5. Pruebas y Resultados.

Para las pruebas de la aplicación decidimos que el mejor enfoque sería implementando ejemplos de aplicaciones que tuvieran los requerimientos de dependencias para ser refactorizadas, una vez hecho esto, mostrar como tomar ventaja y añadir nuevas funcionalidades. Por esta razón, se desarrollaron dos aplicaciones ejemplo para el escenario de dependencias contenidas. Ambos ejemplos no son aplicaciones reales, son simples, porque el objetivo es mostrar las características de la herramienta.

5.1 Ejemplo de aplicación: Current Date.

El propósito de esta aplicación es bastante simple. Imprime la fecha actual en la consola. Después de aplicar la refactorización podremos proveer de una segunda versión que en lugar de imprimir en la consola, imprimirá la fecha en una ventana. En resumen, cambiaremos una aplicación que fue diseñada para ser usada en consola, y modificarla para convertirla en una aplicación con una interfaz gráfica.

La aplicación Current-date está compuesta por distintos ensamblados. Cada uno representa las entidades básicas que se deben cumplir, como una clase abstracta, una clase concreta, y ensamblado que es dependiente.

Comencemos describiendo el tipo base. El tipo base está guardado en una librería llamada CurrentDateBaseLibrary.dll que contiene un tipo denominado BaseCurrentDate.

BaseCurrentDate es una clase abstracta que contiene 2 métodos:

- GetCurrentDate, que regresa la fecha actual del sistema.
- Un método abstracto llamado DisplayDate, el cual tiene como objetivo dado una fecha, mostrarla de alguna forma.

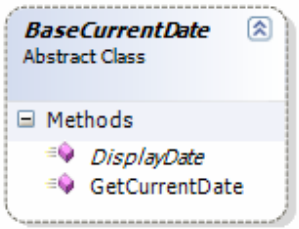

Figure 1. Diagrama de clases de BaseCurrentDate.

La implementación de BaseCurrentDate es la siguiente:

```
public abstract class BaseCurrentDate
{
   public DateTime GetCurrentDate()
   {
      return DateTime.Now;
   }

   public abstract void DisplayDate(DateTime date);
}
```

Por otro lado, tenemos la implementación concreta de BaseCurrentDate. Este tipo está guardado en el ensamblado CurrentDateConsoleLibrary.dll, y tiene un tipo que hereda de BaseCurrentDate. Esta clase está encargada de sobrecargar el método DisplayDate y proporcionar una implementación. La implementación es muy sencilla, sólo imprime la fecha usando el método ToString() de la fecha y luego lo imprime en la consola usando el método estático Console.WriteLine().

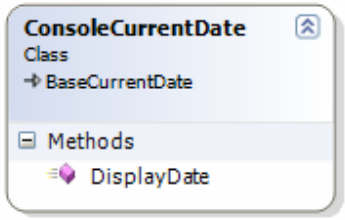

Figure 2. Diagrama de clase de ConsoleCurrentDate.

La implementación es la siguiente:

```
public class ConsoleCurrentDate : BaseCurrentDate
{
   public override void DisplayDate(DateTime date)
   {
      Console.WriteLine(date.ToString());
      Console.ReadKey();
   }
}
```

Finalmente, tenemos la librería que es dependiente de este componente, CurrentDateLibrary. Esta librería tiene una dependencia hacia la última librería de solo-implementación en el método Show.

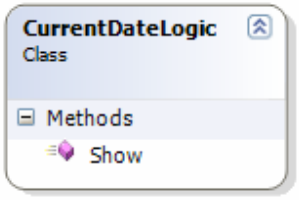

Figure 3. Diagrama de clase de CurrentDateLogic.

En la implementación de CurrentDateLogic podemos claramente ver la dependencia con ConsoleCurrentDateLibrary.

```
public class CurrentDateLogic
{
   public void Show()
   {
      BaseCurrentDate currentDate = new ConsoleCurrentDate();
      DateTime now = currentDate.GetCurrentDate();
      currentDate.DisplayDate(now);
   }
}
```

Además de todo esto existe una aplicación que se encarga de lanzar la última librería. Si corremos la aplicación, tendríamos la siguiente salida.

