2. Marco teórico y trabajo relacionado.

Este capítulo provee el marco teórico en distintas áreas relacionadas directamente con el proyecto como el .NET Framework, inyección de código, generación de código, y otras áreas de interés como el patrón inyección de dependencias. Una breve descripción es dada acerca de cada uno de estos temas, al igual de una pequeña descripción de su uso en el proyecto, sin embargo, más detalles sobre estos son presentados en el siguiente capítulo.

2.1 .NET Framework.

El .NET Framework es el modelo de programación propuesto por Microsoft para la construcción de aplicaciones con experiencias visuales impresionantes, comunicaciones seguras, y principalmente tiene la habilidad de poder modelar un sinfín de procesos de negocios [9]. Para propósitos de nuestra investigación, el .NET Framework fue la tecnología escogida para realizar la experimentación debido a que tiene un lenguaje intermedio que permite que distintos lenguajes de programación puedan usarse con él,

maximizando las posibilidades de ser utilizado, incluso con distintos paradigmas de programación. El .NET Framework incluye una enorme librería de soluciones pre-programadas que resuelven problemas comunes de programación, así como también de una máquina virtual que se encarga de la ejecución de los programas escritos usando el framework. La librería pre-programada es conocida como la librería de clases base (BCL) y cubre un número grande de necesidades comunes de programación en distintas áreas que incluyen: interfaces, acceso a datos, conexión a bases de datos, conectividad, criptografía, desarrollo de aplicaciones web, algoritmos numéricos y comunicaciones de red [26]. La BCL es usada por los desarrolladores que combinadas con su propio código pueden producir aplicaciones funcionales.

Los programas escritos usando el .NET Framework son ejecutados en un ambiente de software especial que se encarga de administrar los requerimientos de cada programa en tiempo de ejecución. Este ambiente de tiempo de ejecución, que también es parte del .NET Framework, es conocido como el Common Language Runtime (CLR) [28]. El CLR es como una máquina virtual similar a la de Java, lo que permite que los programadores no se tengan que preocupar por las capacidades específicas del CPU o arquitectura en el que va a correr el programa.

El .NET Framework en resumen busca proveer las siguientes características:

- **Independencia del Lenguaje.** El .NET Framework introduce la noción de Common Type System (CTS). El CTS es básicamente una especificación que define los posibles tipos de datos así como los tipos que se pueden construir en un lenguaje de programación soportado por el CLR. También

específica como estos datos pueden o no pueden interactuar entre cada uno. Gracias al CTS es que el .NET Framework soporta más de 25 lenguajes de programación distintos, incluyendo: C#, Visual Basic .NET, J#, Delphi, COBOL, etc.

- **Interoperabilidad.** Debido a que a lo largo del tiempo se han creado distintas librerías siguiendo el modelo de COM, el .NET Framework provee métodos para permitir la interoperabilidad entre el código administrado y el código no administrado.

- **Seguridad.** El .NET Framework provee varios mecanismos para proteger los recursos y códigos de personas no autorizadas.

Algo importante del .NET Framework es que sus características principales están dictadas por el Common Language Infrastructure (CLI). El propósito del CLI es de proveer una plataforma con un lenguaje-neutral para el desarrollo y ejecución de aplicaciones, incluyendo funciones para el manejo de excepciones, recolector de basura, seguridad e interoperabilidad [5].

La implementación del CLI en Microsoft es precisamente el CLR. Pero además de este, existen otras implementaciones, siendo una de las más importantes el proyecto Mono. El proyecto mono es un proyecto de código abierto que implementa estos estándares dictados por ECMA y permite poder ejecutar programas desarrollados en sus tecnologías en otros sistemas operativos como Unix, MacOS X, y Windows [30]. El proyecto incluye un compilador de C# y un CLR.

Microsoft comenzó el desarrollo del .NET Framework a finales de los 90's. A partir de la primera versión oficial, se han lanzado distintas versiones, siendo la más nueva el .NET Framework 3.5 En esta nueva versión la mayoría de los cambias estuvieron relacionados con LINQ, ASP .NET AJAX, y algunas mejorar en la BCL [9].

