6.4 Patrones de diseño GoF

Uno de los hitos más importantes en el diseño orientado a objetos fue la publicación del libro "*Design Patterns*" por Gamma, Helm, Johnson y Vlissides en 1995; conocidos como *"Gang of Four", GoF*. En este libro se muestran 23 patrones ampliamente utilizados. En este apartado se tratarán algunos de ellos.

6.4.1 Adaptador

Problema: ¿Cómo resolver interfaces incompatibles, o proporcionar una interfaz estable para componentes parecidos con diferentes interfaces?

Solución: Convierta la interfaz original de una componente en otra, mediante un objeto adaptador intermedio.

El propósito de este patrón es convertir la interfaz de una clase en otra interfaz que es la que esperan los clientes. Este patrón permite que cooperen clases que de otra forma no podrían colaborar por no tener compatibilidad entre ellas.

Debería usarse el patrón Adaptador cuando:

- Se quiere utilizar una clase existente y su interfaz no concuerda con lo que se espera.

- Se quiere crear una clase reutilizable que coopere con clases no relacionadas o que no han sido previstas, i.e. clases que no tienen por qué tener interfaces compatibles.

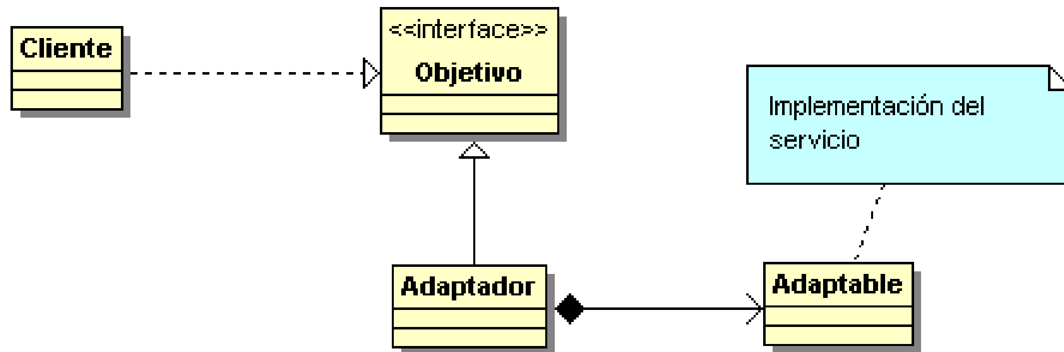
Los participantes en este patrón realizan los siguientes roles:

- Objetivo: define los servicios del dominio que usa el cliente. Representa un interfaz estable con los servicios tal cual espera el cliente.

- Cliente: utiliza los servicios del paquete a través del interfaz Objetivo.

- Adaptable: Implementa los servicios del paquete.

- Adaptador: adapta la interfaz de Adaptable a la interfaz Objetivo.



Los clientes llaman a operaciones a través de la interfase estable. A su vez el adaptador llama a operaciones de Adaptable que son las que satisfacen las peticiones.

En una implementación en C++ de un adaptador de clases, Adaptador debería heredar públicamente de objetivo y tener como atributo privado a un objeto de la clase Adaptable.

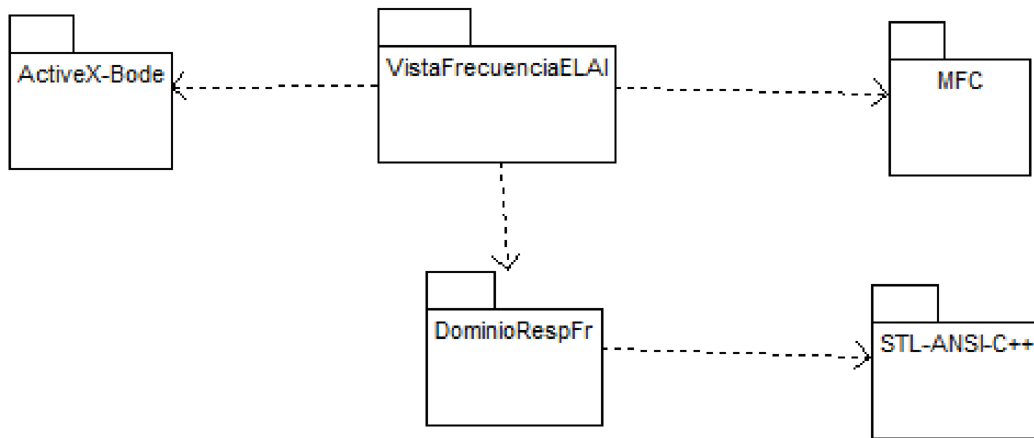
Nótese que los nombres de los tipos incluyen el nombre de patrón "Adaptador".

La aplicación del Adaptador es una especialización de Variaciones Protegidas, Indirección y Polimorfismo. En GRASP es el polimorfismo, aquí es una especialización que se llama Adaptador. Se puede analizar muchos patrones más complejos y especializados en función de la familia GRASP. Existen muchos publicados y es el alfabeto del DOO.

Ejemplo 6.13

La aplicación de Respuesta en Frecuencia no depende sólo del algoritmo de calcular el módulo y argumento de un filtro, sino también de su visualización en un diagrama de Bode. Realizar un diseño para el paquete de representación gráfica.

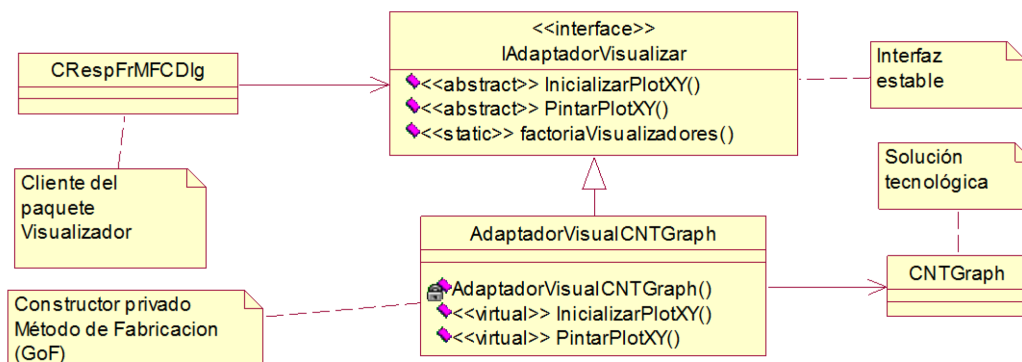
Tal cual se presentó en la vista de gestión, la aplicación tenía un paquete para la visualización del diagrama de Bode. Se había elegido una solución basada en software prefabricado (ActiveX).



Entre las posibles soluciones tecnológicas actuales se ha elegido NTGraph[4]. En un diseño robusto, esta inserción supone un punto caliente. Por varios motivos:

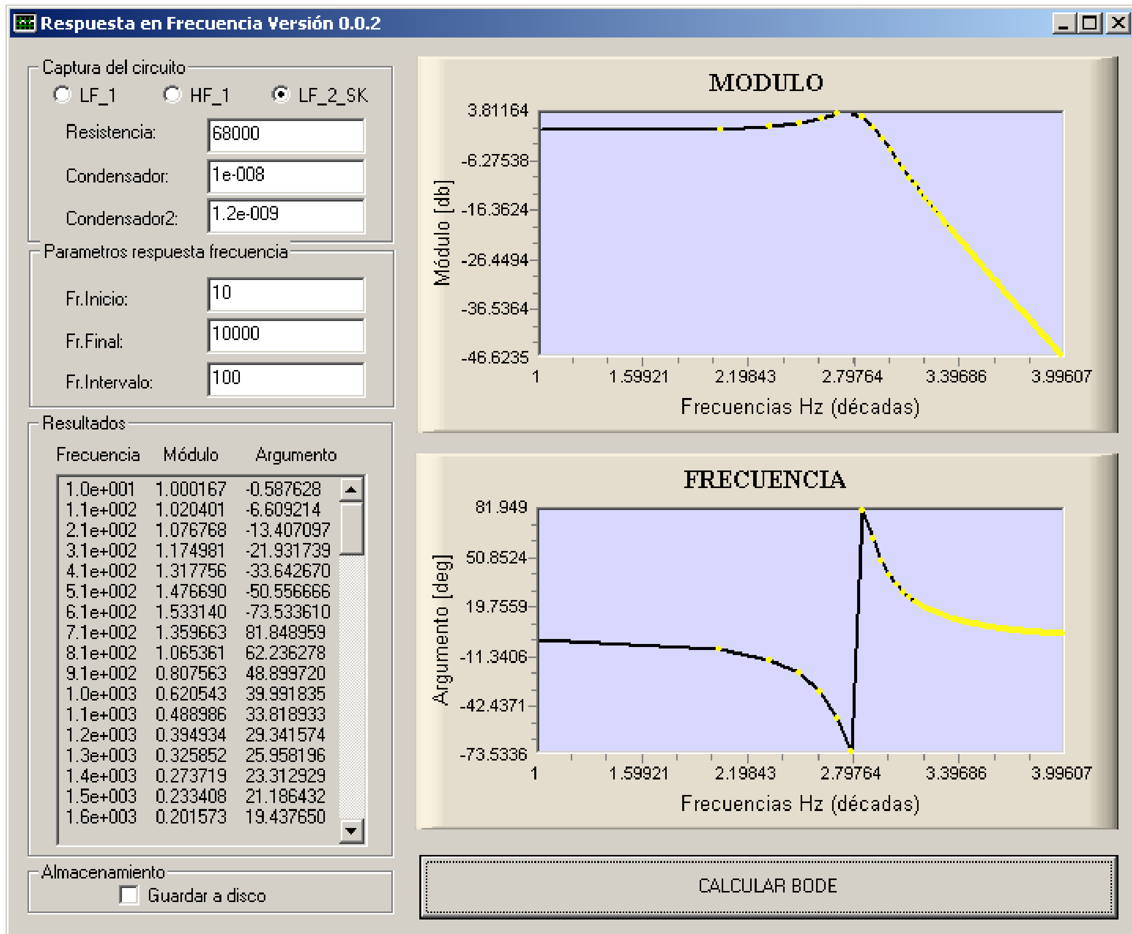
- En el futuro se puede cambiar de componente, i.e. salto a otra nueva tecnología.
- Ampliación de nuevos servicios en la representación del diagrama de Bode.

La aplicación de Variaciones Protegidas supone, de momento, el uso de patrones GoF: Adaptador y Factoría; mientras en GRASP implica Indirección, Polimorfismo y Fabricación Pura.



[4] http://www.codeproject.com/miscctrl/ntgraph_activex.asp

```
// De: "Apuntes de Sistemas Informáticos Industrial" Carlos Platero.
// Ver permisos en licencia de GPL

#include "../../ntgraph.h"

//Tipos de visualizadores
enum PlataformaVisual{NTGRAPH} ;

class IAdaptadorVisualizar
{
public:
	virtual void InicializarPlotXY(void) = 0;
	virtual void PintarPlotXY(float,float,float, double *)= 0;
	//Factoria de Visualizadores
	static IAdaptadorVisualizar *factoriaVisualizadores(enum PlataformaVisual,
		CNTGraph *p1 = NULL);

};


class AdaptadorVisualCNTGraph : public IAdaptadorVisualizar
{
	CNTGraph *graph;
	AdaptadorVisualCNTGraph(CNTGraph *gr): graph(gr){}
	friend class IAdaptadorVisualizar;
public:
	virtual void InicializarPlotXY(void);
	virtual void PintarPlotXY(float,float,float, double *);
};
```



Ejemplo 6.14

Las series de Fibonacci se hicieron famosas en la Edad Media por que planteó el problema de procreación de los conejos de manera formal. Estudió que si se partía de una pareja de conejos cómo éstos se multiplicaban con el tiempo. La solución se encuentra en la Serie de Fibonacci. Ésta se construye con la suma de los dos últimos valores. Los dos primeros términos de la serie son el 1 y el 1. Los demás se obtienen con la regla mencionada, la suma de los valores anteriores:

$$F_n = F_{n-1} + F_{n-2}$$

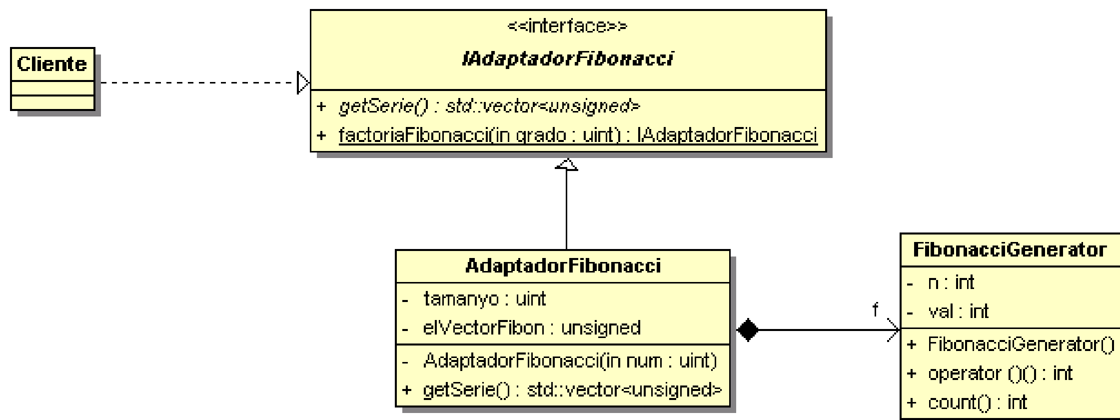
$$\{1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad \ldots\}$$

Un generador de la serie de Fibonacci ha sido tomado de Bruce Eckel y Check Allison[5]. Adaptarlo para emplear los algoritmos dados por las STL.

```
// From "Thinking in C++, Volume 2", by Bruce Eckel & Chuck Allison.
// (c) 1995-2004 MindView, Inc. All Rights Reserved.
// See source code use permissions stated in the file 'License.txt',
// distributed with the code package available at www.MindView.net.
#ifndef FIBONACCIGENERATOR_H
#define FIBONACCIGENERATOR_H

class FibonacciGenerator {
  int n;
  int val[2];
public:
  FibonacciGenerator() : n(0) { val[0] = val[1] = 1; }
  int operator()() {
    int result = n > 2 ? val[0] + val[1] : 1;
    ++n;
    val[0] = val[1];
    val[1] = result;
    return result;
  }
  int count() { return n; }
};

#endif // FIBONACCIGENERATOR_H ///:~
```

Se trata de diseñar un adaptador que permita utilizar los algoritmos dados por las STL como *for_each( )* o *accumulate()*. Estos servicios requieren que la información esté preparada como un tipo vector de las STL, *std::vector<>*. Para tal fin, se emplea el patrón Adaptador:

[5] *"Thinking in C++, Volume 2"*, by Bruce Eckel & Chuck Allison. (c) 1995-2004 MindView, Inc.



Se emplea el siguiente código de test:

```
// De: "Apuntes de Informática Industrial" Carlos Platero.
// (R) 2005 Con licencia GPL.
// Ver permisos en licencia de GPL
//
//Ejemplo del patrón GoF de adaptador
//Desde un generador de números de Fibonnaci, dado en FibonacciGenerator.h,
//Adaptación para emplear las STL
//Código de prueba (XP)

#include <iostream>
#include "AdaptadorFibonacci.h"
#include <numeric>
#include <algorithm>

IAdaptadorFibonacci* IAdaptadorFibonacci::factoriaFibonacci(unsigned grado)
{
        return (new AdaptadorFibonacci(grado));
}
using namespace std;

void imprimir(unsigned);

int main()
{
  const unsigned numeroFibo = 20;
  IAdaptadorFibonacci *pAdaptadorFibo =
        IAdaptadorFibonacci::factoriaFibonacci(numeroFibo);

  cout << "Tabla de Fibonacci" <<endl;
  cout << "------------------" <<endl;

  for_each(pAdaptadorFibo->getSerie().begin(),pAdaptadorFibo->getSerie().end(),
        imprimir);
  //Ver GRASP "No hable con extraños" y el código maduro

  cout << "Valor acumulado total: "
    << accumulate(pAdaptadorFibo->getSerie().begin(),
             pAdaptadorFibo->getSerie().end(), 0) << endl;
  return 0;

}
```

Las clases de los roles objetivo y adaptador quedarán definidas como:

```
// De: "Apuntes de Sistemas Informáticos Industriales" Carlos Platero.
// Ver permisos en licencia de GPL
//
// Ejemplo de adaptador GoF entre el generador de Fibonacci
// y las librerías STL
#ifndef _ADAPTADOR_FIBO_INC_
#define _ADAPTADOR_FIBO_INC_

#include <vector>
#include "FibonacciGenerator.h"

class IAdaptadorFibonacci
{
public:
        virtual std::vector<unsigned> & getSerie() = 0;
        static IAdaptadorFibonacci *factoriaFibonacci(unsigned grado);
};

class AdaptadorFibonacci: public IAdaptadorFibonacci
{
        FibonacciGenerator f;
        unsigned tamanyo;
        std::vector<unsigned> elVectorFibon;
        friend class IAdaptadorFibonacci;
        AdaptadorFibonacci(unsigned num): tamanyo(num)
          {
          for (unsigned i=0;i<=tamanyo;i++)
                        elVectorFibon.push_back(f());
          }
public:

          virtual std::vector<unsigned> & getSerie()
          {return elVectorFibon;}
};

#endif
```

El cliente, ahora, podrá emplear al generador de la serie de Fibonacci utilizando las funciones algorítmicas de las STL.

6.4.2 Factoría

Problema: ¿Quién debe ser responsable de la creación de los objetos cuando existen consideraciones especiales, como una lógica de creación compleja, el deseo de separar las responsabilidades de la creación para manejar la cohesión, etc.?

Solución: Crear un objeto de Fabricación Pura denominado Factoría que maneje la creación.

Cuando aparece una nueva variación del tipo de los datos se aplica el patrón Polimorfismo [GRASP], tal cual se comentó en el apartado 6.3.6. Al principio parece que sólo es necesario implementarlo en el punto de la aplicación que se introduce la variación. Sin embargo, mayoritariamente sucede que además se requiere un constructor para la nueva variación y su repercusión se extiende por todo el código. Para estos casos se debe aplicar Variaciones Protegidas. Por tanto, se considera la creación de un punto de variación o punto caliente y se coloca una interfase estable a través del Polimorfismo, la Indirección y el Adaptador. La construcción de estos objetos deben ser forzados a ser creados en una única Factoría.

Por ejemplo, cuando al aplicar Variaciones Protegidas y Adaptador sobre la representación en el diagrama de Bode de la aplicación Respuesta en Frecuencia, ¿quién crea el adaptador de NTGraph? ¿Y cómo determinar qué clase de adaptador crear?. Si los creará algún objeto del dominio, estas responsabilidades exceden de la pura lógica de la aplicación y entra en otras cuestiones relacionadas con la conexión con componentes de software externos.

Este punto subraya otro principio de diseño fundamental: mantener siempre una separación de intereses. La elección de un objeto del dominio para crear los adaptadores no está de acuerdo con los objetivos de separación de intereses. Además disminuye su cohesión. La lógica de qué clase Adaptador se instancia es resuelto en la Factoría, leyendo una fuente externa y después cargando la clase dinámicamente.

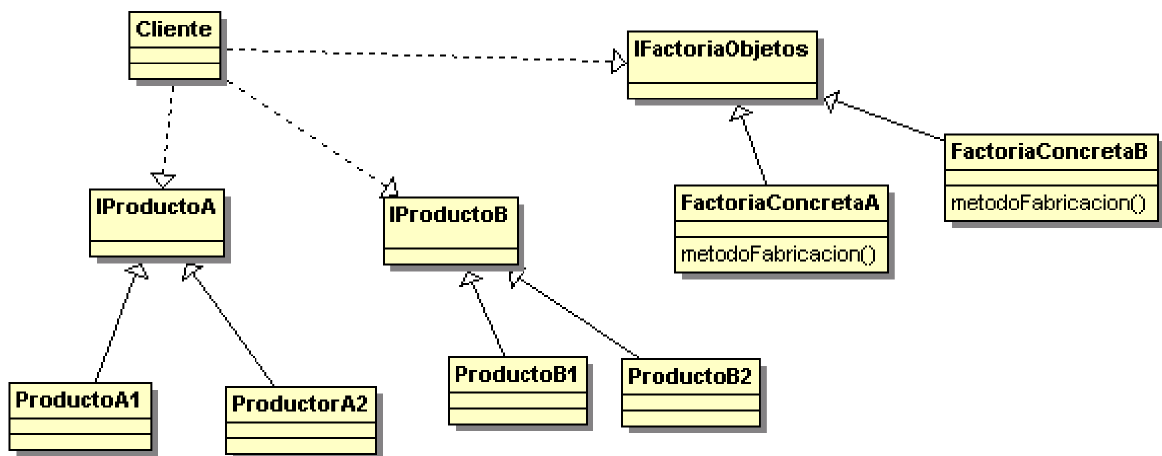
Una alternativa típica en este caso es aplicar el patrón Factoría. Los objetos Factoría tienen varias ventajas:

- Separación de responsabilidades en la creación compleja en objetos de apoyo cohesivo.
- Ocultan la lógica de creación potencialmente compleja.
- Permite introducir estrategias para mejorar el rendimiento de la gestión de la memoria, como objetos caché o de reciclaje.

