6 Diseño orientado a objetos

En las etapas de captura de los requisitos y del análisis orientado a objetos se han centrado en aprender a realizar la definición del proyecto sin decir cómo. En esta otra etapa se pondrá el énfasis en implantar las especificaciones con eficiencia y fiabilidad.

En el Proceso Unificado, UP, por cada iteración, tendrá lugar una transacción desde un enfoque centrado en los requisitos, a un enfoque centrado en el diseño y en la implementación.

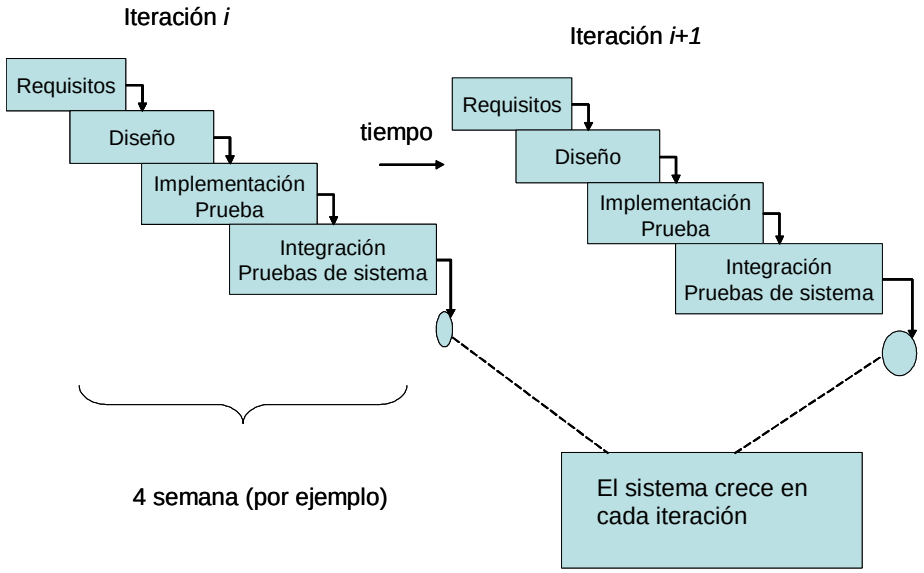


Figura 6. 1 evolución del proyecto por iteraciones

El diseño orientado a objetos requiere tener conocimientos en:

- Los principios de asignación de responsabilidades.
- Los usos de los patrones de diseño.

Para el desarrollo de las técnicas de diseño se emplearán los diagramas de interacción y los diagramas de clase de diseño, DCD. Ambos artefactos pertenecen a la disciplina UP de Modelado del Diseño.

Este capítulo se organiza en tres apartados. El primero tratará sobre las bases del diseño y de la implementación, para luego pasar a entrar de lleno en el diseño con patrones. Los apartados segundo y tercero se estudiarán los patrones GRASP y GoF respectivamente.

6.1 De los diagramas de clase de diseño a la implementación

Los diagramas de clase de diseño, DCD, se crean en paralelo con los diagramas de interacción. En los DCD se encuentran reflejados:

- Las clases, asociaciones y atributos.
- Los patrones.
- Los interfaces, con sus operaciones y constantes.
- Los métodos o servicios.
- La información acerca del tipo de atributos.
- La navegabilidad.
- Las dependencias.

A diferencia de las clases conceptuales[1] del AOO, las clases de diseño de los DCD muestran las esencias de las futuras clases implementadas o de software.

El primer paso para crear un DCD es identificar aquellas clases que participan en la solución del paquete a diseñar. Se pueden encontrarlas examinando el modelo del dominio, donde algunas clases conceptuales pueden ser tomadas como clases de diseño. También pueden ser localizadas en los diagramas de interacción y listando las clases que se mencionan.

[1] Clases conceptuales ≡ abstracciones de conceptos del mundo real

El siguiente paso es dibujar un diagrama de clases e incluir los atributos que se identificaron previamente en el modelo del dominio que también se utilizan en el diseño.

En cuanto a los servicios, éstos se pueden identificar analizando los diagramas de interacción y observando los nombres de los mensajes mandados entre los objetos.

El mensaje *"create()"* es una forma de independizar UML de los lenguajes de programación. En C++ implica la asignación dinámica con el operador *new*, seguido de una llamada al constructor.

Los métodos de acceso a la información, capaces de recuperar *(getX())* o de establecer *(setX())* los valores de atributo, son definidos automáticamente o manualmente. En algunos lenguajes, como en *Java*, es un estilo común tener un *get()* y un *set()* por cada atributo y declarar todos los atributos privados.

Un mensaje a un multiobjeto se interpreta como un mensaje al propio objeto contenedor/colección. Normalmente, estas interfaces o clases de contenedores/colecciones son elementos de las librerías predefinidas y no es útil mostrar explícitamente estas clases en el DCD.

Los tipos de los atributos, los argumentos de los servicios y los valores de retorno se podrían mostrar. La cuestión sobre si se muestra o no esta información se debe de considerar en el siguiente contexto:

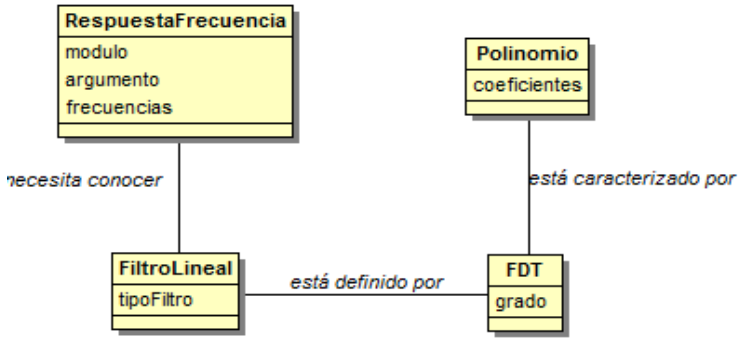
> *"Si se emplea alguna herramienta CASE con generación automática del código, son necesarios todos los detalles. Si se hace para que lo lean los desarrolladores, los detalles podrían influir negativamente por el ruido visual que produce tanta información en los DCD."*

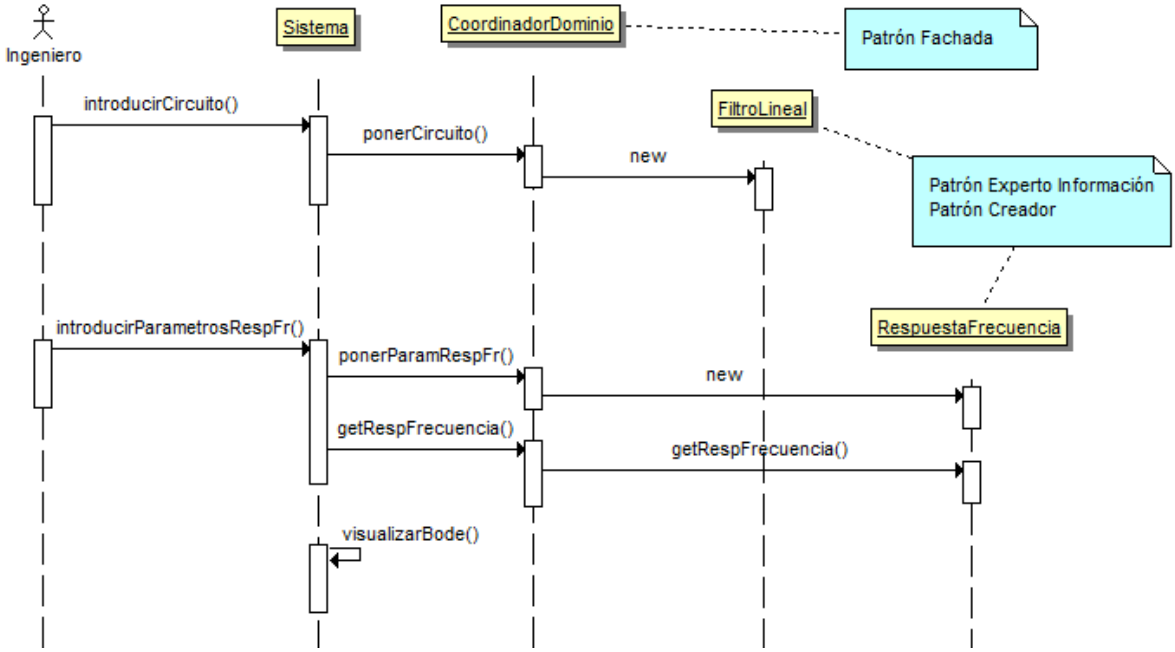
En el Modelo del Dominio sólo se expresaba relaciones semánticas entre las clases conceptuales, intentando definir un diccionario visual del problema. Por el contrario, en DCD se eligen las asociaciones de acuerdo al criterio de necesito conocer. Es en el DCD donde tiene sentido emplear toda la rica notación de UML sobre relaciones entre clases de diseño.

La visibilidad y las asociaciones entre clases de diseño se dan a conocer en el DCD. Para su determinación se ayudara con los diagramas de interacción.

Ejemplo 6.1

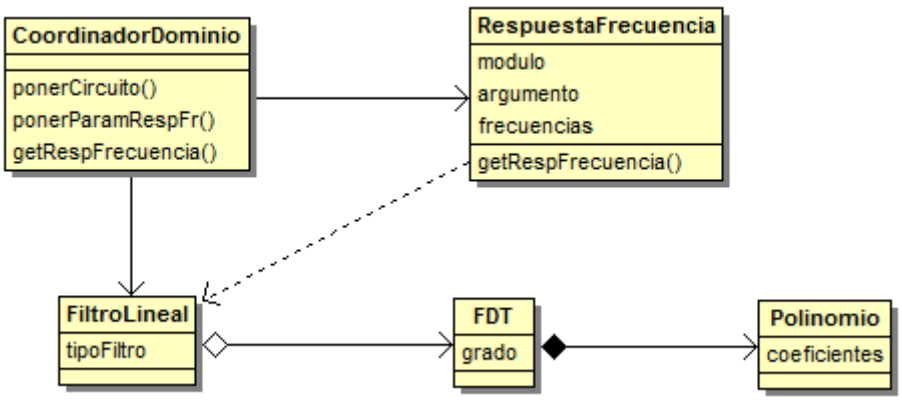
Determinar el DCD del paquete del dominio de *RespuestaFrecuencia* considerando el modelo del dominio y los diagramas de interacción, obtenidos de anteriores ejercicios.