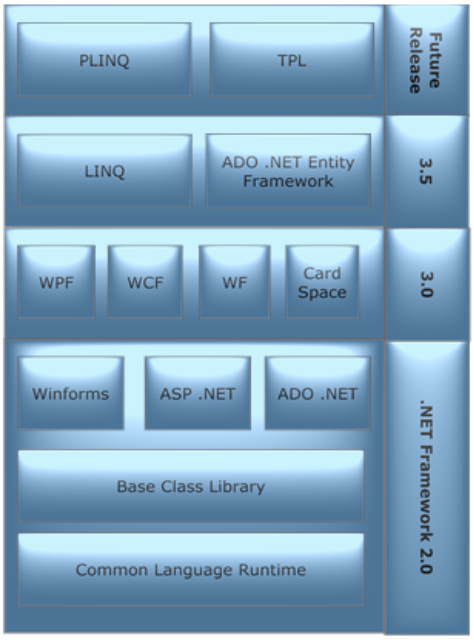

Figure 1. Descripción general del .NET Framework

La versión 3.5 del .NET Framework, también añadió características importantes que fueron usadas en el desarrollo del proyecto. Una de ellas son las extensiones de métodos que permiten crear métodos estáticos para ciertos tipos, y poder usarlos como si fueran originalmente parte del tipo, sin recompilar ó cambiar una clase manualmente. Otra característica usada fue LINQ para XML, el cual se ocupó para el análisis de documentos XML así como lo generación de los mismos. Finalmente se tomó ventaja de los delegados genéricos, para la implementación de las factories y los cuáles son detallados más adelante en el capítulo 4 que habla sobre la implementación. Una ventaja es que estas características aunque fueron introducidas con esta nueva versión, fueron construidas en base al mismo

CLR, por lo que versiones antiguas del .NET Framework pueden correr aplicaciones con código que use estas.

2.1.1 CLI

El Common Language Infrastructure (CLI) provee una especificación para el código ejecutable así como del ambiente de ejecución en el que el código correrá. La especificación describe estos 4 aspectos:

- El Common Type System (CTS).
- Los metadatos.
- El Common Language Specification (CLS).
- Y el sistema virtual de ejecución (VES).

Todos ellos son parte fundamental del .NET Framework y juntos forman una infraestructura unificada para el diseño, desarrollo, despliegue y ejecución de componentes distribuidos y aplicaciones. El CLI está completamente documentado por el ECMA [5].

El uso del CLI es muy importante, ya que usar lenguajes de programación que siguen el estándar, nos da la posibilidad de poder desarrollar en múltiples lenguajes, promoviendo la reusabilidad de código a través de distintos lenguajes. El CLI permite qué los módulos sean auto-registrables, lo que significa que puedan correr en procesos remotos, hacer manejo de sus versiones e incluso administrar errores a través de manejo de excepciones.

2.1.1.1 CTS

El CTS es un estándar que específica como las definiciones de tipos, así como sus valores son representados en la memoria de un computadora [39]. Al compartir un mismo modulo de objetos, es posible que componentes escritos en diferentes lenguajes de programación puedan interactuar uno con el otro y viceversa. Más que poder solo llamar a funciones de librerías escritas en otro lenguaje, con el .NET Framework es posible extender las capacidades de estos componentes.

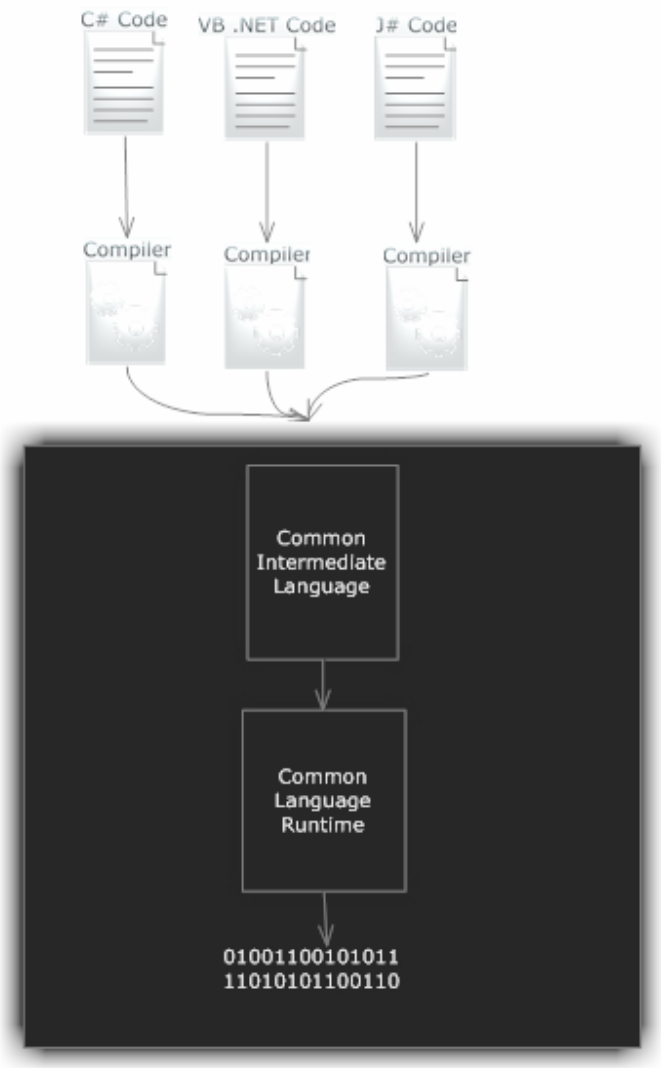

Figure 2. Descripción general del CLI.