El cliente sólo utilizará las interfases de sus paquetes servidores, dejando que sea la lógica externa quien decida sobre la implementación y la factoría quien crea los objetos que implementan los servicios. A este patrón GoF se le llama Factoría Abstacta. Se empleará cuando:

- un sistema debe ser independiente de cómo se crean, componen y representan sus productos.
- un sistema debe ser configurado como una familia de productos entre varias.
- quiere proporcionar una biblioteca de clases de productos y sólo quiere revelar sus interfaces, no sus implementaciones.

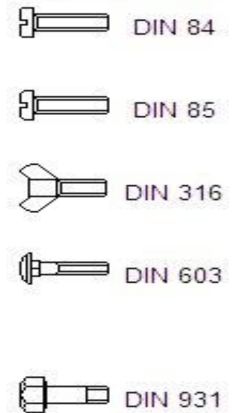
La estructura del patrón queda reflejada en el siguiente diagrama de clases:



Los roles desempeñados son:

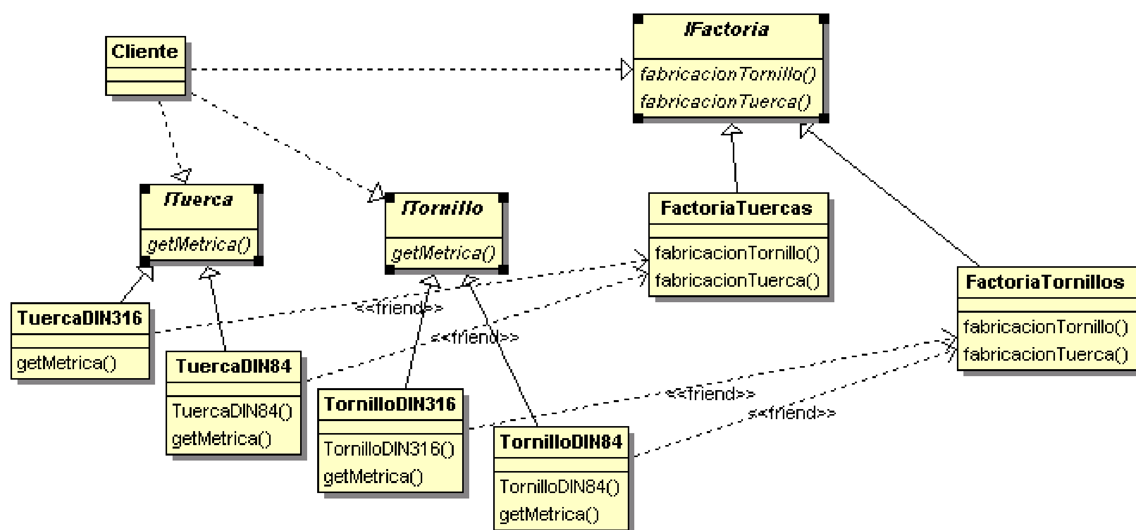
- Cliente: sólo usa interfaces declarados por las clases de Fabricación Abstracta y Productos Abstractos
- Producto Abstracto: declara una interfaz para un tipo de objeto (p.ej. IClaseA)
- Producto Concreto: define un objeto producto para que sea creado por la Factoría correspondiente. Implementa la interfaz de Producto Abstracto.
- Factoría Abstracta: declara una interfaz para operaciones que crean objetos productos abstractos
- Factoría Concreta: implementa las operaciones para crear objetos producto concretos

Ejemplo 6.15



Se pretende simular el evento de sacar de una bolsa un tornillo y una tuerca y saber si se pueden ensamblar. La bolsa puede contener tornillos y tuercas de diferentes métricas. Hágase para el caso concreto de elementos DIN84 y DIN316.

Se diseñará de manera que el cliente sólo utilice el concepto de tornillos y tuercas. La responsabilidad de saber si pertenece a la misma métrica quedará a los objetos concretos creados por las factorías abstractas. El diseño sería:



```cpp
typedef enum{DIN84, DIN316} metrica;
class FactoriaTuercas;
class ITuerca
{
public:
	virtual metrica getMetrica() = 0;

};
class TuercaDIN84 : public ITuerca
{
	metrica laMetrica;
	friend class FactoriaTuercas;
	TuercaDIN84() {laMetrica = DIN84;}
public:
	virtual metrica getMetrica() {return laMetrica;}
};
class TuercaDIN316 : public ITuerca
{
	metrica laMetrica;
	friend class FactoriaTuercas;
	TuercaDIN316() {laMetrica = DIN316;}
public:
	virtual metrica getMetrica() {return laMetrica;}

};
```

```cpp
class FactoriaTornillos;
class ITornillo
{
public:
	virtual metrica getMetrica() = 0;
};
class TornilloDIN84 : public ITornillo
{
	metrica laMetrica;
	friend class FactoriaTornillos;
	TornilloDIN84() {laMetrica = DIN84;}
public:
	virtual metrica getMetrica() {return laMetrica;}
};
class TornilloDIN316 : public ITornillo
{
	metrica laMetrica;
	friend class FactoriaTornillos;
	TornilloDIN316() {laMetrica = DIN316;}

public:
	virtual metrica getMetrica() {return laMetrica;}

};
```

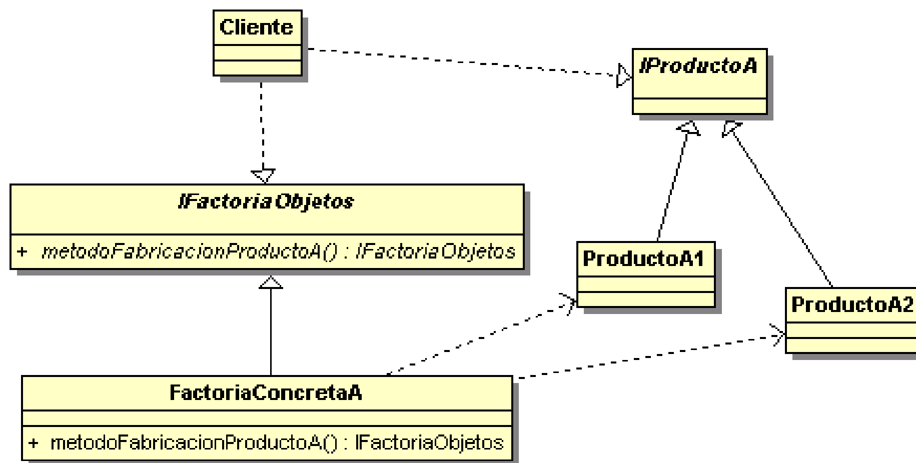
```
"C:\cpd\InfoInd\TeorÝa\6_dise±o\Factorias\Ferreteria\Debug\...
Simulacion de sacar un tornillo y una tuerca de forma aleatoria de una bolsa
La bolsa contiene tornillos y tuercas de metrica DIN84 y DIN316
Pulsar c o C para sacar tornillo y tuerca
c
Ensamblaje correcto: metrica DIN84
Pulsar c o C para sacar tornillo y tuerca
c
Ensamblaje correcto: metrica DIN316
Pulsar c o C para sacar tornillo y tuerca
c
Ensamblaje incorrecto
Pulsar c o C para sacar tornillo y tuerca
c
Ensamblaje correcto: metrica DIN316
Pulsar c o C para sacar tornillo y tuerca
c
Ensamblaje correcto: metrica DIN316
Pulsar c o C para sacar tornillo y tuerca
c
Ensamblaje correcto: metrica DIN84
Pulsar c o C para sacar tornillo y tuerca
c
Ensamblaje correcto: metrica DIN84
Pulsar c o C para sacar tornillo y tuerca
```

```
class IFactoria
{
public:
	virtual ITornillo * fabricacionTornillo(metrica) = 0;
	virtual ITuerca   * fabricacionTuerca(metrica) = 0;
};
class FactoriaTornillos : public IFactoria
{
public:
	virtual ITornillo * fabricacionTornillo(metrica laMetrica)
	{
		if(laMetrica == DIN84) return new TornilloDIN84;
		else if (laMetrica == DIN316) return new TornilloDIN316;
		else return 0;
	}
	virtual ITuerca   * fabricacionTuerca(metrica laMetrica)
		{return 0;}
};
class FactoriaTuercas: public IFactoria
{
public:
	virtual ITornillo * fabricacionTornillo(metrica laMetrica)
	{return 0;}

	virtual ITuerca * fabricacionTuerca(metrica laMetrica)
	{
		if(laMetrica == DIN84) return new TuercaDIN84;
		else if (laMetrica == DIN316) return new TuercaDIN316;
		else return 0;
	}
};
```

```
#include "IFactoria.h"
#include <iostream>
#include <stdlib.h>

int main()
{
 IFactoria *pFactoriaTornillos = new FactoriaTornillos;
 IFactoria *pFactoriaTuercas   = new FactoriaTuercas;
 std::cout<<"Simulacion de sacar tornillo y tuerca de forma aleatoria" <<std::endl;
 std::cout<<"La bolsa contiene tornillos y tuercas DIN84 y DIN316"<<std::endl;
 std::cout<<"Pulsar c o C para sacar tornillo y tuerca"<<std::endl;
 char opcion;  std::cin>> opcion;
 while(opcion == 'c' || opcion == 'C') {
	ITornillo *pTornillo =
	pFactoriaTornillos->fabricacionTornillo(rand() % 2 == 1 ? DIN84 : DIN316);
	ITuerca   *pTuerca =
	pFactoriaTuercas->fabricacionTuerca(rand() % 2 == 1 ? DIN84 : DIN316);
	if(pTornillo->getMetrica() == pTuerca->getMetrica()){
	  char *mensaje = pTuerca->getMetrica() == DIN84 ? "DIN84" : "DIN316";
	  std::cout<<"Ensamblaje correcto: metrica " << mensaje <<std::endl;
	}else
	  std::cout<<"Ensamblaje incorrecto" << std::endl;

	std::cout<<"Pulsar c o C para sacar tornillo y tuerca"<<std::endl;
	std::cin>> opcion;
	delete pTornillo, pTuerca;
	}
	delete pFactoriaTornillos, pFactoriaTuercas;
	return 0;
}
```

Para la elaboración de las Factorías se emplea el patrón Método de Fabricación (GoF). Se define una interfaz de factoría para crear los objetos, pero se deja que sean los métodos de fabricación quien instancia las subclases. Se utiliza cuando:

- una clase no puede prever la clase de objetos que debe crear.
- una clase quiere que sean sus subclases quienes especifiquen los objetos que ésta crea.
- las clases delegan la responsabilidad en una de entre varias clases auxiliares y se desea localizar qué subclase de auxiliar concreta es en la que se delega.



Los roles que se desempeñan en este patrón son:

- Producto: Interfaz de los objetos que crea el método de fabricación
- ProductoConcreto: Realización de la interfaz Producto
- Factoría: Declara el servicio de MétodoFabricación que retorna un objeto de tipo producto abstracto
- FactoríaConcreto: redefine el método de fabricación para devolver una instancia de un ProductoConcreto

La Factoría se apoya en sus subclases para definir el método de fabricación de manera que éste devuelva una instancia del producto concreto adecuado.

Hay que tener mucho cuidado con la creación con las Factorías. En C++ es el programador el que debe posteriormente liberar, con posterioridad, el objeto creado en la factoría.

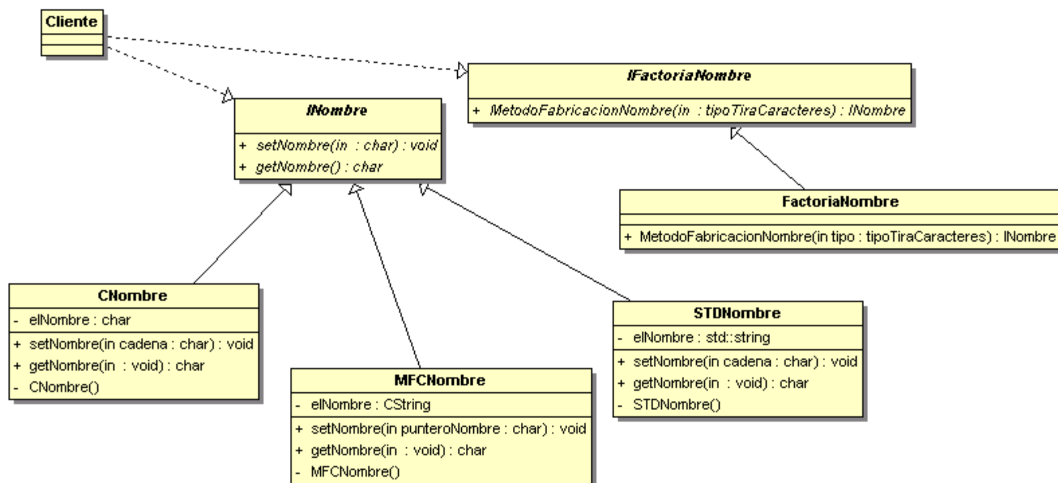
A menudo se accede a las factorías con el patrón *Singleton*.

Ejemplo 6.16

Realizar una factoría de objetos de Nombres. El cliente utilizará la clase abstracta *INombre* y las clases concretas serán empleando *std::string, CString* y en estilo C (ver ejemplo 4.4).

Primera solución

Se empleará un diseño basado en Factoría Abstracta. El Cliente sólo utilizará los interfases para recibir los servicios.




```
#ifndef _INOMBRE_INC_
#define _INOMBRE_INC_
enum Plataforma{ESTANDAR_STL, ESTILO_C,
CADENA_MFC};

class INombre {
public:
  virtual void setNombre (const char *) = 0;
  virtual const char * getNombre () = 0;
};

#endif
```

```
#ifndef _INC_MFCNOMBRE_
#define _INC_MFCNOMBRE_
#include <afx.h>
#include "INombre.h"
class FactoriaNombre;
class MFCNombre : public INombre {
public:
 virtual void setNombre(const char *cadena)
        { elNombre=cadena; }
 virtual const char * getNombre (void)
        { return (elNombre);}
private:
       CString elNombre;
       MFCNombre () {}
       friend class FactoriaNombre;
};
#endif
```

```
// De: "Apuntes de Informática Industrial"
// Ver permisos en licencia de GPL
#ifndef _INC_STDNOMBRE_
#define _INC_STDNOMBRE_

#include <string>
#include "INombre.h"
class FactoriaNombre;
class STDNombre : public INombre
{
public:
 virtual void setNombre(const char *cadena)
        { elNombre = cadena; }
  virtual const char * getNombre (void)
        { return (elNombre.c_str());}

private:
       std::string elNombre;
       STDNombre () {}
       friend class FactoriaNombre;


};

#endif
```

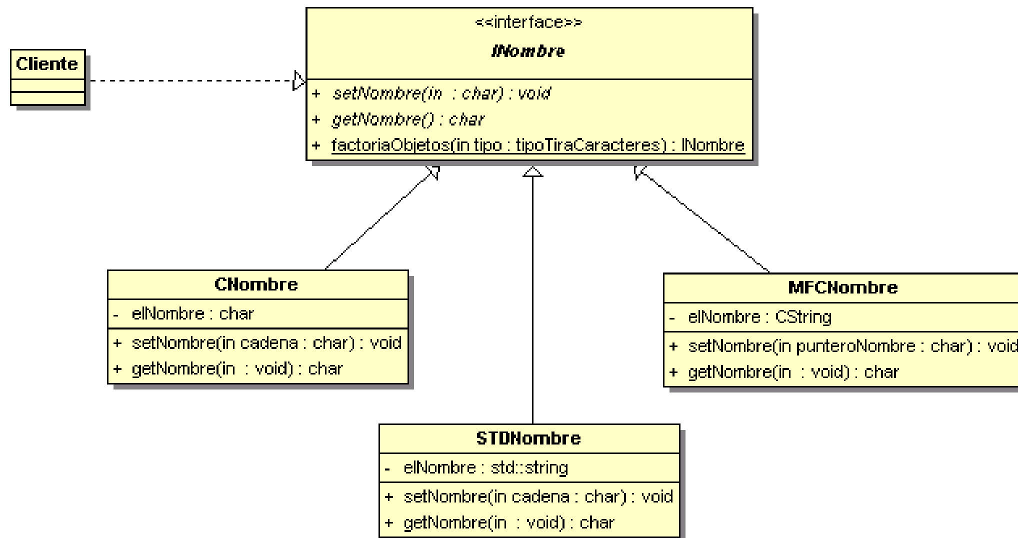
```
// De: "Apuntes de Informática Industrial"
// Ver permisos en licencia de GPL
#ifndef _INC_CNOMBRE_
#define _INC_CNOMBRE_

#include <string>
#include "INombre.h"
class FactoriaNombre;
class CNombre : public INombre {
public:
virtual void setNombre(const char *cadena)
        { strcpy (elNombre, cadena); }

virtual const char * getNombre (void)
        { return (elNombre);}
private:
        char elNombre[80];
        CNombre () {}
        friend class FactoriaNombre;

};

#endif
```

```
// Ver permisos en licencia de GPL
#ifndef _IFACTORIA_INC_
#define _IFACTORIA_INC_

#include "STDNombre.h"
#include "CNombre.h"
#include "MFCNombre.h"

class IFactoriaNombre
{
public:
 virtual INombre* MetodoFabricacionNombre
(enum Plataforma) = 0;

};

class FactoriaNombre: public IFactoriaNombre
{
public:
 virtual INombre* MetodoFabricacionNombre
        (enum Plataforma tipo)
 {
 if(tipo==ESTANDAR_STL)return new STDNombre;
 else if(tipo==ESTILO_C) return new CNombre;
 else if(tipo==CADENA_MFC) return new
        MFCNombre;
 else return NULL;
 }
};
#endif
```

```
// De: "Apuntes de Informática Industrial"
// Ver permisos en licencia de GPL
#include <iostream>
#include "../includes/INombre.h"
#include "../includes/FactoriaNombres.h"

using namespace std;

int main ( void )
{
        //Solo utiliza referencias abstractas
        IFactoriaNombre *pFactoria = new (FactoriaNombre);
        INombre *pNombre1 = pFactoria->MetodoFabricacionNombre (ESTANDAR_STL);
        INombre *pNombre2 = pFactoria->MetodoFabricacionNombre (ESTILO_C);
        INombre *pNombre3 = pFactoria->MetodoFabricacionNombre (CADENA_MFC);

        pNombre1->setNombre("Manolo Gonzalez");
        pNombre2->setNombre("Pedro Lopez");
        pNombre3->setNombre("Ana Rodriguez");

        cout << pNombre1->getNombre() << endl;
        cout << pNombre2->getNombre() << endl;
        cout << pNombre3->getNombre() << endl;

        delete pNombre1, pNombre2, pNombre3;
        delete pFactoria;
        return 0;
}
```

<u>Segunda solución:</u>

El método de fabricación se coloca como un servicio estático dentro de la clase de interfase de Nombre. Para asegurarse de que sólo se utilizará este punto de creación se ha colocado los constructores privados y se ha declarado a la clase abstracta como amiga.




```
// De: "Apuntes de Informática Industrial"
// Ver permisos en licencia de GPL

#ifndef _INOMBRE_INC_
#define _INOMBRE_INC_

typedef enum {ESTANDAR_STL, ESTILO_C,
CADENA_MFC} tipoTiraCaracteres;

class INombre {
public:
 virtual void setNombre (const char *) = 0;
 virtual const char * getNombre () = 0;

 //Factoria de objetos
static INombre *factoriaObjetos
(tipoTiraCaracteres a);
};
#endif
```

```
// De: "Apuntes de Informática Industrial"
// (R) 2005 Con licencia GPL.
// Ver permisos en licencia de GPL
#ifndef _INC_STDNOMBRE_
#define _INC_STDNOMBRE_

#include <string>
#include "INombre.h"

class CNombre : public INombre
{
public:

virtual void setNombre(const char *cadena)
        { strcpy (elNombre, cadena); }

virtual const char * getNombre (void)
        { return (elNombre);}
private:
       char elNombre[80];
};

#endif
```

```
// De: "Apuntes de Informática Industrial"
// (R) 2005 Con licencia GPL.
// Ver permisos en licencia de GPL
#ifndef _INC_STDNOMBRE_
#define _INC_STDNOMBRE_

#include <string>
#include "INombre.h"

class STDNombre : public INombre
{
public:
 virtual void setNombre(const char *cadena)
        { elNombre = cadena; }
  virtual const char * getNombre (void)
        { return (elNombre.c_str());}

private:
 std::string elNombre;
 STDNombre () {}//Desactivar al constructor
 friend class INombre; // Sólo se fabrica
  //desde el método de fabricación
};
#endif
```

```
#ifndef _INC_MFCNOMBRE_
#define _INC_MFCNOMBRE_

#include <afx.h>
#include "INombre.h"

class MFCNombre : public INombre
{
public:
 virtual void setNombre(const char *cadena)
        { elNombre=cadena; }
 virtual const char * getNombre (void)
        { return (elNombre);}
private:
       CString elNombre;
};
#endif
```

```
// De: "Apuntes de Informática Industrial"
// (R) 2005 Con licencia GPL.
// Ver permisos en licencia de GPL

#include <iostream>
#include "../includes/STDNombre.h"
#include "../includes/CNombre.h"
#include "../includes/MFCNombre.h"
//Método único para producir los objetos nombres
INombre* INombre::factoriaObjetos(tipoTiraCaracteres  tipo)
{
  if(tipo == ESTANDAR_STL) return new STDNombre;
  else if(tipo == ESTILO_C) return new CNombre;
  else if(tipo == CADENA_MFC) return new MFCNombre;
  else return NULL;
}

using namespace std;

int main ( void )
{
        INombre *pNombre1 = INombre::factoriaObjetos(ESTANDAR_STL);
        INombre *pNombre2 = INombre::factoriaObjetos(ESTILO_C);
        INombre *pNombre3 = INombre::factoriaObjetos(CADENA_MFC);

        pNombre1->setNombre("Manolo Gonzalez");
        pNombre2->setNombre("Pedro Lopez");
        pNombre3->setNombre("Ana Rodriguez");

        cout << pNombre1->getNombre() << endl;
        cout << pNombre2->getNombre() << endl;
        cout << pNombre2->getNombre() << endl;


        delete pNombre1, pNombre2, pNombre3;
        return 0;
}
```

6.4.3 Singleton

Problema: ¿Cómo garantizar que una clase sólo tenga una instancia y proporciona un punto de acceso global a ella?.

