Inyección de Dependencias en el Lenguaje de Programación Go

Dependency injection in GO programming language

Carlos Eduardo Gutiérrez Morales
Instituto Tecnológico de Colima
g0402748@itcolima.edu.mx

Jesús Alberto Verduzco Ramírez
Instituto Tecnológico de Colima
averduzco@itcolima.edu.mx

Nicandro Farías Mendoza
Instituto Tecnológico de Colima
nmendoza@ucol.mx

Resumen

Actualmente, una de las características más buscadas en un proyecto de software es la flexibilidad debido a que los requerimientos tienden a cambiar durante el proceso de desarrollo. Una manera de obtener código desacoplado es mediante el uso de Inyección de Dependencias (DI por sus siglas en inglés). Este documento trata sobre la construcción de una librería de código abierto para el uso de DI en proyectos desarrollados con el lenguaje de programación Go, también conocido como Golang. Esta librería fue construida siguiendo el principio de Inversión de Control (IoC por sus siglas en inglés), tratando de seguir con la estructura común encontrada en los contenedores de DI más populares y

teniendo en cuenta las limitaciones que el lenguaje posee. El producto final es una librería fácil de usar, que permite construir el código más fácil de mantener.

Palabras clave: Inversión de Control, Inyección de Dependencias, Flexibilidad del software.

Abstract

Currently, one of the features most sought in a software project is the flexibility since the requirements tend to change during the development process. One way to get uncoupled code is through the use of Dependency Injection (DI). This document is about the construction of a library of open code for the use of DI in projects developed with programming language Go, also known as Goland. This library was built following the principle of Inversion of Control (IoC), trying to follow the common structure found in the most popular DI containers and taking into account the limitations that the language has. The final product is an easy to use library, which allows writing code easier to maintain.

Keywords: Inversion of Control, dependency injection, software flexibility.

Fecha Recepción: Septiembre 2014 **Fecha Aceptación:** Diciembre 2014

Introducción

Según la descripción dada en la página web oficial[1], Go es un lenguaje de código abierto que hace simple la construcción de código sencillo, confiable y eficiente. Este lenguaje lleva relativamente poco tiempo en el mercado, por lo que aún carece de una gran cantidad de frameworks y librerías que otros lenguajes poseen. Por dicho motivo, este proyecto trata sobre la creación de una librería para inyección de dependencias en Go que contribuyan a

[1] Recuperado de http://golang.org

llenar el nicho de las librerías para inyección de dependencias, permitiendo crear proyectos de software con un mayor nivel de flexibilidad, es decir, mejor preparados para ser modificados.

Existen varios métodos para lograr tener código flexible y desacoplado. Un ejemplo es el patrón de diseño conocido como estrategia o strategy, el cual define un conjunto de algoritmos encapsulados que pueden ser cambiados para obtener un comportamiento específico (Erich Gamma et al, 1998).

Otro ejemplo es el principio Open-Close (Meyer, 1997), el cual menciona que las entidades de software (clases, módulos, funciones, etcétera) deben estar abiertas para extensión, pero cerradas para modificación.

Ambos ejemplos tienen algo en común: usualmente el código se modifica para depender en abstracciones en lugar de implementaciones específicas, aumentando la flexibilidad del código. Este es el mismo principio que hace posible la inyección de dependencias.

El objetivo del uso de inyección de dependencias en un proyecto tiene como objetivo aumentar la mantenibilidad, la cual es muy difícil de medir debido a que depende de varios factores, algunos de los cuales son muy subjetivos.

Existen algunos estudios como el realizado por Ekaterina Razina y David Janzen (2007) que mencionan que el mantenimiento del software consume alrededor del 70 % del ciclo de vida del mismo. Igualmente mencionan que algunos estudios (Arisholm, 2002 y Rajaraman, 1992, et al.) muestran que módulos pequeños, desacoplados y con una alta cohesión conllevan un incremento en la mantenibilidad.

En este mismo estudio se midieron 2 factores basados en la investigación de L. Briand, J. Daly, and J. Wust (1999): cohesión y acoplamiento.