El CTS soporta dos categorías de tipos, el cual a su vez es dividido en sub-categorías [29]:

- Los tipos por valor que contienen sus propios datos, y los cuales son guardados en la pila o asignados a estructura en línea. Existen tipos por valor ya construidos, pueden ser definidos por el usuario, o pueden ser enumeraciones.

- Los tipos por referencia, sus valores son guardados y alojados en el montículo. Los tipos por referencia pueden ser tipos auto descriptivos, apuntadores o interfaces.

Las variables que son tipos por valor tienen cada una su propia copia de los datos, por tanto las operaciones en una variable no afecta a las demás. Por otro lado, las variables por referencia pueden apuntar a un mismo objeto; por tanto, las operaciones en una variable pueden afectar a otra que apunta al mismo objeto.

Todos los tipos se derivan de un objeto base, System.Object. Los valores de estos tipos son representaciones binarias de datos, y los mismos tipos deben proveer mecanismos para interpretar estos datos. Un tipo por valor es guardado directamente en su representación binaria. El valor de un tipo por referencia es simplemente una locación de una secuencia de bits que representan el dato.

Esto significa que cada valor tiene un tipo que define la semántica de sus datos. A los valores que son auto-descriptivos se les llama objetos [32]. Cuando es normalmente imposible determinar el tipo exacto de un objeto examinando su valor, esto no se puede realizar con un tipo por valor o por referencia. Esto se debe a que un valor puede tener más de un tipo. Por ejemplo el valor de un tipo que implementa una interfaz, también es un valor de esa interfaz. Del mismo modo, el valor de un tipo que deriva de una clase base es también valor de esa clase.

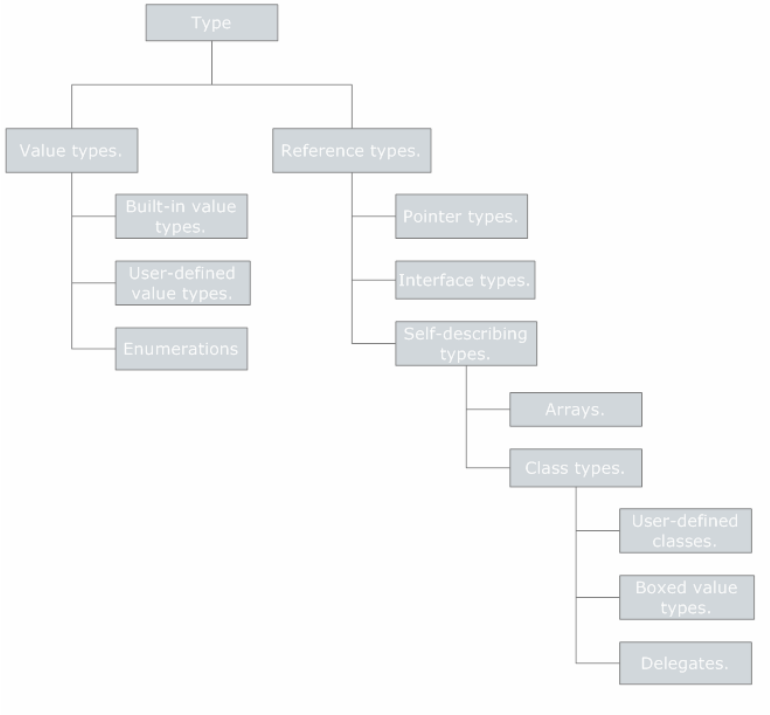

Figure 3. Jerarquía de tipos.

2.1.1.2 CLS

El CLS es un conjunto de reglas con el objetivo de promover la interoperabilidad de lenguajes [5]. Esto significa que el CLS provee las características necesarias para que el .NET Framework pueda interactuar con cualquier tipo de objeto no importando el lenguaje en el que fue implementado. Básicamente el CLS define un subconjunto de las reglas del CTS; esto significa que todas las reglas que aplican para el CTS también aplican par el CLS, excepto algunas reglas muy estrictas que solo están definidas para el CLS. El CLS también establece los requerimientos para que un programa este escrito conforme a sus reglas. Esto es muy importante ya que ayuda a determinar cuándo una pieza de código administrado fue escrita conforme al CLS.

Los componentes que se adhieren a las reglas del CLS y usan solo características incluidas en el CLS se dice son componentes compatibles con el CLS.

La mayoría de los miembros definidos en la BCL son compatibles. Sin embargo, algunos tipos en él, tienen uno o más miembros que no son compatibles con el CLS. Estos miembros permiten el soporte de ciertas características para el lenguaje que no están soportadas por el CLS.

El CLS fue diseñado lo suficientemente grande para incluir las necesidades más comunes para los desarrolladores en un lenguaje de programación, y al mismo lo suficientemente pequeño para que otros lenguajes de programación lo puedan soportar fácilmente. Además, cualquier característica de un lenguaje que no fuera posible verificar rápidamente su seguridad de tipos, fue excluida del CLS, de tal forma que todos los lenguajes de programación compatibles con el CLS pudieran producir código verificable si así lo desearan. Esta verificación de seguridad de tipos es hecha por la compilación justo a tiempo.

