5 UML. El modelo dinámico y de implementación

UML estructural estaba basado esencialmente en los diagramas de clases. En el modelo dinámico hay más variedad de diagramas, aunque no todos deben ser empleados en el proyecto.

Los aspectos dinámicos o de evolución temporal de la aplicación se pueden modelar con los siguientes diagramas:

- Diagramas de interacción (de secuencia y de colaboración)
- Diagramas de estado
- Diagramas de actividad

Mientras la implementación del proyecto emplea los gráficos de:

- Diagrama de componentes
- Diagrama de despliegue

5.1 Diagramas de interacción

UML incluye los diagramas de interacción para ilustrar el modo en que los objetos interactúan por medio de mensajes. Esta visión proporciona una vista integral del comportamiento del sistema; muestra el flujo de control a través de los mensajes entre objetos.

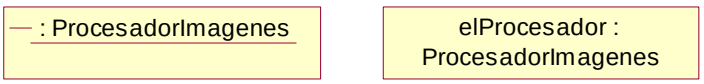
Se debería dedicar un tiempo y esfuerzo no trivial en la creación de los diagramas de interacción. Es en esta etapa donde se requiere la aplicación de las técnicas de diseño, en términos de patrones, estilos y principios. La creación de los casos de uso, modelo del dominio y otros artefactos vistos resultan ser más sencillos que las asignaciones de responsabilidades y su representación en los diagramas de interacción. La realización de los diagramas de interacción, en otras palabras, decide sobre los detalles del diseño de objetos, es una etapa muy creativa del AOO/D. Es donde se suele aplicar los patrones, principios y estilos para mejorar la calidad de los diseños.

Hay dos tipos de diagramas de interacción, ambos centrados en aspectos complementarios:

- Diagramas de secuencia
- Diagrama de interacción

5.1.1 Notación general

Los objetos son representados por instancias de la clase con o sin identificador.



Los mensajes emplearán una sintaxis igual que los servicios de las clases. De hecho, la mayoría de los mensajes son los servicios de las clases:

Nombre-devolucion ‘:=’ nombre-mensaje’ (‘parámetro’:’tipo Parametro’):’tipo-devolucion

Ejemplo

laListaCaracteristica := procesarFichImag(nomFichImag:string):ListaCaracterística

5.1.2 Diagramas de secuencia

Un diagrama de secuencia muestra un conjunto de mensajes dispuestos en una secuencia temporal. Se encuentra estructurado en dos dimensiones. El tiempo se representa verticalmente y evoluciona hacia abajo. No suele estar representado

necesariamente a escala. En la dirección horizontal, hay franjas verticales sucesivas que corresponde a los diferentes objetos que participan en la interacción.

La línea de vida de un objeto simboliza la existencia de éste en un cierto periodo de tiempo. Se representa mediante una línea discontinua vertical que va desde su creación hasta la destrucción.

Las actividades representan los tiempos de procesamiento de los objetos y se representan mediante rectángulos verticales alargados insertados en las líneas de vida.

Los mensajes se indican con flechas que comienzan en una activación (al principio de ésta o en una posición intermedia) y acaban en otra. También, se puede indicar los mensajes de retorno al final de una activación, en forma de flecha discontinua y punta abierta. Lo normal es que se excluyan por quienes utilizan UML.

El uso de un diagrama de secuencia es para mostrar la secuencia de comportamiento de un caso de uso. Cuando se implementa el comportamiento del caso de uso, cada mensaje en un diagrama de secuencia corresponde a:

1. Una operación en una clase.

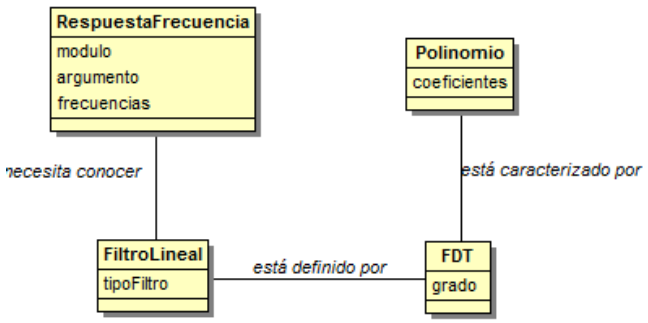
2. A un evento externo.