El acoplamiento es definido como el grado de interdependencia entre las partes de un diseño (Chidamber, 1994, et al.). Para esta se utilizaron dos medidas:

- Acoplamiento entre objetos (CBO). Se refiere al número de clases a la cual una clase en particular está acoplada.

- Respuesta por clase (RFC). Es un conjunto de métodos que pueden ejecutarse potencialmente en respuesta a un mensaje recibido por un objeto de la clase.

La cohesión es el grado de similitud de los métodos (Chidamber, 1994, et al. y Briand, 1999, et al.). También podemos decir que es el grado en el cual cada parte del módulo está asociada con la otra (Razina, 2007, et al.).

Lamentablemente dicho estudio no pudo comprobar la hipótesis planteada de que la inyección de dependencias reduce significativamente las dependencias en módulos de software. Sin embargo, durante el mismo se encontró una tendencia de un menor acoplamiento en aquellos módulos con un porcentaje mayor de inyección de dependencias (mayor al 10 %), tal como se muestra en la figura 1.

Proj	%DI
3	3.1
9	41.7
11	10.43
13	10.09
14	19.39

Figura 1. Proyectos con menor acoplamiento

Inyección de dependencias

Esta sección explica de una manera más concisa la inyección de dependencias, de tal manera que el lector tenga una mejor idea de dicho concepto.

La Inyección de Dependencias es un conjunto de principios de diseño de software y patrones que nos permite desarrollar código débilmente acoplado (Seemann, 2011).

Sus ventajas son las siguientes:

Extensibilidad: Esta es la propiedad que permite fácilmente añadir nueva funcionalidad al código. Permite actualizar partes correctamente aisladas en lugar de modificar pequeñas partes a través de todo el código.

Ataduras tardías: Es la habilidad de escoger cuáles componentes usar en tiempo de ejecución en lugar de en tiempo de compilación. Esto solo se puede lograr si el código está débilmente acoplado; nuestro código solo interactúa con abstracciones en lugar de tipos concretos. Lo anterior nos permite cambiar componentes sin tener que modificar nuestro código.

Desarrollo paralelo: Si nuestro código esta débilmente acoplado, es mucho más fácil para diferentes equipos trabajar en el mismo proyecto. Podemos tener un equipo trabajando en la capa de negocios, otro en la de servicios y, debido a que las capas son independientes, los equipos trabajan en código fuente que no afecta directamente a los demás.

Facilidad de mantenimiento: Cuando nuestros componentes son independientes, la funcionalidad está aislada. Esto significa que si necesitamos buscar errores en el código o ajustar alguna funcionalidad, sabemos exactamente dónde buscar.

Facilidad para probar: Las pruebas unitarias son un tema muy importante. Su propósito es probar aisladamente pequeñas partes del código. Cuando tenemos código débilmente acoplado, fácilmente podemos usar dobles de prueba o dependencias falsas para aislar fácilmente las partes del código que deseamos probar.

Existen tres tipos de Inyección de dependencias:

- **Inyección por interfaz**: Esta consiste en definir una interfaz, la cual se usa para realizar la inyección. Dicha interfaz debe ser implementada por aquella clase que desee obtener la o las referencias definidas en la interfaz.
- **Inyección por constructor**: Utiliza un constructor para decidir cómo inyectar las dependencias necesarias. Por ejemplo, en el lenguaje de programación Java, estas

dependencias se pasan al hacer uso de la palabra clave *new* seguida del nombre de la clase, pasando las dependencias como parámetro.

- **Inyección por setter**: En este comúnmente se utiliza la convención de definir el método que se utiliza para inyectar una dependencia nombrándolo con la palabra *Set*, seguida del nombre de la dependencia.

Como ejemplo práctico, supongamos que tenemos una estructura A, la cual tiene una dependencia del tipo BI, la cual es una interfaz tal .Como se puede observar en la figura 2, podemos crear una estructura B que implemente la interfaz BI y añadirla a la estructura A. Finalmente imaginemos que la implementación B es creada internamente por A.