2.1.1.3 CLR

Como hemos mencionado anteriormente, el CLR es la implementación de Microsoft del estándar dictado por el CLI, el cuál define un ambiente de ejecución del código de los programas. El CLR corre un tipo de byte-code llamado Common Intermediate Language (CIL). En otras palabras el CLR es como una capa intermedia que trabaja entre el sistema operativo y las aplicaciones escritas en algún lenguaje de programación del .NET Framework y que es compatible con el CLS.

El CLR hace fácil diseñar componentes y aplicaciones en las que sus objetos interactúan entre distintos lenguajes [28]. Objetos escritos en distintos lenguajes de programación pueden comunicarse el uno al otro, y sus comportamientos pueden ser

fuertemente integrados. Por ejemplo, se puede definir una clase que usa un lenguaje diferente que el lenguaje de programación ocupado para la clase base, ó hacer llamadas a métodos en la clase original. Incluso se puede pasar una instancia de una clase a un método de otra clase escrita en un lenguaje de programación diferente. Está integración de lenguajes es posible debido a que los compiladores de los lenguajes y herramientas que fueron hechas para el CLR usan el CTS. Estos lenguajes siguen las reglas para definir nuevos tipos, así como para crearlos, usarlos, guardarnos y re-usarlos entre clases [29].

Los compiladores de los lenguajes de programación así como las herramientas exponen sus funcionalidades en formas intuitivas y útiles para el desarrollador. Esto significa, que algunas características del CLR son más notables en un ambiente que en otro. La experiencia que se tenga con el CLR depende del compilador y herramientas usadas.

Estos son algunos beneficios del CLR [28]:

- Mejoras de rendimiento.
- La capacidad para usar componentes desarrollados en otros lenguajes de forma fácil.
- Tipos extensibles proveídos por la librería de clases.
- Características nuevas de lenguajes como herencia, interfaces y sobrecarga de operadores; soporte explícito de hilos libres que permiten la creación de aplicaciones multi-hilos, aplicaciones escalables; y soporte para manejo de excepciones estructuradas así como atributos personalizados.

Específicamente hablando del compilador del lenguaje C#, podemos escribir código administrado con los siguientes beneficios [28]:

- Un diseño totalmente orientado a objetos.
- Una alta seguridad de tipos.
- Una buena mezcla de la simplicidad de Visual Basic y del poder de C++.
- Recolector de basura.
- Sintaxis y palabras clave parecidas a las de C y C++.
- Uso de delegados en lugar de apuntadores a funciones para incrementar seguridad de los tipos así como la seguridad de la aplicación.

El proceso de ejecución de aplicaciones administradas incluye los siguientes pasos [27]:

- Escoger un compilador.
- Compilar el código fuente para obtener el código intermedio.
- Compilar el código intermedio a código nativo.
- Correr el código.

Básicamente esto nos dice que para obtener los beneficios del CLR, es necesario escoger un compilador que sean compatibles con el CLR como C# o Visual Basic. Este compilador lo que hace es establecer la sintaxis del código fuente, se sugiere que los tipos usados sean tipos incluidos en el CLS [34]. El siguiente paso es compilar el código fuente

del lenguaje de programación seleccionado y convertirlo en código CIL, incluyendo los metadatos. El código CIL incluye instrucciones para cargar, guardar, inicializar y llamar los métodos de los objetos, así como también instrucciones para operaciones lógicas y aritméticas, control de flujo, acceso directo a memoria, manejo de excepciones y otras operaciones relevantes. El código CIL y los metadatos son guardados en archivo PE. Este formato de archivo permite al sistema operativo reconocer imágenes CLR. La presencia de los metadatos permite que el código sea auto-descriptivo, lo que significa que no hay necesidades de definir algún tipo de lenguaje para la definición de librerías. El CLR localiza y extrae los metadatos del archivo según sea necesario durante el tiempo de ejecución [35]. El siguiente paso es compilar el código CIL a código nativo. Esto se realiza en el momento de ejecución mediante el uso de un compilador justo a tiempo, el cual es encargado de traducir CIL en código nativo. Este proceso se hace en demanda, durante el tiempo de ejecución de la aplicación, ya que el JIT toma en cuenta que ciertas partes del código puede que no sean ejecutas durante el tiempo de ejecución. Es por eso que en lugar de convertir todo el código CIL en código nativo, lo va convirtiendo mientras sea necesario, y guarda el resultado en una especie de caché, que permite que este sea accesible las veces que sea necesario [36]. Finalmente el último paso es correr el código, para esto el CLR provee de una infraestructura que permite la ejecución de los programas así como otros servicios que son usados durante la ejecución. Algunos de estos servicios son el recolector de basura, seguridad, interoperabilidad con código no administrado, soporte para depuración de código en distintos lenguajes de programación, y un soporte mejorado de despliegue y versiones [37].