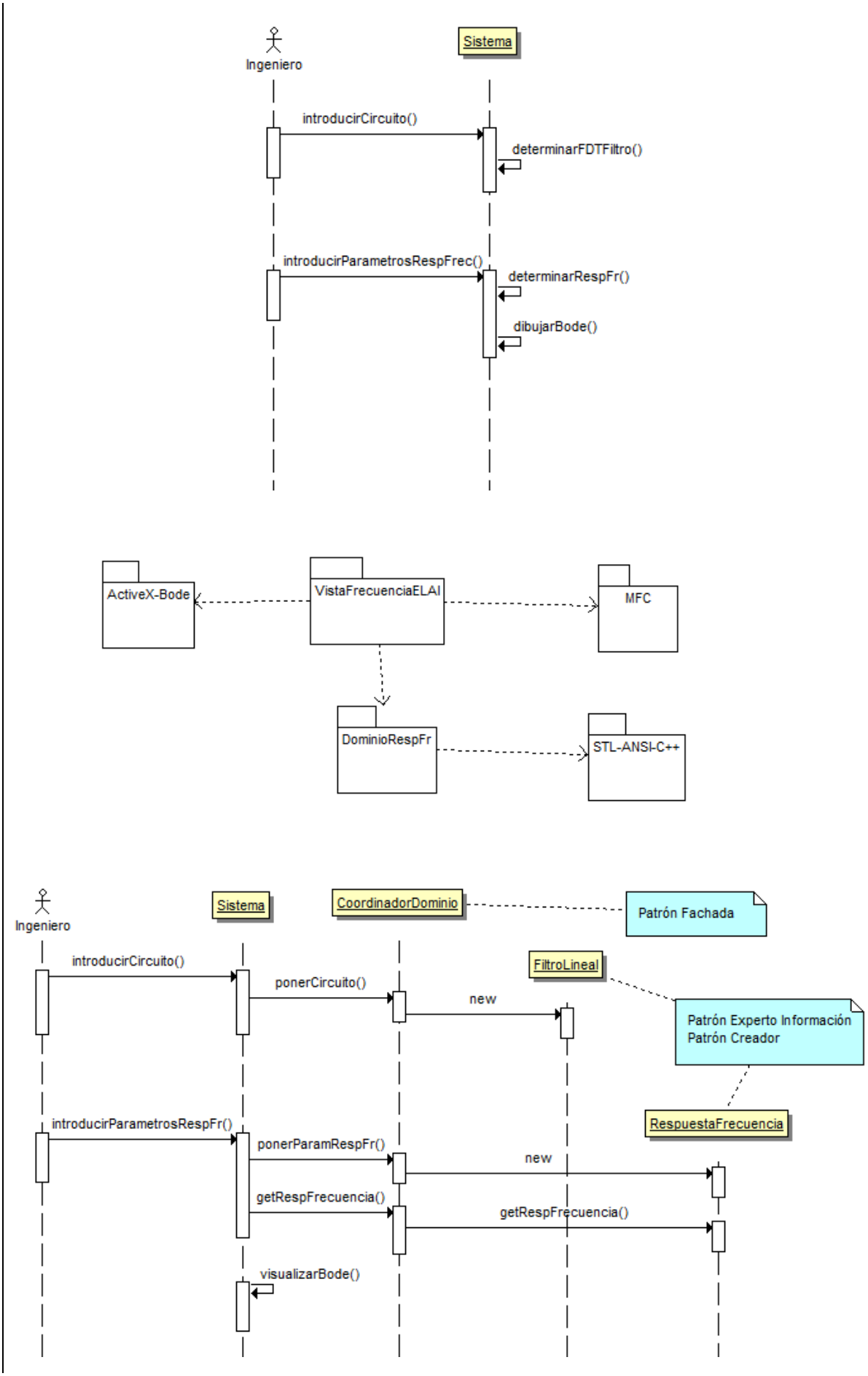
3. A una transición de una máquina de estados.

Ejemplo 5.1

Para el caso de uso RespuestaFrecuencia realizar el diagrama de secuencia del paquete del dominio.

Partiendo del Modelo del Dominio, DSS, contratos de operación y vista de gestión, visto en el capítulo 3 y 4:




Ingeniero
Sistema
introducirCircuito()
determinarFDTFiltro()
introducirParametrosRespFrec()
determinarRespFr()
dibujarBode()
ActiveX-Bode
VistaFrecuenciaELAI
MFC
DominioRespFr
STL-ANSI-C++
Ingeniero
Sistema
CoordinadorDominio
Patrón Fachada
introducirCircuito()
ponerCircuito()
new
FiltroLineal
Patrón Experto Información
Patrón Creador
RespuestaFrecuencia
introducirParametrosRespFr()
ponerParamRespFr()
new
getRespFrecuencia()
getRespFrecuencia()
visualizarBode()

Para indicar la creación de un objeto se emplea el mensaje *create()*, colocándose el objeto a la altura del mensaje de creación. Mientras, el mensaje estereotipado con *<<destroy>>*, con una gran X y la línea de vida cortada, indica la destrucción explícita del objeto.

UML ofrece una rica notación para la representación de distintos tipos de mensajes:

- Mensajes condicionales: el mensaje está condicionado a que se verifique una condición, representa una bifurcación condicionada. UML emplea la sintaxis para las condiciones de:

  '[' condición ']'

- Mensajes condicionales mutuamente exclusivos: Bifurcación del tipo *if-else* de carácter excluyente. La notación es un tipo de línea de mensaje con forma de ángulo que nace desde el mismo punto y se dirige a diferentes actividades.

- Iteración para un único mensaje: representación de bucle de iteración. La notación es emplear un índice que recorre desde una posición inicial hasta otra final:

  '[' indice '=' posición ']'

- Iteración sobre una colección (multiobjetos): La colección de objetos se representa con una sobre línea en la parte superior del objeto.

- Mensaje a <<self>> o <<this>>: mensaje sobre el propio objeto. Se representa mediante una flecha que se inicia en la actividad y termina sobre ella.

En la figura adjunta se ha rescatado el diagrama de secuencia ofrecido en los documentos de la OMG en la versión 1.3 de UML. En el diagrama se pretende mostrar los distintos conceptos definidos.