Solución: Definir un método estático de la clase que devuelva el singleton.

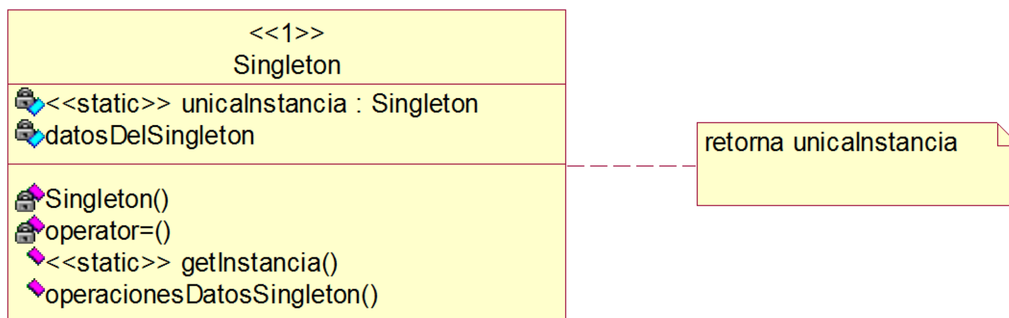
Del uso del patrón Factoría surge un nuevo problema de diseño ¿quién crea a la Factoría?. Sólo se necesita una única instancia de Factoría y ésta puede ser llamada desde distintos lugares del código. Una solución sería ir pasando la instancia de Factoría en los métodos, pero este proceder no es conveniente, ya que implica acoplamiento por necesitar visibilidad. La solución está en el patrón Singleton.

Ocasionalmente es conveniente mantener visibilidad global o un único punto de acceso a una única instancia de una clase. Es importante que algunas clases tengan exactamente una instancia. Por ejemplo, aunque puede haber muchas impresoras, sólo debería de haber una única cola de impresión. De la misma manera, en aplicaciones de control de procesos, sólo debería de existir una única instancia para el driver del convertidor analógico/digital, aunque pudieran existir varios filtros digitales. Son ejemplos de este problema patrón.

¿Cómo se puede asegurar que una clase tenga una única instancia y que sea fácilmente accesible?. Una variable global hace accesible a un objeto, pero no se previene de crear múltiples instancias de esta clase.

Una solución es hacer que sea la propia clase la responsable de su única instancia. La clase debe garantizar que no se pueda crear ninguna otra instancia y proporciona un modo de acceder a su instancia.

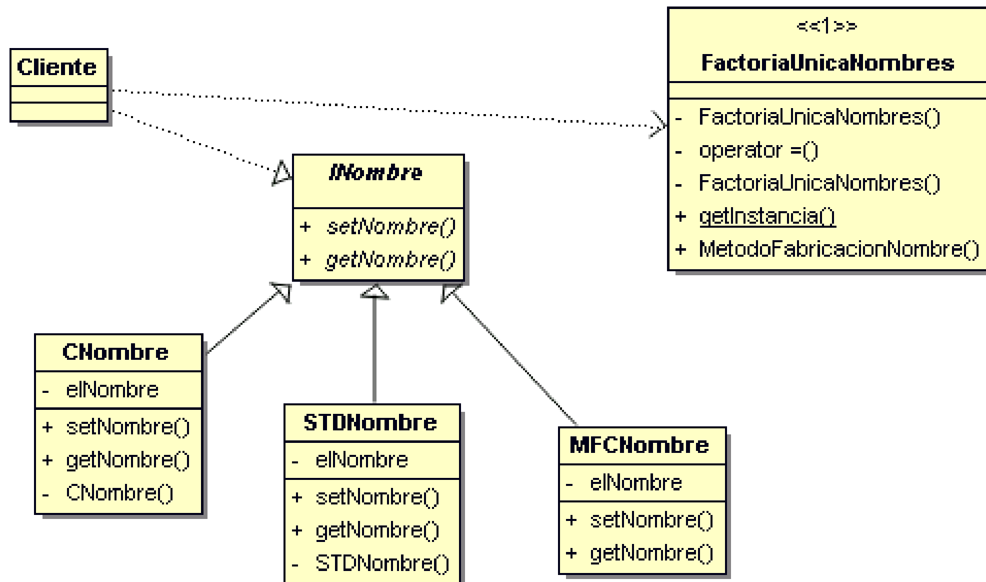
La clave del Singleton es evitar que el cliente no tenga el control sobre la vida del objeto. Para tal fin se declaran todos los constructores como privados y se previene para que el compilador no produzca constructores por defecto ni sobrecarga de operadores de asignación. Se construye el servicio de tipo estático para obtener la referencia Singleton, *getInstancia()*, y también se colocará como estática la propia referencia. En la notación se puede emplear el estereotipo <<*1*>> para indicar que sólo existe una única instancia de esta clase




```
#include <iostream>
using namespace std;

class Singleton {
  int i; //Dato por ejemplo
  Singleton(int x) : i(x) { }
  void operator=(Singleton&);  // Para desactivar
  Singleton(const Singleton&); // Para desactivar
public:
  static Singleton& getInstancia() {
    static Singleton unicaInstancia(47); //P.ej valor 47
    return unicaInstancia;
  }
  int getValor() { return i; }
  void setValor(int x) { i = x; }
};

int main() {
  Singleton& s = Singleton::getInstancia();
  cout << s.getValor() << endl;
  Singleton& s2 = Singleton::getInstancia();
  s2.setValor(9);
  cout << s.getValor() << endl;
  return 0;
}
```

Ejemplo 6.17

Realizar una única factoría de objetos Nombre que sea accesible en cualquier parte de la aplicación (ver problema 6.12).




```
// De: "Apuntes de Informática Industrial" Carlos Platero.
// Ver permisos en licencia de GPL
#ifndef _IFACTORIA_INC_
#define _IFACTORIA_INC_

#include "STDNombre.h"
#include "CNombre.h"
#include "MFCNombre.h"

class FactoriaUnicaNombres
{
    FactoriaUnicaNombres(){};
    void operator=(FactoriaUnicaNombres&);  // Para desactivar
    FactoriaUnicaNombres(const FactoriaUnicaNombres&); // Para desactivar
public:
    static FactoriaUnicaNombres& getInstancia()
        {
            static FactoriaUnicaNombres unicaInstancia;
            return unicaInstancia;
        }

    INombre* MetodoFabricacionNombre (enum Plataforma tipo)
        {
        if(tipo == ESTANDAR_STL) return new STDNombre;
        else if(tipo == ESTILO_C) return new CNombre;
         else if(tipo == CADENA_MFC) return new MFCNombre;
        else return NULL;
        }
};

#endif
```

```
// De: "Apuntes de Informática Industrial" Carlos Platero.
// Ver permisos en licencia de GPL
#include <iostream>
#include "../includes/INombre.h"
#include "../includes/FactoriaUnicaNombres.h"
using namespace std;
int main ( void )
{
	//Solo utiliza referencias abstractas
	void otrosNombres (void);
	FactoriaUnicaNombres &laFactoria = FactoriaUnicaNombres::getInstancia();
	INombre *pNombre1 = laFactoria.MetodoFabricacionNombre (ESTANDAR_STL);
	INombre *pNombre2 = laFactoria.MetodoFabricacionNombre (ESTILO_C);

	pNombre1->setNombre("Manolo Gonzalez");
	pNombre2->setNombre("Pedro Lopez");

	cout << pNombre1->getNombre() << endl;
	cout << pNombre2->getNombre() << endl;

	delete pNombre1, pNombre2;
	otrosNombres();
	return 0;
}

void otrosNombres ( void )
{
	//Solo utiliza referencias
	FactoriaUnicaNombres &laMismaFactoria = FactoriaUnicaNombres::getInstancia();
	INombre *pNombre3 = laMismaFactoria.MetodoFabricacionNombre (CADENA_MFC);

	pNombre3->setNombre("Ana Blanco");
	cout << pNombre3->getNombre() << endl;

	delete pNombre3;
}
```

6.4.4 Estrategia

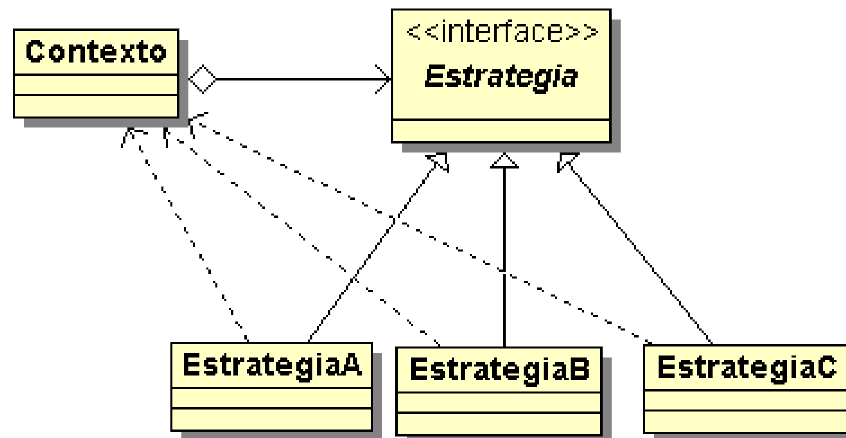
Problema: ¿Cómo diseñar algoritmos que están relacionados? ¿Cómo diseñar que estos algoritmos se puedan variar dinámicamente?

Solución: Defina cada algoritmo o estrategia en una clase independiente, con una interfaz común.

Este patrón define una familia de algoritmos relacionados, los encapsula y los hace intercambiables. La consecuencia de esta estructura es la variación dinámica de los algoritmos sin que los clientes se vean afectados. Los roles de este patrón son:

- Estrategia: declara una interfaz común a todos los algoritmos permitidos.
- Estrategia concreta: implementa el algoritmo concreto usando la interfaz Estrategia.

- Contexto: este objeto usa la interfaz Estrategia para llamar al algoritmo concreto definido por una Estrategia Concreta. Tiene como atributo a un objeto de Estrategia Concreta, a través de su interfaz.



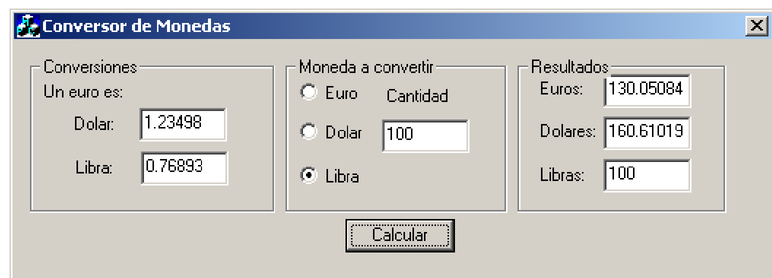
Un objeto estrategia se agrega a un objeto de contexto, i.e. el objeto contexto necesita tener visibilidad de atributo de su estrategia. Además es habitual que el objeto contexto pase una referencia de él al objeto estrategia, de manera que la estrategia tenga visibilidad de parámetro del objeto contexto para futuras colaboraciones.

Al existir diferentes algoritmos que cambian con el tiempo, la pregunta sería: ¿quién debería de crear la estrategia?. Un enfoque directo es aplicar, de nuevo, el patrón Factoría. Una Factoría de Estrategias debe ser responsable de crear todas las estrategias (algoritmos o políticas conectables o cambiantes). Con este diseño, uno puede cambiar dinámicamente, m la estrategia de actuación, mientras se está ejecutando la aplicación.

Como con la mayoría de las factorías será construido empleando un Singleton y se accederá mediante el patrón Singleton.

Con los patrones Estrategia y Factoría se ha conseguido Variaciones Protegidas con respecto a las estrategias que varían dinámicamente. La Estrategia se fundamenta en el Polimorfismo y en la interfaz para permitir algoritmos conectables en un diseño de objetos.

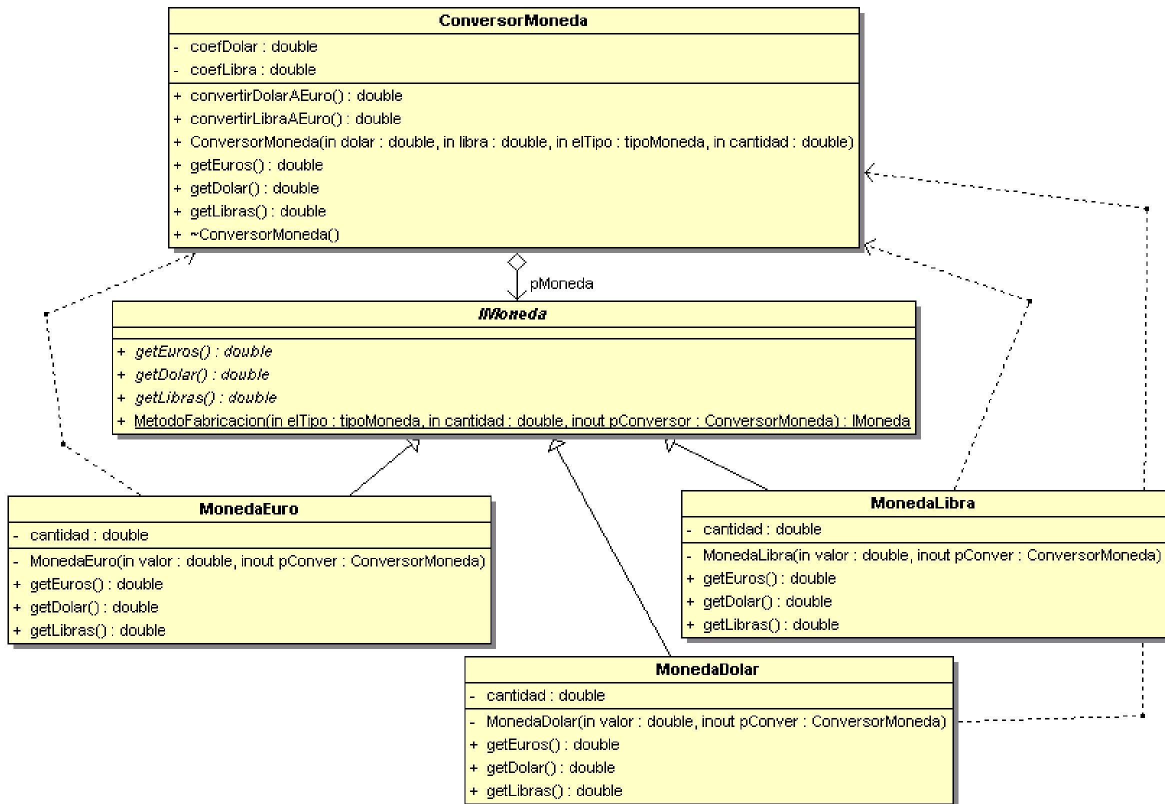
Ejemplo 6.18



Siguiendo el Proceso Unificado, diseñe una aplicación que sea un conversor de monedas. En una primera versión inicial, considere sólo euros, dólares y libras esterlinas. A la aplicación se le facilitará los valores de conversión entre las monedas y la cantidad de una moneda concreta a convertir en el resto de monedas.

```
void CConversorMonedasDlg::OnCalcular()
{
	// TODO: Add your control notification handler code here
	UpdateData(TRUE);
	ConversorMoneda elConversor(this->m_factorDolar,this->m_factorLibra,
		this->m_elTipo,this->m_Cantidad);
	this->m_Euros = elConversor.getEuros();
	this->m_Dolares = elConversor.getDolar();
	this->m_Libras = elConversor.getLibras();
	UpdateData(FALSE);

}
```

Solución



```
typedef enum {EURO,DOLAR,LIBRA} tipoMoneda;
class ConversorMoneda;
class IMoneda
{
public:
	virtual double getEuros() = 0;
	virtual double getDolar() = 0;
	virtual double getLibras() = 0;
	static IMoneda* MetodoFabricacion(tipoMoneda,double,ConversorMoneda *);
};
class ConversorMoneda
{
	IMoneda *pMoneda; double coefDolar; double coefLibra;
public:
	double convertirDolarAEuro()  {return 1/coefDolar;}
	double convertirLibraAEuro() {return 1/coefLibra;}
	ConversorMoneda(double dolar, double libra, tipoMoneda elTipo, double cant):
	coefDolar(dolar), coefLibra(libra)
	{
		pMoneda = IMoneda::MetodoFabricacion(elTipo, cant, this);
	}
	double getEuros() {return pMoneda->getEuros(); }
	double getDolar() {return pMoneda->getDolar(); }
	double getLibras(){return pMoneda->getLibras(); }
	~ConversorMoneda()
	{	delete pMoneda; }
};

class MonedaEuro: public IMoneda
{
	double cantidad; ConversorMoneda *pConversor;
	MonedaEuro(double valor, ConversorMoneda *pConver):
	cantidad(valor), pConversor(pConver) {}
	friend class IMoneda;
public:
	virtual double getEuros() {return cantidad;}
	virtual double getDolar()
	{return cantidad/this->pConversor->convertirDolarAEuro();}
	virtual double getLibras()
	{return cantidad/this->pConversor->convertirLibraAEuro();}
};

class MonedaDolar: public IMoneda
{
	double cantidad; ConversorMoneda *pConversor;
	MonedaDolar(double valor, ConversorMoneda *pConver):
	cantidad(valor),	pConversor(pConver) {}
	friend class IMoneda;
public:
	virtual double getEuros()
	{return cantidad*this->pConversor->convertirDolarAEuro();}
	virtual double getDolar() {return cantidad;}
	virtual double getLibras()
	{return cantidad*this->pConversor->convertirDolarAEuro()/
		this->pConversor->convertirLibraAEuro();}
};
class MonedaLibra: public IMoneda
{
	double cantidad; ConversorMoneda *pConversor;;
	MonedaLibra(double valor,ConversorMoneda *pConver):
	cantidad(valor), pConversor(pConver) {}
	friend class IMoneda;
public:
	virtual double getEuros()
	{return cantidad*this->pConversor->convertirLibraAEuro();}
	virtual double getDolar()
	{return cantidad*this->pConversor->convertirLibraAEuro()/
		this->pConversor->convertirDolarAEuro();}
	virtual double getLibras() {return cantidad;}
};
```

6.4.5 Observador

Problema: Diferentes tipos de objetos receptores están interesados en el cambio de estado o eventos de un objeto emisor y quieren reaccionar cada uno a su manera cuando el emisor genere un evento. Además, el emisor quiere mantener bajo acoplamiento con los receptores ¿qué hacer?

Solución: Defina una interfaz "suscriptor" u "oyente". Los suscriptores implementan esta interfaz. El emisor dinámicamente puede registrar los suscriptores que están interesados en un evento y notificarles cuando ocurre un evento.

Este patrón define una dependencia de uno-a-muchos, de forma que cuando un objeto cambie de estado se notifica y se actualiza automáticamente a todos los objetos que dependan de él. También es conocido como *Dependents* (Dependientes), *Publish-Subscribe* (Publicar-Suscribir).

Una consecuencia habitual de dividir un sistema en una colección de clases cooperantes es la necesidad de mantener una consistencia entre los objetos relacionados. No se desea alcanzar esa consistencia haciendo a las clases fuertemente acopladas, ya que reduciría su reutilización. Por ejemplo, muchos interfaces gráficas de usuario separan los aspectos de presentación del dominio del problema. Las clases que definen los datos de las aplicaciones y las representaciones pueden utilizarse de forma independiente. Así, un objeto de hoja de cálculo y un gráfico de barras pueden representar una información contenida en el mismo objeto de datos del dominio. La hoja de cálculo y el gráfico de barras no se conocen entre sí, permitiéndose de esta manera reutilizarse, pero gracias a este patrón se comportan como si se conocieran. Cuando el usuario cambia la información de la hoja de cálculo, la barra de gráfica refleja los cambios inmediatamente y viceversa.