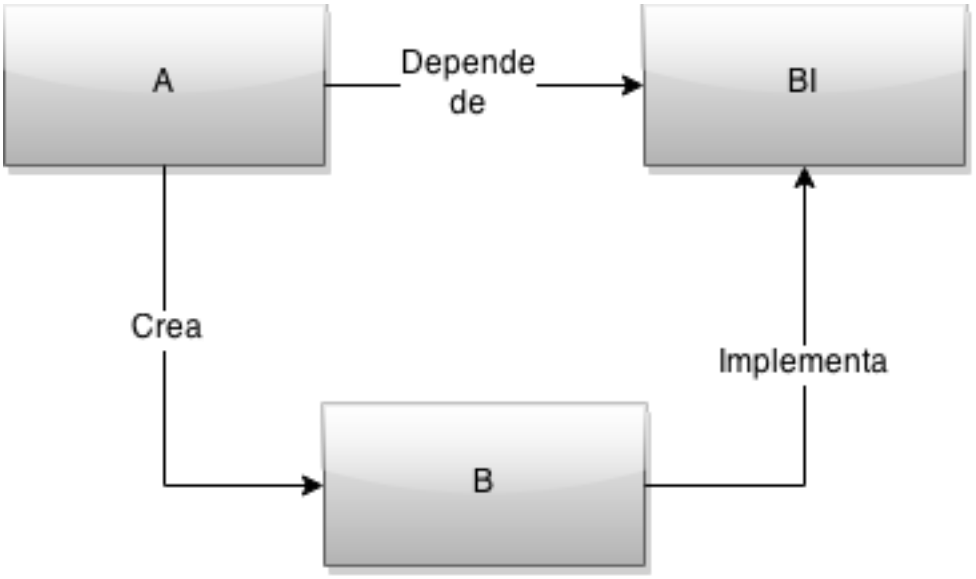

Figura 2. Ejemplo de dependencia

A crea directamente a B, lo que provoca un fuerte acoplamiento entre ambas estructuras y, por lo tanto, una menor flexibilidad en el código. Una respuesta a este problema es seguir el principio de inversión de control.

Inversión de control

Se trata de un método de programación donde el flujo de ejecución se invierte, lo que lo diferencia de los métodos de programación convencionales en los que las interacciones se hacen mediante llamadas explícitas a procedimientos o funciones. En el caso de la inversión de control, solo se especifican respuestas deseadas a eventos definidos y se deja que alguna entidad externa tome el control de la ejecución.

La inversión de control es el principio subyacente a la inyección de dependencias debido a que es el contenedor quien inyecta las dependencias cuando crea los objetos en lugar de ser estos últimos quienes controlen la instanciación o localización de dichas dependencias.

Por lo anterior, se dice que este método es una implementación del principio de Hollywood ("no nos llames, nosotros te llamamos"), comúnmente usado por algunos frameworks como el Spring Framework de Java.

Una característica importante de un framework es que los métodos definidos por el usuario para modificar el framework comúnmente serán llamados dentro de este y no desde el código del usuario. El framework comúnmente juega el rol del programa principal al coordinar y secuenciar la actividad de la aplicación. Esta inversión de control le da a los frameworks el poder de servir como un esqueleto extensible. Los métodos provistos por el usuario modifican los algoritmos genéricos definidos en el framework para una operación en particular (Cheney, 2013).

Aplicación de la inyección de dependencias

Tras definir el concepto de inversión de dependencias, retomemos el ejemplo anterior. Tenemos que la estructura A es la que actualmente se encarga de crear la instancia de la implementación B. Siguiendo el principio de Inversión de Control, podemos delegar la creación de la instancia a un elemento externo a A, lo que permitirá que A no tenga idea de los detalles acerca de la implementación especifica de BI con la que está tratando. Así, se obtiene un código débilmente acoplado y, por ende, más flexible.