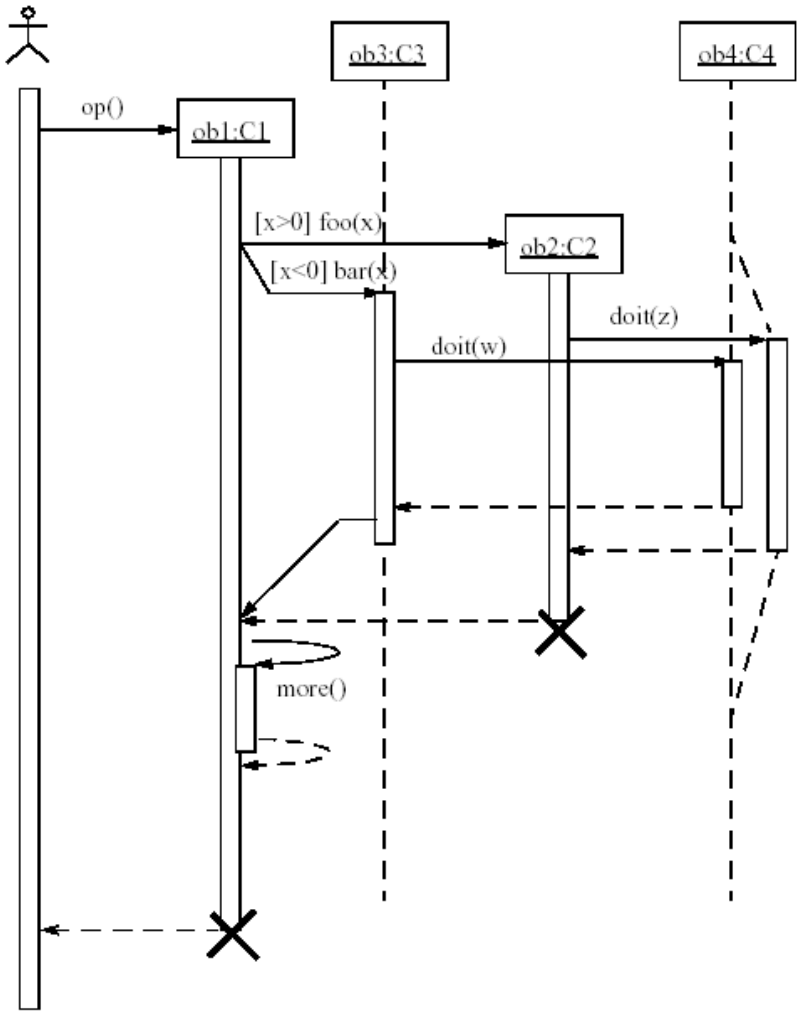


Figura 5. 1. Diagrama de secuencia: ejemplo tomado de la documentación de UML v1.3

5.1.3 Diagramas de colaboración

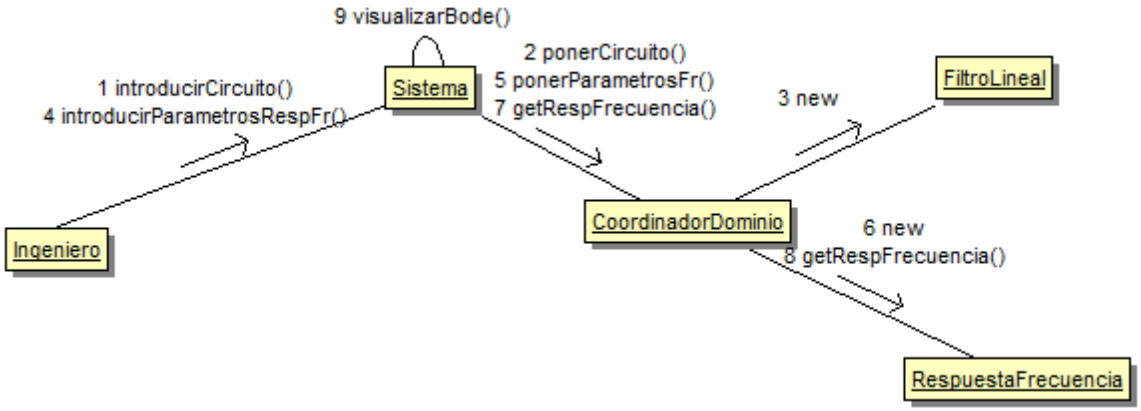
Una colaboración modela los objetos y los enlaces significativos dentro de una interacción. Un enlace es un camino de conexión entre dos objetos; indica que es posible alguna forma de navegación y visibilidad entre objetos. De manera más formal, un enlace es una instancia de una asociación. A lo largo de un enlace pueden fluir múltiples mensajes.

Cada mensaje entre objetos se representa con una expresión de mensaje, una pequeña flecha que indica la dirección del mensaje y un número de secuencia para mostrar el orden.

El orden de los mensajes se representa mediante números de secuencia. El esquema de numeración realiza un anidamiento de los mensajes según el orden de salida.

Ejemplo 5.2

Para el caso de uso Respuesta en frecuencia realizar el diagrama de colaboración[1].



Tanto los diagramas de secuencias como de colaboración muestran interacciones entre los objetos pero acentúan aspectos diferentes. Un diagrama de secuencias muestra secuencias en el tiempo, pero las relaciones entre roles son implícitas. Un diagrama de colaboración presenta las relaciones de roles, pero las secuencias están menos claras, porque vienen dadas por los números de secuencia. Cada diagrama debe ser utilizado cuando su aspecto principal sea el foco de atención.

Para la creación de instancias, UML emplea el convenio de utilizar el mensaje denominado *create()*, o bien emplear otro nombre de mensaje pero acompañándolo del estereotipo <<*create*>>. Adicionalmente, podría añadirse la propiedad UML {new} para resaltar la creación. Igualmente, UML tiene notación específica para los diferentes tipos de mensajes, ya vistos anteriormente en el diagrama de secuencia.

[1] Rational Rose convierte automáticamente los diagramas de secuencia a diagramas de colaboración y viceversa.

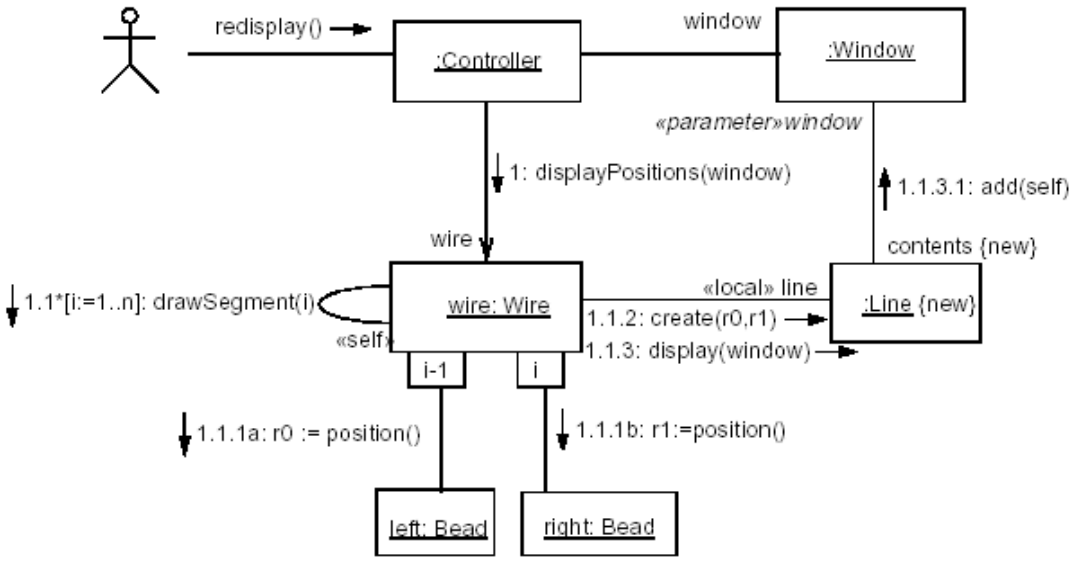


Figura 5. 2. Diagrama de colaboración: ejemplo tomado de la documentación de UML v1.3

5.2 Diagramas de estado

Una maquina de estados modela los posibles estados que puede tener en vida un objeto de una clase. Cada objeto se trata como una entidad aislada que se comunica con el resto del mundo recibiendo eventos y respondiendo a ellos.