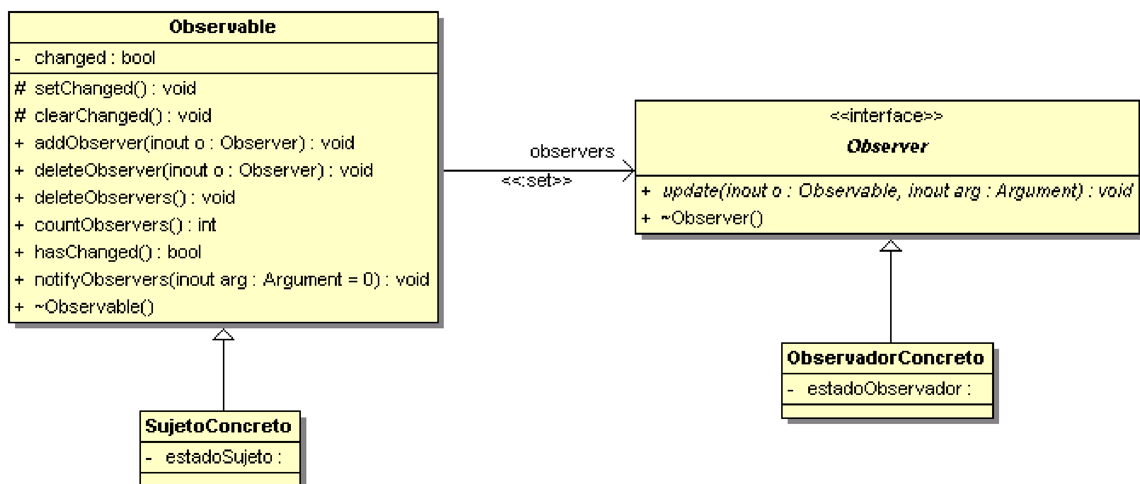
¿Por qué no se puede mandar un mensaje desde el dominio a la vista?. Esta solución supone que los objetos del dominio deben saber el interfaz de la vista, por tanto, se rompería el principio de Bajo Acoplamiento. La separación Dominio-Vista mantiene Variaciones Protegidas con respecto a los cambios en la interfaz de usuario.

El patrón Observador describe cómo establecer estas relaciones. Los principales objetos de este patrón son el *Observable* y el *Observador*. Un *Observable* puede tener cualquier número de observadores dependientes de él. Cada vez que el *Observable* cambia de estado se notificará a sus observadores. Este tipo de interacción también se conoce como publicar-suscribir. El *Observable* es quien publica las notificaciones. Envía estas notificaciones sin tener que conocer quiénes son sus observadores. Pueden suscribirse un número indeterminado de observadores para recibir estas notificaciones.

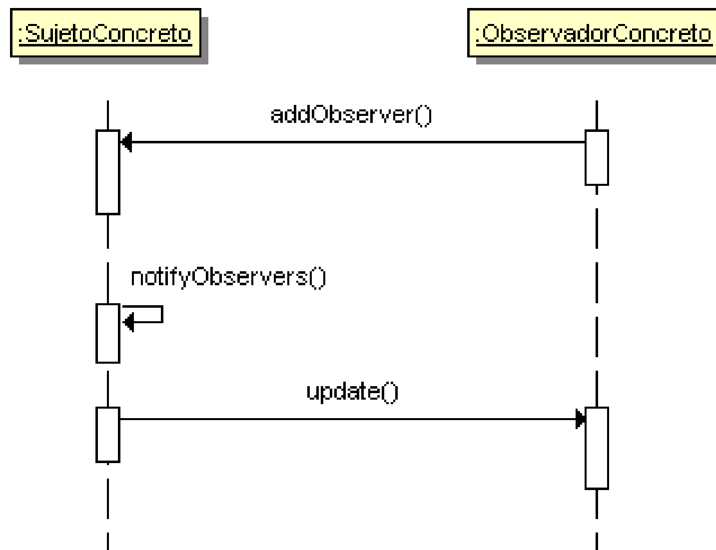
Los roles que se desempeñan en este patrón son:

- *Observable*
  - un Observable puede ser observado por cualquier número de objetos Observador.
  - proporciona una interfaz para asignar y quitar objetos Observador.
- *Observador*
  - define una interfaz para actualizar los objetos que deben ser notificados ante cambios en un sujeto observable.
- *ObservableConcreto*
  - almacena el estado de interés para los objetos ObservadorConcreto.
  - envía una notificación a sus observadores cuando cambia su estado.
- *ObservadorConcreto*
  - mantiene una referencia a un objeto ObservableConcreto.
  - guarda un estado que debería ser consistente con el sujeto observable.
  - Implementa la interfaz de actualización del Observador para mantener su estado consistente con el sujeto observable.

Así, el *ObservableConcreto* notifica a sus observadores cada vez que se produce un cambio en su estado. Después de ser informado de un cambio en el *ObservableConcreto*, un objeto *ObservadorConcreto* puede pedirle al observable más información. *ObservadorConcreto* usa esta información para sincronizar su estado con el del observable.



El siguiente diagrama de secuencias muestra los métodos entre el emisor y receptor.



El patrón Observador permite modificar los observables y los observadores de forma independiente. Es posible reutilizar los observables sin conocer a sus observadores y viceversa. Esto permite añadir observadores sin modificar al observable u otros observadores.

Otras ventajas e inconvenientes del patrón Observador son los siguientes:

1. Acoplamiento abstracto entre *Observable* y *Observador*. Todo lo que un Observable sabe es que tiene una lista de observadores, cada uno de los cuales se ajusta a la interfaz simple de la clase abstracta *Observador*. El *Observable* no conoce la clase concreta de ningún observador. Por tanto, el acoplamiento entre el sujeto observable y los observadores es mínimo (patrón GRASP de Bajo Acoplamiento).

Gracias a que *Observable* y *Observador* no están fuertemente acoplados, pueden pertenecer a diferentes capas de abstracción de un sistema. Un sujeto de bajo nivel puede comunicarse e informar a un observador de más alto nivel, manteniendo de este modo intacto la estructura de capas del sistema (patrón Capas de POSA).

2. Capacidad de comunicación mediante difusión. A diferencia de una petición ordinaria, la notificación enviada por un *Observable* no necesita especificar su receptor (patrón GRASP de Alta Cohesión). La notificación se envía automáticamente a todos los objetos interesados que se hayan suscrito a ella. Al *Observable* no le importa cuántos objetos interesados pudieran existir; su única responsabilidad es notificar a los observadores. Esto da la libertad de añadir y quitar observadores en cualquier momento. Se deja al observador manejar u obviar una notificación del observable.

3. Actualizaciones inesperadas. El mayor inconveniente de este patrón es que los observadores no saben de la presencia de los otros observadores y no pueden saber el coste último de cambiar el estado. Una operación aparentemente inofensiva sobre el *Observable* puede dar lugar a una serie de actualizaciones en cascada de los observadores y sus objetos dependientes. Más aún, los criterios de dependencia que no

están bien definidos o mantenidos suelen provocar falsas actualizaciones que pueden ser muy difíciles de localizar.

6.4.5.1 Implementación

Dos tipos de objetos se utilizan para poner el patrón del observador. La clase ***Observable*** que se encarga de que el estado del Observable concreto sea escuchado por sus observadores y la clase ***Observer*** que está a la escucha. Primero, se presenta ***Observer***:

```
// From "Thinking in C++, Volume 2", by Bruce Eckel & Chuck Allison.
// (c) 1995-2004 MindView, Inc. All Rights Reserved.
//
//: C10:Observer.h
// The Observer interface
#ifndef OBSERVER_H
#define OBSERVER_H

class Observable;
class Argument {};

class Observer {
public:
  // Called by the observed object, whenever
  // the observed object is changed:
  virtual void update(Observable* o, Argument* arg) = 0;
  virtual ~Observer() {}
};
#endif // OBSERVER_H ///:~
```

Puesto que el ***Observer*** obra recíprocamente con ***Observable***, el ***Observable*** se debe declarar primero. La clase ***Observer*** es la interfaz para cualquier observador concreto. La clase ***Argument***, declarada en este ejemplo como vacía, será la que defina los atributos que van a estar a la escucha. El servicio *update()* es llamada por el objeto que está en la escucha cuando se produce una notificación del emisor.

La clase ***Observable*** mantiene la lista de objetos que desean estar a la escucha de las modificaciones que produce el emisor.

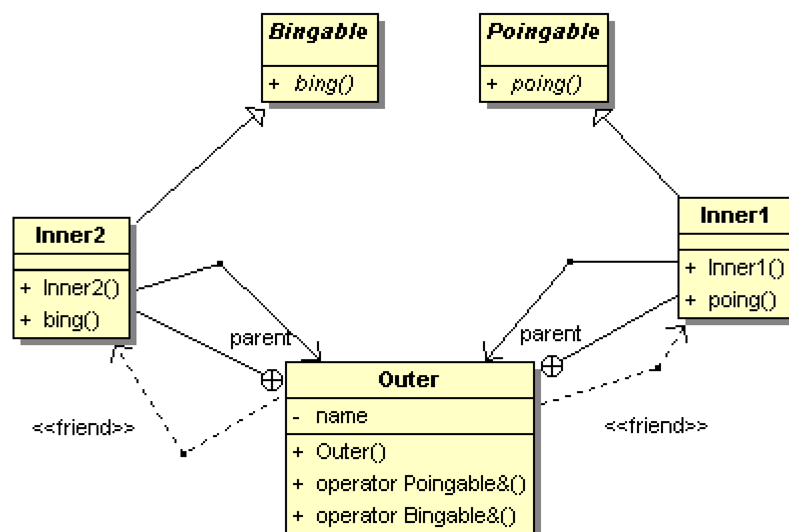
```
// From "Thinking in C++, Volume 2", by Bruce Eckel & Chuck Allison.
// (c) 1995-2004 MindView, Inc. All Rights Reserved.
//
//: C10:Observable.h
// The Observable class.
#ifndef OBSERVABLE_H
#define OBSERVABLE_H
#include <set>
#include "Observer.h"

class Observable {
  bool changed;
  std::set<Observer*> observers;
protected:
  virtual void setChanged() { changed = true; }
  virtual void clearChanged() { changed = false; }
public:
  virtual void addObserver(Observer& o) {
    observers.insert(&o);
  }
  virtual void deleteObserver(Observer& o) {
    observers.erase(&o);
  }
  virtual void deleteObservers() {
    observers.clear();
  }
  virtual int countObservers() {
    return observers.size();
  }
```

```
    virtual bool hasChanged() { return changed; }
    // If this object has changed, notify all
    // of its observers:
    virtual void notifyObservers(Argument* arg = 0) {
      if(!hasChanged()) return;
      clearChanged(); // Not "changed" anymore
      std::set<Observer*>::iterator it;
      for(it = observers.begin();it != observers.end(); it++)
        (*it)->update(this, arg);
    }
    virtual ~Observable() {}
  };
  #endif // OBSERVABLE_H ///:~
```

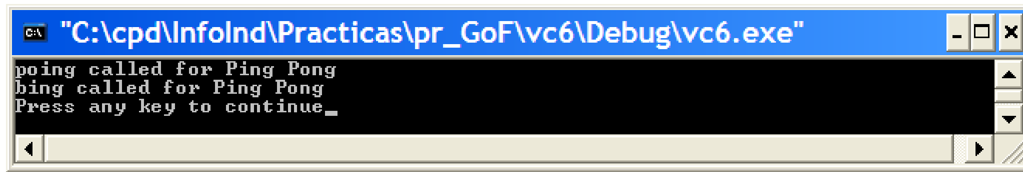
El objeto ***Observable*** tiene un *flag* para indicar si ha cambiado el estado de los atributos a escuchar. La colección de objetos observador se mantiene en un lista *STL set<Observer*>* para prevenir los duplicados; los servicios de ***insert(), erase(), clear(),*** y ***size()*** se exponen para permitir que se agreguen y se quiten a los observadores en cualquier momento, proporcionando flexibilidad en tiempo de ejecución. La mayoría del trabajo se hace en ***notifyObservers()***. La clase ***Observable*** llama a la función miembro ***notifyObservers()*** para cada observador en la lista de observadores a la escucha.Mientras no se produzcan cambios, las llamadas repetidas al ***notifyObservers()*** no hacen nada y no pierden el tiempo. Cuando hay un cambio en los atributos del emisor se produce la llamada al ***update()*** de todos los observadores que estén en la lista.

6.4.5.2 Las clases internas

Tanto las clases que están a la escucha como las observables requieren interfaces que las permitan mantener la propiedad de bajo acoplamiento. Para este fin se requiere de algo que tiene Java y no C++: Las clases internas. Se trata de jerarquizar las clases de forma que se puedan tener acceso a los datos de la clase que las contiene, i.e. una instancia de la clase interna tenga acceso a datos de la clase que la ha creado, siendo la clase interna una subclase de la interfaz deseada. Se presenta un ejemplo del idioma interno. Véase cómo la clase ***Outer*** puede ser vista a través de dos interfaces distintos ***Bingable*** y ***Poingable***:



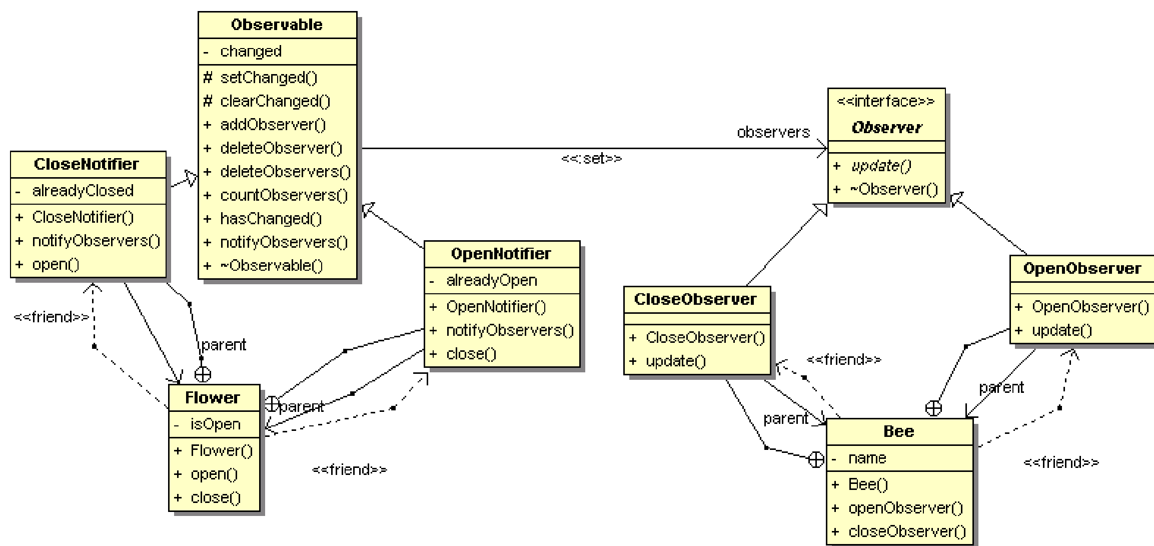
```
// From "Thinking in C++, Volume 2", by Bruce Eckel & Chuck Allison.
// (c) 1995-2004 MindView, Inc. All Rights Reserved.
//
//: C10:InnerClassIdiom.cpp
// Example of the "inner class" idiom.
#include <iostream>
#include <string>
using namespace std;

class Poingable {
public:
  virtual void poing() = 0;
};

class Bingable {
public:
  virtual void bing() = 0;
};


class Outer {
  string name;
  // Define one inner class:
  class Inner1;
  friend class Outer::Inner1;
  class Inner1 : public Poingable {
    Outer* parent;
  public:
    Inner1(Outer* p) : parent(p) {}
    void poing() {
      cout << "poing called for "
        << parent->name << endl;
      // Accesses data in the outer class object
    }
  } inner1;
  // Define a second inner class:
  class Inner2;
  friend class Outer::Inner2;
  class Inner2 : public Bingable {
    Outer* parent;
  public:
    Inner2(Outer* p) : parent(p) {}
    void bing() {
      cout << "bing called for "
        << parent->name << endl;
    }
  } inner2;
public:
  Outer(const string& nm)
  : name(nm), inner1(this), inner2(this) {}
  // Return reference to interfaces
  // implemented by the inner classes:
  operator Poingable&() { return inner1; }
  operator Bingable&() { return inner2; }
};


void callPoing(Poingable& p) {
  p.poing();
}

void callBing(Bingable& b) {
  b.bing();
}

int main() {
  Outer x("Ping Pong");
  // Like upcasting to multiple base types!:
  callPoing(x);
  callBing(x);
} ///:~
```



El ejemplo, previsto para demostrar la sintaxis más simple para el idioma interno, comienza con los interfaces de ***Poingable*** y de ***Bingable*** (cada uno contiene una sola función del miembro). Los servicios de ***callPoing()*** y de ***callBing()*** requieren que el objeto entregue los interfaces de ***Poingable*** y de ***Bingable***, respectivamente. La clase ***Outer*** contiene ciertos datos confidenciales (nombre), y desea proporcionar un interfaz de ***Poingable*** y un interfaz de ***Bingable***, así que puede ser utilizado con el servicio ***callPoing()*** y con ***callBing()***. Para proporcionar un objeto de ***Poingable*** *sin derivar de* ***Poingable***, se utiliza el idioma interno de la clase. Primero, la clase interna se declara y luego se la declara ***friend*** de la clase que la contiene, generándose la jerarquía de clases. Note que la clase interna guarda un indicador de quien la creó, y este indicador se debe inicializar en el constructor. El cual servirá para tener acceso a los atributos de la clase "padre".