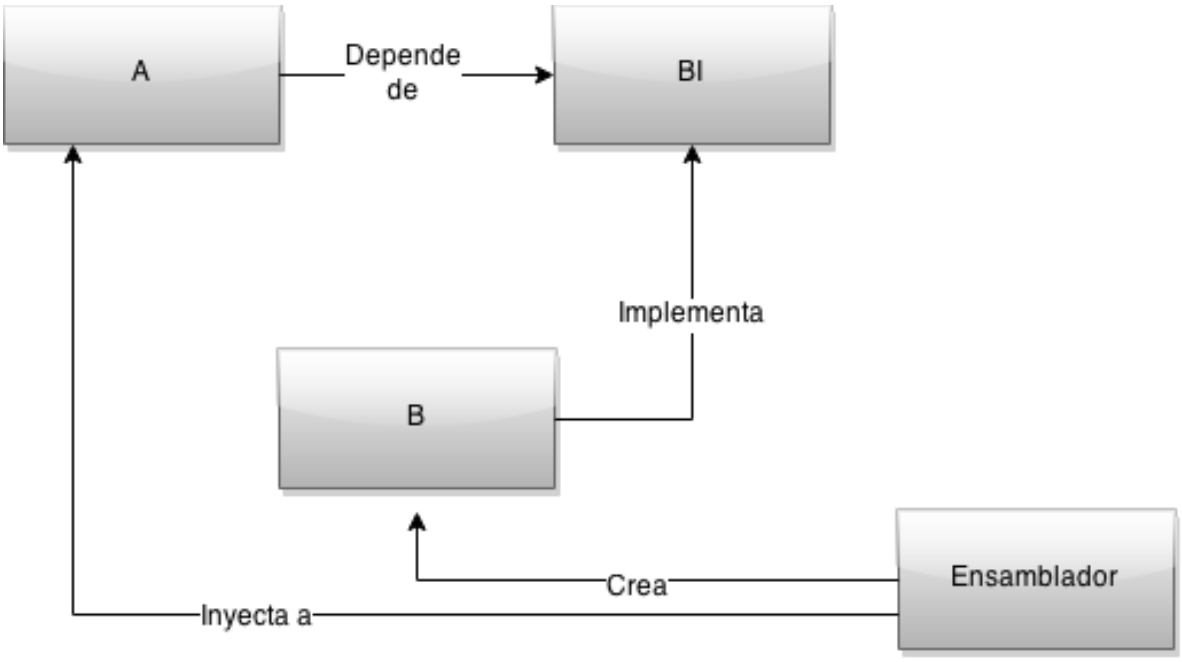

Figura 3. Aplicación de la inyección de dependencias

Como podemos ver en la figura 3, es el objeto Ensamblador el que crea la instancia de B y la inyecta a A, invirtiendo el control sobre la creación de dicha dependencia y ayudando al desacoplo de ambas estructuras.

Diseño de la librería

Se propuso la creación de una librería para la inyección de dependencias que sea capaz de leer las configuraciones de un archivo en formato JSON y crear las dependencias e inyectarlas a los elementos correspondientes para formar el árbol de dependencias. Ver la figura 4.