Las máquinas de estados se muestran a través de los diagramas de estados. Éstos están constituidos por eventos, estados y transiciones. Un evento es una ocurrencia significativa o relevante. Un estado es la condición de un objeto en un instante de tiempo; hace referencia a los valores de sus atributos en un determinado tiempo. Una transición es una relación entre dos estados que indica cuando tiene lugar un evento; el objeto pasa de su estado al siguiente.

Las transiciones se representan por flechas, etiquetadas con sus eventos. Los estados se representan por rectángulos de esquinas redondeadas. Es habitual incluir un pseudo-estado inicial que pasa automáticamente a otro estado cuando se crea la instancia.

Una máquina de estados contiene los estados de un objeto conectados por transiciones. Cada estado modela un periodo de tiempo, durante la vida de un objeto, en el que se satisface ciertas condiciones. Cuando ocurre un evento se puede desencadenar una transición que lleve al objeto a un nuevo estado. Al dispararse una transición se puede ejecutar una acción unida a la transición.

Un diagrama de estado muestra el ciclo de vida de un objeto: qué eventos experimenta, sus transiciones y los estados en los que se encuentran entre estos eventos. No es necesario ilustrar todos los posibles estados. Por lo tanto, el diagrama de estado

describe el ciclo de vida de un objeto a un nivel de detalle arbitrario, simple o complejo dependiendo de las necesidades.

El diagrama de estado servirá para:

- Comprobar que los eventos ocurren en orden correcto.
- Habilitar/deshabilitar elementos según el desarrollo del diagrama de estados.
- En un dominio con muchos eventos del sistema, la concisión y minuciosidad del diagrama de estado ayuda al diseñador asegurarse a que no se ha omitido ninguno.

5.2.1 Notación adicional

Una transición puede provocar que se dispare una acción. En una implementación SW, esto supone la invocación de un método de una clase. Las acciones se representan mediante la barra invertida, /.

Una transición podría tener una condición de guarda –o condición booleana-. Sólo ocurre la transición si se cumple la condición.

'[' condición ']'

Un estado puede contener subestados. Los subestados heredan la transición del estado al que pertenece. Esta utilidad permite tener varias vistas con diferentes niveles de resolución o complejidad del modelo.

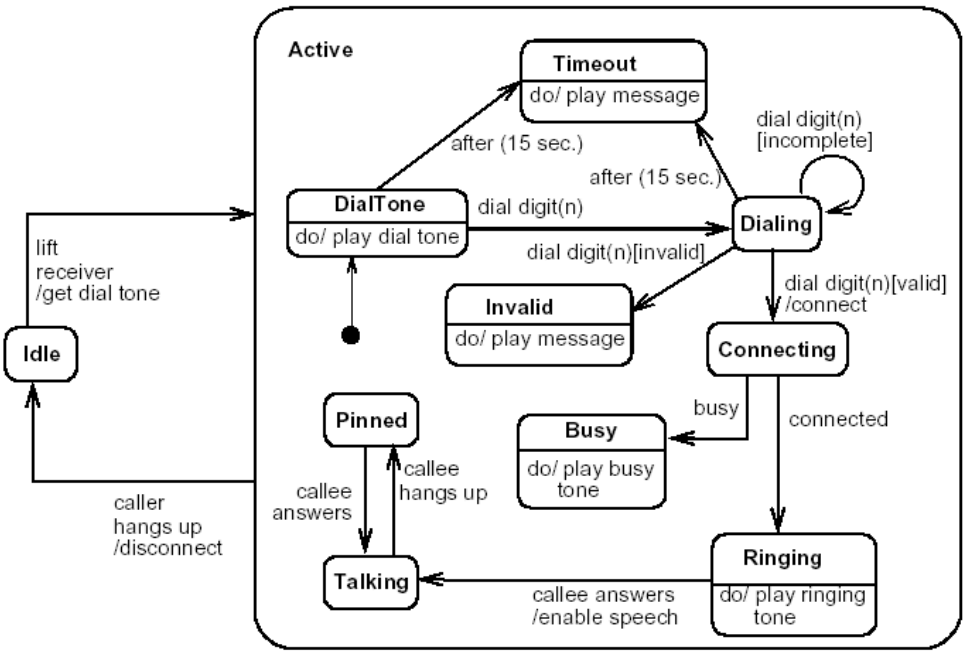
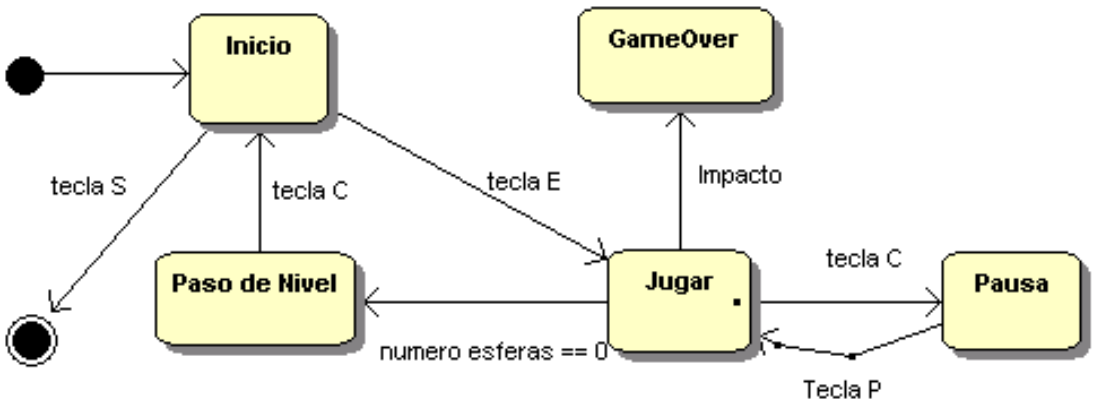


Figura 5. 3. Diagrama de estado: ejemplo tomado de la documentación de UML v1.3

Ejemplo 5.3

Realizar el diagrama de estado para el juego del Pang.