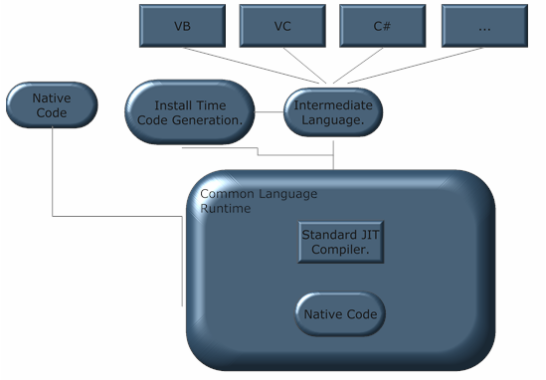

Figure 4. El proceso de ejecución de código administrado.

2.1.2 Common Intermediate Language (CIL).

CIL ó IL es un lenguaje de bajo nivel, específicamente diseñado para describir todas las características funcionales del CLR [6]. Cuando tú compilas un programa hecho con el .NET Framework, el compilador traduce el código fuente a CIL. Y este a su vez luego es traducido a código nativo por un compilador justo a tiempo.

A diferencia de la mayoría de los lenguajes de programación de alto nivel, así como otros lenguajes ensambladores, CIL está orientado a la plataforma en lugar de a los conceptos. Un lenguaje ensamblador normalmente es una lingüística que tiene un mapeo exacto con la plataforma escogida, el cual es el caso del CLR. De hecho, es un mapeo tan exacto, que este lenguaje fue usado para describir varios aspectos del ambiente en los documentos de la estandarización del ECMA/ISO acerca del CIL [5].

CIL es un lenguaje basado en pilas, a diferencia de otros lenguajes máquina que son basados en registros. Esto significa que cada vez que se quiere manipular un objeto este tiene que haberse guardado en la pila y después llamar un código op, que es una instrucción

del lenguaje para aplicar esta operación. Por supuesto, la pila debe estar vacía al final de la ejecución [6].

Un ejemplo simple de un Hola Mundo usando CIL es el siguiente:

```
.method static void main()
{
   .entrypoint
   .maxstack 1
   ldstr "Hello world!"
   call void [mscorlib]System.Console::WriteLine(string)
   ret
}
```

A diferencia de C#, el código CLI no tiene el requerimiento que los métodos deban pertenecer a una clase. Incluso el punto de entrada no tiene que llamarse main.

El método main anterior es conocido como la definición del método, ya que contiene tanto la firma como el cuerpo del método. En cambio, cuando solo la firma es provista, se dice que es una declaración del método. La declaración del método es normalmente usado como objetivos para ser llamados, mientras que la definición del método provee de la implementación del mismo.

La definición de un método comienza con la directiva *.method* la cual puede ser definida como global ó a nivel de clase. El punto de entrada de la aplicación debe ser estático, lo que significa que no se necesita instancia para poder llamarla, esto es indicado por la palabra clave *static*.

La directiva *.entrypoint* es la que indica al CLR que el método es el punto de entrada de la aplicación. Solo un método de la aplicación puede tener esta directiva.

La directiva *.maxstack* indica cuantas entradas la aplicación puedo usar como máximo de la pila. Por ejemplo, para sumar 2 números, estos deben ser introducidos a la pila, y luego aplicar el método *add*, y sacar ambos números. Esto significa que como máximo se necesitan 2 espacios dentro de la pila.

La instrucción *ldstr* añade la cadena que es pasada como parámetro al método *WriteLine* debe ser guarda en la pila. Y la instrucción *call* invoca el método estático WriteLine que se encuentra en el espacio de nombres System.Console que a su vez se encuentra en el ensamblado mscorlib.

Finamente la instrucción *ret*, retorna la ejecución al método que la llamó. En el caso del método de punto de entrada, finaliza la aplicación.

2.1.3 Ensamblados.

Un ensamblado es la unidad de despliegue, un bloque de construcción clave para las aplicaciones administradas [6]. Los ensamblados son también reusables, lo que permite que distintas aplicaciones usen el mismo. Estos contienen auto descripción completa en sus metadatos, incluyendo información acerca de la versión. Los ensamblados están preparados para resolver el viejo problema conocido como el infierno del DLL. Este sucede cuando se actualiza una aplicación, pero otra aplicación que hace uso de la anterior se vuelve inoperable porque ambos usan DLLs con el mismo nombre, pero diferente versión [7]. Otro problema resuelto con los ensamblados es el versionado. Ahora todo el manejo de versiones es hecho a nivel de ensamblados. También, los ensamblados son usados para compartir código entre aplicaciones.