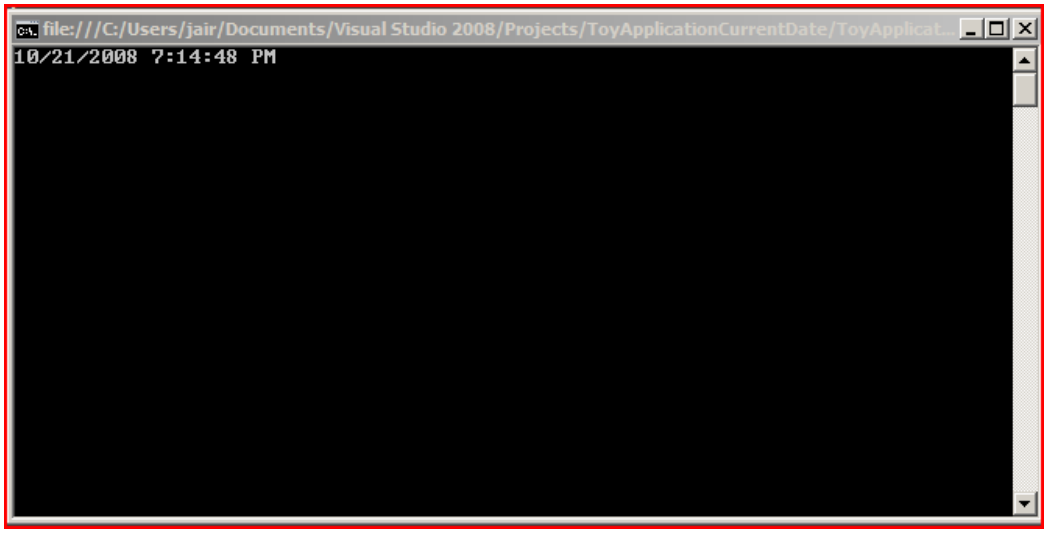

Figure 4. Salida de la aplicación original.

Esencialmente imprime la fecha en la consola. Con el objetivo de darle flexibilidad y extensibilidad a la aplicación correremos Afterex con los argumentos adecuados. Como mencionamos anteriormente el uso de Afterex es el siguiente:

USAGE: Refactor.exe <Library DLL> <Concrete Type DLL owner> <Concrete Type full name> <Base Type DLL owner> <Base Type full name>

En la qué:

- Library DLL es CurrentDateLibrary.dll
- Concrete Type DLL es CurrentDateConsoleLibrary.dll
- Concrete Type full name es ConsoleCurrentDate.
- Base Type DLL owner es BaseCurrentDateLibrary.dll
- Y Base Type full name es BaseCurrentDate.

Siguiendo los nombres de los argumentos, deberíamos poder correr la herramienta de la siguiente forma, suponiendo que todos los DLLs están en el mismo directorio:

Refactor.exe CurrentDateLibrary.dll CurrentDateConsoleLibrary.dll ConsoleCurrentDate CurrentDateBaseLibrary.dll BaseCurrentDate

Después de correrlo satisfactoriamente, los siguientes archivos son generados:

- Factory.dll
- Afterex.xml
- Un nuevo CurrentDateBaseLibrary.dll

Para terminar el proceso debemos substituir el nuevo CurrentDateBaseLibrary.dll y sobre-escribir el viejo. También será necesario copiar Factory.dll, SettingsManager.dll y Afterex.xml al directorio donde se encuentra localizada la aplicación consumidora o el ejecutable CurrentDateConsumer.exe.

Ahora, que el proceso de refactorización ha terminado, los clientes como la aplicación consumidora deberán funcionar exactamente como antes. Si corriéramos la aplicación debería mostrar el mismo resultado, pero con una fecha diferente.

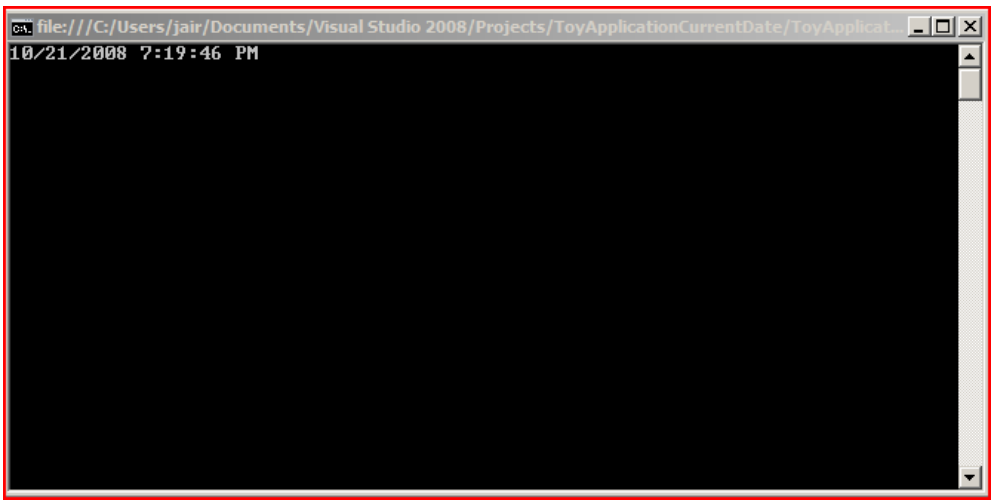

Figure 5. Salida despues de la refactorización.