En general, en las aplicaciones informáticas de gestión, las clases dependientes del estado son una minoría. En cambio, en los proyectos de control de procesos o en aplicaciones de telecomunicaciones son frecuentes que muchos objetos sean dependientes de sus estados.

Tipos de eventos:

- Evento externo: También conocido como evento hacia el sistema, lo origina algo fuera del sistema.

- Evento interno: Causado por algo dentro del sistema.

- Evento del tiempo: Causado por la ocurrencia de una fecha y hora específicas. Un evento temporal lo dirige un reloj de tiempo real.

Es preferible utilizar los diagramas de estado para ilustrar los eventos externos y de tiempo, y las reacciones a ellos; mientras los eventos internos son representados en los diagramas de interacción.

Cuando se desear tener una idea más amplia de los efectos del comportamiento dinámico de un sistema, se emplean las vistas de interacción. Por contra, las máquinas de estados son útiles para entender los mecanismos de control, tales como interfaces de usuario o controladores de dispositivos.

5.3 Diagrama de actividades

Un diagrama de actividades de UML ofrece una notación rica para representar una secuencia de actividades. Podría aplicarse a cualquier propósito, pero se considera especialmente útil para visualizar los flujos de trabajo y los procesos del negocio. Formalmente, un diagrama de actividades se considera un tipo especial de diagrama de estados de UML, en el que los estados son acciones y las transiciones de los eventos se disparan automáticamente al completarse la acción. Un grafo de actividades describe grupos secuenciales y concurrentes de actividades.

Un diagrama de actividades tiene el propósito de modelar los procesos reales de una organización humana. El modelado de tales negocios es un propósito importante de los diagramas de actividades, pero también se pueden utilizar para modelar actividades software de alto nivel de ejecución de un sistema, sin profundizar en los detalles internos de los mensajes.

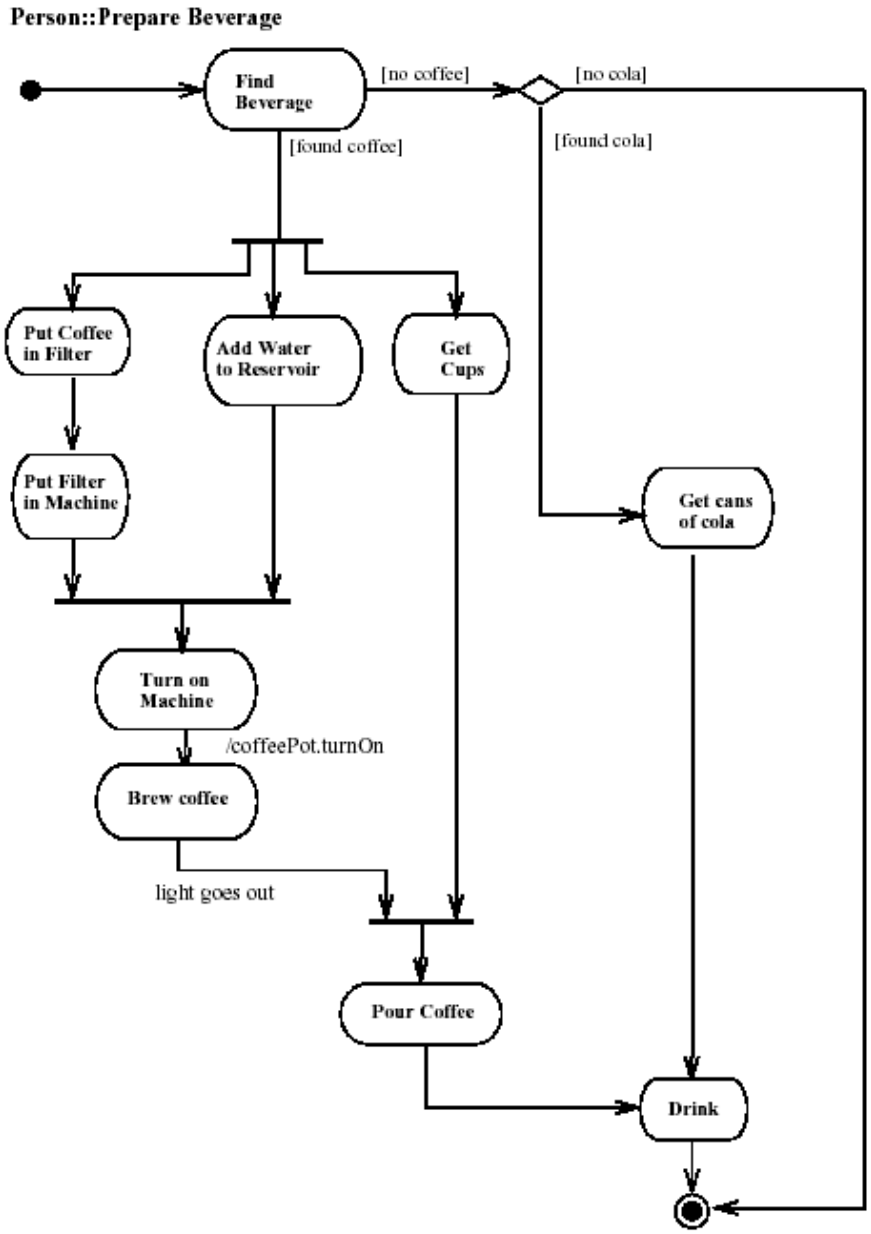


Figura 5. 4. Diagrama de actividades: ejemplo tomado de la documentación de UML v1.3

5.4 La vista física

Las vistas anteriores modelan los conceptos de la aplicación desde un punto de vista lógico. Las vistas físicas describen las estructuras de implementación de la aplicación, su organización en componentes y su despliegue en nodos de ejecución. Estas vistas proporcionan una oportunidad de establecer correspondencias entre las clases y los componentes de implementación y nodos. Pero además, los aspectos de implementación son importantes para los propósitos de reutilización y rendimiento. UML define dos tipos de vistas que se pueden utilizar para ilustrar los detalles de la implementación: la vista de implementación y la vista de despliegue.

