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

Introduction

According to the description given on the official1 website, Go is an open source language that makes it easier to build simple, reliable and efficient code. This language has been relatively little time in the market, so it still lacks of lots of frameworks and libraries that other languages have. For this reason, this project deals with the creation of a library for dependency injection in Go contributing to fill the niche of the libraries for dependency

injection, allowing to create software projects with a higher level of flexibility, i.e., better prepared to be modified.

There are several methods to achieve flexible and decoupled code. An example is the design pattern known as strategy, which defines a set of encapsulated algorithms that can be changed to obtain a specific behavior (Erich Gamma et al, 1998).

Another example is the Open-Close principle (Meyer, 1997), which mentions that software entities (classes, modules, functions, etc.) should be open for extensibility, but closed for modification.

Both examples have something in common: usually the code is modified to depend on abstractions rather than specific implementations, increasing the flexibility of the code. This is the same principle which enables dependency injection.

The objective of the use of dependency injection in a project aims to increase the maintainability, which is very difficult to measure since it depends on several factors, some of which are very subjective.

There are some studies like the one done by Ekaterina Razina and David Janzen (2007) which mentions that the maintenance of the software consumes around 70% of its life cycle. Also mentions that some studies (Arisholm, 2002 and Rajaraman, 1992, et al.) exhibit that small uncoupled modules with high cohesion involve an improvement in maintainability.

Cohesion and coupling: In this same study two factors based on the research of L. Briand, J. Daly, and J. Wust (1999) were measured.

The coupling is defined as the degree of interdependence between the parts of a design (Chidamber, 1994, et al.). For the two measures were used:

- Coupling between objects (CBO). It refers to the number of classes to which a particular class is coupled.

- Response for Class (RFC). It is a set of methods that can potentially be executed in response to a message received by an object class.

Cohesion is the degree of similarity of the methods (Chidamber, 1994, et al. And Briand, 1999, et al.). We can also say that it is the degree to which each of the module is associated with the other (Razina, 2007, et al.).

Unfortunately the study could not prove the hypothesis that dependency injection significantly reduces dependencies on software modules. However, during the same trend of lower coupling modules found in those with a higher percentage of injection units (greater than 10%), as shown in Figure 1.

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

Figure 1. Projects with lower coupling

Dependency Injection

This section explains in a more concise manner dependency injection, so that the reader has a better idea of the concept.

Dependency Injection is a set of software design principles and patterns that allow us to develop loosely coupled code (Seemann, 2011).

Its advantages are:

Extensibility: This is the property that makes it easy to add new functionality to the code. It allows updating properly isolated parts rather than modify small parts throughout the code.

Late bonds: The ability to choose which components to use at runtime rather than at compile time. This can only be achieved if the code is loosely coupled; our code interacts only with abstractions rather than specific types. This allows us to change components without having to modify our code.

Parallel development: If our code is loosely coupled, it is much easier for different teams working on the same project. We can have a team working in the business layer, one in the service and, because the layers are independent, the teams work in source code that does not directly affect the other.

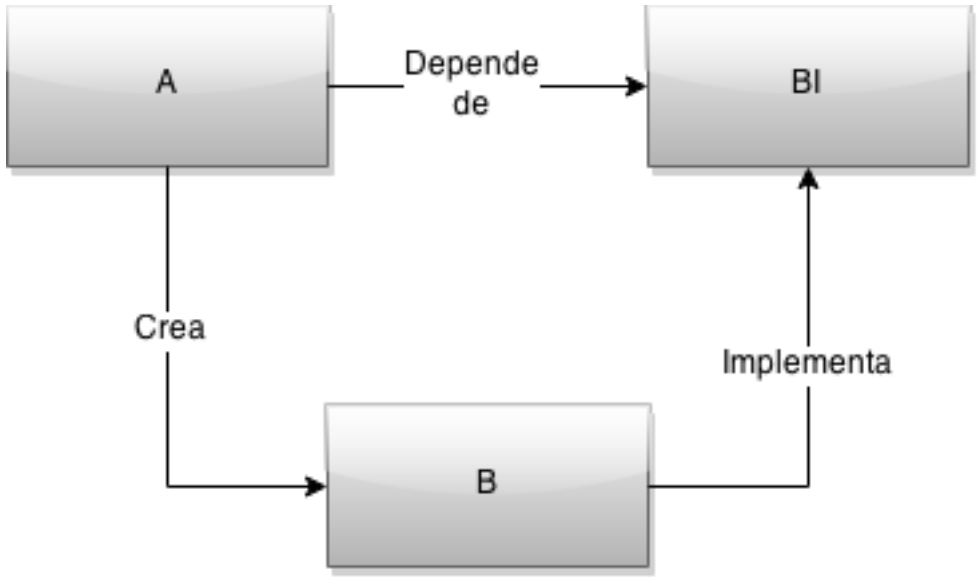
Ease of maintenance: When our components are independent, the functionality is isolated. This means that if you need to look for errors in the code or set some functionality, we know exactly where to look.