En resumen un ensamblado .NET consiste de las siguientes elementos [8] (Solo el manifiesto es estrictamente requerido, pero tipos y recursos son necesarios para darle un significado):

- Información del manifiesto de metadatos, como la versión, el nombre, la cultura y cualquier requerimiento de seguridad.
- Metadatos que describen los tipos.
- Uno o más módulos de código intermedio.
- Cualquier recurso requerido.

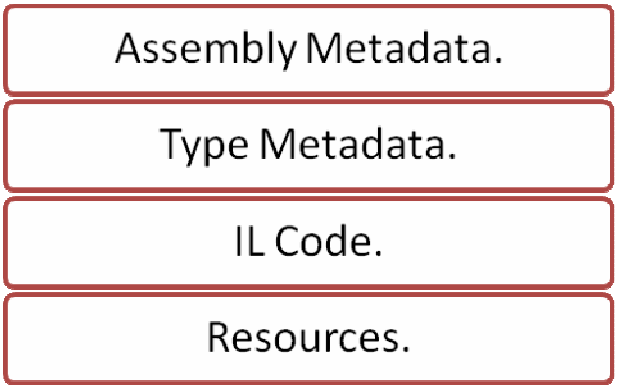

Figure 5. DLL que contiene todos los elementos de un ensamblado.

Los ensamblados están clasificados ya sea en privados ó en compartidos [6]. La diferencia entre ambos es básicamente en cómo son nombrados y desplegados.

Los ensamblados privados son considerados una parte particular de una aplicación en específico, por tanto es considerado responsabilidad del autor de la aplicación. El único requisito es que el nombre del ensamblado debe ser único en esa aplicación. Por otro lado, los ensamblados compartidos son más restrictivos ya que varias aplicaciones pueden hacer

uso de los mismos. El nombre de un ensamblado compartido debe ser único globalmente. Esto se logra mediante el uso de *strong names*, los cuales usan algoritmos criptográficos con llave pública/privada para asegurar que los nombres sean únicos.

Una parte importante de la librería de clases, que está relacionada con los ensamblados es el API propuesto para leer los metadatos de los ensamblados conocido como reflexión, y que se encuentra en System.Reflection. Este API provee de una vista lógica de los metadatos, por tanto puede ser usado para inspeccionar clases y miembros, e incluso invocar métodos de un ensamblado. En la versión 2.0 del Framework, la reflexión ya permitía obtener el CIL de un método [11].

Algo muy importante para la investigación realizada en esta tesis, fue saber dónde estaban definidas las dependencias externas de un ensamblado o módulo. La respuesta es que esta información está guardada en la tablada de metadatos AssemblyRef. Solo existe un caso especial, para mscorlib.dll, el cual es el ensamblado raíz de las clases del .NET Framework, y por tanto tiene una tabla AssemblyRef vacía [6].

La razón de porque esto es tan importante, es porque estamos concentrados en romper las dependencias entre ensamblados, no entre tipos, módulos u otra clase de unidades. El problema con usar clases o tipos, es que el diseño orientado a objetos está predicado a la noción de agrupaciones de estados y comportamientos autónomos, los cuales son conocidos como clases, y que de alguna forma colaboran para realizar una tarea. Por tanto, a algún nivel, queremos tener relaciones entre estas clases, y por tanto esperamos que algunas sean dependientes unas a otras. Desafortunadamente, el número de clases en una aplicación, tiende a ser muy grande, por tanto administrar las dependencias a nivel de clase

no funcionaría. Por otro lado, un ensamblado es la unidad de despliegue, mantenimiento y versionado. También, se sabe que un buen diseño orientado a objetos, las clases que colaboran cerca para realizar tareas específicas son denominadas como cohesivas y son agrupadas juntas en un mismo ensamblado. De hecho, si se quisiera cambiar la funcionalidad de una aplicación existente en .NET, el esfuerzo realizado tendrá que ver con añadir o reemplazar un ensamblado. Por tanto, administrar dependencias entre ensamblados es un objetivo mucho más real y más valioso.

2.2 Refactorización.

Refactorizar es el proceso de cambiar un software de tal forma que no se altere su comportamiento externo, sino más bien mejore su estructura interna. Básicamente, es una forma disciplinada de limpiar el código, minimizando generar errores por estos cambios [3]. En esencia, cuando un desarrollador refactoriza su código, mejora el diseño del código fuente después de que se ha terminado una versión funcional.

El concepto de refactorización es importante dentro de esta investigación, ya que sus guías nos ofrecieron la forma correcta de hacer cambios al código existente. Por ejemplo, cómo se debe mover un método a otro tipo, manteniendo la funcionalidad. Este último principio también es clave dentro del trabajo, ya que uno de los objetivos es proporcionar nuevos puntos de extensión dentro de la aplicación existente pero al mismo tiempo mantener la misma funcionalidad.