6.4.5.3 Entre flores, abejas y colibríes

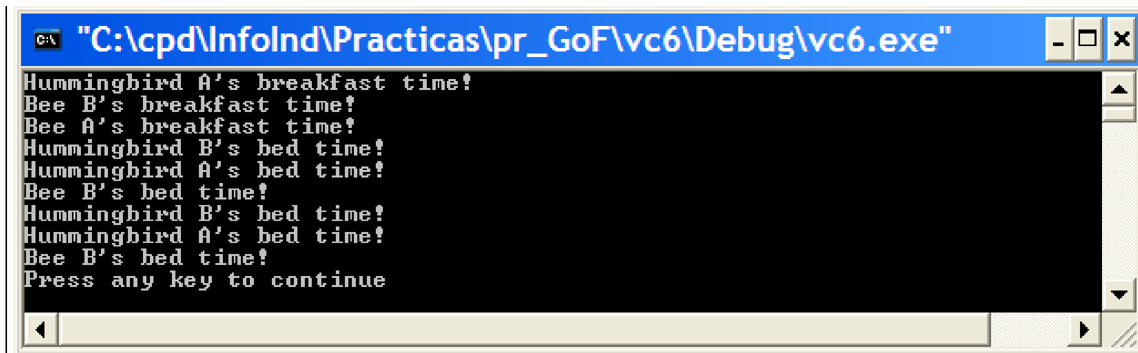
Armado con el interfaz observado, la clase observables y el idioma interno de las clases, se propone un ejemplo del patrón del observador. Hay una instancia de 'flor' y dos instancias de 'abeja' y 'colibrí'. Cuando la flor abre sus pétalos, las abejas y colibrís interesados en esa flor son notificados. Si cierra sus pétalos también será notificado a quién esté interesado. Empleando el patrón observador y las clases internas, el DCD queda determinado para este problema de la siguiente forma:




```
// From "Thinking in C++, Volume 2", by Bruce Eckel & Chuck Allison.
// (c) 1995-2004 MindView, Inc. All Rights Reserved.
//
//: C10:ObservedFlower.cpp
// Demonstration of "observer" pattern.
#include <algorithm>
#include <iostream>
```

```
#include <string>
#include <vector>
#include "Observable.h"
using namespace std;

class Flower {
  bool isOpen;
public:
  Flower() : isOpen(false),
    openNotifier(this), closeNotifier(this) {}
  void open() { // Opens its petals
    isOpen = true;
    openNotifier.notifyObservers();
    closeNotifier.open();
  }
  void close() { // Closes its petals
    isOpen = false;
    closeNotifier.notifyObservers();
    openNotifier.close();
  }
  // Using the "inner class" idiom:
  class OpenNotifier;
  friend class Flower::OpenNotifier;
  class OpenNotifier : public Observable {
    Flower* parent;
    bool alreadyOpen;
  public:
    OpenNotifier(Flower* f) : parent(f),
      alreadyOpen(false) {}
    void notifyObservers(Argument* arg = 0) {
      if(parent->isOpen && !alreadyOpen) {
        setChanged();
        Observable::notifyObservers();
        alreadyOpen = true;
      }
    }
    void close() { alreadyOpen = false; }
  } openNotifier;
  class CloseNotifier;
  friend class Flower::CloseNotifier;
  class CloseNotifier : public Observable {
    Flower* parent;
    bool alreadyClosed;
  public:
    CloseNotifier(Flower* f) : parent(f),
      alreadyClosed(false) {}
    void notifyObservers(Argument* arg = 0) {
      if(!parent->isOpen && !alreadyClosed) {
        setChanged();
        Observable::notifyObservers();
        alreadyClosed = true;
      }
    }
    void open() { alreadyClosed = false; }
  } closeNotifier;
};

class Bee {
  string name;
  // An "inner class" for observing openings:
  class OpenObserver;
  friend class Bee::OpenObserver;
  class OpenObserver : public Observer {
    Bee* parent;
  public:
    OpenObserver(Bee* b) : parent(b) {}
    void update(Observable*, Argument *) {
      cout << "Bee " << parent->name
        << "'s breakfast time!” << endl;
    }
  } openObsrv;
  // Another "inner class" for closings:
  class CloseObserver;
  friend class Bee::CloseObserver;
  class CloseObserver : public Observer {
    Bee* parent;
  public:
```

```
    CloseObserver(Bee* b) : parent(b) {}
    void update(Observable*, Argument *) {
      cout << "Bee " << parent->name
        << "'s bed time!" << endl;
    }
  } closeObsrv;
public:
  Bee(string nm) : name(nm),
    openObsrv(this), closeObsrv(this) {}
  Observer& openObserver() { return openObsrv; }
  Observer& closeObserver() { return closeObsrv;}
};

class Hummingbird {
  string name;
  class OpenObserver;
  friend class Hummingbird::OpenObserver;
  class OpenObserver : public Observer {
    Hummingbird* parent;
  public:
    OpenObserver(Hummingbird* h) : parent(h) {}
    void update(Observable*, Argument *) {
      cout << "Hummingbird " << parent->name
        << "'s breakfast time!" << endl;
    }
  } openObsrv;
  class CloseObserver;
  friend class Hummingbird::CloseObserver;
  class CloseObserver : public Observer {
    Hummingbird* parent;
  public:
    CloseObserver(Hummingbird* h) : parent(h) {}
    void update(Observable*, Argument *) {
      cout << "Hummingbird " << parent->name
        << "'s bed time!" << endl;
    }
  } closeObsrv;
public:
  Hummingbird(string nm) : name(nm),
    openObsrv(this), closeObsrv(this) {}
  Observer& openObserver() { return openObsrv; }
  Observer& closeObserver() { return closeObsrv;}
};

int main() {
  Flower f;
  Bee ba("A"), bb("B");
  Hummingbird ha("A"), hb("B");
  f.openNotifier.addObserver(ha.openObserver());
  f.openNotifier.addObserver(hb.openObserver());
  f.openNotifier.addObserver(ba.openObserver());
  f.openNotifier.addObserver(bb.openObserver());
  f.closeNotifier.addObserver(ha.closeObserver());
  f.closeNotifier.addObserver(hb.closeObserver());
  f.closeNotifier.addObserver(ba.closeObserver());
  f.closeNotifier.addObserver(bb.closeObserver());
  // Hummingbird B decides to sleep in:
  f.openNotifier.deleteObserver(hb.openObserver());
  // Something changes that interests observers:
  f.open();
  f.open(); // It's already open, no change.
  // Bee A doesn't want to go to bed:
  f.closeNotifier.deleteObserver(
    ba.closeObserver());
  f.close();
  f.close(); // It's already closed; no change
  f.openNotifier.deleteObservers();
  f.open();
  f.close();
} ///:~
```


"C:\cpd\InfoInd\Practicas\pr_GoF\vc6\Debug\vc6.exe"
Hummingbird A's breakfast time!
Bee B's breakfast time!
Bee A's breakfast time!
Hummingbird B's bed time!
Hummingbird A's bed time!
Bee B's bed time!
Hummingbird B's bed time!
Hummingbird A's bed time!
Bee B's bed time!
Press any key to continue

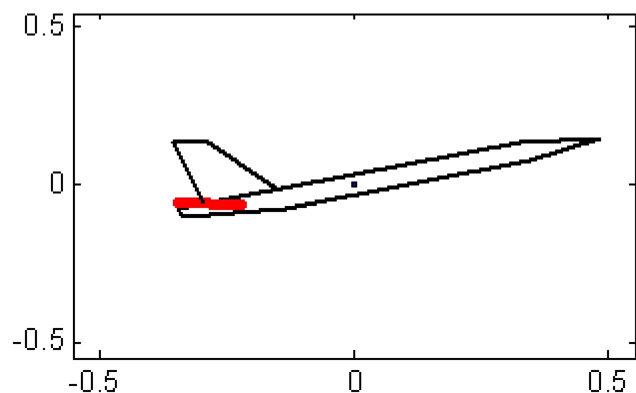
Propósito	Patrones de diseño GoF	Tema que trata
De creación	Factoría Abstracta Método de Fabricación *Singleton*	Uso de familias de referencias abstractas, ocultado las implementaciones concretas. Creación de las clases concretas. la única instancia de una clase
Estructurales	Adaptador Fachada	la interfaz de un objeto la interfaz de un subsistema
De comportamiento	Observador Estrategia	cómo se mantiene actualizado el objeto dependiente cómo se varía dinámicamente un algoritmo

6.5 Ejercicios

1. Habilidades necesarias para el diseño orientado a objetos.
2. Tipos de visibilidad en la programación orientada a objetos.
3. ¿Qué es la Programación Extrema, XP?.
4. ¿Qué se entiende por patrón en el diseño orientado a objetos?.
5. Describir el par problema/solución de los patrones GRASP y GoF estudiados.

Problema 1

Realizar el primer diseño para el simulador del sistema de control de elevación de una aeronave (ver este problema en los ejercicios de los capítulos 2 y 3).



Un modelo simplificado del control de elevación en transformadas en Z está dado por la siguiente función de transferencia:

$$M(z)=\frac{0.0895z-0.085}{z^2-1.883z+0.888}$$

El periodo de muestreo es de 25ms. Por tanto, si la señal de mando se la denota por δ_k y el ángulo de elevación de la aeronave por α_k. El algoritmo de control está definido por la siguiente ecuación en diferencias:

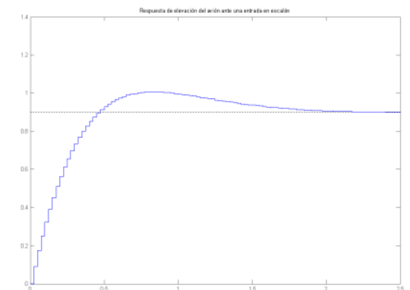
$$\alpha_k = 0.0895\,\delta_k - 0.085\,\delta_{k-1} + 1.883\,\alpha_{k-1} - 0.888\,\alpha_{k-2}$$

Como prueba de concepto se puede suponer una entrada en escalón unitario. La dinámica de elevación de la aeronave seguirá la siguiente trayectoria:

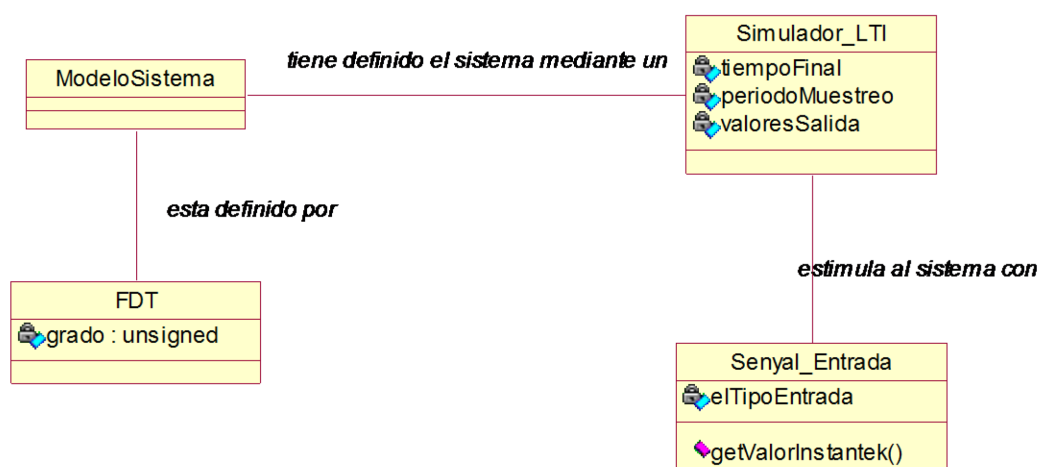
k	δ_k	δ_{k-1}	α_k	α_{k-1}	α_{k-2}
0	1	0	0.0895	0	0
1	1	1	0.173	0.0895	0
2	1	1	0.25	0.173	0.0895
3	1	1	0.323	0.25	0.173

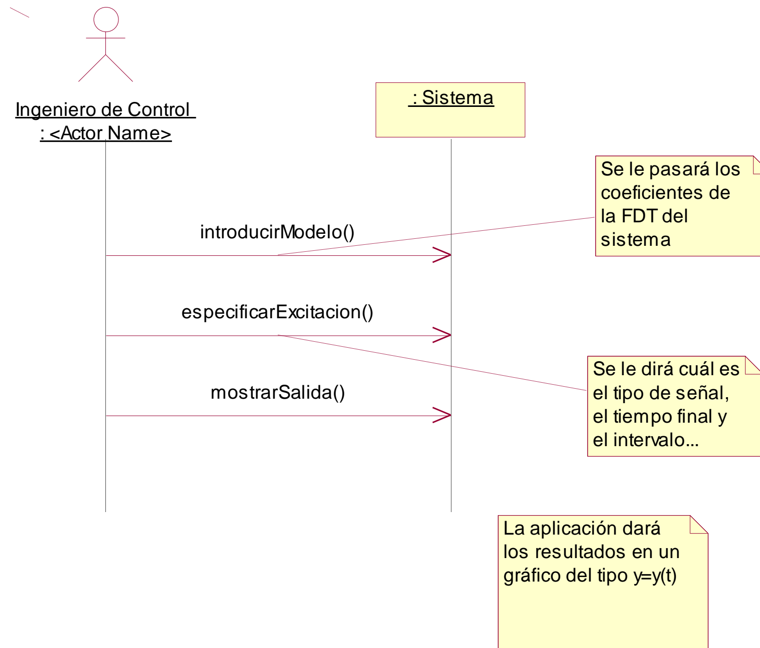
Empleando Matlab, la solución quedaría como:

```
>> g1=tf([.0895 -.085],[1 -1.883 .888],.025)

>>step(g1)
```

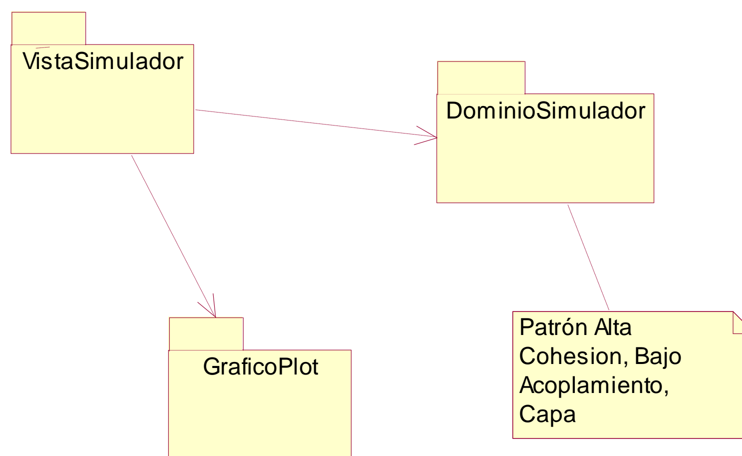


Una vez realizado las pruebas de conceptos se pasa al modelo del dominio y al DSS:

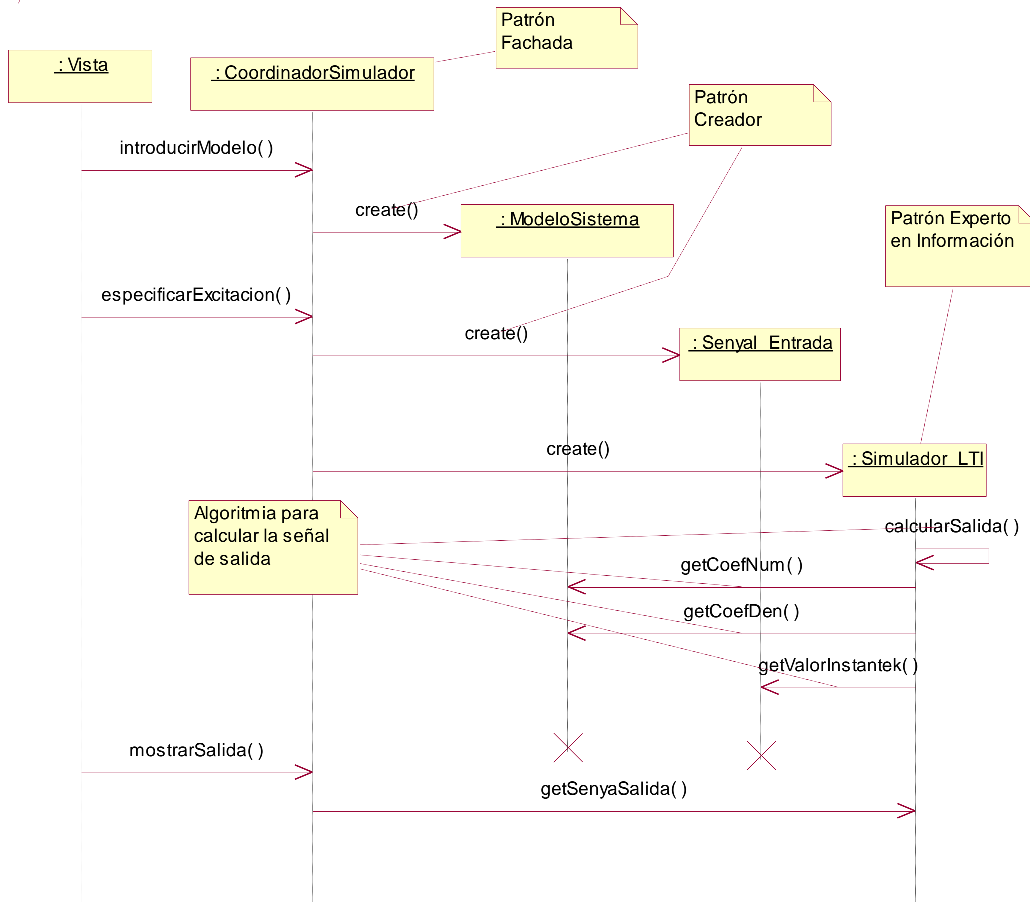


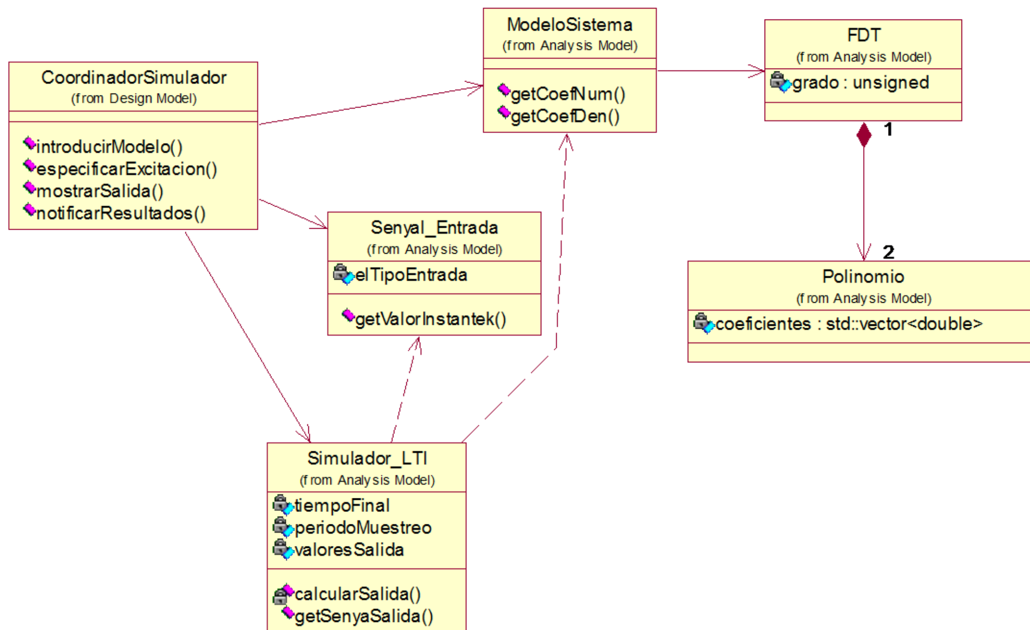


Aplicando los patrones de Alta Cohesión, Bajo Acoplamiento y Modularidad-Capas, la vista de gestión de la aplicación estaría formada por los siguientes paquetes:



El diagrama de interacción y el DCD del paquete del dominio estará constituido por:





Problema 2

Para el código adjuntado se pide:

a. Ingeniería inversa: Diagrama de clases.

b. Ingeniería inversa: Diagrama de secuencia.

c. Resultado de su ejecución en la consola.

d. Indicar los patrones GRASP empleados en este patrón.

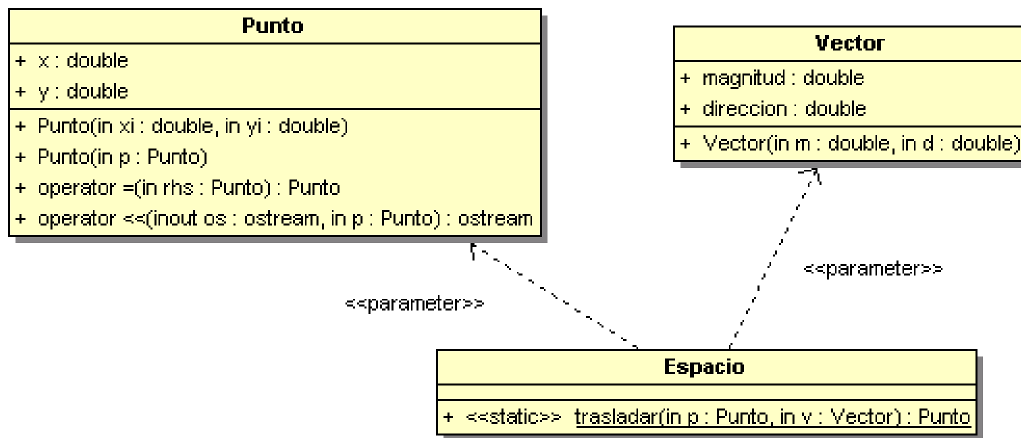
e. Diseñar e implementar el servicio rotar( ), tal que

$$\begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$$

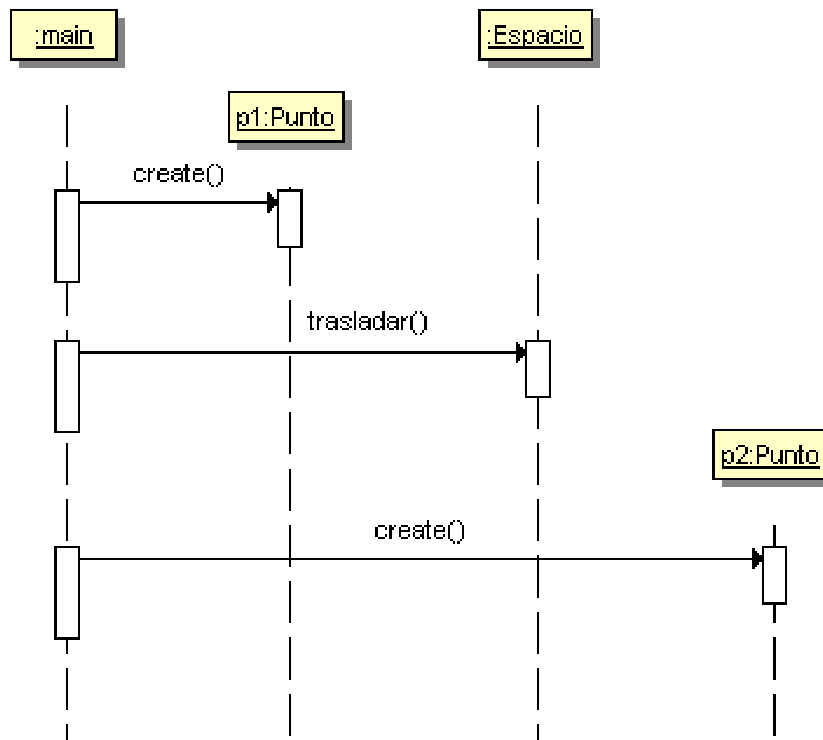
Empléese sobre el punto p2.

```
#include <iostream>
#include <string>
#include <cmath>
using namespace std;

class Punto {
public:
  double x, y;
  Punto(double xi, double yi) : x(xi), y(yi) {}
  Punto(const Punto& p) :  x(p.x), y(p.y) {}
  Punto& operator=(const Punto& rhs) {
    x = rhs.x;
    y = rhs.y;
    return *this;
  }
  friend ostream&
  operator<<(ostream& os, const Punto& p) {
    return os << "x=" << p.x << " y=" << p.y;
  }
};

class Vector {
public:
  double magnitud, direccion;
  Vector(double m, double d) : magnitud(m), direccion(d) {}
};

class Espacio {
public:
  static Punto trasladar(Punto p, Vector v) {
    p.x += (v.magnitud * cos(v.direccion));
    p.y += (v.magnitud * sin(v.direccion));
    return p;
  }
};

int main() {
  Punto p1(1, 2);
  Punto p2 = Espacio::trasladar(p1, Vector(3, 3.1416/3));
  cout << "p1: " << p1 << " p2: " << p2 << endl;

  return 0;
}
```

a) Diagrama de clases de la ingeniería inversa



b) Diagrama de secuencia de la ingeniería inversa



c) p1: x=1 y=2 p2: x=2.5 y=4.6

d) Se ha aplicado Experto de Información en la clase Punto y Vector. Para evitar el acoplamiento entre ambas clases se ha aplicado el patrón Indirección y por tanto una Fabricación Pura con la clase Espacio.