Easy to test: Unit testing is a very important issue. Its purpose is to test small pieces of code in isolation. When we loosely coupled code, we can easily use double or false test to easily isolate parts of the code that we want to test dependencies.

There are three types of Dependency Injection:

- **Injection Interface**: This is to define an interface, which is used for injection. This interface should be implemented by this class or want to get the references defined in the interface.
- **Injection builder**: Use a constructor to decide how to inject the necessary dependencies. For example, in the Java programming language, these units are passed to make use of the new keyword followed by the name of the class, passing as a parameter dependencies.
- **Injection setter**: In this common convention to define the method used to inject a dependency naming it with the word Set, followed by the name of the unit is used.

As a practical example, suppose we have an A, which has a dependency type IB, which is such an interface .As can be seen in Figure 2, B can create a structure that implements the BI interface and add it to the structure A. Finally imagine that implementation B is created internally by A



.

Figure 2. Example of dependency

A creates directly to B, which causes a strong coupling between the two structures and, therefore, less flexibility in the code. One answer to this problem is to follow the principle of inversion of control.

Inversion of Control

This is a programming method where the execution flow is reversed, making Unlike conventional programming methods in which the interactions are done through explicit calls to procedures or functions. In the case of investment control, only desired responses specified defined events and allow some external entity to take control of the execution.

Inversion of control is the principle underlying dependency injection because it is the container who injects dependencies when you create the objects rather than the latter who control the instantiation or location of such units.

Therefore, it is said that this method is an implementation of the principle of Hollywood ("do not call us, we will call"), commonly used by some frameworks like Spring Framework Java.

An important feature of a framework is that the methods defined by the user to modify the framework commonly be called within this and not from the user code. The framework often plays the role of principal to coordinate and sequence the application activity program. This inversion of control gives frameworks the power to serve as an extensible framework. The methods provided by the user modify the generic algorithms defined in the framework for a particular operation (Cheney, 2013).

Application Dependency Injection

After defining the concept of investment units, we return to the example above. We have the structure A is currently responsible for creating the instance of the implementation B. Following the principle of Inversion of Control, can delegate instance creation to an external element A, which will allow to have no idea details about the specific implementation of BI with which you are dealing. Thus, an therefore more flexible and loosely coupled code is obtained.