Del diagrama de interacciones se observa que el coordinador del paquete del dominio recibe dos mensajes (*ponerCircuito() y ponerParamResFr()*). Del primero se observa que se creará el objeto filtro con sus características que según el modelo del dominio será la FDT de un sistema tipo LTI-SISO, constituyéndolo con dos polinomios, uno en el numerador y otro en el denominador. Por tanto, son asociaciones de necesito conocer y relaciones del todo con las partes. Se les pondrá flechas de navegación y de tipo de agregación y composición. No se ha elegido, para este caso, relaciones de generalización, por que es más robusto la composición que la herencia. Nótese que en las clases conceptuales no se ha colocado ninguna relación jerárquica.

Cuando se recibe la información sobre los parámetros de respuesta en frecuencia, el coordinador se lo pasará a RespuestaFrecuencia y éste deberá de calcular el Bode. La conexión entre RespuestaFrecuencia y FiltroLineal, necesaria en el cálculo de Bode, se realizará por argumento en este servicio, generando una relación de dependencia.



6.1.1 Determinación de la visibilidad

La visibilidad es la capacidad de un objeto de tener una referencia a otro objeto. Para que un objeto emisor envíe un mensaje a un objeto receptor[2], el receptor debe ser visible al emisor. Hay cuatro formas comunes de alcanzar la visibilidad:

1. Visibilidad de atributo: El emisor tiene entre sus atributos al receptor. Es una visibilidad permanente porque persiste mientras existan el emisor y el receptor. Ésta es una visibilidad muy común en AOO/D. Se usa el estereotipo <<*association*>> para definir esta visibilidad en UML.

2. Visibilidad de argumento: El objeto-atributo es pasado como argumento en un servicio solicitado a otro objeto. La visibilidad de parámetro desde el emisor al receptor existe cuando el emisor pasa como un parámetro el atributo al receptor. Es una visibilidad relativamente temporal. Es la segunda forma más común en AOO/D. Es habitual transformar la visibilidad de argumento en visibilidad local. El estereotipo UML para su identificación es <<*parameter*>>.

3. Visibilidad local: El receptor es declarado dentro de algún servicio del emisor. La visibilidad local, desde el emisor al receptor, existe cuando el receptor se declara como un objeto local en un método del emisor. Es una visibilidad relativamente temporal. Es la tercera forma de visibilidad más común en POO. Se declara con el estereotipo <<*local*>>.

4. Visibilidad global: Cuando el receptor es un objeto global y puede ser manejado por cualquier objeto de la aplicación. Es una visibilidad permanente en el tiempo y es la forma menos común de visibilidad. El método para conseguir visibilidad global es utilizar el patrón *Singleton* [GoF]. Se declara con el estereotipo <<*global*>>.

[2] Recuerde que el objeto receptor es el que realiza la operación.

La navegabilidad implica visibilidad, normalmente, visibilidad del atributo. Las asociaciones en DCD deberían adornarse con las flechas de navegación necesarias. Es aquí donde tiene más sentido destacar las distintas relaciones que se establecen en UML. Así, por ejemplo, en los diagramas de clase de diseño, la relación de dependencia es útil para describir la visibilidad entre clases que no son de atributos, esto es, para declarar una visibilidad de parámetro, local o global.

6.1.2 Modelo de implementación

Una vez finalizado los DCD se dispone de los suficientes detalles para generar el código de la capa del dominio de los objetos.

Los artefactos UP creados durante el trabajo de diseño -diagramas de interacción y los DCDs- se utilizarán como entradas en el proceso de generación de código.

En UP se define el Modelo de Implementación. Éste contiene los artefactos de implementación como el código fuente, las definiciones de bases de datos, las páginas XML/HTML, etc.

Una ventaja del AOO/D y la POO, cuando se utiliza UP, es que proporciona una guía de principio a fin, esto es, se presenta un conjunto de artefactos, procedimientos y técnicas que van desde los requisitos hasta la generación del código.

Durante el trabajo de diseño se tomaron algunas decisiones. Ahora, en la fase de implementación, no es una etapa de generación de código trivial, más bien lo contrario. En realidad, los resultados generados durante el diseño son un primer paso incompleto. Durante la fase de producción del código aparecerán nuevas cuestiones que habrán de resolverse *in situ*. Es una tarea que costará mucho tiempo, pero mucho menos que si no se hubiera puesto esfuerzo en la captura de los requisitos, en el análisis y en el diseño. Producir código sin realizar estos estudios es algo que es improductivo, tedioso y extremadamente peligroso.

Después de haber acabado una iteración de UP, es deseable, para la siguiente vuelta del ciclo, que los diagramas generados se actualicen de manera semiautomática con el trabajo surgido de la implementación. Éste es un aspecto de la ingeniería inversa (ver ejemplo 6.2). El código producido hará actualizar el modelo definido en el Proceso Unificado.

6.1.3 Transformación del diseño al código

La transformación en un lenguaje OO requiere la escritura del código fuente para:

- La definición de las clases e interfaces.
- Las definiciones de los métodos o servicios.

Para la creación de las definiciones de las clases se emplearán básicamente los DCD. Las definiciones de los métodos y de los atributos simples son inmediatas de obtener a partir de los DCD y de los diagramas de interacción. Sin embargo, para los

atributos complejos se emplearán atributos de referencia. Los atributos de referencia se deducen de las asociaciones y de la navegabilidad de los diagramas de clases. Los atributos de referencia de una clase a menudo están implícitos, en lugar de explícitos. Cuando el atributo es simple, por ejemplo, un carácter, éste está de forma explícita en la definición de la clase; pero cuando es complejo, como por ejemplo una frase, se utiliza una referencia a una instancia de ese atributo complejo.

En este curso se ha ignorado el manejo de las excepciones en el desarrollo de la solución. De hecho no se plantea la inserción de código para el control de excepciones. Sin embargo, habrá que contemplarlas en la producción de código industrial.

El método para traducir los diagramas de clases de diseño y los diagramas interacción a código se basará en el procedimiento *"eXtreme Programming"*, XP. Se basa en escribir el código de pruebas antes que el código de producción. La secuencia es escribir un poco de código de prueba, luego escribir un poco de código de producción, hacer las pruebas y cuando éste se hayan superado, entonces se escribirá más código de prueba y más de producción y así sucesivamente. Se empezará por implementar desde las clases menos acopladas a las más acopladas.

Ejemplo 6.2

Implementar la aplicación Respuesta en Frecuencia v0.0.0

Primero se realizará el código de prueba, utilizando el diagrama de secuencia del sistema (DSS), los contratos de operación y los diagramas de interacción empleados anteriormente. Por tanto, se implementará la función *main()* y la clase *Vista*:

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

#include <iostream>
using namespace std;

#include "../Dominio/CoordinadorFrecELAI.h"

class VistaFrecuenciaELAI
{
        tipoFiltro elTipo;
        float resistencia;
        float condensador;
        float frecInicial, frecFinal, frecIntervalo;
        CoordinadorFrecELAI elCoordinador;

public:
        void introducirCircuito(void);
        void introducirParametrosRespFr(void);
}; /*VistaFrecuencia.h*/
```

```
// De: "Apuntes de Sistemas Informáticos Industriales" Carlos Platero.
// Ver permisos en licencia de GPL
#include "../../include/Vista/VistaFrecuenciaELAI.h"
void VistaFrecuenciaELAI::introducirCircuito(void)
{
	cout << "Elegir entre:\n1.Filtro paso bajo primer orden.\n2.Filtro paso alto";
	int eleccion;
	cin >> eleccion;
	elTipo = eleccion == 1 ? LF_1 : HF_1;
	cout << "\nValor de la resistencia: ";
	cin     >> resistencia;
	cout << "\nValor del condensador: ";
	cin >> condensador;
	elCoordinador.ponerCircuito(elTipo,resistencia,condensador);
}
void VistaFrecuenciaELAI::introducirParametrosRespFr(void)
{
	cout << "\nCual es la frecuencia inicial [Hz]: ";
	cin >> frecInicial;
	cout << "\nCual es la frecuencia final [Hz]: ";
	cin >> frecFinal;
	cout << "\nCual es el intervalo empleado para el cálculo [Hz]: ";
	cin >> frecIntervalo;
	elCoordinador.ponerParamResFr(frecInicial,frecFinal,frecIntervalo);

	//Visualizar los resultados
	std::vector<double> elVectorModulo;
	std::vector<double>::iterator iteradorModulo;
	elCoordinador.getModuloRespFr(elVectorModulo);
	iteradorModulo = elVectorModulo.begin();
	for (unsigned i =0; i<elVectorModulo.size(); i++)
		std::cout<<*(iteradorModulo+i)<<std::endl;
	cout << "Pulsar cualquier tecla para finalizar";
}
void main(void)
{
	VistaFrecuenciaELAI laVista;
	laVista.introducirCircuito();
	laVista.introducirParametrosRespFr();
}
```

El siguiente paso será escribir el código de las clases menos acopladas a las que más lo están. Se implementarán por el siguiente orden: Polinomio, FDT, FiltroLTI, RespuestaFrecuencia y Coordinador:

```
// De: "Apuntes de Sistemas Informáticos Industriales" Carlos Platero.
// Ver permisos en licencia de GPL
#ifndef _POLINOMIO_INC_
#define _POLINOMIO_INC_
#include <vector>

class Polinomio
{
	std::vector<double> coeficientes;
public:
	Polinomio(){}
	Polinomio(unsigned grado, double *pCoef)
	{
	  for (unsigned i=0;
	       i<=grado;coeficientes.push_back(*(pCoef+i)),i++);
	}
	double getCoeficiente(unsigned n)
	{return( coeficientes[n]));}
};

#endif /*Polinomio.h*/
```

```
// "Apuntes de Sistemas Informáticos Industriales" Carlos Platero.
// Ver permisos en licencia de GPL
#ifndef _FDT_INC_
#define _FDT_INC_
#include "Polinomio.h"
class FDT
{
        unsigned grado;
        Polinomio numerador;
        Polinomio denominador;
public:
        FDT(unsigned n, double *pNum, double *pDen):
          grado(n),numerador(n,pNum),denominador(n,pDen){}
        unsigned getGrado(void){return grado;}
        double getCoefNum(unsigned n)
        {return n<=grado ? numerador.getCoeficiente(n) : 0;}
        double getCoefDen(unsigned n)
        {return n<=grado ? denominador.getCoeficiente(n) : 0;}
};

#endif /*FDT.h*/
```

```
#ifndef _FILTRO_LINEAL_INC_
#define _FILTRO_LINEAL_INC_
#include "FDT.h"
typedef enum{LF_1,HF_1} tipoFiltro;

class FiltroLineal
{
        tipoFiltro elTipo;
        FDT *pFDT;
public:
        FiltroLineal(tipoFiltro, float, float);
        unsigned getGradoFiltro(void){return pFDT->getGrado();}
        double getCoefNum(unsigned n)
        {return pFDT != NULL ? pFDT->getCoefNum(n) : 0;}
        double getCoefDen(unsigned n)
        {return pFDT != NULL ? pFDT->getCoefDen(n) : 0;}
        ~FiltroLineal(){if(pFDT) delete pFDT;}
};

#endif /*FiltroLineal.h*/
```

```
// "Apuntes de Sistemas Informáticos Industriales" Carlos Platero.
// Ver permisos en licencia de GPL
#ifndef _RESPFREC_INC_
#define _RESPFREC_INC_
#include <vector>
#include "FiltroLineal.h"
class RespuestaFrecuencia
{
        float freInicio, freFinal, freIntervalo;
        std::vector<double> modulo;
        std::vector<double> argumento;
        double calcularModulo(float,FiltroLineal *);
        double calcularArgumento(float,FiltroLineal *);
public:
        RespuestaFrecuencia(float,float,float,FiltroLineal *);
        float getFrInicio(void){return freInicio;}
        float getFrFinal(void){return freFinal;}
        float getFrIntervalo(void){return freIntervalo;}
        void getModuloRespFr(std::vector<double> &elVectorModulo)
         {elVectorModulo = modulo;}
};

#endif /*RespuestaFrecuencia.h*/
```

```
// De: "Apuntes de Informática Industrial" Carlos Platero.
// Ver permisos en licencia de GPL
#ifndef _COORDINFRECELAI_INC_
#define _COORDINFRECELAI_INC_
#include "FiltroLineal.h"
#include "RespuestaFrecuencia.h"

class CoordinadorFrecELAI
{
	FiltroLineal *pFiltro;
	RespuestaFrecuencia *pRespFr;

public:
	int ponerCircuito(tipoFiltro ,float , float );
	int ponerParamResFr(float,float,float);
	int getModuloRespFr(std::vector<double> &);
	~CoordinadorFrecELAI()
		{if(pFiltro) delete pFiltro; if(pRespFr) delete pRespFr;}

};
```

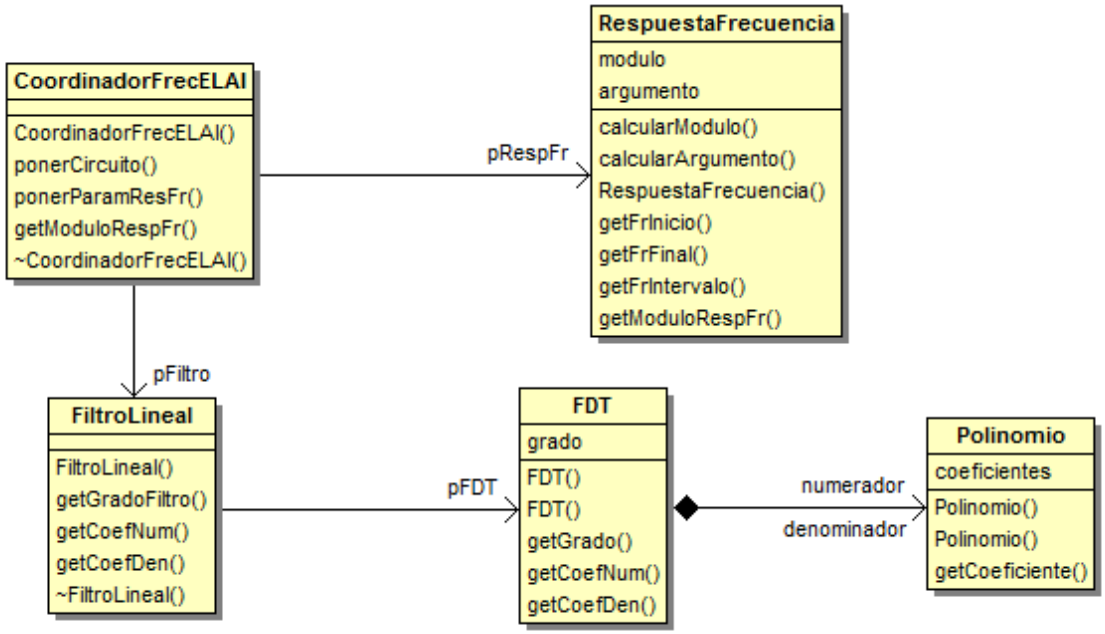
Las algoritmias de los métodos serán implementados en los fuentes de las clases. También de la menos acopladas a la de más acoplamiento. Se usa un código de test para visualizarlo en consola.

```
// Ver permisos en licencia de GPL
#include "../../include/Dominio/FiltroLineal.h"
FiltroLineal::FiltroLineal(tipoFiltro tipo, float resistencia, float condensador)
{
	elTipo = tipo;
	double numerador[2]; double denominador[2];
	if (elTipo == LF_1) {
		numerador[0]=1; numerador[1]=0;
	}
	else{
		numerador[0]=0; numerador[1]=resistencia*condensador;
	}
	denominador[0]= 1;denominador[1]=resistencia*condensador;
	pFDT = new FDT(1,numerador,denominador);
}
```

```
// De: "Apuntes de Sistemas Informáticos Industriales" Carlos Platero.
// Ver permisos en licencia de GPL
#include "../../include/Dominio/RespuestaFrecuencia.h"
#include <iostream>
#include <math.h>
#define PI     3.1416
#define PRUEBA_BODE
#ifdef PRUEBA_BODE
#include <iostream>
#endif
RespuestaFrecuencia::RespuestaFrecuencia(float frInicio, float frFin,float
frIntervalo,FiltroLineal *pFiltro)
{
	for (float f=frInicio; f< frFin; f+=frIntervalo)
	{
	modulo.push_back(this->calcularModulo(f,pFiltro));
	//argumento.push_back(this->calcularArgumento(f,pFiltro));
	//A implementar
	}
#ifdef PRUEBA_BODE
	this->iteradorModulo = modulo.begin();
	for (unsigned i =0; i<modulo.size(); i++)
		std::cout<<*(iteradorModulo+i)<<std::endl;
#endif
}

double RespuestaFrecuencia::calcularModulo(float frecuencia,FiltroLineal *pFiltro)
{
	//...
}
```

