Interfaces & Make

Esp. Ing. José María Sola, profesor.

Revisión 2.1.0
Abr 2017

Tabla de contenidos

1. Introducción 1
2. Abstracciones e Interfaces 3
3. Interfaces en el Lenguaje C y Derivados 5
4. Make 9
5. Lectura Adicional 13

1

Introducción

Este documento presenta los siguientes conceptos y técnicas fundamentales de la programación en general y del Lenguaje C y sus derivados.

- Construcción de abstracciones.
- Dependencia del cliente con respecto a una interfaz, no a una implementación.
- Archivos encabezados como interfaz y guardas de inclusión.
- Proceso de compilación y compilación separada.
- Automatización de construcción mediante `make`.

2

Abstracciones e Interfaces

Un **componente** es una unidad que **provee servicios** a otros componentes, el mecanismo que **implementa** ese servicio es **abstraído** de los componentes mediante una **interfaz pública**.

De esa forma, el componente implementa una abstracción, la cual es provista mediante una interfaz.

La interfaz establece el **contrato** de comunicación, que establece las responsabilidades del **componente proveedor** y del **componente consumidor**.

Al diseñar la interfaz de la abstracción buscamos que nuestros consumidores cumplan el siguiente objetivo:

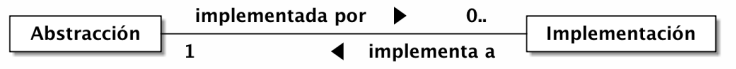
> **Depender de la abstracción, no de la implementación.**

Para ello la interfaz del componente no debe exponer detalles de implementación, lo cual permite que cambios en el componente no afecten a los consumidores.

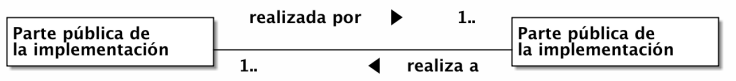
La relación entre el Cliente y la Interfaz puede describirse como que **el cliente Importa la interfaz** o también como que **el cliente depende de la interfaz**.

Asimismo, la relación entre la Implementanción y la Interfaz puede describirse como que **el proveedor exporta la interfaz** o también como que **el proveedor implementa la interfaz**.

El objetivo final es **construir abstracciones para resolver problemas**. Una abstracción puede implementarse en diferentes lenguajes de programación y de diferentes formas, pero cada implementación siempre tiene una *parte pública* o *interfaz* y una *parte privada*.



El diseño de la implementación debe permitir cambios en su parte privada, sin requerir cambios en su parte pública.



3

Interfaces en el Lenguaje C y Derivados

En el leguaje C, y sus derivados, las interfaces se definen en archivos **header** (encabezado), con extensión `.h`, y los consumidores y proveedores en archivos `.c`.

Otras tecnologías aplican los conceptos de forma similar con otros nombres, por ejemplo C# y Java usan `interface` y `class`, y Smalltalk usa `protocol` y clases.

Tanto la relación **importa** como la relación **exporta** en C se realiza con la ayuda de la directiva `#include` del preprocesador.

Si cumplimos la regla que

> **tanto el consumidor como el proveedor deben incluir `Interfaz.h`.**

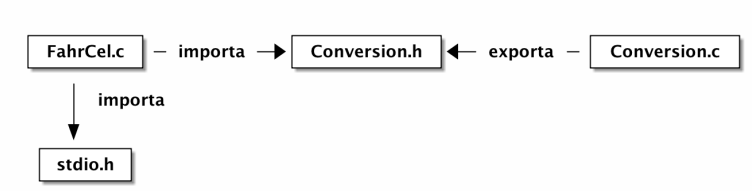
podemos basarnos en el **compilador** para forzar que ambas partes cumplan el contrato. Al ser incluido `Interfaz.h` por ambas partes, el compilador puede detectar los siguientes tipos de errores:

- **Invocación** incorrecta por parte del consumidor.
- **Definción** incorrecta por parte del proveedor.

Como ejemplo, supongamos el caso del programa de conversión de temperaturas de la sección 1.2 y el ejercicio 1-15 de [K&R1988].

Un programa que imprime una tabla de conversión de temperaturas de fahrenheit a celsius depende de un componente que provea el servicio de conversión de forma tal que lo abstraiga de la expresión que implementa la fórmula.

La **abstracción** se logra mediante la función de conversión `GetCelsFromFahr`; la cual se declara en la **interfaz** `Conversion.h` y se implementa en el **proveedor** `Conversion.c`. El programa que imprime la tabla es `FahrCel.c`, el cual también depende de un mecanismo para enviar datos a la salida estándar, por eso, `FahrCel.c` depende de `Conversion.h` y de `stdio.h`.