e)

```
class Espacio {
public:
  static Punto trasladar(Punto p, Vector v) {
    p.x += (v.magnitud * cos(v.direccion));
    p.y += (v.magnitud * sin(v.direccion));
    return p;
  }
  static Punto rotar(Punto p, double theta) {
    Punto res(0,0);
    res.x = (p.x * cos(theta)) - (p.y *sin(theta));
    res.y = (p.x * sin(theta)) + (p.y *cos(theta));
    return res;
  }
};

int main() {
  Punto p1(1, 2);
  Punto p2 = Espacio::trasladar(p1, Vector(3, 3.1416/3));
  Punto p3 = Espacio::rotar(p2,3.1416/6);

  cout << "p1: " << p1 << " p2: " << p2 << " p3: " << p3 <<endl;

  return 0;
}
```

Problema 3

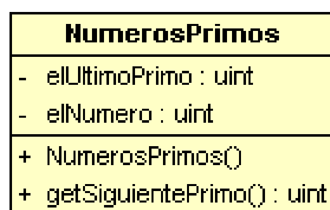
En el cuadro se entrega el código sobre un generador de números primos. Se trata de diseñar un componente tal que el cliente le entregue el número límite de números primos, *n*, y el servidor retorne con un vector que contenga los *n* primeros números primos. En la figura se presenta el resultado del cliente. Se

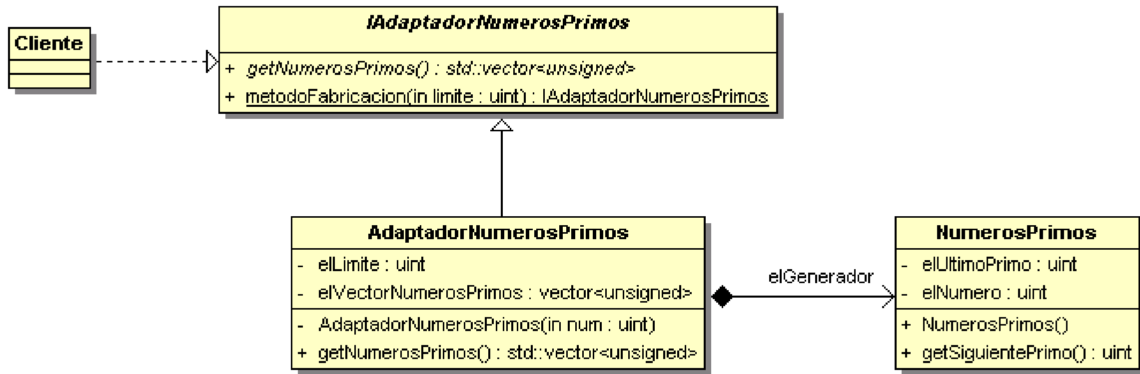
```
"f:\cpd\InfoInd\TeorÝa\6_dise±o\Adaptador\NumerosPrimos\Debug\NumerosPrimos.exe"
Cuantos numeros primos desea que aparezcan : 200

Tabla de Numeros Primos
-----------------------
    2    3    5    7   11   13   17   19   23   29
   31   37   41   43   47   53   59   61   67   71
   73   79   83   89   97  101  103  107  109  113
  127  131  137  139  149  151  157  163  167  173
  179  181  191  193  197  199  211  223  227  229
  233  239  241  251  257  263  269  271  277  281
  283  293  307  311  313  317  331  337  347  349
  353  359  367  373  379  383  389  397  401  409
  419  421  431  433  439  443  449  457  461  463
  467  479  487  491  499  503  509  521  523  541
  547  557  563  569  571  577  587  593  599  601
  607  613  617  619  631  641  643  647  653  659
  661  673  677  683  691  701  709  719  727  733
  739  743  751  757  761  769  773  787  797  809
  811  821  823  827  829  839  853  857  859  863
  877  881  883  887  907  911  919  929  937  941
  947  953  967  971  977  983  991  997 1009 1013
 1019 1021 1031 1033 1039 1049 1051 1061 1063 1069
 1087 1091 1093 1097 1103 1109 1117 1123 1129 1151
 1153 1163 1171 1181 1187 1193 1201 1213 1217 1223
Press any key to continue_
```

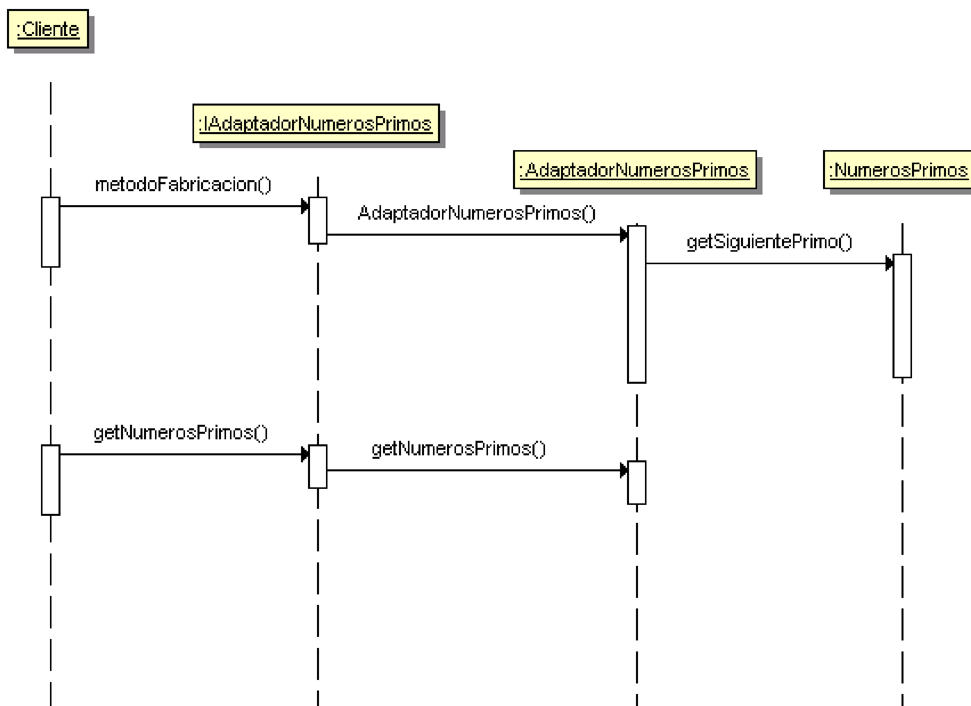
```cpp
#ifndef NUMEROSPRIMOS_H
#define NUMEROSPRIMOS_H

class NumerosPrimos {
  unsigned elUltimoPrimo;
  unsigned elNumero;
public:
  NumerosPrimos() : elUltimoPrimo(1) { elNumero = elUltimoPrimo+1;}
  unsigned getSiguientePrimo()
  {
        do{
          for(unsigned divisor = 2;elNumero % divisor != 0; divisor++) ;
               if (divisor == elNumero)
                 elUltimoPrimo = elNumero;
               else
                 elNumero++;
        } while ( elUltimoPrimo != elNumero );
        elNumero++;
        return (elUltimoPrimo);
  }

};
#endif
```

pide:

1. Realizar ingeniería inversa sobre el generador de números primos.
2. Obtener el diagrama de clase de diseño, DCD, así como el diagrama de secuencia del componente servidor.
3. Implementación del cliente en C++.
4. Implementación de las clases en C++ del componente servidor.




Cliente
IAdaptadorNumerosPrimos
+ getNumerosPrimos() : std::vector<unsigned>
+ metodoFabricacion(in limite : uint) : IAdaptadorNumerosPrimos
AdaptadorNumerosPrimos
- elLimite : uint
- elVectorNumerosPrimos : vector<unsigned>
- AdaptadorNumerosPrimos(in num : uint)
+ getNumerosPrimos() : std::vector<unsigned>
elGenerador
NumerosPrimos
- elUltimoPrimo : uint
- elNumero : uint
+ NumerosPrimos()
+ getSiguientePrimo() : uint


:Cliente
:IAdaptadorNumerosPrimos
:AdaptadorNumerosPrimos
:NumerosPrimos
metodoFabricacion()
AdaptadorNumerosPrimos()
getSiguientePrimo()
getNumerosPrimos()
getNumerosPrimos()

```
#include <iostream>
#include "AdaptadorNumerosPrimos.h"
#include <numeric>
#include <algorithm>

IAdaptadorNumerosPrimos* IAdaptadorNumerosPrimos::metodoFabricacion(unsigned limite)
{
        return (new AdaptadorNumerosPrimos(limite));

}
using namespace std;
void imprimir(unsigned);

int main()
{
  unsigned elLimiteNumerosPrimos;
  cout<<"Cuantos numeros primos desea que aparezcan : ";
  cin >> elLimiteNumerosPrimos;

  IAdaptadorNumerosPrimos *pAdaptadorNumPrimos =
         IAdaptadorNumerosPrimos::metodoFabricacion(elLimiteNumerosPrimos);

  cout << endl <<"Tabla de Numeros Primos" <<endl;
  cout << "-----------------------" <<endl;

  for_each(pAdaptadorNumPrimos->getNumerosPrimos().begin(),
             pAdaptadorNumPrimos->getNumerosPrimos().end(),imprimir);
  delete pAdaptadorNumPrimos;
  return 0;
}

void imprimir(unsigned numPrimo)
{
        static unsigned indice;
        cout.width(5);
        cout << numPrimo;
        if(++indice % 10 == 0)
               cout << endl;
}
```

```
#ifndef _ADAPTADOR_NUMEROSPRIMOS
#define _ADAPTADOR_NUMEROSPRIMOS

#include <vector>
#include "NumerosPrimos.h"

class IAdaptadorNumerosPrimos
{
public:
    virtual std::vector<unsigned> & getNumerosPrimos() = 0;
    static IAdaptadorNumerosPrimos *metodoFabricacion(unsigned);
};

class AdaptadorNumerosPrimos: public IAdaptadorNumerosPrimos
{
    NumerosPrimos elGenerador;
    unsigned elLimite;
    std::vector<unsigned> elVectorNumerosPrimos;
    friend class IAdaptadorNumerosPrimos;
    AdaptadorNumerosPrimos(unsigned num): elLimite(num)
          {
           for (unsigned i=1;i<=elLimite;i++)
                 elVectorNumerosPrimos.push_back
                             (elGenerador.getSiguientePrimo());
          }
public:
      virtual std::vector<unsigned> & getNumerosPrimos()
          {return elVectorNumerosPrimos;}
};

#endif
```

Problema 4

Se desea hacer una aplicación que sirva para calcular las nominas de una compañía. Al salario base de cada empleado hay que quitarle un 20% de retención del IRPF para calcular su salario neto. Como existen diferentes políticas salariales en la empresa, se desea hacer un programa fácilmente extensible a nuevas políticas. De momento se pretende abordar dos de ellas: a) el sueldo ordinario b) el sueldo con bonus, consistente en aumentar el salario base (antes de la retención) un 35%. De momento se ha desarrollado el siguiente programa:

```
main()
{
	Nomina* nomina;
	cout<<"1. Nomina ordinaria"<<endl;
	cout<<"2. Nomina con bonus"<<endl;
	cin>>opcion;

	//Completar codigo aqui

	cout<<"Salario base: ";
	float salario;
	cin>>salario;
	nomina->SetSalarioBase(salario);
	float total=nomina->GetSueldoNeto();
	cout<<"El salario neto es: "<<total<<endl;
}
```

Se pide:

1. Diagrama de Clases de Diseño de una arquitectura que permita una fácil extensión a nuevas políticas. Indicar los patrones utilizados.

2. Implementación en C++ de la solución, completando el main().

Problema 5

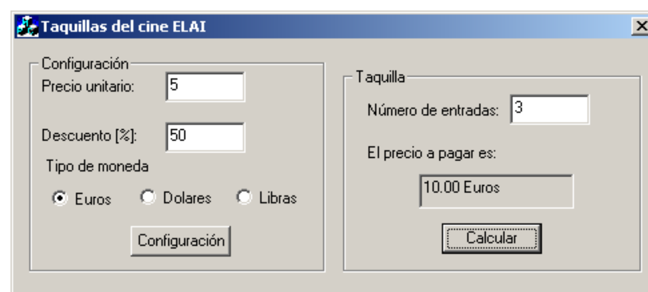
Para la aplicación informática de una sala de cine, se desea que un taquillero introduzca el número de localidades que desea un comprador y se le aplica el mejor descuento para el espectador. En esta primera versión se considera que las monedas pueden ser EUROS, DOLARES y LIBRAS. Por defecto, se considerará que el dinero se expresa en EUROS y el precio de butaca es de 5€. Las políticas de ventas a introducir, en esta primera iteración, son:

a) Si el día de la semana es miércoles se le aplicará un descuento del 50%.

b) En cualquier día de la semana, la compra de tres entradas será al precio de dos. Esta política se aplica para número de localidades que sea múltiplo de tres.

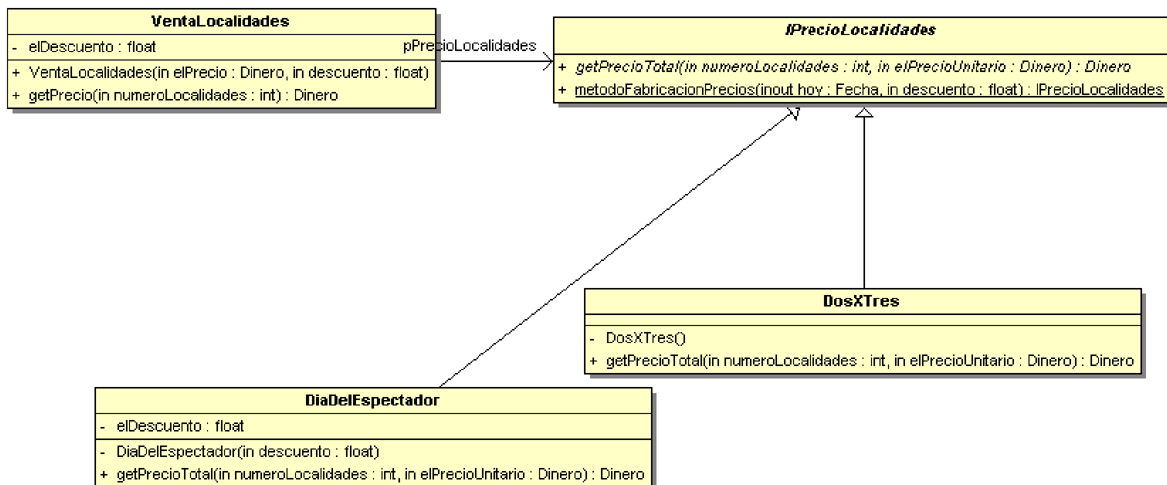
El arquitecto del programa ha realizado un primer borrador del diagrama de clase de diseño. Obviamente no está toda la información para la codificación.

```
int main() {
float precio;
cout <<"\nPrecio unitario de taquilla en euros: ";
cin  >> precio;
Dinero elPrecioUnitario(precio);
float descuento;
cout <<"\nCual es el descuento del dia del espectador [\%]: ";
cin  >> descuento;
VentaLocalidades laVentanilla(elPrecioUnitario,descuento/100);
//Bucle de taquilla
bool bSalir = false;
Dinero elValorVenta;
while (bSalir != true){
        int numeroLocalidades;
        cout << "\nNumero de localidades a vender :";
        cin  >> numeroLocalidades;
        elValorVenta = laVentanilla.getPrecio(numeroLocalidades);
        cout << "A cobrar " << elValorVenta.getCantidad() <<" euros.";
         cout << "\nIntroducir mas ventas (s/n) ?";
        char eleccion;
        cin >> eleccion;
        bSalir = ((eleccion == 'n')||(eleccion == 'N')) ? true:false;
        }
        return 0;
}
```



1. Implementar la clase Dinero (por defecto se considera que el tipo de moneda es el EURO).

2. Para las estrategias de los precios se ha utilizado un interfaz, de manera que las políticas de ventas se puedan variar en el futuro. Realizar su implementación.



3. Implementar la clase de Venta de Localidades

```
#ifndef _DINERO_INC_
#define _DINERO_INC_

typedef enum {EURO, DOLAR, LIBRA} TipoDinero;

class Dinero
{
	TipoDinero elTipoMoneda;
	float cantidad;
public:
	Dinero(): elTipoMoneda(EURO), cantidad(0) {}
	Dinero(float valor, TipoDinero elTipo): elTipoMoneda(elTipo),cantidad(valor) {}

	Dinero(Dinero &elValor)
	{
		elTipoMoneda = elValor.elTipoMoneda;
		cantidad = elValor.cantidad;
	}

	Dinero& operator= (Dinero &elValor)
	{
		elTipoMoneda = elValor.elTipoMoneda;
		cantidad = elValor.cantidad;
		return(*this);
	}

	void setCantidad(float laCantidad){cantidad=laCantidad;}
	float getCantidad(void){return cantidad;}
	void setTipoDinero(TipoDinero elTipo){elTipoMoneda=elTipo;}
	TipoDinero getTipoDinero(void){return elTipoMoneda;}
};

#endif
```

```
#ifndef _VENTA_LOCALIDADES_INC_
#define _VENTA_LOCALIDADES_INC_

#include "Fecha.h"
#include "Dinero.h"
#include "OfertasTaquillas.h"

class VentaLocalidades
{
	Fecha elDiaSemana;
	Dinero elPrecioUnitario;
	float elDescuento;
	IPrecioLocalidades *pPrecioLocalidades;

public:
  VentaLocalidades(Dinero elPrecio, float descuento) :
		elPrecioUnitario(elPrecio),elDescuento(descuento) {}
  Dinero getPrecio(int numeroLocalidades)
  {
    pPrecioLocalidades =
	IPrecioLocalidades::metodoFabricacionPrecios(elDiaSemana,elDescuento);
    Dinero elDinero =
	pPrecioLocalidades->getPrecioTotal(numeroLocalidades,elPrecioUnitario) ;
    delete pPrecioLocalidades;
    return(elDinero);
  }
};

#endif
```

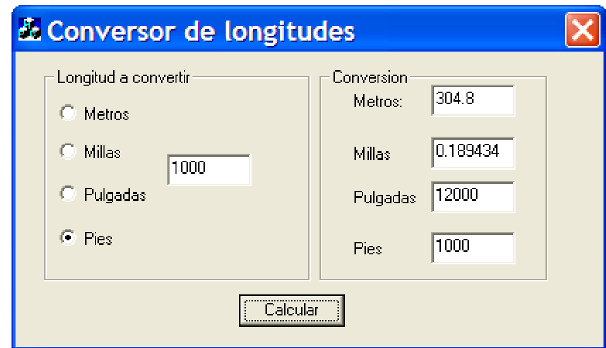
```
#ifndef _OFERTAS_TAQUILLA_INC_
#define _OFERTAS_TAQUILLA_INC_
#include "Dinero.h"
#include "Fecha.h"

class IPrecioLocalidades
{
public:
  virtual Dinero getPrecioTotal(int numeroLocalidades, Dinero elPrecioUnitario) = 0;
  static IPrecioLocalidades *metodoFabricacionPrecios(Fecha &, float);
};

class DiaDelEspectador : public IPrecioLocalidades
{
        float elDescuento;
        friend class IPrecioLocalidades;
        DiaDelEspectador(float descuento): elDescuento(descuento) {}
public:
virtual Dinero getPrecioTotal(int numeroLocalidades, Dinero elPrecioUnitario)
 {
 Dinero elValorTotal;
 elValorTotal.setCantidad(elPrecioUnitario.getCantidad()*numeroLocalidades*elDescuento);
 elValorTotal.setTipoDinero(elPrecioUnitario.getTipoDinero());
 return (elValorTotal);
 }
};

class DosXTres : public IPrecioLocalidades
{
        friend class IPrecioLocalidades;
        DosXTres() {}
public:
        virtual Dinero getPrecioTotal(int numeroLocalidades, Dinero elPrecioUnitario)
        {
        Dinero elValorTotal;
        int multiplosDeTres = 0;
        for(int i=1;i<=numeroLocalidades;i++)
                if( i%3 == 0)
                        multiplosDeTres++;
        int localidadesSueltas = numeroLocalidades - (multiplosDeTres*3);
        elValorTotal.setCantidad(elPrecioUnitario.getCantidad()*
                (localidadesSueltas+(multiplosDeTres*2)));
        elValorTotal.setTipoDinero(elPrecioUnitario.getTipoDinero());
        return (elValorTotal);
        }
};

#endif
```

```
#include "includes/VentaLocalidades.h"

IPrecioLocalidades * IPrecioLocalidades::metodoFabricacionPrecios
(Fecha &hoy, float descuento = 0)
{
        if ((hoy.isMiercoles() == true ) && (descuento > 100/3) )
                return new DiaDelEspectador(descuento);
        else return new DosXTres;
}
```