```
// De: "Apuntes de Sistemas Informáticos Industriales" Carlos Platero.
// Ver permisos en licencia de GPL
#include "../../include/Dominio/CoordinadorFrecELAI.h"
int CoordinadorFrecELAI::ponerCircuito(tipoFiltro elTipo,float resistencia,
float condensador)
{
        pFiltro = new FiltroLineal(elTipo,resistencia,condensador);
        return(0);
}
int CoordinadorFrecELAI::ponerParamResFr(float frInicio,float frFinal,
                                                        float frIntervalo)
{
        if (pFiltro == NULL) return (-1);
        pRespFr = new RespuestaFrecuencia(frInicio,frFinal,frIntervalo,pFiltro);
        return(0);
}
int CoordinadorFrecELAI::getModuloRespFr(std::vector<double> &elVectorModulo)
{
        if (pRespFr == NULL) return (-1);
        pRespFr->getModuloRespFr(elVectorModulo);
        if(pFiltro) delete pFiltro;
        if(pRespFr) delete pRespFr;
        return (0);
}
```

Una vez depurada la aplicación se procederá a aplicar ingeniería inversa para obtener el nuevo DCD y pasar a la siguiente iteración UP:



6.2 Diseño de objetos con responsabilidad

Para el diseño OO se requiere tener conocimientos en:

- Los principios de asignación de responsabilidades
- Patrones de diseño

Para su desarrollo se emplearán los diagramas de interacción y los diagramas de clase, ambos artefactos forman parte del Modelo del Diseño.

Las responsabilidades están relacionadas con las obligaciones de un objeto en su comportamiento. Estas obligaciones son de dos tipos: a) los objetos deben saber qué información manejan (conocer) y b) las cosas qué deben de hacer.

Las responsabilidades se asignan a las clases de los objetos durante la etapa de diseño. Una responsabilidad no es lo mismo que un método o servicio, pero los métodos se implementan para llevar a cabo las responsabilidades. El objetivo de este capítulo es aplicar sistemáticamente los principios fundamentales para asignar responsabilidades a los objetos.

6.2.1 Patrones

En la tecnología de objetos, un patrón es una descripción de un problema y su solución; a la que se le da un nombre y se puede aplicar a nuevos contextos. Son guías sobre el modo en el que debería asignarse las responsabilidades a los objetos.

Resumiendo:

1. La asignación habilidosa de responsabilidad es extremadamente importante en el diseño de objetos.
2. La decisión acerca de la asignación de responsabilidades tiene lugar durante la creación de los diagramas de interacción, de los DCD y, posteriormente, en la programación.
3. Los patrones son pares problemas/solución con un nombre que codifican buenos consejos y principios relacionados, con frecuencia, con la asignación de responsabilidades.

Dos tipos de patrones se explicarán: los patrones GRASP y los GoF. GRASP es el acrónimo de *General Responsibility Assignment Software Patterns*. Se tratarán los patrones: Experto en Información, Creador, Alta Cohesión, Bajo Acoplamiento, Controlador, Polimorfismo, Indirección, Fabricación Pura y Variaciones Protegidas.

Mientras GoF es la abreviatura de *Gangs of Four*, de los que se tratarán los patrones: Adaptador, Factoría, Singleton, Estrategia, Composición y Observador.

6.3 Patrones GRASP

6.3.1 Experto en Información

Problema: ¿Cuál es el principio general para asignar responsabilidades?

Solución: Asignar la responsabilidad al que tenga la información.

El patrón Experto indica qué hacen los objetos con la información que contienen. Sucede muchas veces que la información está dispersa por diferentes clases de objetos. Esto implica que hay muchos expertos con información "parcial" que colaboran en la tarea, mediante el paso de mensajes para compartir el trabajo. Por ejemplo, se han visto varios ejemplos que cuando una tarea llega a un objeto, el trabajo es distribuido a otros objetos asociados.

Para asignar la responsabilidad se emplearán las clases del DCD. En la primera iteración se utilizará el Modelo del Dominio, en versiones posteriores se consultaran los resultados de la ingeniería inversa de las iteraciones anteriores. La idea es ir ampliando o actualizando las nuevas clases del diseño.

Si se empieza el trabajo del diseño se buscará los expertos en información en el Modelo del Dominio. Hay que hacer una tabla de responsabilidades generando las primeras clases de diseño.

Ejemplo 6.3

Realizar una tabla de responsabilidad sobre la aplicación RespuestaFrecuencia

Clase de diseño	Información	Responsabilidad
Filtro	Tiene la FDT del filtro y el tipo de filtro	Definir matemáticamente la estructura del filtro
RespuestaFrecuencia	Los parámetros de frecuencia	Aplicar los algoritmos para calcular el Bode

En algunos casos, el Experto en Información genera problemas de acoplamiento y cohesión. Este problema se pone de manifiesto cuando el patrón Experto une los datos del dominio, con los datos de la presentación. Por ejemplo, en una hoja de cálculo, el Experto uniría los datos de la base de datos con la presentación en barras gráficas. El patrón Experto se equivoca. Siempre hay que mantener separada la lógica de la aplicación de la lógica de la base de datos o la lógica del dominio de la vista.

Beneficios del patrón experto:

- Mantiene el encapsulamiento de la información, puesto que los objetos utilizan su propia información para llevar a cabo las tareas.

- Se distribuye el trabajo entre clases, haciéndolas más cohesivas y ligeras, lo que conlleva a que sean más fáciles de entender y de mantener.

Este patrón también se conoce como: "Colocar la responsabilidad con los datos", "Eso que conoces, hazlo", "Hacerlo yo mismo", "Colocar los servicios con los atributos con los que trabaja".

6.3.2 Creador

Problema: ¿Quién debería ser el responsable de la creación de una nueva instancia de una clase?

Solución: Asignar a la clase *B* la responsabilidad de crear una instancia de clase *A* si se cumple uno o más de los siguientes casos:

- B contiene objetos de A

- B se asocia con objetos de A

- B registra instancias de objetos de A

- B utiliza más estrechamente objetos de A

- B tiene datos de inicialización que se pasarán a un objeto de A

El patrón creador está relacionado con la asociación y especialmente con la agregación y la composición (relación del *Todo* con las *Parte*s).

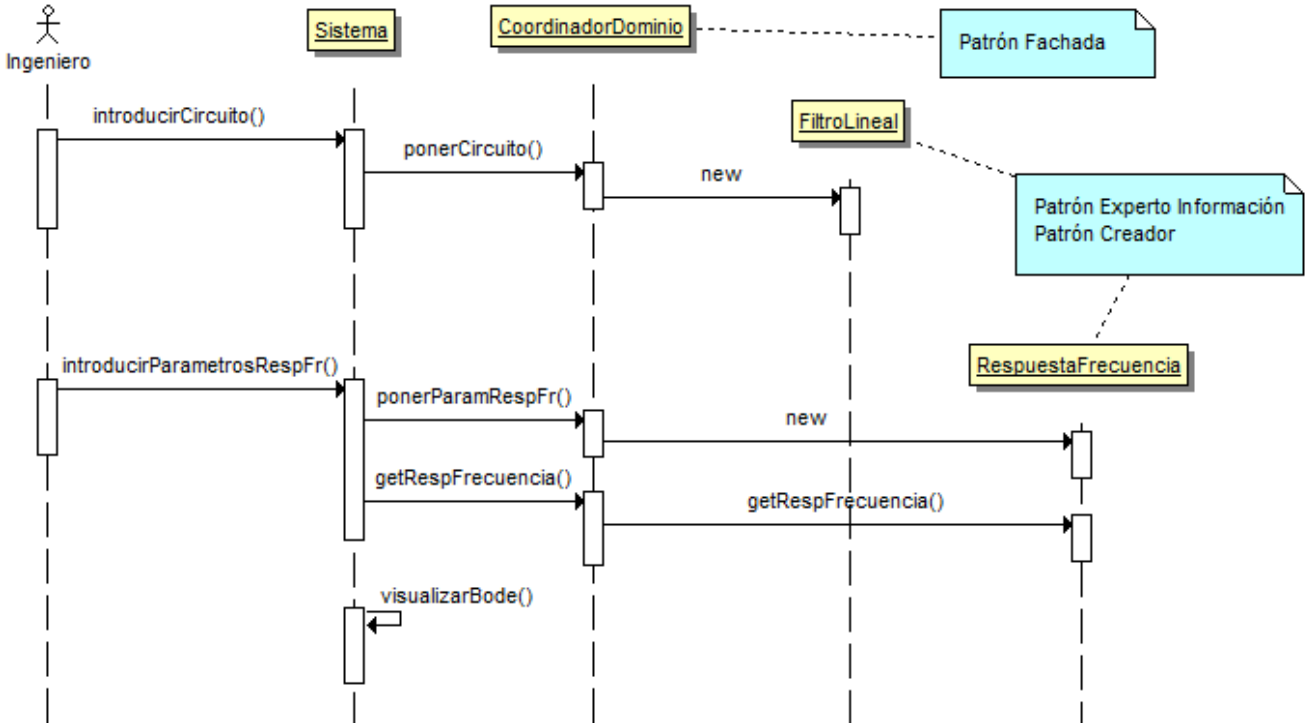
Estas responsabilidades se asignarán durante la elaboración de los diagramas de interacción. A veces se encuentra el creador buscando las clases que tienen los datos de inicialización que se pasará durante la creación.

A menudo, la creación requiere una complejidad significativa, como utilizar instancias recicladas por motivos de rendimientos o crear instancias de forma

condicional. En estos casos es aconsejable delegar la creación a una clase auxiliar denominada Factoría. Esta clase deriva del patrón Factoría (GoF) que se verá más adelante.

Ejemplo 6.4

Utilizar el patrón creador en la aplicación RespuestaFrecuencia




```
// De: "Apuntes de Informática Industrial" Carlos Platero.
// Ver permisos en licencia de GPL
#include "../../include/Coordinador/CoordinadorFrecELAI.h"