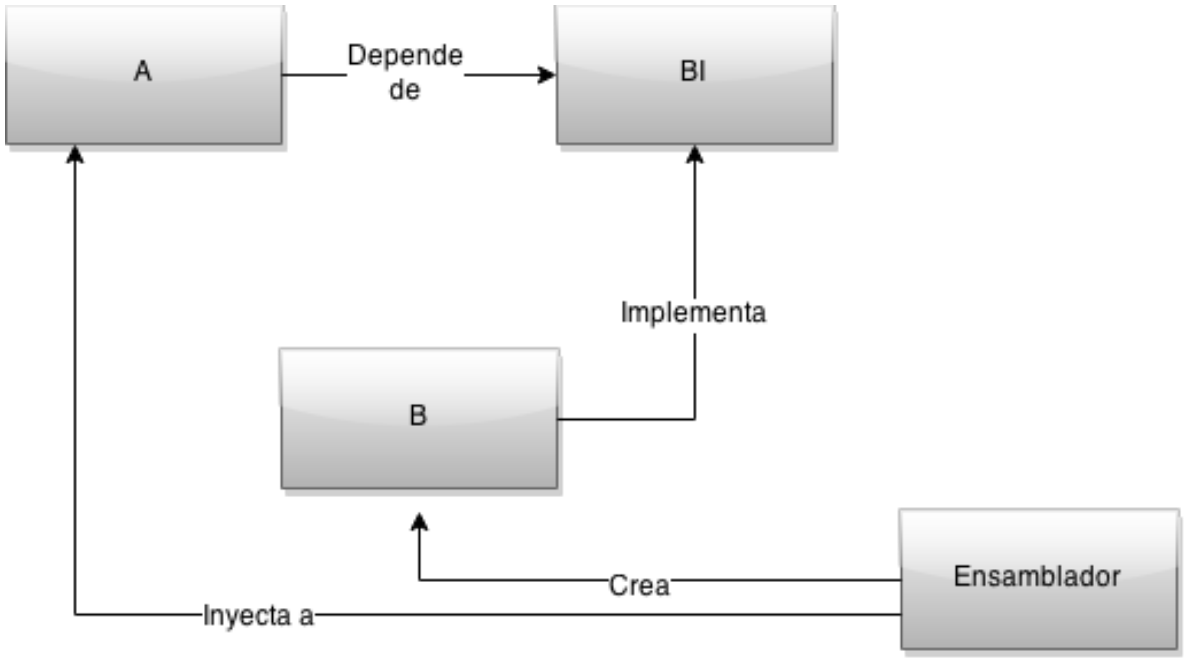


Figure 3. Application Dependency Injection

As we can see in Figure 3, is the assembler object that instantiates injected B and A, investing control over the creation of that agency and helping the decoupling of both structures.

Bookseller design

It is creating a library for dependency injection to be able to read the settings from a file in JSON format and create dependencies and inject the corresponding elements to form the dependency tree was proposed. See Figure 4.

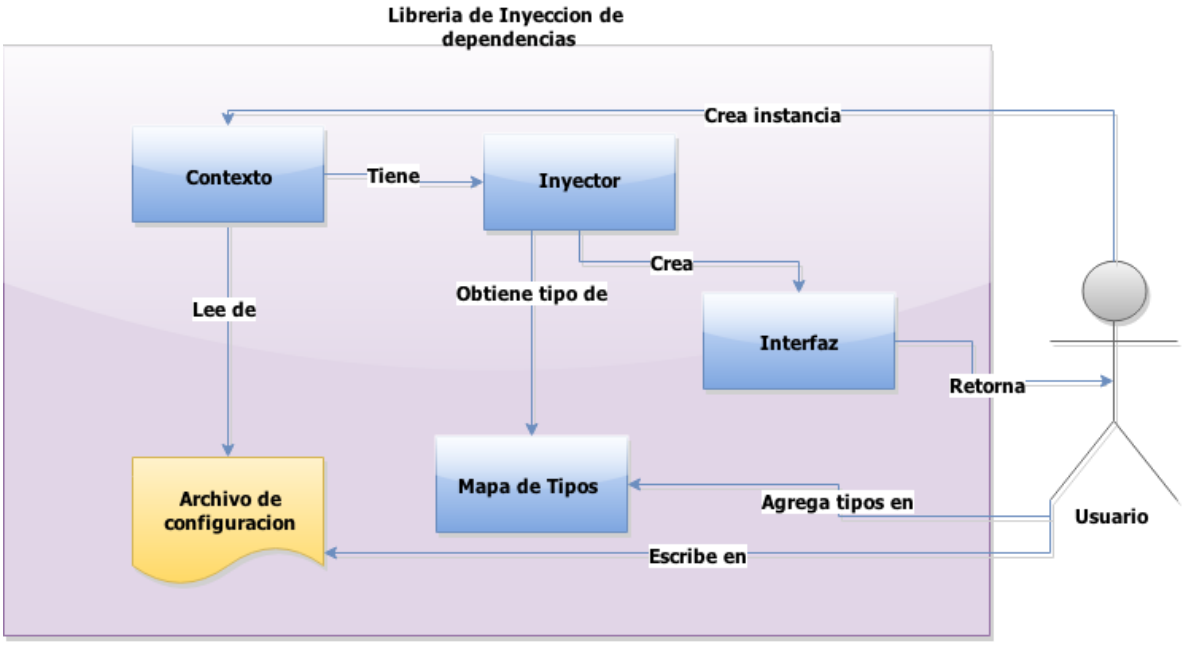


Figure 4. Conceptual Model

Components

Context. It allows us to inject dependencies from a given configuration. This allows us to change the behavior of the system.