Problema 6

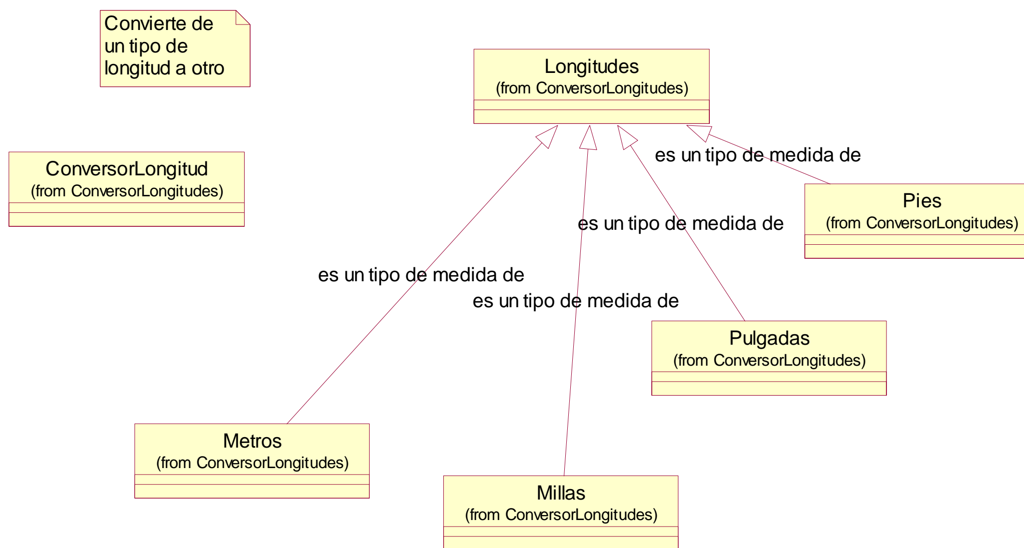
Desarrollar una aplicación que convierta las magnitudes de longitud de un sistema a otro, sabiendo que:

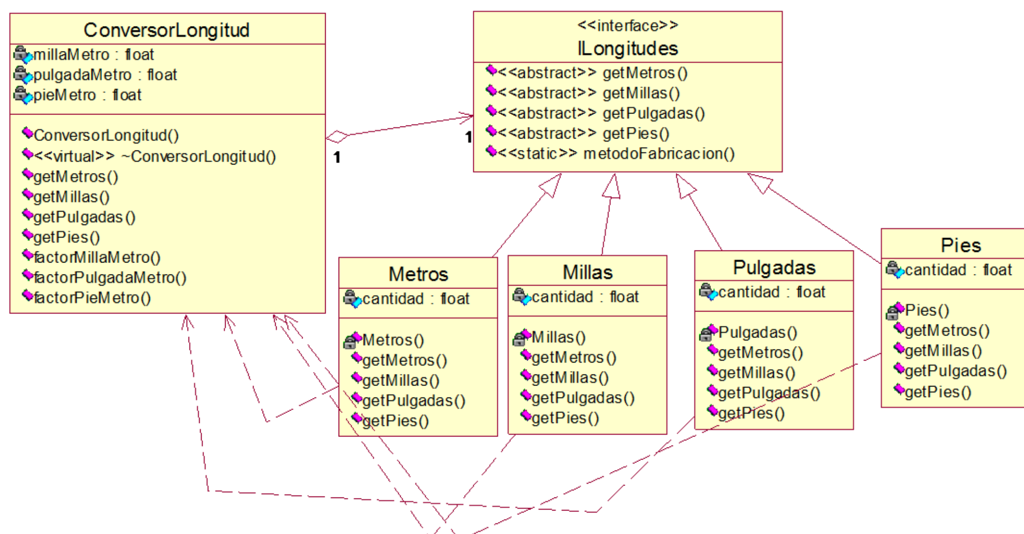
1 milla = 1609 m
1 pulgada = 25.4 mm
1 pie = 30.48 cm



Se pide:

1. AOO: Modelo del dominio.
2. DOO: Diagrama de clases de diseño. Indicar los patrones que se están aplicando.
3. Implementación en C++.





Los patrones utilizados son: Estrategia(GoF) que incluye Variaciones Protegidas (GRASP), Método de Fabricación (GoF) y Polimorfismo (GRASP)

El código de test sería:

```
void CConversorLongitudesDlg::OnCalcular()
{
    // TODO: Add your control notification handler code here
    UpdateData(TRUE);

    ConversorLongitud elConversor(this->m_tipoLongitud,this->m_longitud);
    this->m_Metros = elConversor.getMetros();
    this->m_Millas = elConversor.getMillas();
    this->m_Pulgadas = elConversor.getPulgadas();
    this->m_Pies = elConversor.getPies();

    UpdateData(FALSE);
}
```

Mientras que el interfaz, el conversor y las subclases de longitudes serían:

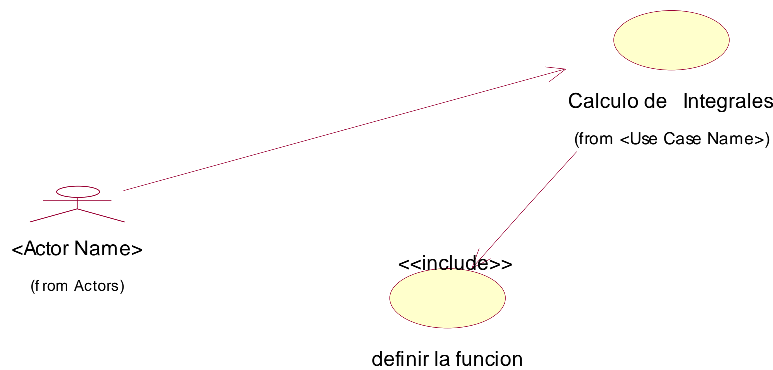
```
#if !defined(AFX_LONGITUDES_H)
#define AFX_LONGITUDES_H

typedef enum {METROS,MILLAS,PULGADAS,PIES}  tipoLongitudes;

class ConversorLongitud;
class ILongitudes
{
public:
        virtual float getMetros() = 0;
        virtual float getMillas() = 0;
        virtual float getPulgadas() = 0;
        virtual float getPies() = 0;
static ILongitudes * metodoFabricacion(tipoLongitudes,float,
                      ConversorLongitud *);
};
#endif
------------------------------------------------------------------------
#if !defined(AFX_CONVERSORLONGUITUDES_H)
#define AFX_CONVERSORLONGUITUDES_H

#include "Longitudes.h"

class ConversorLongitud
{
        ILongitudes *plongitudes;
        float millaMetro;
        float pulgadaMetro;
        float pieMetro;

public:
        ConversorLongitud(tipoLongitudes,float);
        virtual ~ConversorLongitud();

        float getMetros() { return plongitudes->getMetros(); }
        float getMillas() { return plongitudes->getMillas(); }
        float getPulgadas() { return plongitudes->getPulgadas(); }
        float getPies() { return plongitudes->getPies(); }

        float factorMillaMetro() {return millaMetro;}
        float factorPulgadaMetro() {return pulgadaMetro;}
        float factorPieMetro() {return pieMetro;}

};
```

```
#if !defined(AFX_LONGUITUDES_SIS_H)
#define AFX_LONGUITUDES_SIS_H

#include "ConversorLongitud.h"


class Metros : public ILongitudes
{
        float cantidad; ConversorLongitud *pConversor;
        Metros(float valor,ConversorLongitud *pC):
                cantidad(valor), pConversor(pC) {}
        friend class ILongitudes;
public:
        float getMetros() { return cantidad;}
        float getMillas() {return cantidad/pConversor->factorMillaMetro();}
float getPulgadas() {return cantidad/pConversor->factorPulgadaMetro();}
        float getPies() {return cantidad/pConversor->factorPieMetro();}
};

class Millas : public ILongitudes
{
        float cantidad; ConversorLongitud *pConversor;
        Millas(float valor,ConversorLongitud *pC):
                cantidad(valor), pConversor(pC) {}
        friend class ILongitudes;
public:
        float getMetros() { return cantidad*pConversor->factorMillaMetro();}
        float getMillas() {return cantidad;}
float getPulgadas() {return cantidad*pConversor->factorMillaMetro()
                /pConversor->factorPulgadaMetro();}
        float getPies() {return cantidad*pConversor->factorMillaMetro()
                /pConversor->factorPieMetro();}
};

class Pulgadas : public ILongitudes
{
        float cantidad; ConversorLongitud *pConversor;
        Pulgadas(float valor,ConversorLongitud *pC):
                cantidad(valor), pConversor(pC) {}
        friend class ILongitudes;
public:
float getMetros() { return cantidad*pConversor->factorPulgadaMetro();}
        float getMillas() {return cantidad*pConversor->factorPulgadaMetro()
                        /pConversor->factorMillaMetro();}
        float getPulgadas() {return cantidad;}
        float getPies() {return cantidad*pConversor->factorPulgadaMetro()
                                /pConversor->factorPieMetro();}
};

class Pies : public ILongitudes
{
        float cantidad; ConversorLongitud *pConversor;
        Pies(float valor,ConversorLongitud *pC):
                cantidad(valor), pConversor(pC) {}
        friend class ILongitudes;
public:
        float getMetros() { return cantidad*pConversor->factorPieMetro();}
        float getMillas() {return cantidad*pConversor->factorPieMetro()
                        /pConversor->factorMillaMetro();}
        float getPulgadas() {return cantidad*pConversor->factorPieMetro()
                        /pConversor->factorPulgadaMetro();}
        float getPies() {return cantidad;}

};
#endif
```

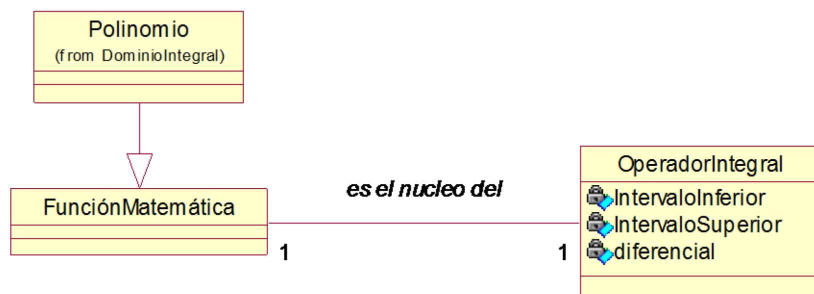
Problema 7

Cálculo de una integral sobre una función monovariable (en este caso, sólo polinómica), pudiendo elegir la estrategia de integración: a) Lineal o b) Bilineal.

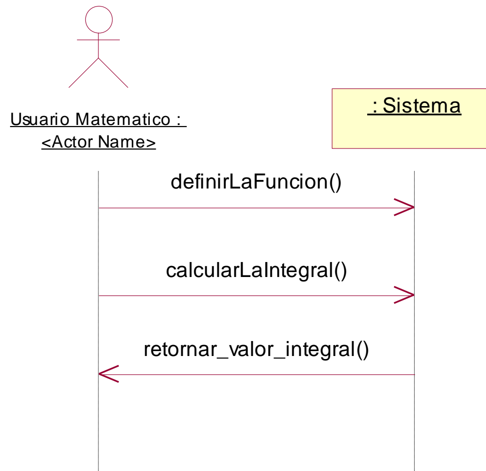
a) Caso de uso
b) Modelo del dominio
c) DSS
d) Vista de gestión
e) Diagramas de interacción
f) DCD
g) Implementación



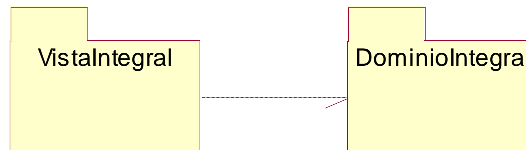
El cálculo de la integral requiere saber cual es la función a quien se le va aplicar el operador. Un modelo del dominio podría quedar reflejado como:



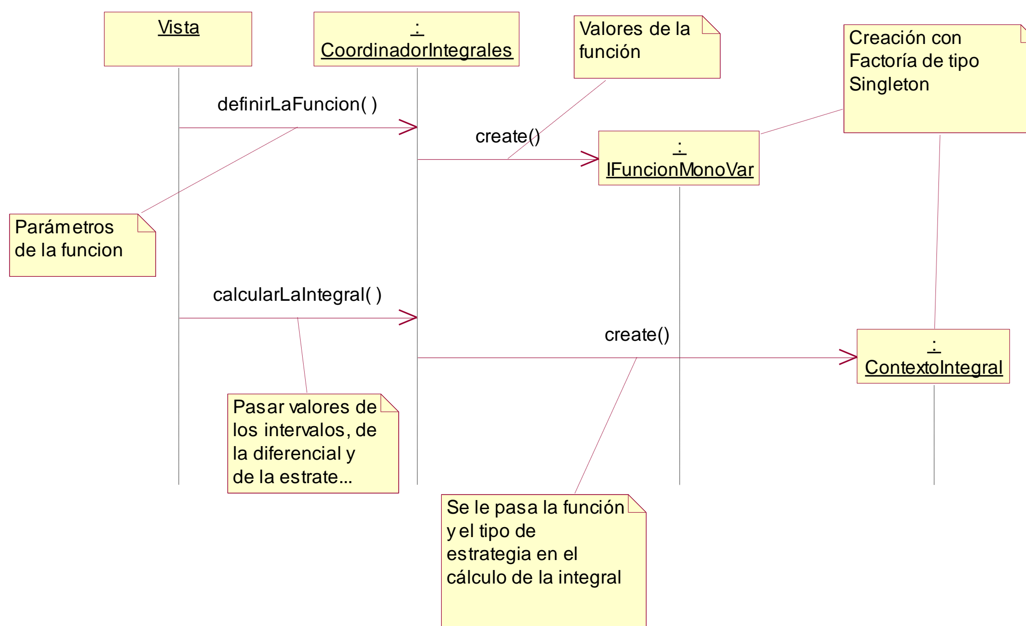
En esta aplicación habrá que definir la función y los parámetros de la integración. Se propone el siguiente DSS:

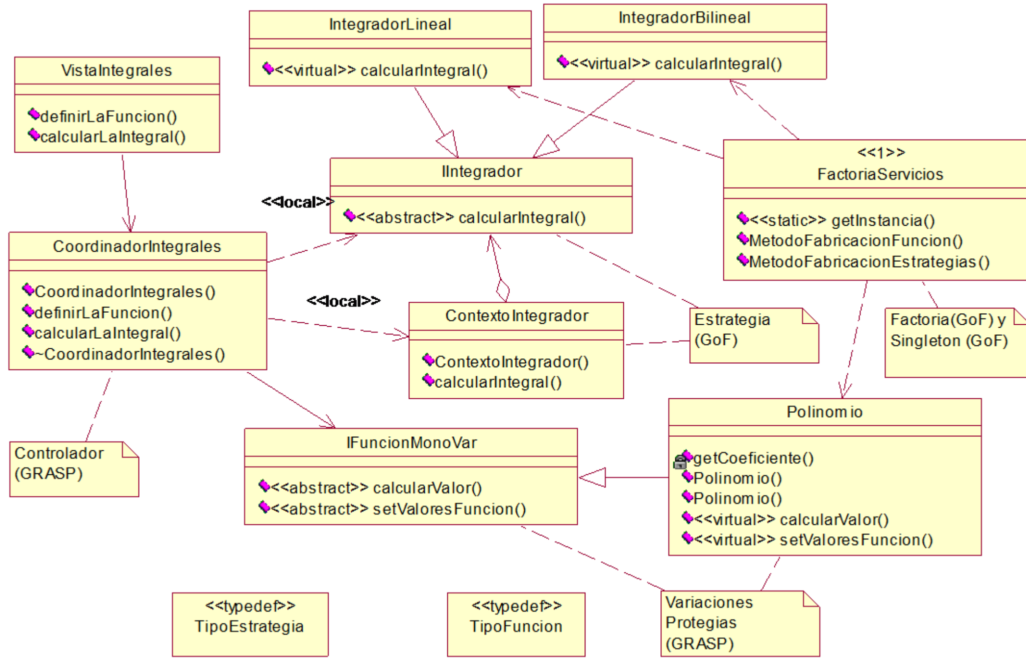


En esta fase se dividirá el modelo en dos paquetes:



Desgranado el DSS y aplicando los patrones se proponen los diagramas de interacción y de clases de diseño:






```
e:\cpd\infoindus\teoría\6_diseño\estrategia\integralesv002\integralesv002\debug\integralesv002....

Funciones polinomicas
a.Cual es el grado: 2

Valor para el coeficiente 0: 1

Valor para el coeficiente 1: 2

Valor para el coeficiente 2: 1

Cual es el intervalo inferior de integracion: 0

Cual es el intervalo superior de integracion: 1

Cual es el intervalo de la diferencial: .01

Tipo de integracion:
1.Lineal
2.Bilineal
2

El resultado de la integracion es: 2.34325
Pulsar cualquier tecla para salir excepto intro o espacio_
```

```
// De: "Apuntes de Informática Industrial" Carlos Platero.
// Ver permisos en licencia de GPL
#ifndef _INTEGRADOR_INC_
#define _INTEGRADOR_INC_
#include "../Funcion/FuncionMonoVar.h"
typedef enum tipoEstr{LINEAL,BILINEAL} TipoEstrategia;
class IIntegrador
{
public:
	virtual double calcularIntegral(IFuncionMonoVar *, float intInf,
		float intSup, float diferencial) = 0;
};
class IntegradorLineal: public IIntegrador
{
public:
	virtual double calcularIntegral(IFuncionMonoVar * pLaFuncion, float intInf,
		float intSup, float diferencial){
		double resultado = 0;
		for(float i= intInf;i<=intSup;i+=diferencial)
			resultado += pLaFuncion->calcularValor(i)*diferencial;
		return (resultado);
	}
 };
class IntegradorBilineal: public IIntegrador
{
public:
	virtual double calcularIntegral(IFuncionMonoVar * pLaFuncion, float intInf,
		float intSup, float diferencial){
		double resultado = 0;
		for(float i= intInf;i<=intSup;i+=diferencial)
			resultado +=
			(pLaFuncion->calcularValor(i)+pLaFuncion->calcularValor
			(i-diferencial))/2 *diferencial;
		return (resultado);
	}
 };
class ContextoIntegrador
{
	IIntegrador* pEstrategia;
public:
	ContextoIntegrador
		(IIntegrador* pTipodeIntegrador) : pEstrategia(pTipodeIntegrador) {}
	double calcularIntegral(IFuncionMonoVar* pLaFuncion, float intInf,
		float intSup, float diferencial)
	{return(pEstrategia->calcularIntegral(pLaFuncion,inInf,intSup,diferencial));}
};

#endif
```

```
#ifndef _IFUNCION_MONO_INC_
#define _IFUNCION_MONO_INC_
#include <vector>
#include <math.h>
typedef enum tipoFun{POLINOMIO,TRIGONOMETRICA} TipoFuncion;
class IFuncionMonoVar
{
public:
	virtual double calcularValor(float) = 0;
	virtual void setValoresFuncion( unsigned , double *) = 0;

};
class Polinomio : public IFuncionMonoVar
{
	std::vector<double> coeficientes;
	std::vector<double>::iterator iteratorCoeficientes;

	double getCoeficiente(unsigned n)
	{return(*(iteratorCoeficientes+n));}

public:
	Polinomio(){}
	Polinomio(unsigned grado, double *pCoef){
		for (unsigned i=0;i<=grado;coeficientes.push_back(*(pCoef+i)),i++);
		iteratorCoeficientes = coeficientes.begin();
	}
	virtual double calcularValor(float valorInd){
		double resultado = 0;
		for(int i=0;i<= this->coeficientes.size() ;i++)
			resultado += getCoeficiente(i)*pow(valorInd,i);
		return resultado;
	}
    virtual void setValoresFuncion( unsigned grado, double *pCoef){
		for (unsigned i=0;i<=grado;coeficientes.push_back(*(pCoef+i)),i++);
		iteratorCoeficientes = coeficientes.begin();
	}
};
#endif
```

```
#ifndef _FACTORIA_INC_
#define _FACTORIA_INC_
#include "../Funcion/FuncionMonoVar.h"
#include "../Integrador/Integrador.h"
class FactoriaServicios
{
    FactoriaServicios(){};
    void operator=(FactoriaServicios&);  // Para desactivar
    FactoriaServicios(const FactoriaServicios&); // Para desactivar
public:
    static FactoriaServicios& getInstancia(){
        static FactoriaServicios unicaInstancia;
        return unicaInstancia;
    }
    //Factoria de funciones matematicas
    IFuncionMonoVar* FactoriaServicios::
                  MetodoFabricacionFuncion(TipoFuncion elTipoFuncion){
        if (elTipoFuncion == POLINOMIO) return new Polinomio();
        else return NULL;
    }
    //Factoria de estrategias
    IIntegrador* FactoriaServicios::
                  MetodoFabricacionEstrategia(TipoEstrategia elTipoEstr){
        if (elTipoEstr == LINEAL) return new IntegradorLineal;
        else if (elTipoEstr == BILINEAL) return new IntegradorBilineal;
        else return NULL;
    }
};

#endif
```