int CoordinadorFrecELAI::ponerCircuito(tipoFiltro elTipo,float resistencia,
float condensador)
{
        pFiltro = new FiltroLineal(elTipo,resistencia,condensador);
        return(0);
}
int CoordinadorFrecELAI::ponerParamResFr(float frInicio,float frFinal,
                                                        float frIntervalo)
{
        if (pFiltro == NULL) return (-1);
        pRespFr = new RespuestaFrecuencia(frInicio,frFinal,frIntervalo,pFiltro);
        return(0);
}
int CoordinadorFrecELAI::getModuloRespFr(std::vector<double> &elVectorModulo)
{
        if (pRespFr == NULL) return (-1);
        pRespFr->getModuloRespFr(elVectorModulo);
        if(pFiltro) delete pFiltro;
        if(pRespFr) delete pRespFr;
        return (0);
}
```

6.3.3 Alta Cohesión

Problema: ¿Cómo mantener la complejidad manejable?

Solución: Asignar una responsabilidad de manera que la cohesión permanezca alta.

En AOO/D, la cohesión es una medida de la fuerza con la que se relacionan los elementos de un conjunto o paquete y del grado de focalización de sus responsabilidades. Un elemento, concepción genérica de UML, de alta responsabilidad y que no hace gran cantidad de trabajo, tiene alta cohesión. Estos elementos pueden ser clases, paquetes, subsistemas, etc.

Una clase con baja cohesión hace muchas cosas no relacionadas o tareas relacionadas pero con mucho trabajo. Las clases de baja cohesión adolecen de los siguientes problemas:

- Difíciles de entender.
- Difíciles de reutilizar.
- Difíciles de mantener.
- Delicadas, constantemente afectadas por los cambios.

A menudo las clases con baja cohesión representan bien un grado grande de abstracción o bien se les han asignado demasiadas responsabilidades que deberían de haberse delegado en otras clases.

Se establece que existe alta cohesión funcional cuando los elementos de un componente "trabajan todos juntos para proporcionar algún comportamiento bien delimitado".

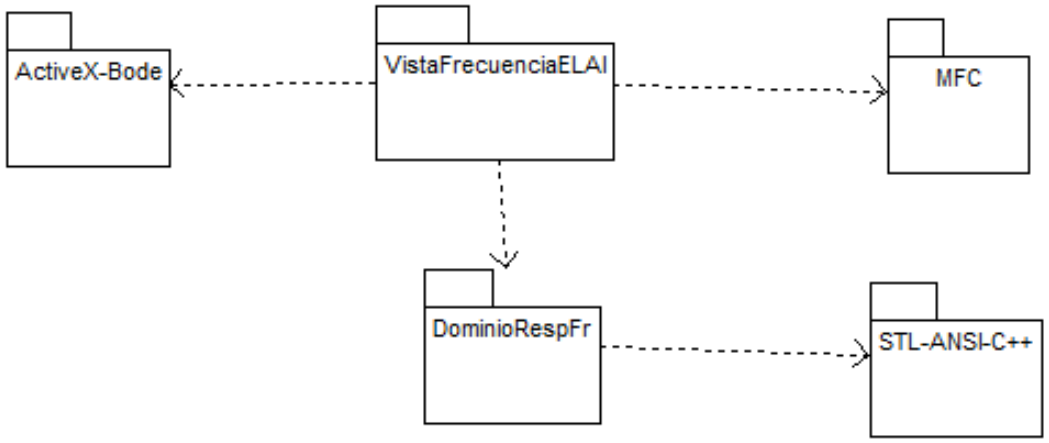
Como regla empírica, una clase con alta cohesión tiene un número relativamente pequeño de métodos, con funcionalidad altamente relacionada y no realiza mucho trabajo. En el caso de que la tarea sea extensa, colaborará con otros objetos para compartir el esfuerzo.

El Bajo Acoplamiento y la Alta Cohesión son viejos principios del diseño SW. Otro de estos principios es promover el diseño modular. La modularilidad es la propiedad del sistema de haberse descompuesto en un conjunto de módulos cohesivos y débilmente acoplados. En UML se emplea la vista de gestión del proyecto para la aplicación de la modularidad. Con un doble motivo: a) organización de las tareas entre los desarrolladores que van a participar en el proyecto y b) diseño de componentes altamente cohesivas y con bajo acoplamiento.

Ejemplo 6.5

Visión arquitectónica o de gestión de la aplicación de Respuesta en Frecuencia.

Los paquetes se deben de diseñar de forma altamente cohesiva y con bajo acoplamiento. Esta tarea también servirá para la organización del trabajo entre los desarrolladores de la aplicación.



En la práctica, el nivel de cohesión no se puede considerar de manera aislada a otras responsabilidades y a otros principios como son los patrones Experto y Bajo Acoplamiento.

Beneficios:

- Se incrementa la claridad y facilita la comprensión del diseño
- Se simplifica el mantenimiento y las mejoras
- Se soporta a menudo bajo acoplamiento
- El grano fino de funcionalidad altamente relacionada incrementa la reutilización. Un paquete o clase altamente cohesiva puede ser aplicado en otro contexto.

6.3.4 Bajo Acoplamiento

Pregunta: ¿Cómo soportar el bajo impacto del cambio e incrementar la reutilización?

Solución: Asignar una responsabilidad de manera que el acoplamiento permanezca bajo.

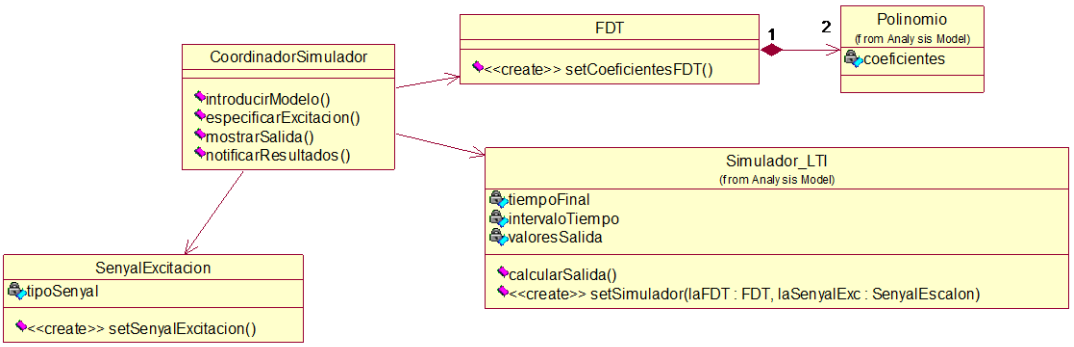
El acoplamiento es una medida de la fuerza con que un elemento está conectado a otros elementos.

Una clase con alto acoplamiento confía en muchas otras clases. Tales clases suelen ser no deseables. Estas clases adolecen de los siguientes problemas:

- Son difíciles de mantener de manera aislada.
- Los cambios en estas clases fuerzan cambios locales.
- Son difíciles de reutilizar.

Ejemplo 6.6

Un simulador de sistemas LTI-SISO requiere para su definición la FDT del sistema. Por tanto, se puede emplear las clases de FDT y Polinomio que se han definido en la Respuesta en Frecuencia para esta otra aplicación. Una definición de clases altamente cohesivas y de bajo acoplamiento muestra su facilidad de reutilización.



En general, las clases que son muy genéricas y con una alta probabilidad de reutilización alta, deberían de tener un acoplamiento especialmente bajo. Por ejemplo, en el anterior ejercicio, se ha colocado las clases *Polinomio* y *FDT* para una aplicación de Simulación que habían sido definidas en Respuesta en Frecuencia. Ambas se caracterizan por un Bajo Acoplamiento.

No suele ser problema el acoplamiento alto entre objetos estables y elementos de generalización. Se entiende como objetos estables aquellos que provienen de las librerías estándar, tales como las STL o el uso de *frameworks* como las *MFC* o las *Qt*.

6.3.5 Controlador

Problema: ¿Quién debe ser el responsable de gestionar un evento de entrada al sistema?

Solución: Asignar la responsabilidad a una clase que represente una de las siguientes opciones:

- Representa el sistema global, dispositivo o subsistema. Se le llamará Controlador de Fachada.

- Representa un escenario de caso de uso. A menudo se denominan Coordinador o Manejador o Sesión acompañado con el nombre del caso de uso. Utilice la misma clase controlador para todos los eventos del sistema en el mismo escenario de caso de uso.

Un controlador es un objeto que no pertenece al interfaz o vista, responsable de recibir o manejar los eventos del sistema. Un controlador define un método para cada operación del sistema. Sus servicios pueden ser establecidos a partir del Diagrama de Secuencia del Sistema, DSS, o de los contratos de operación.

El controlador es una especie de fachada del paquete que recibe los eventos externos y organiza las tareas. No sólo el interfaz genera eventos, también puede hacerlo el tiempo, si es una aplicación en tiempo real. Otro caso son las aplicaciones de control de procesos; los sensores y/o dispositivos generan interrupciones que se deben de atender. Cada uno de estos eventos debería ser mandado al correspondiente Coordinador.

Un error típico del diseño de los controladores es otorgarles demasiadas responsabilidades. Normalmente, un controlador debería delegar en otros objetos el trabajo que se necesita hacer, coordinar o controlar la actividad. No realiza mucho trabajo por sí mismo.

La primera categoría de controlador es el controlador de fachada que representa al sistema global, dispositivo o subsistema. Los controladores de fachada son adecuados cuando no existen "demasiados" eventos del sistema. Si se elige un controlador de casos de uso, entonces hay un controlador diferente para cada caso de uso.

Antiguamente se empleaban los conceptos de objetos *frontera*, objetos *entidad* y objetos *control*. Precisamente los objetos control eran los manejadores de los casos de uso y que se describen en este patrón. Las otras dos categorías pertenecían a los que se relacionaban con la vista (*frontera*) y a los objetos del dominio (*entidad*).

Resumiendo, el controlador recibe la solicitud del servicio desde una capa superior y coordina su realización, normalmente delegando a otros objetos, aumentando el potencial para reutilizar. Se asegura que la lógica de la aplicación no se maneja en la capa interfaz. También el controlador o coordinador sirve para analizar la *"máquina de estado"* del sistema o de caso del uso, según sea el tipo.

Un corolario importante del patrón Controlador es que los objetos interfaz **no deberían ser responsables de manejar los eventos del sistema**. Este patrón incrementa el potencial de reutilización.

El patrón Controlador crea un objeto artificial que no procede del análisis del dominio. Se dice que es una Fabricación Pura, detalle que se analizará más adelante. La implementación del Controlador hace uso de los patrones GRASP de Fabricación Pura y de Indirección.

Signos de un controlador saturado:

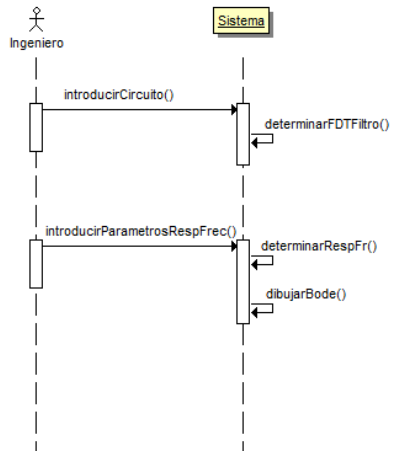
- Existe una única clase controlador que recibe todos los eventos del sistema.
- El propio controlador realiza muchas de las tareas necesarias para llevar a cabo los eventos del sistema, sin delegar trabajo. Lleva a la ruptura de los patrones del Experto y Alta cohesión.
- Un controlador tiene muchos atributos y mantiene información significativa sobre el sistema o el dominio.

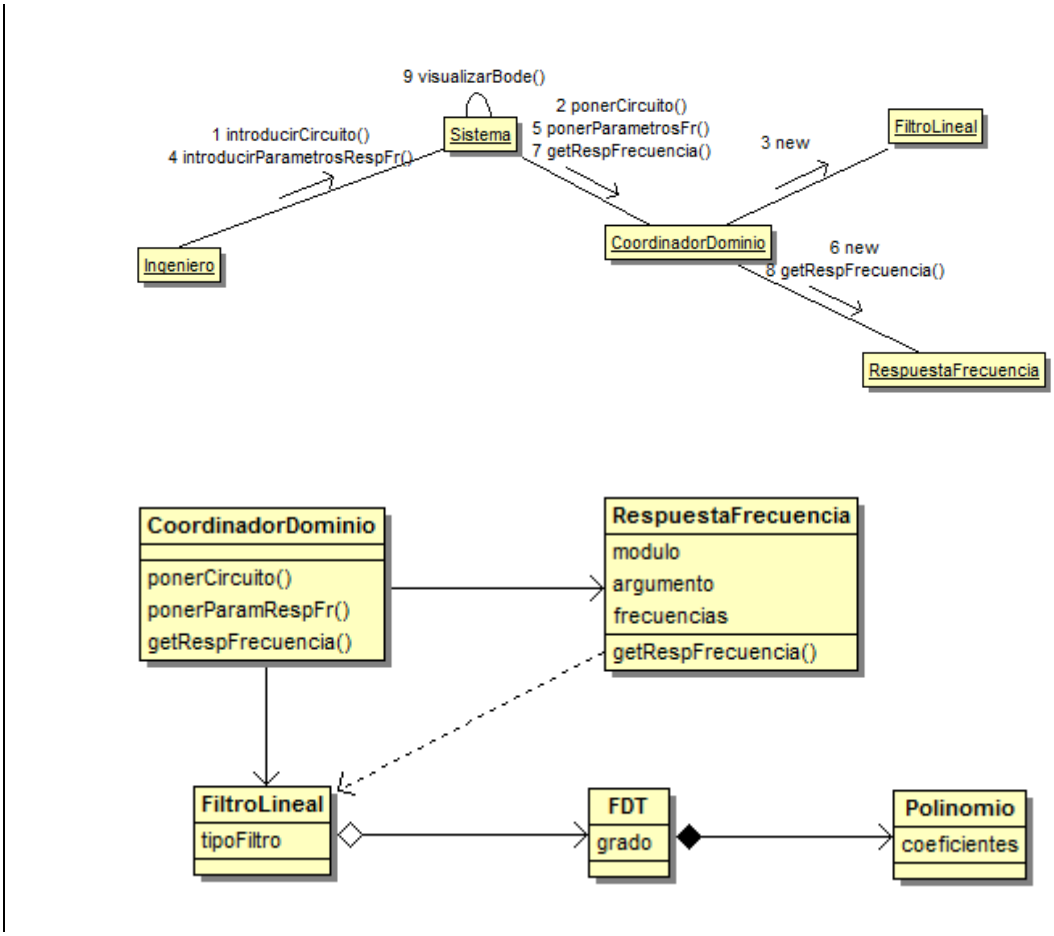
Remedios:

1. Añadir más controladores.
2. Delegar las tareas a otros objetos.
3. Este patrón está relacionado con el patrón Comand [GoF], Fachada [GoF], Capas [POSA] y Fabricación Pura [GRASP].

Ejemplo 6.7

Emplear un Coordinador para la aplicación de Respuesta en Frecuencia que organice las tareas del escenario de caso de uso y que delegue las tareas.