Sin embargo, ahora tenemos la habilidad de cambiar la implementación de CurrentDateBaseLibrary, en especifico podríamos cambiar el comportamiento del método DisplayDate.

Para lograr esta tarea, tenemos que crear un nuevo ensamblado, con un tipo que heredará de BaseCurrentDate, e implementará el método DisplayDate como queramos.

Digamos que llamamos a este nuevo WinformsCurrentDate, también necesitamos crear un método estático que regresará instancias de WinformsCurrentDate. Esto podría realizarse de la siguiente manera:

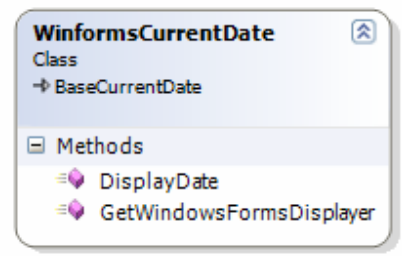

Figure 6. Diagrama de clases de WinformsCurrentDate.

```
public class WinformsCurrentDate : BaseCurrentDate
{
   public override void DisplayDate(DateTime date)
   {
      DateDialog dialog = new DateDialog();
      dialog.DateLabel.Text = date.ToString();
      Application.EnableVisualStyles();
      Application.Run(dialog);
   }

   public static BaseCurrentDate GetWindowsFormsDisplayer()
   {
      return new WinformsCurrentDate();
   }
}
```

En este ejemplo podemos ver como el método DisplayDate es implementado así como también el método estático que regresa instancia de ese objeto. Digamos que este tipo está guardado en el ensamblado CurrentDateWinformsLibrary.dll.

A partir de esto, lo único que se tiene que hacer para cambiar el comportamiento de la aplicación es editar el archivo de configuración XML y copiar el nuevo ensamblado.

Esto se puede realizar con el editor de texto de preferencia. Si abrimos el archivo de configuración debe contener algo parecido a lo siguiente:

```
<Settings>
<Assembly>Factory.dll</Assembly>
<Type>Afterex.Factory.FactoryBuilder</Type>
<Method>ReturnDefaultTypeInstance</Method>
</Settings>
```

Estos son los datos correspondientes a la dependencia original: CurrentDateConsoleLibrary.dll. Para que se pueda inyectar una dependencia debemos editar este archivo, y cambiarlo para que diga algo como lo siguiente:

```
<Settings>
<Assembly>CurrentDateWinformsLibrary.dll</Assembly>
<Type>WinformsCurrentDate<Type>
<Method>GetWindowsFormsDisplayer</Method>
</Settings>
```

Salvamos el archivo, y ahora cuando corramos la aplicación consumidora, tomará la nueva implementación WinformsCurrentDate en lugar de ConsoleCurrentDate, y el resultado final será algo como:

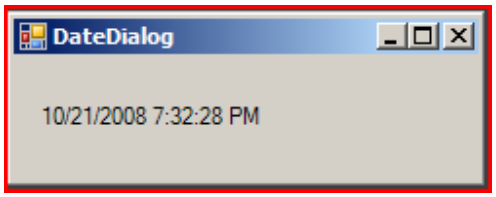

Figure 7. Nuevo comportamiento de la aplicación Current Date.

Ahora el programa original usa una implementación diferente sin recompilar, proveyendo los siguientes beneficios:

- La aplicación ya no es responsable de encontrar sus dependencias.
- El contenedor de encarga de encontrar esas dependencias.
- Añadimos flexibilidad a la aplicación para futuros cambios.
- Promovimos la disminución del acoplamiento entre clases y subsistemas, por lo cual facilitamos pruebas de unidad en los mismos.

5.2 Ejemplo de aplicación: Generación de bitácoras.

La segunda aplicación ejemplo es una más realista. Se trata de un bien conocido componente, conocido como cross-cutting concern. Se trata de un componente que genera bitácoras de errores de una aplicación.

La jerarquía de componentes de esta aplicación es la misma que para la aplicación current date. El propósito de ejemplo es el de mostrar cómo podemos tomar ventaja para cambiar el comportamiento de un componente más real. La aplicación comenzará guardando los mensajes de error en un archivo de texto. Y después cambiaremos esta dependencia directa para que tomen otras implementaciones y podemos guardar los mensajes de error en un archivo XML ó incluso mostrarlos en tiempo real en una consola.