El comando para construir el programa es

```
$ cc FahrCel.c Conversion.c -o FahrCel
```

El contenido de los tres archivos está a continuación:

FahrCel.c.

```
/* K&R
 * Exercise 1.15. Rewrite the temperature conversion program
 * of Section 1.2 to use a function for conversion.
 * JMS
 * 2016
 */

#include <stdio.h>
#include "Conversion.h"
```

```
int main(void){
  const int LOWER = 0;   // lower limit of table
  const int UPPER = 300; // upper limit
  const int STEP  = 20;  // step size

  for(int fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
    printf("%3d %6.1f\n", fahr, GetCelsFromFahr(fahr) );
}
```

Conversion.h.

```
/* K&R
 * Exercise 1.15. Rewrite the temperature conversion program
 * of Section 1.2 to use a function for conversion.
 * JMS
 * 2016
 */

#ifndef CONVERSION_H_INCLUDED
#define CONVERSION_H_INCLUDED

double GetCelsFromFahr(double);

#endif
```

Conversion.c.

```
/* K&R
 * Exercise 1.15. Rewrite the temperature conversion program
 * of Section 1.2 to use a function for conversion.
 * JMS
 * 2016
 */

#include "Conversion.h"

double GetCelsFromFahr(double f){
    return (5.0/9.0)*(f-32);
}
```

4

Make

Compilar un proyecto resulta complicado si el proyecto está compuesto por varios archivos y para compilar se requiere escribir comandos extensos.

Los `makefiles` junto con la utilidad `make` proponen una solución.

Un `makefile` es una notación declarativa que define las dependencias y comandos pra consutruir uno o más productos. Si la definición se encuentra en la carpeta actual, la simple invocación a `make` construye los productos.

Para simplificar el proceso la buena práctica es contener los archivos fuente en y en el `makefile` en una misma carpeta.

La utilidad `make` lee las dependencias declararas en el `makefile` y determina que componentes de las solución fueron actualizados desde la última vez que se construyó el producto, `make` reconstruye solo las componentes que fueron actualizadas y reconstruye el producto.

Del punto de vista más fundamental, un `makefile`, es una secuencia de reglas. Cada regla tiene la siguiente sintaxis:

```
target: prerequisites
[tab]steps
```

Lo semántica de la regla es: Ante la actualización de alguno de los prerequisitos, reconstruir el objetivo según los pasos indicados.

Por ejemplo, para el reconocido "Hello, World", el make file es el siguiente:

```
hello: hello.o
  cc hello.o -o hello

hello.o: hello.c
  cc -c hello.c -o hello.o
```

Si desde la línea de comando se escribe `make`, se construirá el programa ejecutable `hello`.

Por defecto, `make` busca un la especificación de construcción un archivo llamado `makefile`. Si se necesita llamarlo de otra manera o se necesita tener más de una especificación, `make` acepta la opción `-f`.

```
make -f othermakefile
```

A continuación presento un ejemplo simple de make de `makefile` para el famoso `hello.c` con un solo archivo fuente, y otro para el programa conversor te temperatura, `FahrCel` que se compone por tres archivos.

makefile para `hello`.

```
# Makes Hello.exe
# JMS
# 2016

BIN      = hello.exe
OBJ      = hello.o
CC       = gcc
CFLAGS   = -std=c11 -Wall -pedantic-errors -m32 -D __DEBUG__ -g3 $(INCS)
# LDFLAGS  = -static-libgcc
INCS     = -I"C:/Program Files/Dev-Cpp/MinGW64/x86_64-w64-mingw32/
include"
LDLIBS   = -L"C:/Program Files/Dev-Cpp/MinGW64/x86_64-w64-mingw32/lib32"
RM       = rm -f

$(BIN): $(OBJ)
 $(CC) $(OBJ) -o $(BIN) $(CFLAGS) $(LDFLAGS) $(LDLIBS)

hello.o: hello.c
 $(CC) -c hello.c -o hello.o $(CFLAGS)

.PHONY: clean
```

```
clean:
 $(RM) $(OBJ) $(BIN)
```

makefile para `FahrCel`**.**

```
# Makes FahrCel.exe
# JMS
# 2016
# K&R Exercise 1.15. Rewrite the temperature conversion program
# of Section 1.2 to use a function for conversion.

BIN      = FahrCel.exe
OBJ      = FahrCel.o Conversion.o
CC       = gcc
CFLAGS   = -std=c11 -Wall -pedantic-errors -m32 -D __DEBUG__ -g3 $(INCS)
# LDFLAGS  = -static-libgcc
INCS     = -I"C:/Program Files/Dev-Cpp/MinGW64/x86_64-w64-mingw32/
include"
LDLIBS   = -L"C:/Program Files/Dev-Cpp/MinGW64/x86_64-w64-mingw32/lib32"
RM       = rm -f

$(BIN): $(OBJ)
 $(CC) $(OBJ) -o $(BIN) $(CFLAGS) $(LDFLAGS) $(LDLIBS)

FahrCel.o: FahrCel.c Conversion.h
 $(CC) -c FahrCel.c -o FahrCel.o $(CFLAGS)

Conversion.o: Conversion.h Conversion.c
 $(CC) -c Conversion.c -o Conversion.o $(CFLAGS)

.PHONY: clean
clean:
 $(RM) $(OBJ) $(BIN)
```

5

Lectura Adicional

La utilidad `make` y el compilador `gcc` tiene decenas de funcionalidades, esta es solo alguna de las referencias para profundizarlas.

- Mrbook's stuff
- cs.colby.edu maketutor
- K&R1988
- https://gcc.gnu.org/onlinedocs/gcc-6.1.0/gcc.pdf
- https://www.gnu.org/software/make/manual/make.pdf
- https://github.com/mbcrawfo/GenericMakefile
- https://github.com/jimenezrick/magic-makefile
- http://www.cs.toronto.edu/~penny/teaching/csc444-05f/maketutorial.html (Make y pruebas automatizadas).-