```
// De: "Apuntes de Informática Industrial" Carlos Platero.
// Ver permisos en licencia de GPL
#include "../../include/Dominio/CoordinadorFrecELAI.h"
int CoordinadorFrecELAI::ponerCircuito(tipoFiltro elTipo,float resistencia,
float condensador)
{
        pFiltro = new FiltroLineal(elTipo,resistencia,condensador);
        return(0);
}
int CoordinadorFrecELAI::ponerParamResFr(float frInicio,float frFinal,
                                                        float frIntervalo)
{
        if (pFiltro == NULL) return (-1);
        pRespFr = new RespuestaFrecuencia(frInicio,frFinal,frIntervalo,pFiltro);
        return(0);
}
int CoordinadorFrecELAI::getModuloRespFr(std::vector<double> &elVectorModulo)
{
        if (pRespFr == NULL) return (-1);
        pRespFr->getModuloRespFr(elVectorModulo);
        if(pFiltro) delete pFiltro;
        if(pRespFr) delete pRespFr;
        return (0);
}
```

6.3.6 Polimorfismo

Problema: ¿Cómo manejar las alternativas basadas en tipo?, ¿Cómo crear componentes software conectables (*pluggable*)? o ¿Cómo se puede sustituir un componente servidor por otro, sin afectar al cliente?

Solución: Cuando las alternativas o comportamientos relacionados varían según los tipos de los datos se debe de asignar la responsabilidad utilizando operaciones polimórficas, de forma que varíe el comportamiento según el tipo. No hay que realizar comprobaciones acerca del tipo del objeto. No se requiere emplear la lógica condicional para llevar a cabo alternativas diferentes basadas en el tipo.

Si en un diseño se emplea las sentencias lógicas de bifurcación *if-else* o *switch-case*, cada nueva variación requiere la modificación de esta lógica. Este enfoque dificulta que el programa se extienda con facilidad.

El polimorfismo trata de asignar el mismo nombre de servicio pero con diferentes objetos. El polimorfismo es un principio fundamental para designar cómo se organiza el sistema para gestionar variaciones similares. Según el polimorfismo un diseño basado en la asignación de responsabilidades puede extenderse fácilmente para manejar nuevas variaciones.

Los desarrolladores diseñan sistemas con interfaces y polimorfismos para futuras necesidades frente a posibles variaciones desconocidas. Esta forma de actuar se analizará en el patrón de Variaciones Protegidas.

Beneficios:

- Se añaden fácilmente las extensiones necesarias para nuevas variaciones.
- Las nuevas implementaciones se pueden introducir sin afectar a los clientes.

6.3.7 Indirección

Problema: ¿Donde asignar una responsabilidad, para evitar el acoplamiento directo entre dos o más lógicas de la aplicación?, ¿Cómo desacoplar los objetos de manera que se soporte el Bajo Acoplamiento y el potencial de reutilización permanezca alto?.

Solución: Asignar la responsabilidad a un objeto intermedio entre dos o más elementos o paquetes de manera que no se acoplen directamente.

Algunos patrones como Adaptador (GoF), Controlador (GRASP), Observador (GoF) y muchas Fabricaciones Puras se generan debido a la Indirección. El motivo de la Indirección normalmente es el Bajo Acoplamiento y la Alta Cohesión. Se añade un intermediario para desacoplar los servicios.

Beneficios:

- Disminuir el acoplamiento entre componentes y/o paquetes

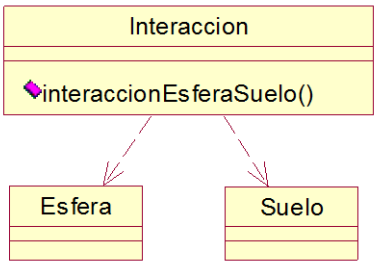
Patrones relacionados: Variaciones Protegidas, Bajo Acoplamiento, muchos patrones GoF.

La mayoría de los intermediarios de Indirección son Fabricaciones Puras.

Ejemplo 6.9

En los programas de simulación donde los objetos cambian de dinámica al chocar con otros objetos, ¿cómo se asignarían las responsabilidades?. Supóngase que se desea simular cómo una esfera en caída libre es arrojada desde una cierta altura respecto al suelo.

Suelo y Esfera son clases conceptuales y por tanto candidatas a ser clases de diseño. Para evitar el acoplamiento entre ambas clases se añade un grado de indirección. Se creará una clase interacción que resuelva la responsabilidad de la interacción entre las instancias de las dos clases.



6.3.8 Fabricación Pura

Problema: ¿Qué objetos deberían de tener las responsabilidades cuando no se quiere romper los objetivos de Alta Cohesión y Bajo Acoplamiento, pero las soluciones que ofrece el Experto no son adecuadas?

Solución: Asignar responsabilidades altamente cohesivas a una clase artificial o de conveniencia que no represente un concepto del dominio del problema. Algo inventado para soportar Alta Cohesión, Bajo Acoplamiento y Reutilización.

El diseño de objetos se puede dividir, en general, en dos grandes grupos

1. Los escogidos de acuerdo a una descomposición de la representación.

2. Los seleccionados según una descomposición del comportamiento.

La descomposición en representación se emplea ya que favorece el objetivo de salto en la representación reducida. En la descomposición por comportamiento se asigna responsabilidades agrupando comportamientos sin estar relacionado con un concepto del dominio del mundo real. En el caso de la aplicación Respuesta en Frecuencia, las clases *Polinomio, FDT, FiltroLineal, RespuestaFrecuencia* están basadas en una

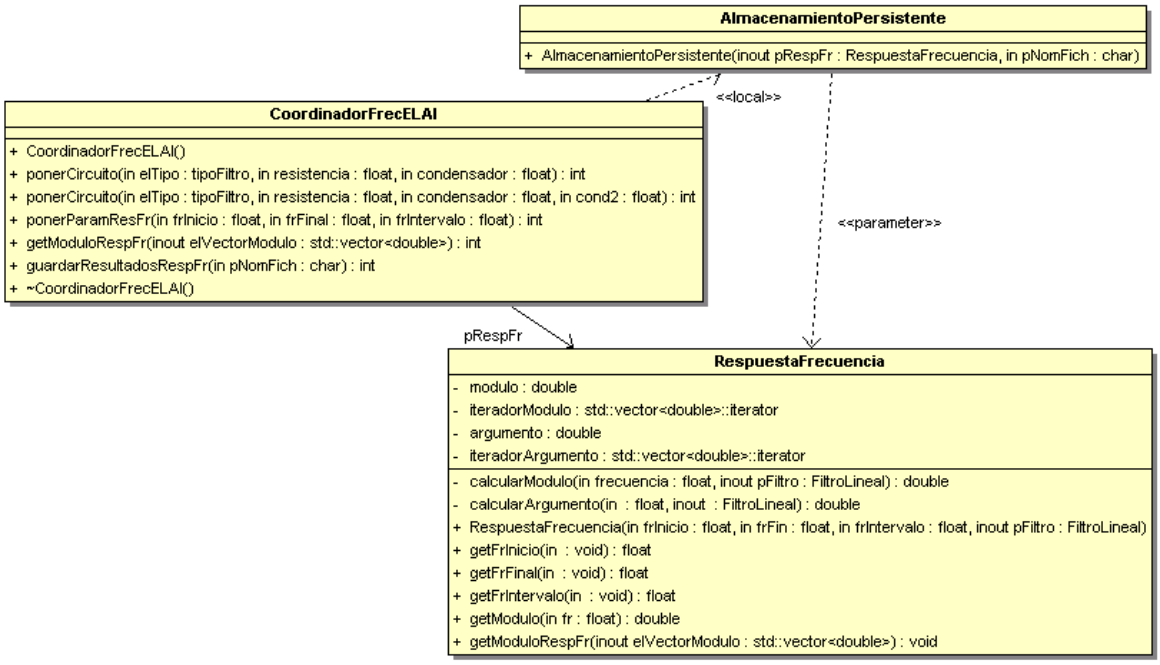
descomposición de la representación del universo del problema. En cambio, la clase *CoordinadorRespuestaFrELAI* está tomada como una descomposición del comportamiento.

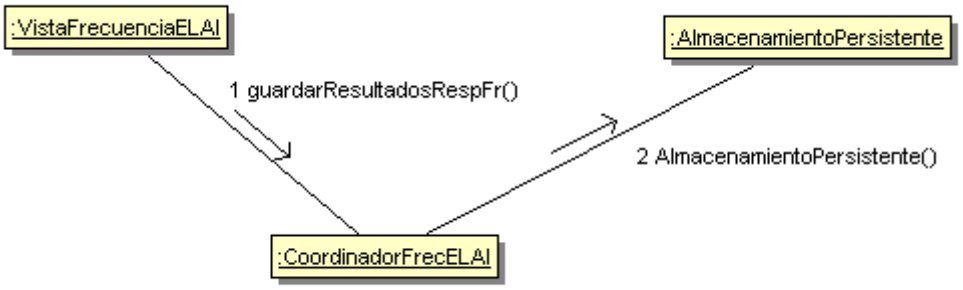
Véase el siguiente problema: el almacenamiento de instancias en una base de datos. El Experto en Información decidiría que estuviera donde están los datos. Sin embargo, rompería los patrones de Alta Cohesión y Bajo Acoplamiento, ya que quiebra la separación de la lógica del problema de la lógica de la base de datos. La solución es crear una nueva clase que sea capaz de almacenar los objetos en algún tipo de almacenamiento persistente; al que se llamará Almacenamiento Persistente. Esta clase no es parte del dominio, sino algo creado artificialmente o fabricado para facilitar las cosas al desarrollar el software.

Ejemplo 6.10

Guardar la información de la respuesta en frecuencia en un fichero.

Para mantener alta la cohesión y no romper la lógica del dominio con el de la base de datos se emplea una Fabricación Pura. Empleando un grado de Indirección se introduce la clase *AlmacenamientoPersistente*. Ésta se encargará de guardar la información en disco. También habrá que añadir este nuevo servicio al Coordinador.