Regresando al concepto de refactorización, sin aplicar su proceso, el diseño de un programa decaería. Esto sucede porque mientras las personas cambian código, este pierde

su estructura. Se vuelve difícil entender el diseño del código solo leyéndolo, por tanto la refactorización es como ordenar el código de tal forma que sea mucho más legible.

Normalmente el código que fue pobremente diseñado, usa más código para realizar una tarea, porque el código hace lo mismo en distintas partes. Por tanto una de las partes más importantes de la refactorización es eliminar el código duplicado [39]. Reducir el número de líneas de código no hará que el sistema corra más rápido, ni nada por el estilo, sin embargo hace una diferencia al momento de modificar el código. Ya que si hay más código, es más difícil de modificarlo correctamente, ya que hay más código que entender.

Otro problema en el que el proceso de refactorización es un aliado del desarrollador ocurre porque normalmente cuando alguien programa un software, no piensa en el futuro desarrollador que se hará cargo del programa. La refactorización te ayuda a tener código más legible. Con un poco de esfuerzo durante el proceso, se puede lograr que el código comunique de una mejor forma sus propósitos [3].

Ayudar a entender la estructura del código y el código mismo también ayuda a prevenir errores de programación [3]. Una vez que se tiene claro la estructura del código de un programa, se pueden clarificar las asunciones hechas, hasta el punto de evitar introducir errores.

Finalmente, un buen diseño es esencial para el desarrollo de software rápido. Sin un buen diseño, es probable que se avance de manera rápida por un tiempo, pero llega el momento en el que el mal diseño comienza a alentar el desarrollo. Esto se debe a que se empieza a ocupar más tiempo en encontrar errores de programación, que añadir funcionalidades. Los cambios comienzan a tomar más tiempo, porque se tiene que entender

todo el sistema así como encontrar el código duplicado. Esto significa que la refactorización ayuda a desarrollar software más rápido, porque evita que un software decaiga [41].

En el caso específico de nuestra investigación, dado que la unidad base para el trabajo son los ensamblados, la forma obvia de romper dependencias sería siguiendo los principios de la refactorización, pero en lugar de hacer los cambios directos al código fuente de la aplicación, hacerlo de una forma automatizada mediante la inyección y cambio de código intermedio directo en los ensamblados. Naturalmente, no estamos haciendo un enfoque basado totalmente en la refactorización como tal, pero estamos siguiendo sus principios más fundamentales, como preservar su estructura externa.

Muchos de los cambios que se realizan automáticamente en el código intermedio, fueron realizando una adaptación del código existente a patrones de diseño de software muy conocidos en el mercado. Uno de ellos, con especial importancia, es el patrón inyección de dependencias, el cual es estudiado en el sub-capítulo 2.3.

2.3 Patrón de diseño: Inyección de dependencias.

Inyección de dependencias (DI) es un estilo de configuración de objetos en la que los campos de objetos y colaboradores son establecidos por una entidad externa [43]. En otras palabras, los objetos son configurados por una tercera entidad. Este patrón es una alternativa a que los objetos se tengan que configurar a sí mismos.

Es uso del patrón de diseño factory es una forma común de implementar DI. Cuando un componente crea una instancia privada de otra clase, la lógica de su inicialización está atada a esa componente. Esta lógica de inicialización es raramente usada

fuera de la creación del componente, por tanto debe ser duplicada para otras clases que requieren instancias de la clase creada.

El enfoque DI ofrece mucho más flexibilidad porque se vuelve más fácil crear implementaciones alternativas del tipo especificado, y después especificar qué implementación usar, mediante un archivo de configuración, sin tener que cambiar los objetos que usan el servicio [44]. Esto es especialmente útil para la prueba de unidades, porque es más fácil inyectar una implementación mock al servicio que se requiere probar.

Martin Fowler identificó 3 formas en las que un objeto puede obtener la referencia de un módulo externo, de acuerdo al patrón usado para proveer la dependencia, se tiene [43]:

- Tipo 1 ó inyección de interfaz, en la que el módulo exportado provee una interfaz que los usuarios deben implementar con el objetivo de obtener la dependencia en tiempo de ejecución.

- Tipo 2 o inyección de establecimiento, en la que el módulo dependiente expone un mecanismo para establecer qué tipo de objeto debe el framework inyectar como dependencia.

- Tipo 3 o inyección al constructor, en la que las dependencias son provistas, usando el constructor de la clase.

Un patrón de diseño clave para DI, es el conocido como Abstract Factory Pattern. El objetivo de este es el de “Construir e instanciar un conjunto de objetos relacionados sin especificar sus objetos concretos” [42]. Utilizar este patrón en las aplicaciones permite

definir clases abstractas para crear familias de objetos. Mediante la encapsulación de las instanciaciones y lógica de construcción de objetos, se retiene el control de las dependencias y de los estados permitidos de los objetos.

Otro tipo de patrón de tipo factory es el llamado factory method. Este patrón es simplemente un método, normalmente definido como estático, con el único propósito de regresar una instancia de una clase [42]. Algunas veces, para facilitar el polimorfismo, una bandera indicando el tipo de objeto a regresar es pasado a esta función.