Comenzaremos igualmente describiendo el tipo abstracto. Es una interfaz que contiene un método llamado Log, el cual recibe una cadena el cual es el mensaje de error. La interfaz está guardada en un componente llamado LoggingBaseLibrary.dll y su implementación es la siguiente:

```
public interface ILog
{
    void Log(string msg);
}
```

También, tenemos la implementación concreta de esta interfaz, el cual tiene como propósito guardar estos mensajes de error en un archivo de texto específico. Este tipo está guardado en un ensamblado llamado TextLoggingLibrary.dll.

```
public class TextLogger : ILog
{
    public void Log(string msg)
    {
        string logMessage = DateTime.Now.ToShortString() + " :" + msg;
        StreamWriter sw;
        sw = File.AppendText("C:\Log.txt");
        sw.WriteLine(logMessage);
        sw.Close();
    }
}
```

Y por supuesto, tenemos otra librería que hace uso de este componente. En una aplicación real el componente de bitácoras debería ser usado en escenarios de manejo de excepciones. Sin embargo, para este ejemplo, hemos creado un simple método que usa la implementación de TextLogger y llama a su método Log. Este tipo está guardado en el ensamblado LogApplication.dll.

```
public class Application
{
    public void LogMessage(string msg)
    {
        ILog logger = new TextLogger();
        logger.Log(msg);
    }
}
```

Finalmente, tenemos una aplicación que se encarga de lanzar a este tipo. Para nuestros propósitos esta aplicación manda datos que hace llamados directos a Application para crear mensajes de error y obtener la salida esperada.

```
public class Executable
{
    public static void Main(string[] args)
    {
        Application app = new Application();
        app.LogMessage("First test");
        app.LogMessageS("Second test");
        app.LogMessage("Third test");
        app.LogMessage("Fourth test");
    }
}
```

Siguiendo la implementación, cuando corremos el programa debemos tener como salida un archivo de texto. Este archivo contendrá 4 diferentes líneas que especifican un mensaje y una fecha de cuando el mensaje fue recibido.

Ahora imaginen que este componente está usado por todos los demás componentes de la aplicación, ya que todos los componentes necesitan una forma de guardar sus mensajes, pero ahora se necesita un archivo más fácil de procesar, por ejemplo un archivo XML. Ó por otro lado, tenemos varios componentes que funcionan en tiempo real, y necesitamos mostrar los mensajes de error en monitores cuando ocurran en tiempo real. Para lograr cualquiera de estos objetivos será necesario refactorizar nuestros ensamblados binarios usando Afterex, con los siguientes argumentos:

Refactor.exe LogApplication.dll TextLoggingLibrary.dll TextLogger LoggingBaseLibrary.dll ILog

Una vez que se ejecute correctamente, tenemos que sustituir los archivos generados y además copiar los archivos de configuración así como la librería que contiene el inyector de dependencias. Después de hacer esto, la aplicación debe correr como antes.

Ahora, escribamos ejemplos si quisiéramos cambiar el comportamiento actual por los mencionados. El primer que atacaremos es que ahora en lugar de guardarlos en texto, los guarda en un archivo XML. El proceso es exactamente el mismo. Primero necesitamos escribir un nuevo componente que tenga un tipo que herede de la interfaz ILog, este implementará la funcionalidad nueva que requerimos. Adicionalmente, se tendrá que crear un método estático que regrese instancias del tipo creado. Finalmente, se actualizará el archivo de configuración XML para que apunte a los datos del nuevo tipo creado.

```
public class XMLLogger : ILog
{
    public void Log(string msg)
    {
        string logMessage = DateTime.Now.ToShortString() + " :" + msg;
        XDocument document = XDocument.Load("C:\Log.xml");
        XElement root = document.Root;
        root.Add(new XElement("item", logMessage));
    }

    public static ILog GetLoggerInstance()
    {
        return new XMLLogger();
    }

}
```

Dependiendo en donde se haya guardado este tipo, será la información usada para llenar el archivo de configuración. Una vez logrado esto, cuando nuestra aplicación original se ejecute, en lugar de generar un archivo de texto, generará un archivo XML. En el caso de

que simplemente que los mensajes aparecieran en la consola, proporcionaríamos la siguiente implementación:

```
public class ConsoleLogger : ILog
{
    public void Log(string msg)
    {
        string logMessage = DateTime.Now.ToShortString() + " :" + msg;
        Console.WriteLine(logMessage);
    }

    public static ILog GetLoggerInstance()
    {
        return new ConsoleLogger();
    }
}
```

Si cambiamos de nuevo el archivo de configuración para que ahora apunte a este nuevo tipo, nuestra aplicación original imprimirá los mensajes en la consola en lugar de los archivos. Y esto puede extenderse a cualquier escenario que requiera cambiar una implementación concreta de una aplicación existente, librería o framework.

Finalmente, estos ejemplos mostraron escenarios reales dónde la herramienta puede ser aplicada. Solo se mostró las aplicaciones de romper una dependencia contenida, ya que estas son las que pueden aplicarse a sistemas reales y logrando el resultado esperado, el cual es dar la posibilidad al desarrollador de poder cambiar comportamientos e implementaciones de dependencias concretas.