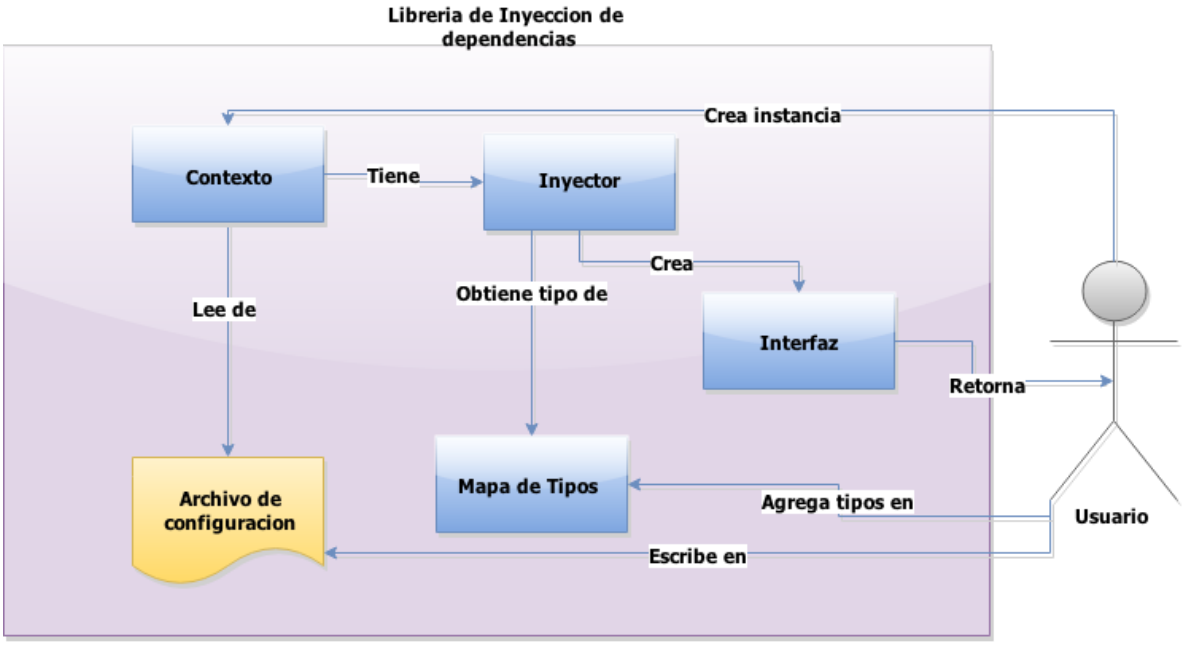

Figura 4. Modelo conceptual

Componentes

Contexto. Nos permite inyectar dependencias a partir de una configuración dada. Lo anterior nos permite poder cambiar el comportamiento del sistema.

Archivo de configuración. En este definimos los tipos de datos que maneja el inyector, así como sus dependencias. Este contiene los datos en formato JSON.

Inyector. Este es el encargado de resolver el árbol de dependencias; contiene una cache para guardar las dependencias de tipo *singleton*.

Mapa de tipos. Nos sirve para registrar los tipos de datos que se van a manejar. El lenguaje Go no permite crear instancias a partir de solamente el nombre de una estructura, así que se requiere registrar los tipos de los datos.

Interfaz resultante. Es el objeto generado, el cual ya contiene las dependencias especificadas en el archivo de configuración. Requiere una afirmación de tipo (type assertion).

El diagrama de clases de la librería desarrollada consta de seis clases que corresponden a los módulos especificados en el modelo conceptual. Ver figura 5.