Ambos tipos de factories permiten a una aplicación unir objetos y componentes sin exponer mucha información acerca de cómo estos componentes funcionan juntos o que dependencias tiene cada componente. En lugar de tener código complejo para crear objetos, los factories ayudan que este código este encapsulado en un lugar central, y por tanto facilitar su reutilización en la misma aplicación.

Sin embargo, los factories también tienen inconvenientes. Uno de ellos es que aunque el valor del factory es muy alto dentro de una aplicación, la mayoría del tiempo no son reusables en otras aplicaciones. Esto se debe a que la lógica que conllevan es normalmente muy atada a la lógica de la aplicación donde se desarrolla, volviendo al factory un clase no extensible. Otra razón es que los objetos que son creados por el factory deben conocerse en tiempo de compilación [44]. Por tanto no hay forma de insertar o alterar la forma en que los objetos son creados, esto se debe hacer en tiempo de diseño. Sin embargo, veremos más adelante que el .NET Framework provee ciertos mecanismos que permiten hacer esto en tiempo de ejecución. Otra razón, es que los factories son personalizados para cada implementación individual. Por ejemplo si un objeto necesita ser

creado en una forma similar que otro, la lógica para hacerlo usando un factory tiene que ser repetido para ambos objetos. Finalmente, los factories necesitan tener un conjunto de interfaces para poder hacer uso del polimorfismo [44]. Para que la implementación de un factory sea capaz de crear diferentes tipos de objetos de forma dinámica se necesita una clase base o una interfaz compartida, la cual es implementada por todas las clases que el factory creará instancias.

La mayoría de estos problemas se pueden resolver usando lo que se conoce como un contenedor de inyección de dependencias. El objetivo de estos contenedores es el de tomar responsabilidad de la administración de los objetos, como instanciaciones, configuraciones así como cosas más específicas.

Una característica importante es que los contenedores te dan la oportunidad de configurar a los objetos en el mismo contenedor, y no en la aplicación. Esto expande la funcionalidad de los contenedores, como el manejo del ciclo de vida de los objetos y la resolución de dependencias.

Para los propósitos de nuestra investigación, es necesario desarrollar un tipo especial de contenedor de inyección de dependencias. Dado que estamos trabajando directamente con el código intermedio de los ensamblados, hacer uso de un contenedor existente sería muy difícil dado todos los factores que habría que cambiar y cumplir para ser compatible con un contenedor en específico. Por tanto, decidimos crear nuestra propia implementación primitiva de un contenedor basados en el patrón factory method y usando algunas de las características que nos provee el .NET Framework 3.5 como reflexión, delegados genéricos y una plantilla para la generación e inyección del código.

Reflexión es el proceso en el cual un programa puede observar y modificar su propia estructura y comportamiento. Reflexión puede ser usado para observar y modificar un programa en tiempo de ejecución. También, puede ser usado para adaptar el programa a diferentes situaciones de forma dinámica.

Específicamente en el .NET Framework se tiene un espacio de nombres que implementa las capacidades especificadas para la reflexión en el framework. Este espacio de nombres provee de objetos que encapsulan los metadatos e información de los ensamblados, módulos y tipos. Con reflexión se puede crear instancias de tipos dinámicamente, añadir ese tipo a un objeto existente ó obtener un tipo de un objeto existente e invocar sus métodos o acceder a sus elementos y propiedades [12].

2.4 Generación de código.

Como programadores, hay veces que debemos lograr la misma funcionalidad pero en distintos contextos. Necesitamos repetir información en diferentes lugares. En estos casos los programadores crean generadores de código. Una vez construido, estos pueden ser usados las veces que sea necesario a lo largo del ciclo de vida del proyecto, sin ningún costo. En otras palabras, un generador de código es un programa que escribe código automáticamente. Existen dos tipos de generadores de código [46]:

- Generadores de código pasivos, que son los que se corren una vez para producir un resultado.
- Los generadores activos que son cada vez que sus resultados son requeridos.

El objetivo de los generadores pasivos es el de evitar un poco de trabajo de escritura de código. Y el de los generadores activos son los que toman la representación de algún

conocimiento y lo convierte en otras formas que son más específicas y necesarias para la aplicación. Esto no es duplicación, porque las formas derivadas estás disponibles y son generadas cada vez que son necesarias.

Aunque la mayoría de los ejemplos son generadores de código que producen código fuente, este no es siempre el caso. Se pueden usar generadores de código para producir cualquier tipo de resultado como por ejemplo HTML, XML, texto plano, etc.

En el desarrollo de nuestra aplicación tomamos ventaja de un generador de código para realizar algunas partes repetitivas del código. Como resultado, tenemos menos código para realizar esta función y un programa automatizado que genera código en base a una plantilla. Detalles de este mecanismo están en el capítulo 4.