5.4.1 La vista de implementación

La vista de implementación muestra el empaquetado físico de las partes reutilizables del sistema en unidades sustituibles, llamadas componentes. Una vista de implementación muestra los elementos físicos del sistema mediante componentes, así como sus interfases y dependencias entre componentes. Los componentes son piezas reutilizables de alto nivel a partir de las cuales se pueden construir los sistemas.

El diagrama de componentes describe la descomposición física del sistema SW en componentes, a efectos de construcción y funcionamiento. La descomposición del diagrama de componentes se realiza en términos de componentes y de relaciones entre los mismos. Los componentes identifican objetos físicos que hay en tiempos de ejecución, de compilación o de desarrollo y tienen identidad propia con una interfaz bien definida. Los componentes incluyen código en cualquiera de sus formatos, DLL, Active X, bases de datos, ... Cada componente incorpora la implementación de ciertas clases del diseño del sistema.

En un diagrama de componentes se muestran las diferentes relaciones de dependencia que se pueden establecer entre componentes. Los componentes bien diseñados no dependen de otros componentes. Un componente en un sistema puede ser sustituido por otro componente que ofrezca las interfaces apropiadas.

Un componente se representa mediante tres rectángulos.

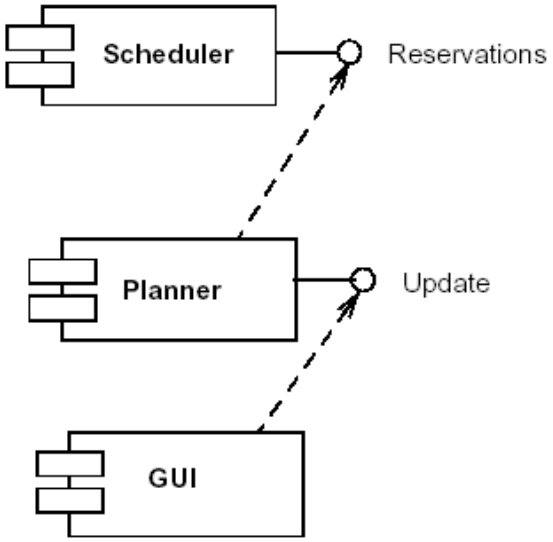
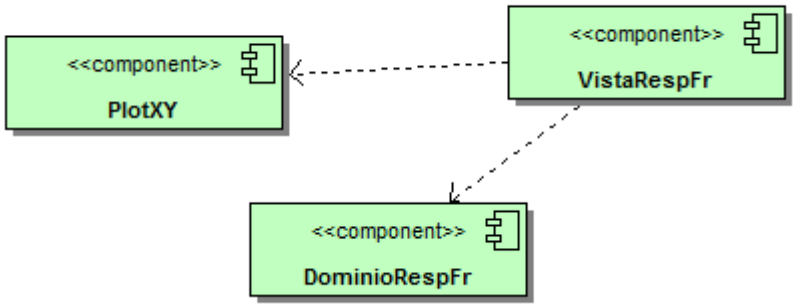


Figura 5. 5. Diagrama de componentes: ejemplo tomado de la documentación de UML v1.3

Ejemplo 5.5

Realizar el modelo de componentes de la aplicación Respuesta en frecuencias.



5.4.2 La vista de despliegue

La vista de despliegue muestra la disposición física de los recursos de la ejecución computacional, tales como los computadores y sus interconexiones. El diagrama de despliegue permite mostrar la arquitectura en tiempo de ejecución del sistema respecto al HW y SW. Es más limitado que el diagrama de componentes ya que sólo se presenta en tiempo de ejecución. Sin embargo, resulta más amplio en el sentido de que puede contener más clases de elementos.

Los nodos representan los objetos físicos existentes en tiempo de ejecución. Éstos sirven para modelar recursos que tienen memoria y capacidad de proceso, los cuales pueden ser computadores, dispositivos o personas. Los componentes participan en los procesos mediante los nodos.

La vista de despliegue puede mostrar cuellos de botella para el rendimiento si las instancias de los componentes con dependencia se ponen en distintos nodos.

Los nodos se presentan mediante paralelepípedos. Las asociaciones entre nodos representan líneas de comunicación.

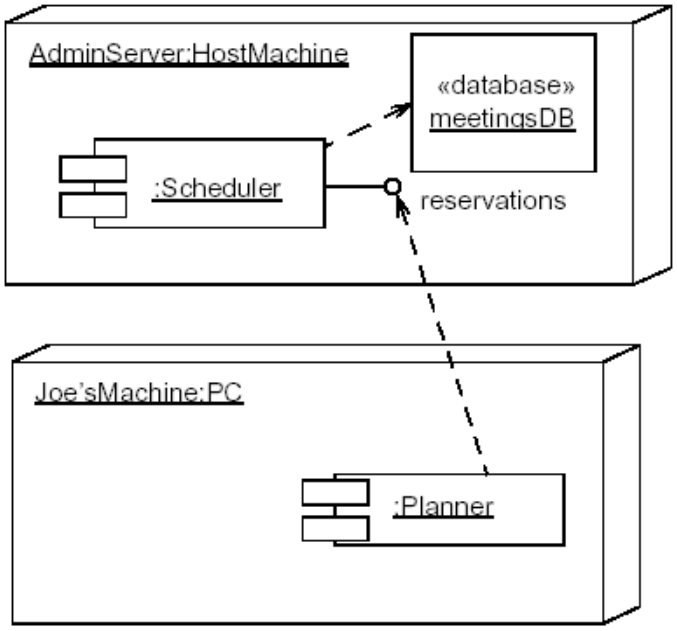


Figura 5. 6. Diagrama de despliegue: ejemplo tomado de la documentación de UML v1.3

5.5 Cuestiones

1. Diferencias entre el diagrama de secuencia con el de colaboración.

2. Obtener el diagrama de secuencias y de colaboración para la aplicación *Juego de dados:* se lanzan dos dados, si la suma de sus caras es siete gana; en caso contrario, pierde.