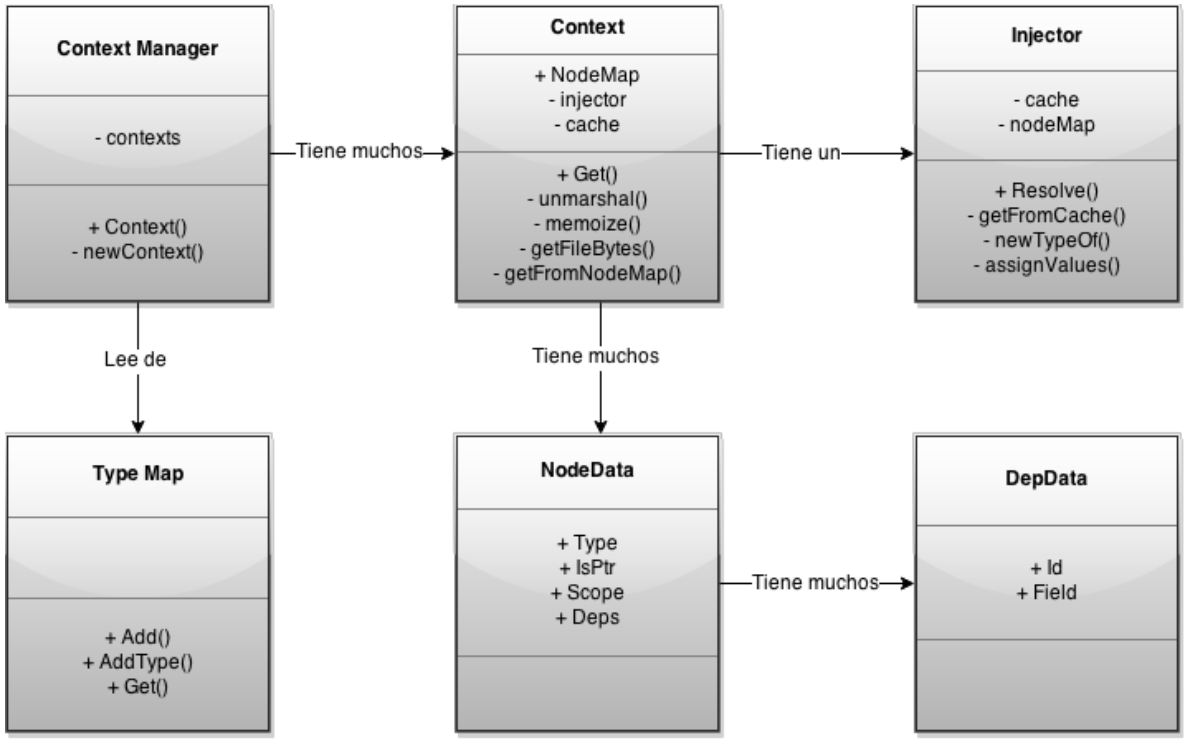

Figura 5. Diagrama de clases de la librería desarrollada.

Creación de la librería

En la etapa inicial se construyó el módulo del Inyector, las estructuras que representan en la configuración de las dependencias y se construyeron pruebas unitarias que pasaban los parámetros necesarios al inyector y verificaban el objeto resultante.

La construcción de esta primera etapa permitió verificar si las dependencias eran inyectadas de manera correcta al objeto resultante.

Posteriormente se agregó el módulo de Contexto e igualmente se agregaron pruebas unitarias para verificar el funcionamiento conjunto con el Inyector y con ciertas configuraciones.

Después, se añadió el módulo del Manejador de Contextos junto con un archivo de configuración y se agregaron las pruebas unitarias correspondiente para verificar si los contextos eran manejados de forma correcta y si era posible procesar las configuraciones del archivo y convertirlo a las estructuras correspondientes.

Por la naturaleza de la librería no es necesaria una base de datos. Aun así se diseñó el modelo para la representación en código de los datos contenidos en el archivo de configuración.

Dicho archivo de configuración se encuentra en formato JSON y consta de varias partes:

- El nodo padre que tiene por llave el nombre "nodes". Este es el nodo que engloba todas las configuraciones.
- Los nodos de información de los objetos, que tiene por llave un alias con el cual se representa el objeto. Este último tiene tres propiedades:
    1. La primera es el tipo del objeto, denotado por la ubicación de la estructura.
    2. La segunda es una bandera que denota si el objeto es un puntero.
    3. La tercera es un arreglo en el cual se especifican las dependencias para un objeto dado, para lo cual se especifica el alias que se le dio a la dependencia y el nombre del campo en el cual se inyecta.

El código de configuración se muestra en la figura 6.

```
{
  "nodes":{
    "super_fridge":{
      "type": "digo.SuperFridge",
      "is_pointer": true
    },
    "old_stove":{
      "type": "digo.OldStove"
    },
    "kitchen":{
      "type": "digo.Kitchen",
      "deps":[
        {
          "id": "super_fridge",
          "field": "MyFridge"
        },
        {
          "id": "old_stove",
          "field": "MyStove"
        }
      ]
    }
  }
}
```

Figura 6. Ejemplo de archivo de configuración

Gran parte de la eficiencia de un usuario al interactuar con un software depende de que su modelo de interacción sea simple, intuitivo y evite el mayor número de errores. En términos generales, un diseño correcto del modelo de interacción tiene como objetivo que el usuario se sienta satisfecho al operar un software.