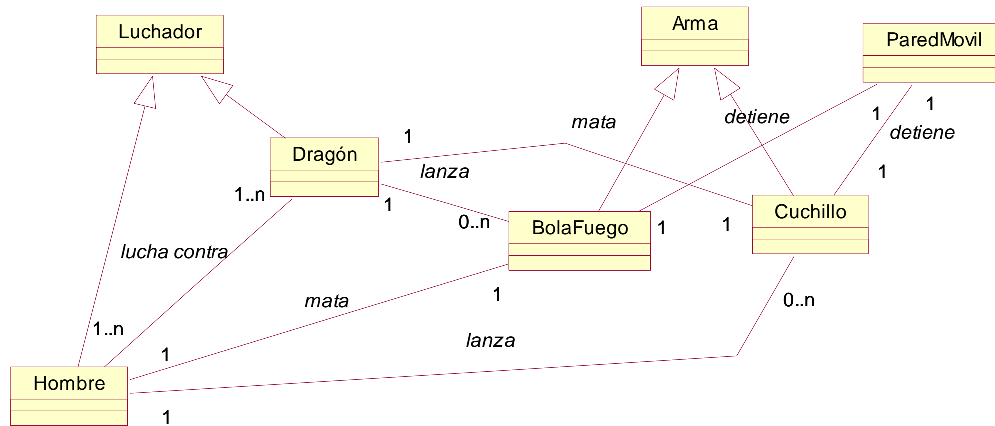
```
// De: "Apuntes de Informática Industrial" Carlos Platero.
// (R) 2005 Con licencia GPL.
// Ver permisos en licencia de GPL
#include "../../include/Coordinador/CoordinadorIntegrales.h"
int CoordinadorIntegrales::definirLaFuncion(TipoFuncion elTipoFun,unsigned grado,
                                                              double *pCoef)
{
    FactoriaServicios &laFactoria = FactoriaServicios::getInstancia();
    this->pFuncionMono =  laFactoria.MetodoFabricacionFuncion (elTipoFun);
    if (this->pFuncionMono == NULL)
               return (-1);
    this->pFuncionMono->setValoresFuncion(grado,pCoef);
    return 0;
}
double CoordinadorIntegrales::calcularLaIntegral(float intInf,
               float intSup, float diferencial, TipoEstrategia elTipoIntegral)
{
 if (this->pFuncionMono == NULL)
       return (0);
 FactoriaServicios &laFactoria = FactoriaServicios::getInstancia();
 IIntegrador *pEstrategia = laFactoria.MetodoFabricacionEstrategia(elTipoIntegral);
 if (pEstrategia == NULL)
       return (0);

 ContextoIntegrador elIntegrador(pEstrategia);
 double resultado = elIntegrador.calcularIntegral( pFuncionMono,
                              intInf, intSup, diferencial );
 delete pEstrategia;
 return(resultado);

}
```

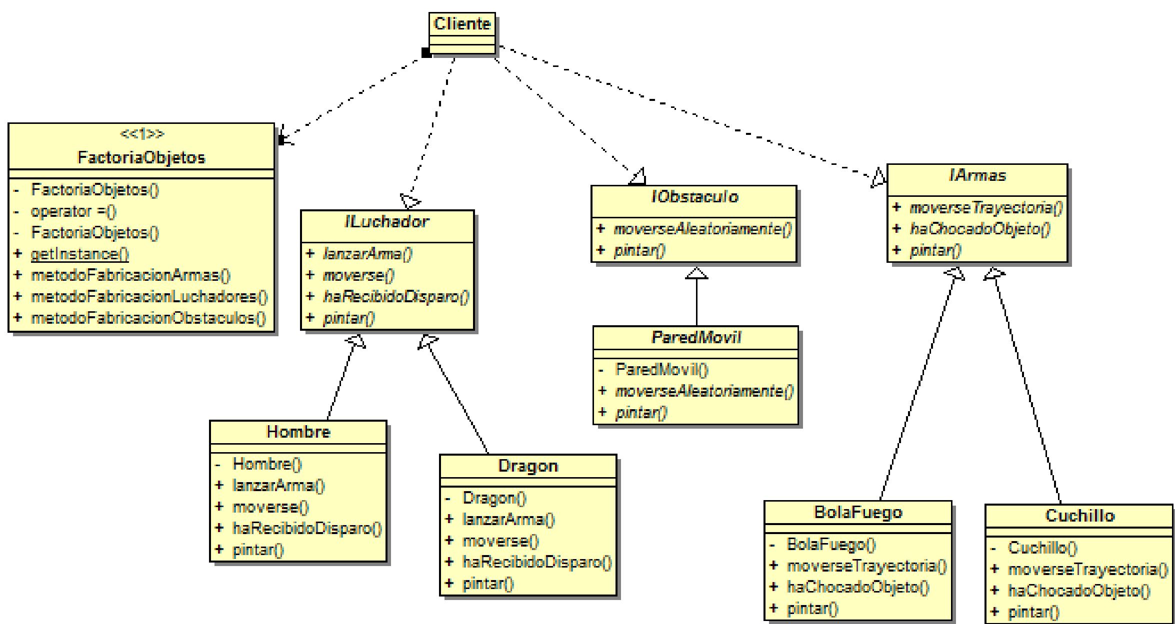
Problema 8

Realizar un juego de batalla los hombres contra los dragones. Los hombres lanzan cuchillos y los dragones bolas de fuego. Los dragones se mueven en el área izquierda de la pantalla y los hombres en el lado derecho. En mitad de la batalla aparecen paredes móviles que se desplazan en el centro de la pantalla. El número de luchadores puede ser variable y dinámico. Se pide:

1. Jerarquía a dos niveles de las características principales.
2. Modelo del dominio.
3. Diagrama de clases de diseño.
4. Implementación en C++ de los ficheros de cabecera de las clases.

1. Video juego de los hombres que luchan contra los dragones
    1.1 Los hombres se mueven en un área restringida de la derecha.
    1.2 Los dragones se mueven en un área restringida de la izquierda.
    1.3 Los hombres lanzan cuchillos que se clavan en la pared o que matan al dragón o que pasan sin hacer daño.
    1.4 Los dragones lanzan bolas de fuego que no pueden atravesar las paredes y que si tocan a un hombre lo mata.
    1.5 Los dragones desaparecen de la pantalla al morir.
    1.6 Los hombres desaparecen de la pantalla al morir.


Luchador
Arma
ParedMovil
Dragón
mata
detiene
detiene
lanza
BolaFuego
Cuchillo
lucha contra
mata
lanza
Hombre
1..n
0..n
1


Cliente
<<1>>
FactoriaObjetos
- FactoriaObjetos()
- operator =()
- FactoriaObjetos()
+ getInstance()
+ metodoFabricacionArmas()
+ metodoFabricacionLuchadores()
+ metodoFabricacionObstaculos()
ILuchador
+ lanzarArma()
+ moverse()
+ haRecibidoDisparo()
+ pintar()
IObstaculo
+ moverseAleatoriamente()
+ pintar()
IArmas
+ moverseTrayectoria()
+ haChocadoObjeto()
+ pintar()
ParedMovil
- ParedMovil()
+ moverseAleatoriamente()
+ pintar()
Hombre
- Hombre()
+ lanzarArma()
+ moverse()
+ haRecibidoDisparo()
+ pintar()
Dragon
- Dragon()
+ lanzarArma()
+ moverse()
+ haRecibidoDisparo()
+ pintar()
BolaFuego
- BolaFuego()
+ moverseTrayectoria()
+ haChocadoObjeto()
+ pintar()
Cuchillo
- Cuchillo()
+ moverseTrayectoria()
+ haChocadoObjeto()
+ pintar()

```
#ifndef _LUCHADORES_INC_
#define _LUCHADORES_INC_

class FactoriaObjetos;

typedef enum{HOMBRE,DRAGON}
TipoLuchadores;

class ILuchador
{
public:
 virtual void lanzarArma() = 0;
 virtual void moverse() = 0;
 virtual bool haRecibidoDisparo() = 0;
 virtual void pintar() = 0;
};

class Hombre : public ILuchador
{
 friend class FactoriaObjetos;
 Hombre();
public:
 virtual void lanzarArma();
 virtual void moverse();
 virtual bool haRecibidoDisparo();
 virtual void pintar();
};

class Dragon : public ILuchador
{
 friend class FactoriaObjetos;
 Dragon();
public:
 virtual void lanzarArma();
 virtual void moverse();
 virtual bool haRecibidoDisparo();
 virtual void pintar();
};

#endif
```

```
#ifndef _ARMAS_INC_
#define _ARMAS_INC_

class FactoriaObjetos;

typedef enum {BOLAFUEGO,CUCHILLO}
TipoArmas;

class IArmas
{
public:
virtual void moverseTrayectoria() = 0;
virtual bool haChocadoObjeto() = 0;
virtual void pintar() = 0;
};

class BolaFuego : public IArmas
{
friend class FactoriaObjetos;
BolaFuego();
 public:
virtual void moverseTrayectoria();
virtual bool haChocadoObjeto();
virtual void pintar();
};

class Cuchillo : public IArmas
{
friend class FactoriaObjetos;
Cuchillo();
public:
virtual void moverseTrayectoria();
virtual bool haChocadoObjeto();
virtual void pintar();
};


#endif
```

```
#ifndef _OBSTACULOS_INC_
#define _OBSTACULOS_INC_

class FactoriaObjetos;

typedef enum {PAREDMOVIL} TipoObstaculos;

class IObstaculo
{
public:
virtual void moverseAleatoriamente() = 0;
virtual void pintar() = 0;
};

class ParedMovil : public IObstaculo
{
friend class FactoriaObjetos;
ParedMovil();
public:
virtual void moverseAleatoriamente() = 0;
virtual void pintar() = 0;
};


#endif
```

```
#ifndef _FACTORIA_INC_
#define _FACTORIA_INC_

#include "Luchadores.h"
#include "Armas.h"
#include "Obstaculos.h"

class FactoriaObjetos {
 FactoriaObjetos(){}; // Para desactivar
 void  operator=(FactoriaObjetos&) {};
 FactoriaObjetos(const FactoriaObjetos&) {};
public:
 static FactoriaObjetos& getInstance() {
 static FactoriaObjetos unicaInstancia;
 return unicaInstancia;
}
IArma* metodoFabricacionArmas (TipoArmas tipo) {
if(tipo == BOLAFUEGO) return new BolaFuego;
      else if(tipo == CUCHILLO) return new Cuchillo;
      else return NULL;}
ILuchador* metodoFabricacionLuchadores (TipoLuchador tipo)
{
      if(tipo == HOMBRE) return new Hombre;
      else if(tipo == DRAGON) return new Dragon;
      else return NULL;}
IObstaculo * metodoFabricacionObstaculos (TipoObstaculos
tipo) {
      if(tipo == PAREDMOVIL) return new ParedMovil;
      else return NULL; }
```

Problema 9

Se tienen las siguientes clases pertenecientes a unas librerías C++, desarrolladas por los fabricantes de impresoras, y que sirven para imprimir un archivo cualquiera en una impresora de un determinado fabricante.

Por tanto, las dos clases anteriores no se pueden modificar, tocar o cambiar. Se desea hacer un programa que permita al usuario teclear el nombre de un archivo, el nombre de la impresora de destino, y que el programa utilice automáticamente la clase de la librería correspondiente. La función main de ese programa (incompleta) seria. Se pide:

1. Diagrama UML de las clases existentes .
2. Diagrama de Clases de Diseño (DCD) de la solución.
3. Explicación (breve, con notas en el anterior diagrama) de los patrones usados.
4. Implementación C++ de la solución propuesta (no olvidar completar el main).

```
class EpsonPrinterDriver
{
public:
      bool Print(char filename[]);
};
class HPControladorImpresora
{
public:
      //este metodo devuelve 1 en caso de exito y -1 en caso de error
      int ImprimeFichero(char* nombre_fichero);
};

int main()
{
      char fichero[255],nombre_impresora[255];

      cout<<"Introduzca en nombre de fichero: ";
      cin>>fichero;
      cout<<"Introduzca nombre impresora: HP o EPSON: ";
      cin>>nombre_impresora;

      Impresora* impresora=NULL;

      //AQUÍ COMPLETAR CODIGO DE CREACION DE LA IMPRESORA ADECUADA
      // en funcion de "nombre_impresora"


      if(impresora==NULL)
      {
            cout<<"Impresora no existe"<<endl;
            return -1;
      }
      if(impresora->Imprime(fichero))
            cout<<"Impresion correcta"<<endl;
      else
            cout<<"Impresion fallida"<<endl;
      return 1;
}
```

Problema 10

Una nueva mejora se propone para el videojuego *Pang* de las prácticas de la asignatura. Se pretende implementar una ÚNICA factoría para la creación compleja de objetos (disparos, esferas, …). Siguiendo el Proceso Unificado, se pide:

1. Ingeniería inversa de la versión actual del videojuego sobre las clases de los disparos.
2. DCD de la nueva mejora, indicando los patrones que se emplean.
3. Ficheros de cabecera en C++ de la nueva versión.
4. Utilice las nuevas prestaciones en el servicio Mundo::Tecla(), sabiendo que *hombre.GetNumBonus()* retorna el número de *bonus*. Si tiene más de uno se generará lanzas, con un *bonus* se creará ganchos especiales y sin *bonus* se construirán ganchos.

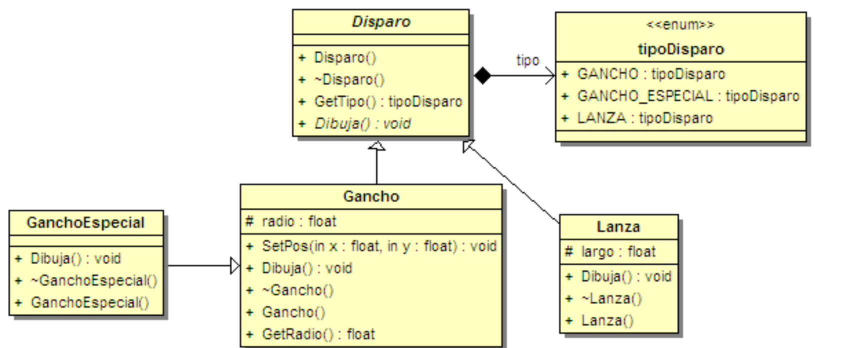
```
#if !defined(_DISPARO_INC_)
#define _DISPARO_INC_

typedef
enum{GANCHO,GANCHO_ESPECIAL,LANZA}
tipoDisparo;

class Disparo
{
public:
	Disparo();
	virtual ~Disparo();

	tipoDisparo GetTipo();
	virtual void Dibuja()=0;

protected:
	tipoDisparo tipo;
};

#endif
```

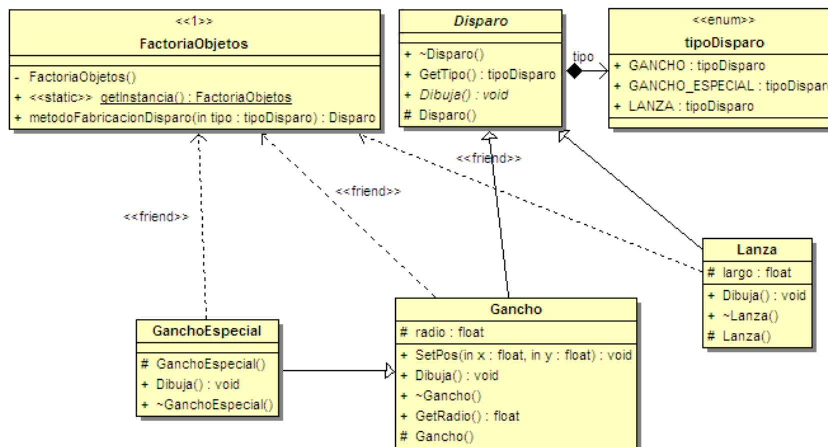
```
#if !defined(_LANZA_INC_)
#define _LANZA_INC_


#include "Disparo.h"

class Lanza : public Disparo
{
public:
	void Dibuja();
	Lanza();
	virtual ~Lanza();

protected:
	float largo;

};

#endif
```

```
#if !defined(_GANCHO_INC_)
#define _GANCHO_INC_

#include "Disparo.h"

class Gancho : public Disparo
{
public:
   void SetPos(float x, float y);
   void Dibuja();
   Gancho();
   virtual ~Gancho();
   float GetRadio(){return radio;}

protected:
	float radio;

};

#endif
```

```
#if
!defined(_GANCHO_ESPECIAL_INC_)
#define _GANCHO_ESPECIAL_INC_

#include "Gancho.h"

class GanchoEspecial : public
Gancho
{
public:
	void Dibuja();
	GanchoEspecial();
	virtual ~GanchoEspecial();

};

#endif
```

```
void Mundo::Tecla(unsigned char key)
{
	switch(key)
	{
		case ' ': if(disparos.GetNumero()<MAX_DISPAROS)
		{
		 Disparo* d=Factoria::CrearDisparo(hombre);
		 disparos.Agregar(d);
		}
		break;
	}
}
```





Se ha empleado los siguientes patrones: Singleton (GoF), Método de Fabricación(GoF) y Factoría Abstracta(GoF).

```
#if !defined(_DISPARO_INC_)
#define _DISPARO_INC_

typedef
enum{GANCHO,GANCHO_ESPECIAL,LANZA}
tipoDisparo;

class Disparo
{
public:
	virtual ~Disparo();

	tipoDisparo GetTipo();
	virtual void Dibuja()=0;

protected:
	tipoDisparo tipo;
	Disparo();
};

#endif
```

```
#if !defined(_LANZA_INC_)
#define _LANZA_INC_


#include "Disparo.h"

class Lanza : public Disparo
{
public:
	void Dibuja();
	virtual ~Lanza();

protected:
 float largo;
 friend class FactoriaObjetos;
 Lanza();

};

#endif
```

```
#if !defined(_GANCHO_INC_)
#define _GANCHO_INC_

#include "Disparo.h"

class Gancho : public Disparo
{
public:
   void SetPos(float x, float y);
   void Dibuja();
   virtual ~Gancho();
   float GetRadio(){return radio;}

protected:
 float radio;
 Gancho();
friend class FactoriaObjetos;

};

#endif
```

```
#if
!defined(_GANCHO_ESPECIAL_INC_)
#define _GANCHO_ESPECIAL_INC_

#include "Gancho.h"

class GanchoEspecial : public
Gancho
{
public:
	void Dibuja();
	virtual ~GanchoEspecial();

protected:
 GanchoEspecial();
friend class FactoriaObjetos;

};

#endif
```

```
#if !defined(_FACTORIA_INC_)
#define _FACTORIA_INC_
#include "GanchoEspecial.h"
#include "Lanza.h"

class FactoriaObjetos
{
	FactoriaObjetos() {}
public:
	static FactoriaObjetos &getInstancia() {
		static FactoriaObjetos laFactoria;
		return (laFactoria);
	}
	Disparo * metodoFabricacionDisparo(tipoDisparo tipo) {
		if(tipo == GANCHO) return new Gancho;
		else if(tipo == GANCHO_ESPECIAL) return new
				GanchoEspecial;
		else if(tipo == LANZA) return new Lanza;
		else return 0;
	}
};
#endif
```

Derecho de Autor © 2014 Carlos Platero Dueñas.

Permiso para copiar, distribuir y/o modificar este documento bajo los términos de la Licencia de Documentación Libre GNU, Versión 1.1 o cualquier otra versión posterior publicada por la Free Software Foundation; sin secciones invariantes, sin texto de la Cubierta Frontal, así como el texto de la Cubierta Posterior. Una copia de la licencia es incluida en la sección titulada "Licencia de Documentación Libre GNU".

La Licencia de documentación libre GNU (GNU Free Documentation License) es una licencia con ***copyleft*** para contenidos abiertos. Todos los contenidos de estos apuntes están cubiertos por esta licencia. La version 1.1 se encuentra en http://www.gnu.org/copyleft/fdl.html. La traducción (no oficial) al castellano de la versión 1.1 se encuentra en http://www.es.gnu.org/Licencias/fdles.html