Configuration file. Here we define the types of data that the injector and its dependencies. This contains the data in JSON format.

Injector. This is the solver dependency tree; It contains a cache to store singleton dependencies.

Type map. We used to record the types of data that are to handle. The Go language does not create instances only from the name of a structure, which is required to record the types of data.

Resulting interface. It is the generated object, which already contains the offices specified in the configuration file. It requires a statement of type (type assertion).

The class diagram of the library developed consists of six classes that correspond to the specified modules in the conceptual model. See Figure 5.

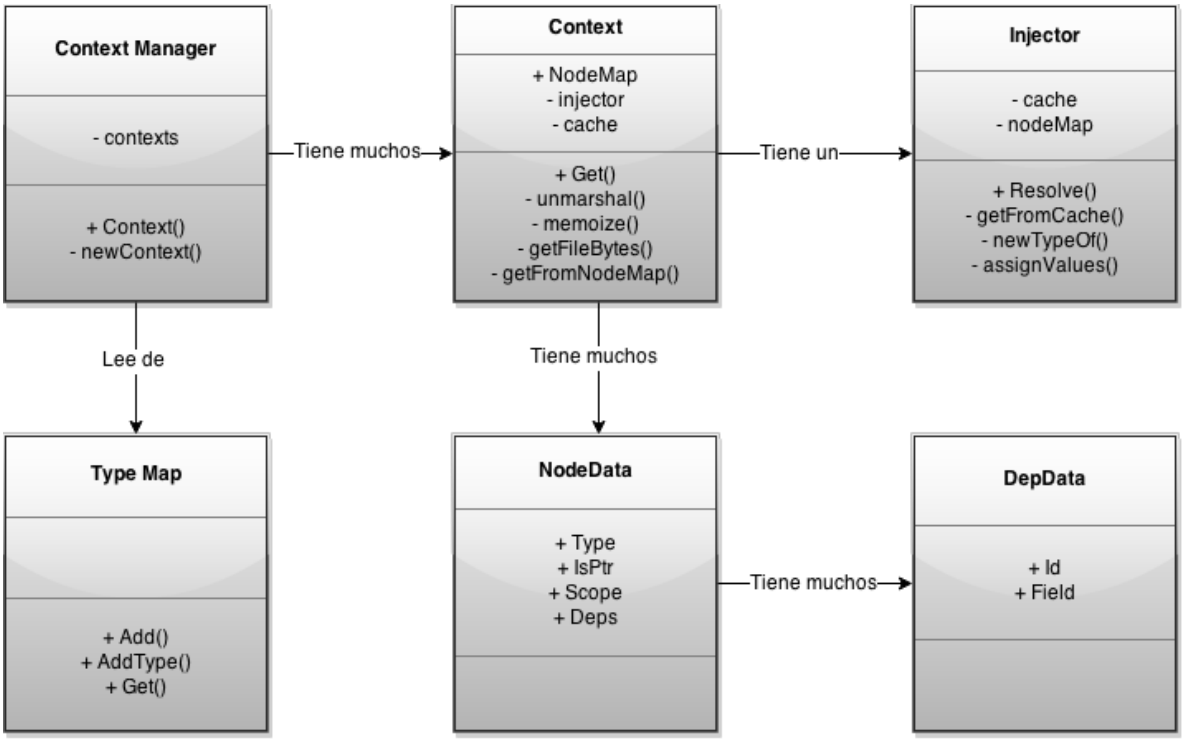


Figure 5. Class diagram of the library developed.

Creating Bookseller

In the initial stage injector module it was built structures representing the configuration of the units and unit tests passing the necessary parameters to the injector and verified the resulting object is constructed.

The construction of this first step possible to verify if the premises were injected the resulting object correct way.

Context module then added and also to verify the performance unit together with the injector and with certain configurations tests were added.

After the module Handler Contexts was added along with a configuration file and the corresponding unit tests were added to verify if the contexts were handled correctly and if it was possible to process settings file and convert it to the appropriate structures.

By the nature of a database library is not required. Yet the model code for the representation of the data contained in the configuration file was designed.

