se pudo recuperar la información.");
    }
  }
  ...
}
```

- **Técnica:** Move Method.
- **Solución:** Trasladar este método al controlador de profesores para implementar el API Rest correctamente.

```
public class ProfesoresController : Controller
{
  ...
  [HttpGet("{id}")]
  public IActionResult GetCursosDisponibles(int id)
  {
    try
    {
      _logger.LogInformation("Recuperando la lista de cursos que dicta
    el profesor.");
      var results = _repository.GetCursosByIdProfesor(id);
      return Ok(Mapper.Map<IEnumerable<CursoViewModel>>(results));
    }
    catch (Exception ex)
    {
      _logger.LogError($"No se pudo recuperar los cursos: {ex}");
      return BadRequest("No se pudo recuperar la información.");
    }
```

```
18    }
19    ...
20 }
```

3.6.3. Atributos que no se usan

- **Síntoma:** La clase Cursos tiene atributos que no usa.

```
1  public class Curso
2  {
3    ...
4    public virtual IEnumerable<Profesor> Profesores { get; set; }
5    public virtual ICollection<ProfesorCurso> ProfesorCurso { get; set; }
6    ...
7  }
```

- **Solución:** Eliminar los atributos que no usa en el modelo.

3.6.4. Cambiar la definición de CronogramaMatricula

- **Síntoma:** La clase CronogramaMatricula que era el modelo de los cronogramas de matrículas; ahora, se utilizará para cualquier tipo de cronograma dentro de un año académico. Además se usando un atributo "Nombreçomo Primary Key.

```
1  public class CronogramaMatricula
2  {
3    [Key]
4    [ForeignKey("AnioAcademico")]
5    public int AnioAcademicoId { get; set; }
6
7    [Key]
8    [StringLength(30)]
9    public string Nombre { get; set; }
10
11   ...
12 }
```

- **Solución:** Renombrar la clase CronogramaMatricula a Cronograma. Agregar un Atributo Id para establecerlo como Primary Key.

```
public class Cronograma
{
  [Key]
  public int Id { get; set; }

  [ForeignKey("AnioAcademico")]
  public int AnioAcademicoId { get; set; }

  [StringLength(30)]
  public string Nombre { get; set; }

  ...
}
```

Referencias bibliográficas

[1] M. Lehman *et al.*, "Metrics and laws of software evolution-the nineties view," in *4th International Software Metrics Symposium*, 1997.

[2] ——, "Programs, life cycles, and laws od software evolution," in *Proccedings of the IEEE*, 1980.

[3] M. Fowler and K. Beck, *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.

[4] W. Opdyke and R. Johnson, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems," in *Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 1990.

[5] W. Opdyke, "Refactoring object-oriented framework," PhD thesis, Cisteseer, 1992.

[6] T. Mens and T. Tourwé, "A survey of software refactoring," in *IEEE Transactions on software enginnering*, 2004, pp. 126–139.

[7] M. A. ana Facundo Klaver, "Aplicación de un proceso de refactoring guiado por escenarios de modificabilidad y Code Smells," Tesis de grado, Buenos Aires, 2015.