```
// De: "Apuntes de Sistemas Informáticos Industriales" Carlos Platero.
// Ver permisos en licencia de GPL
#include "../../include/Dominio/AlmacenamientoPersistente.h"

AlmacenamientoPersistente::AlmacenamientoPersistente(RespuestaFrecuencia *pRespFr,
                                                     const char * pNomFich)
{
        ofstream os(pNomFich);
        os <<"Modulo de la respuesta en frecuencia"<<endl;
        float fr;
        for (fr = pRespFr->getFrInicio();fr <= pRespFr->getFrFinal();
                fr+=pRespFr->getFrIntervalo())
                 os << fr << " :" << pRespFr->getModulo(fr)<<endl;
}

int CoordinadorFrecELAI::guardarResultadosRespFr(const char *pNomFich)
{
        if (pRespFr == NULL) return (-1);
        AlmacenamientoPersistente elAlmacen(this->pRespFr,pNomFich);
        return (0);
}
```

Muchos de los patrones del DOO que se van a ver son ejemplos de Fabricación Pura: Adaptador, Estrategia, Observador, etc. También lo es el Controlador (GRASP) o Fachada (GoF).

Tiene como beneficio el Bajo Acoplamiento y la Alta Cohesión. Usualmente, una Fabricación Pura asume responsabilidades de las clases del dominio a las que se les asignaría esas responsabilidades en base al patrón Experto; pero que no se las da, debido a que disminuiría en cohesión y aumentaría la dependencia.

La Fabricación Pura emplea el patrón de Indirección, al asignar a una clase artificial o de comportamiento, las responsabilidades de una clase del dominio para evitar el acoplamiento y mantener alta la cohesión.

Ejemplo de Fabricación Pura:

- Guardar información en una base de datos.

- Separar la lógica del dominio de la vista (Observador-GoF).

- Coordinador o fachada.

6.3.9 Variaciones Protegidas

Problema: ¿Cómo diseñar objetos, subsistemas y sistemas de manera que las variaciones e inestabilidades en estos elementos no tengan un impacto negativo en otros elementos?

Solución: Identifique los puntos de variaciones previstas e inestabilidad; asigne responsabilidades para crear una interfaz estable alrededor de ellos. Añadiendo Indirección, Polimorfismo y una interfaz se consigue un sistema de Variaciones Protegidas, VP. Las distintas implementaciones del componente y/o paquete ocultan las variaciones internas a los sistemas clientes de éste. Dentro del componente, los objetos internos colaboran en sus tareas con una interfaz estable.

El principal objetivo de este patrón es proteger a los clientes de las variaciones de mejoras de los servicios dados por el componente servidor. Para tal fin se define los puntos calientes y a éstos se les cubre con una interfaz estable, permitiendo variar el componente sin interferir en las aplicaciones clientes.

Hay que distinguir dos tipos de variaciones:

- Puntos de variación: variaciones en el sistema actual.

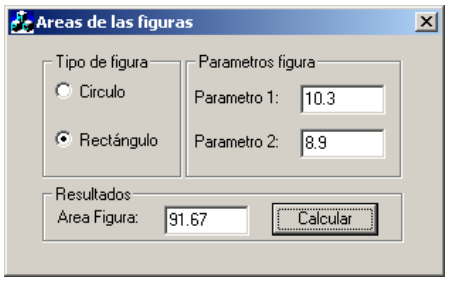
- Puntos de evolución: puntos especulativos de variación que podrían aparecer en el futuro, pero que no están presentes en los requisitos actuales.

La aplicación de Variaciones Protegidas tiene un esfuerzo de diseño que siempre hay que considerar. Si la necesidad de flexibilidad y protección de cambios es realista, entonces está motivada la aplicación de VP. Un diseño debe ser un compromiso entre el coste de cambio y su probabilidad.

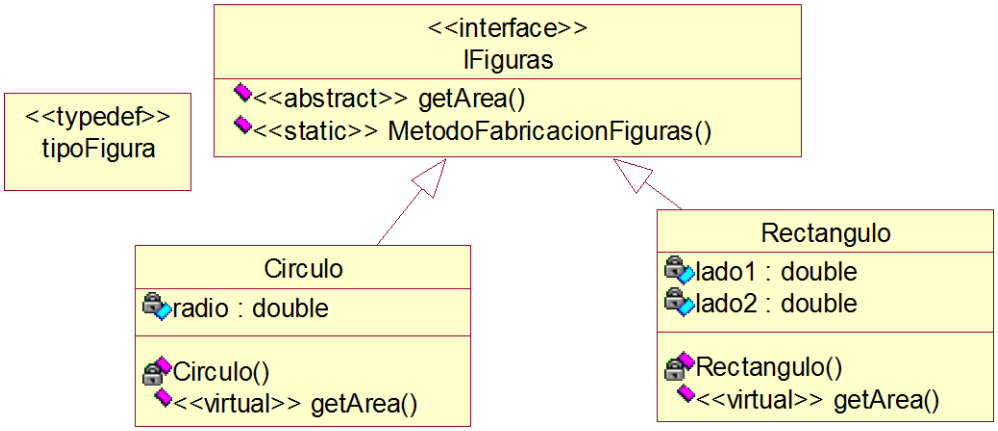
La mayoría de los patrones y principios de diseño son mecanismos para Variaciones Protegidas, entre los que se encuentran: Polimorfismo, Indirección, Encapsulamiento y la mayoría de los patrones GoF.

Ejemplo 6.11

Realizar una aplicación que calcule el área de figuras geométricas. En esta primera versión sólo se considera círculos y rectángulos.



Debido a que el programa debe de crecer, el concepto de Figura debe ser aplicado de forma genérica (interfaz). Se utiliza un punto de variación. De otro lado, la subclase concreta se cargará dependiendo de la elección del usuario. Se empleará un Método de Fabricación GoF (se verá más adelante).