Such a configuration file in JSON format and consists of several parts:

1. The parent node whose name key "nodes". This is the node that includes all configurations.
2. The nodes of object information, which is key to an alias with which the object is represented. The latter has three properties:
3. The first is the type of the object, denoted by the location of the structure.
4. The second is a flag that indicates whether the object is a pointer.
5. The third is an arrangement in which the dependencies for a given object to which the alias that was given to dependence and the name of the field in which injected specified specified.

The configuration code shown in Figure 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"
        }
      ]
    }
  }
}
```

Figure 6. Example of configuration file

Much of the efficiency of a user to interact with a software depends on its interaction model is simple, intuitive and avoid more errors. Overall, a correct model of interaction design aims that the user is satisfied to operate software.

In this phase we were evaluated as the best programmer user could interact with the library intuitive and unobtrusive way. You only have to import the library and use the Context method to create a new context. Once an object of type Context taking may use the Get method on him, for the object and its dependencies.

The interface is very simple and that much of the complexity is performed in the configuration file.

Testing

Go the testing package, which allows us to know the speed of execution of a process to write stress test known as benchmarks used to measure the speed of execution.

As Dave Cheney mentioned, write benchmarks is an excellent way to communicate a performance improvement or regression in a reproducible manner (Cheney, 2013).

Various tests were created to verify that the library meets the purpose for which it was designed. Access to a database, using layered architectures and use of global objects: Because it is a library, several fictitious scenarios for the same test, such as is created. It was also verified that the validations for the configuration file functioned properly.

Integration testing is performed using continuous integration tool called Travis, for which a new project was added and Go specified that the language would be used.

By specifying the language, Travis automatically creates a virtual machine cloned the project from the handler system versions, which was Github in this case, and creates the necessary environment variables, such as GOROOT and GOPATH.

Finally, Travis runs all unit tests that are in the project through the "go test ./... -v" command. In this command the words "go test" means that the project is compiled code and then test files with ending "_test.go" are run. The "-v" flag means an extended description of each test is printed. And finally "./..." it means the tests are searched iteratively throughout the project.

Results

This investigation was limited only to the development of the library and test it in their proper operation. It was not included as part of this research the extent to which help a project to be more maintainable software because it is very difficult to measure this.

Result in a library that allows user to use the principle of dependency injection to create more flexible code was obtained.

The library was tested in a test project previously prepared and the code was compared before and after using the library. The first factor was the flexibility shown in the code. Consider the code shown in Figure 7.

```
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)
}
```

Figure 7. Code initial test

We can see that the structure has two units Controller: Model and Writer, which are interfaces. When we created the controller with the function NewController we create specific implementations of those interfaces.

If at any time you need to change one of the implementations on the other, for example, change the Writer by an implementation that produces a string in JSON format with the product information call JsonWriter, we would have to change the NewController function to add such implementation as It is shown in Figure 8.

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

Figure 8. Change implementation

This creates a link between the function and implementation NewController Writer we want to use. However, we can achieve a decoupling of these elements using the library, as shown in Figure 9.

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

Figure 9. Refactoring with the library built

You can see several advantages to the use of the library:

• The decoupling of the code as the details of specific implementations will be handled by the library using the configuration file.

• Reducing the code to create the controller.

• The ease of switching units without recompiling the code again.

The resulting library source code is hosted on Github repository under an MIT-style license and is available at the following URL: https://github.com/cone/digo.

Conclusion

The purpose of this research is to develop a library that allows dependency injection in the language Go open source. To achieve this we proceeded to do some research about the necessary concepts, such as Inversion of Control (IoC) and the concept of dependency injection DI.

As mentioned above, it is very difficult to measure how dependency injection aid for a project to be more flexible to change and therefore more maintainable. For example, you can use dependency injection but does not guarantee that the rest of the code is uncoupled, or even objects that are being injected they are.

This is outside the scope of this project, whose purpose is only the creation of this library and observed the effectiveness of it. Therefore, the degree is observed that this library helps create more maintainable code, if it helps to reduce lines of code or how much help reduce the development time of a project.

As a user of frameworks that allow dependency injection in my view dependency injection helps create more flexible code and, in the case of providing the option to use an external configuration file, allow you to create code that is easier to try to be able to change the behavior of the system without having to recompile the source code.

Bibliography

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