En esta fase se evaluó la mejor manera como el programador usuario pudiera interactuar con la librería de manera intuitiva y no obstructiva. El usuario solo tendrá que importar la librería y utilizar el método *Context* para crear un nuevo contexto. Una vez teniendo un objeto de tipo Contexto se podrá utilizar el método *Get* sobre él, para obtener el objeto ya con sus dependencias.

La interfaz es muy simple ya que mucha de la complejidad se lleva a cabo en el archivo de configuración.

Pruebas

Para medir la velocidad de ejecución se utilizó el paquete *testing* de Go, el cual nos permite conocer la velocidad de ejecución de un proceso al escribir pruebas de tensión conocidas como *benchmarks*.

Tal y como Dave Cheney menciona, escribir *benchmarks* es una excelente manera de comunicar una mejora en el rendimiento o una regresión de un modo reproducible (Cheney, 2013).

Se crearon diversas pruebas para comprobar que la librería cumpliera con el objetivo para el cual fue diseñada. Debido a que se trata de una librería, se crearon varios escenarios ficticios para la prueba de la misma, tales como: accesos a una base de datos, utilización de arquitecturas en capas y utilización de objetos globales. Asimismo, se verificó que las validaciones para el archivo de configuración funcionaran de manera correcta.

Las pruebas de integración se llevaron a cabo usando la herramienta de integración continua llamada Travis, para lo cual se agregó un proyecto nuevo y se especificó que se utilizaría el lenguaje Go.

Al especificar el lenguaje, Travis automáticamente crea una máquina virtual, clona el proyecto desde el sistema manejador de versiones, el cual fue Github en este caso, y crea las variables de entorno necesarias, tales como GOROOT y GOPATH.

Finalmente, Travis corre todas las pruebas unitarias que hay en el proyecto mediante el comando "go test –v ./…". En dicho comando las palabras "go test" significan que se compila el código del proyecto y luego se corren las pruebas de los archivos con terminación "_test.go". La bandera "-v" significa que se imprime una descripción extendida de cada prueba. Y por último "./…" significa que las pruebas se buscan de forma iterativa en todo el proyecto.

Resultados

Esta investigación se limitó solo al desarrollo de la librería y pruebas de la misma en cuanto a su correcto funcionamiento. No se incluyó como parte de esta investigación la medida en la que ayudaría a un proyecto de software a ser más mantenible debido a que es muy difícil medir lo anterior.

Como resultado se obtuvo una librería de fácil uso que permite utilizar el principio de inyección de dependencias para crear código más flexible.

La librería fue probada en un proyecto de prueba previamente preparado y se comparó el código antes y después de utilizar la librería. El primer factor observado fue la flexibilidad del código. Consideremos el código mostrado en la figura 7.

```go
func main(){
    controller := NewController()
    controller.WriteProducts()
}

func NewController() *Controller{
    return &Controller{
        Model: &ProductModel{},
        Writer: &ProductWriter{},
    }
}

type Controller struct{
    Model products
    Writer writer
}

func (this *Controller) WriteProducts() (error, string){
    prods, err := this.Model.All()
    if err != nil{
        return err
    }

    return this.Writer.Write(prods)
}
```

Figura 7. Código de prueba inicial

Podemos observar que la estructura *Controller* tiene dos dependencias: *Model* y *Writer*, que son interfaces. Cuando creamos el controlador con la función *NewController* nosotros creamos las implementaciones específicas de dichas interfaces.

Si en algún momento se necesita cambiar una de las implementaciones por otra, por ejemplo, cambiar el *Writer* por una implementación que produzca una cadena en formato JSON con la información de los productos llamada *JsonWriter*, tendríamos que cambiar la función *NewController* para agregar dicha implementación como se muestra en la figura 8.

```
func NewController() *Controller{
    return &Controller{
        Model: &ProductModel{},
        Writer: &JsonWriter{},
    }
}
```

Figura 8. Cambio de implementación

Esto genera un acoplamiento entre la función *NewController* y la implementación de *Writer* que queramos utilizar. Sin embargo, podemos lograr un desacoplamiento de dichos elementos al usar la librería, tal como se muestra en la figura 9.

```
func NewController(context digo.Context) *Controller{
    return context.Get("controller")
}
```

Figura 9. Refactorización con la librería construida

Se pueden observar varias ventajas con la utilización de la librería:

- El desacoplo del código ya que los detalles de las implementaciones específicas serán manejados por la librería mediante el archivo de configuración.
- La reducción del código para crear el controlador.
- La facilidad para cambiar de dependencias sin tener que recompilar el código nuevamente.