```
#ifndef _AREAS_FIGURA_INC_
#define _AREAS_FIGURA_INC_
typedef enum tipoFig {CIRCULO, RECTANGULO} tipoFigura;
class IFiguras
{
public:
        virtual double getArea() = 0;
        static IFiguras* MetodoFabricacionFiguras
                (tipoFigura, double, double);
};
class Circulo: public IFiguras
{
        double radio;
        friend class IFiguras;
        Circulo(double param1):radio(param1) {}
public:
        virtual double getArea()
        {return (3.1416*radio*radio);}
};

class Rectangulo: public IFiguras
{
        double lado1;
        double lado2;
        friend class IFiguras;
        Rectangulo(double param1, double param2):
        lado1(param1),lado2(param2) {}
public:
        virtual double getArea()
        {return (lado1*lado2);}
};

#endif
```

```
IFiguras* IFiguras::MetodoFabricacionFiguras(tipoFigura elTipo,
                                             double param1,double param2 = 0)
{
        if (elTipo == CIRCULO) return new Circulo(param1);
        else if(elTipo == RECTANGULO) return new Rectangulo(param1,param2);
        else return 0;

}//////////////////////////////////////////////////////////////////////////

void CAreasFiguraDlg::OnCalcular()
{
        UpdateData(TRUE);
        IFiguras *pFigura= IFiguras::MetodoFabricacionFiguras(
                this->m_Figura == true ? CIRCULO : RECTANGULO,
                this->m_Param1,this->m_Param2);

        this->m_Area = pFigura->getArea();
        delete pFigura;
        UpdateData(FALSE);

}
```

Otra aplicación de Variaciones Protegidas está en los intérpretes de líneas de comando (*script)*. El cliente tiene su sintaxis que se mantiene aunque varíe el servidor. Hay muchos sistemas que ofrecen una línea de comandos para interactuar con él. Por ejemplo, se podría pensar en *Matlab*. Es conocido que los comandos de las versiones

anteriores se pueden usar en las nuevas. El cliente las utiliza pero no sabe si éstas han sido mejoradas. Más aun, los desarrolladores confían en estos servicios y crean aplicaciones, de más alto nivel, basadas en ellas.

Ejemplos de Variaciones Protegidas

- Las máquinas virtuales son ejemplos complejos de Indirección para conseguir VP.
- Lectura y escritura de datos de sistemas externos.
- Diseños dirigidos por un intérprete.

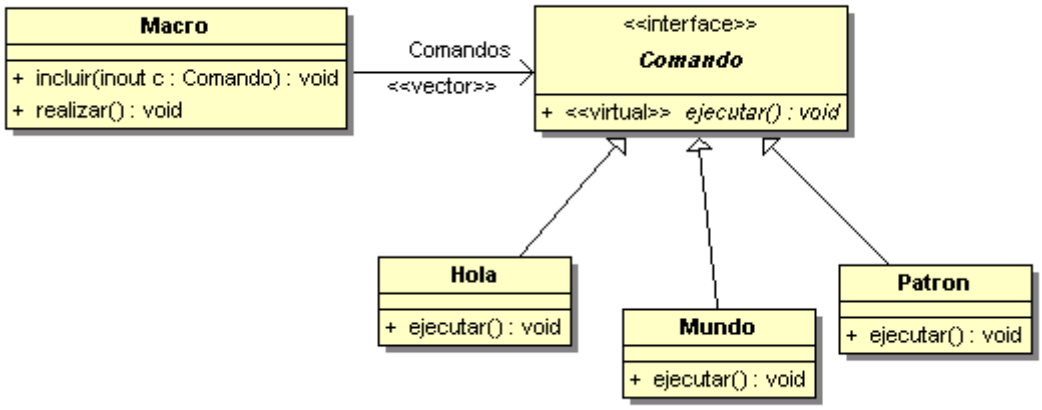
Ejemplo 6.12

El código entregado corresponde con la implementación del patrón comando, de manera que encapsula un objeto y el cliente lo ve como si fuese una función (muy utilizado en lenguajes script). Se pide:

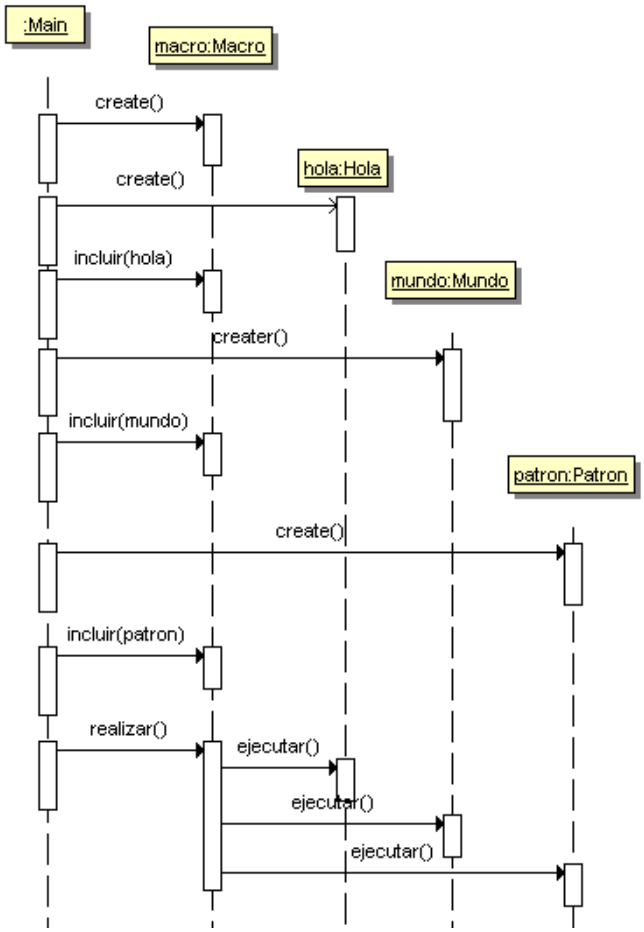
1. Ingeniería inversa: Diagrama de clases.
2. Ingeniería inversa: Diagrama de secuencias de la función *main().*
3. Resultado de su ejecución en la consola.
4. Indicar los patrones GRASP empleados en este patrón.
5. Diseñar e implementar la clase Saludo, de manera que se despida al añadirse al macro.

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

class Comando
{
public:
  virtual void ejecutar() = 0;
};

class Hola : public Comando
{
public:
  void ejecutar() { cout << "Hola "; }
};

class Mundo : public Comando
{
public:
  void ejecutar() { cout << "Mundo! "; }
};

class Patron : public Comando
{
public:
  void ejecutar() { cout << "Soy el comando patron!"; }
};


class Macro
{
  vector<Comando*> Comandos;
public:
  void incluir(Comando* c) { Comandos.push_back(c); }
  void realizar() {
    for(int i=0;i<Comandos.size();i++)
      Comandos[i]->ejecutar();
  }
};

int main()
{
  Macro macro;
  macro.incluir(new Hola);
  macro.incluir(new Mundo);
  macro.incluir(new Patron);
  macro.realizar();
}
```

1.



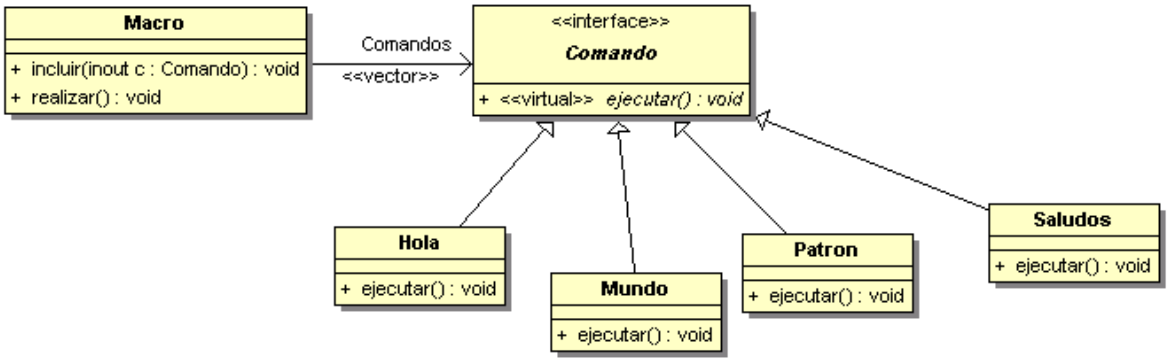
2.



3. **Hola Mundo! Soy el comando patron!**

4. El patrón Comando emplea Variaciones Protegidas (GRASP), de forma que el cliente no ve las modificaciones que está realizando el servidor.

5.




```
class Saludo : public Comando
{
public:
  void ejecutar() { cout << " Un saludo. "; }
};
```

La sostenibilidad depende de las VP. Estas aplicaciones se basan en el principio de sustitución de Liskov:

"*El software que hace referencia a un tipo T debería de trabajar correctamente con cualquier implementación o subclase T que la sustituya*".

Por otro lado, uno de los patrones antiguos GRASP era "*No hable con Extraños"* o Ley de Demeter. Este patrón incide en evitar crear diseños que recorran largos caminos de la estructura de los objetos. No se podía enviar mensajes a objetos distantes, indirectos o extraños. Tales diseño son frágiles con respecto a los cambios en las estructuras de los objetos.

No hable con Extraños establecía que un método, sólo, debería enviar mensajes a los siguientes objetos:

1. A él mismo (objeto *this*).
2. A un parámetro de un servicio propio.
3. A un atributo de él.
4. A una colección de él.
5. A un objeto creado en un método propio.

La intención es evitar el acoplamiento entre un cliente con objetos indirectos.

Beneficios de *No hable con Extraños*:

- Se añaden fácilmente las extensiones que se necesitan.
- Se puede introducir nuevas implementaciones sin afectar a los clientes.
- Se reduce el acoplamiento.
- Se puede disminuir el impacto o coste de los cambios.

Hay que saber escoger las batallas. En sistemas maduros, la estructura es más estable y se puede hablar con extraños. En cambio, en sistemas nuevos es recomendable utilizar este antiguo patrón GRASP[3]. Si se emplea Variaciones Protegidas es posible no utilizar el patrón *No hable con Extraños*.

[3] Para los programadores noveles se aconseja utilizar este patrón. Empléese en el trabajo de curso.