El código fuente de la librería resultante se encuentra alojado en un repositorio de Github bajo una licencia de tipo MIT y puede ser consultado en la URL siguiente: https://github.com/cone/digo.

Conclusión

El propósito de esta investigación es el desarrollo de una librería open source que permita la inyección de dependencias en el lenguaje Go. Para lograr lo anterior se procedió a hacer una investigación acerca de los conceptos necesarios tales como inversión de control (IoC) y el mismo concepto de inyección de dependencias DI.

Como se mencionó anteriormente, es muy difícil medir en qué grado la inyección de dependencias ayuda a un proyecto a ser más flexible al cambio y, por ende, mas mantenible. Por ejemplo, se puede usar la inyección de dependencias pero eso no garantiza que el resto del código esté desacoplado, o incluso que los objetos que están siendo inyectados lo estén.

Lo anterior se encuentra fuera del alcance del presente proyecto, cuyo propósito es solo la creación de dicha librería y no observa la eficacia de la misma. Por lo tanto, no se observa el grado en que esta librería ayuda a crear código más mantenible, si ayuda a reducir las líneas de código o qué tanto ayuda a reducir el tiempo de desarrollo de un proyecto.

Como usuario de frameworks que permiten la inyección de dependencias, en mi opinión la inyección de dependencias ayuda a crear código más flexible y, en el caso de los que proporcionan la opción de utilizar un archivo de configuración externo, permiten crear código que es más fácil de probar al ser capaces de cambiar el comportamiento del sistema sin tener que recompilar el código fuente.

Bibliografía

E. Gamma, R. Helm, R. Johnson, and J. Vlissides (1995). Design patterns: Elements of reusable object-oriented software. Reading, Mass.: Addison-Wesley.

Gamma, E. (1995). Design patterns: Elements of reusable object-oriented software. Reading, Mass.: Addison-Wesley.

E. Arisholm (2002). Dynamic coupling measures for object oriented software. IEEE Symposium on Software Metrics, 30(8), pp. 33-34.

C. Rajaraman and M.R. Lyu (1992). Reliability and maintainability related software coupling metrics in c++ programs. In Third International Symposium on Software Reliability, North Carolina, USA, pp. 303-311.

S. R. Chidamber and C. F. Kemerer (1994). A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 20(6), pp. 476-493.

L. Briand, J. Daly, and J. Wust (1999). A unified framework for coupling measurement in object-oriented systems. IEEE Transactions on Software Engineering, 24(1), pp. 91-121.

M. Seemann. (2011). Dependency Injection in .Net. Recuperado de: http://www.manning.com/seemann/MEAP_Seemann_01.pdf

S. Chacon and B. Straub (2010, Agosto 2). Pro Git. Recuperado de: http://labs.kernelconcepts.de/downloads/books/Pro%20Git%20-%20Scott%20Chacon.pdf

S. Chacon and B. Straub. (2010, Agosto 2). Pro Git. Recuperado de: http://git-scm.com/book/en/v2/GitHub-Account-Setup-and-Configuration

M. Fowler. (2006, Mayo 1). Continuous Integration. Recuperado de: http://www.martinfowler.com/articles/continuousIntegration.html

R. Johnson and B. Foote. (1988, Junio/Julio). Designing Reusable Classes. Recuperado de: http://www.laputan.org/drc/drc.html

D. Cheney. (2013, Junio 30). How to write benchmarks in go. Recuperado de: http://dave.cheney.net/2013/06/30/how-to-write-benchmarks-in-go

E. Razina and D. Janzen. (2007, Noviembre 19-21). Effects of dependency injection on maintainability. Recuperado de: http://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1035&context=csse_fac