3. Utilidades de los diagramas de estado.

4. Cuándo se empleará un diagrama de estado y cuándo de interacción.

5. Sugiera un diagrama de componentes y otro de despliegue para una aplicación de subasta por Internet, para una galería de arte.

5.6 Problemas

Ejercicio 1

Realizar una aplicación que ordene de forma creciente los números dados por el usuario.

1. Caso de Uso

2. Modelo del dominio y DSS

3. Vista de Gestión.

4. Diagrama de secuencia y diagrama de clases de diseño

5. Implementación en C++

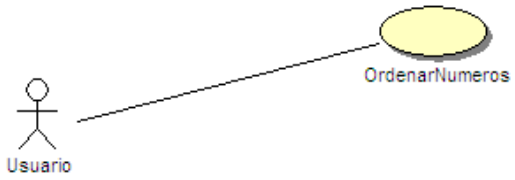
1. El caso de uso EBP se llama *"OrdenarNumeros"* y tendrá un curso de éxito como:

I. Ordenar los números de forma creciente

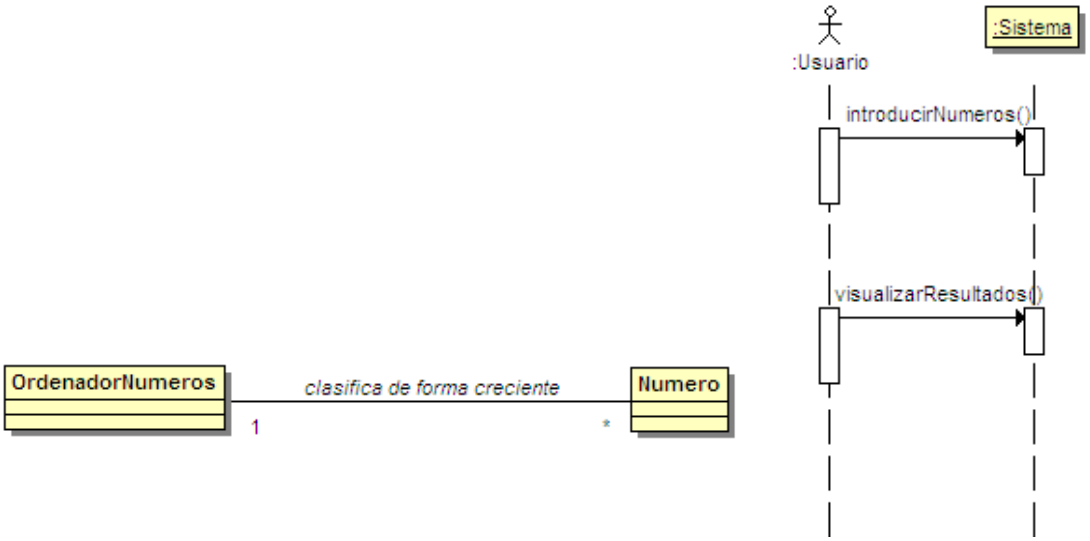
I.a. Solicitar al usuario el vector de números a ordenar

I.b. Ordenar de forma creciente el vector

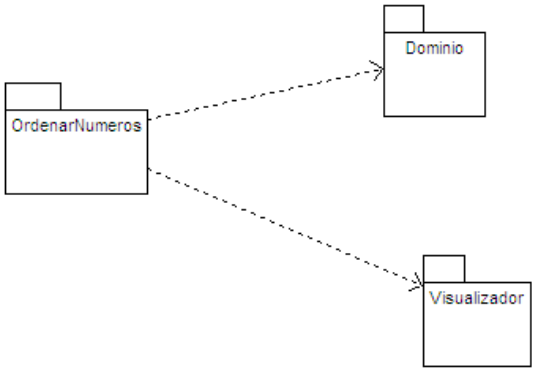
I.c. Visualizar los resultados



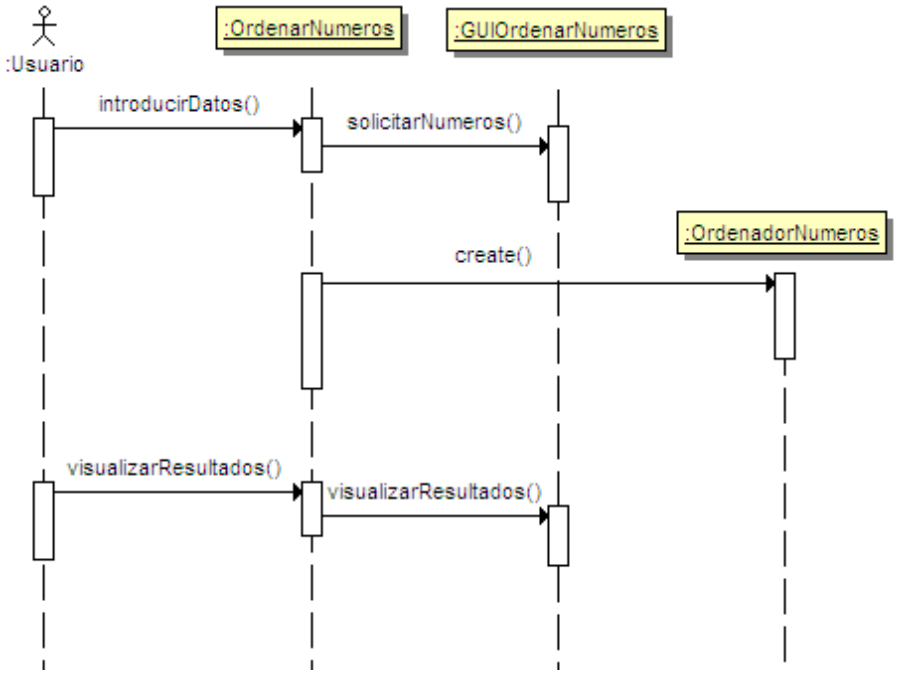
2. Del análisis de los documentos aparece los conceptos de *OrdenadorNumeros* y *Número* proponiéndose el siguiente Modelo del Dominio y DSS:

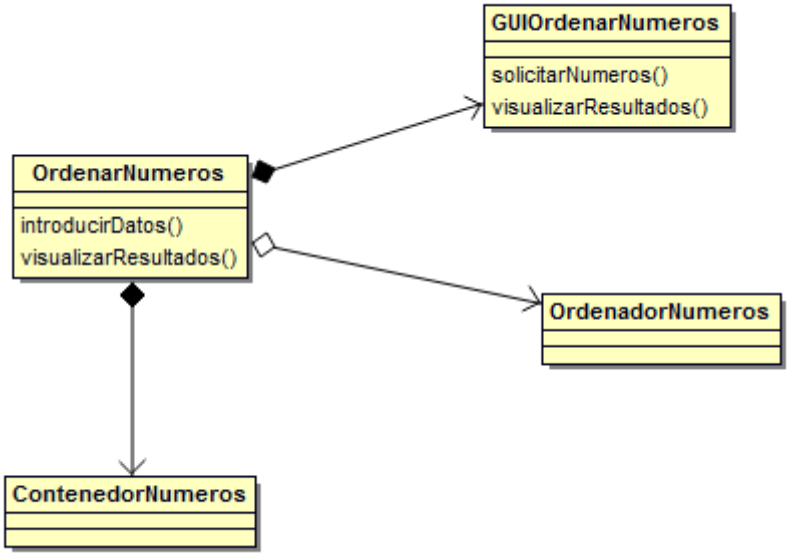


3. Se propone la siguiente arquitectura:



4. Diseño de la aplicación





5. Implementación en C++

```
#if !defined(_ORDENADORNUMEROS__INCLUDED_)
#define _ORDENADORNUMEROS_INCLUDED_

#include <algorithm>
#include <vector>

class OrdenadorNumeros
{
public:
	OrdenadorNumeros(std::vector<double> & elVectorNumeros)
	{std::sort(elVectorNumeros.begin(),elVectorNumeros.end());}
};

#endif
```

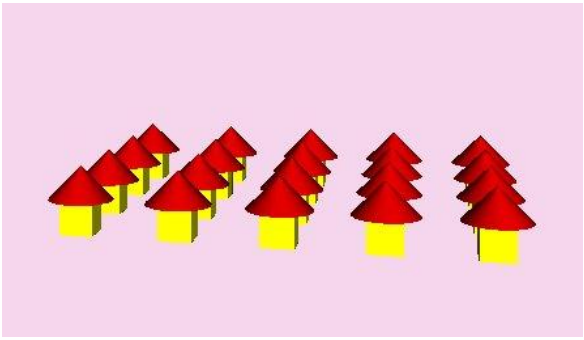
```
#if !defined(_GUIORDENARNUMEROS__INCLUDED_)
#define _GUIORDENARNUMEROS__INCLUDED_

#include <vector>

class GUIOrdenarNumeros
{
public:
	void visualizarResultados(std::vector<double> &);
	void solicitarNumeros(std::vector<double> &);
};

#endif
```

```
#include "..\..\CABECERAS\VISUALIZADOR\GUIOrdenarNumeros.h"
#include <algorithm>
#include <iostream>
void GUIOrdenarNumeros::solicitarNumeros(std::vector<double> &elVectorNumeros)
{
	bool introducirDatos = true; double valor;
	std::cout<<"Esta aplicacion ordena los valores de forma creciente"<<std::endl;
	std::cout<<"Introducir la lista de numeros y poner cero para salir"<<std::endl;
	while(introducirDatos == true){
		std::cin>>valor;
		if(valor != 0)
			elVectorNumeros.push_back(valor);
		else
			introducirDatos = false;
	}
}
void visualizarDatos(double);
void GUIOrdenarNumeros::visualizarResultados(std::vector<double> &elVectorNumeros)
{
	std::cout<<"Lista ordenada"<<std::endl;
	std::for_each(elVectorNumeros.begin(),elVectorNumeros.end(),visualizarDatos);
}
void visualizarDatos(double valor)
{
	std::cout<<valor<<std::endl;
}
```

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

#include <vector>
#include "../Dominio/OrdenadorNumeros.h"
#include "../Visualizador/GUIOrdenarNumeros.h"

class OrdenarNumeros
{
	GUIOrdenarNumeros elVisualizador;
	std::vector<double> elVectorNumeros;
public:
	void introducirDatos();
	void visualizarResultados();
};

#endif
```

```
#include "..\..\CABECERAS\ORDENARNUMEROS\OrdenarNumeros.h"
void OrdenarNumeros::introducirDatos()
{
	this->elVisualizador.solicitarNumeros(this->elVectorNumeros);
	OrdenadorNumeros elOrdenador(this->elVectorNumeros);
}
void OrdenarNumeros::visualizarResultados()
{
	this->elVisualizador.visualizarResultados(this->elVectorNumeros);
}

int main()
{
	OrdenarNumeros elOrdenar;
	elOrdenar.introducirDatos();
	elOrdenar.visualizarResultados();
	return 0;
}
```

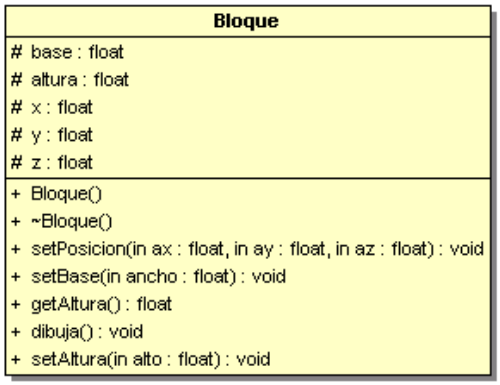
Ejercicio 2

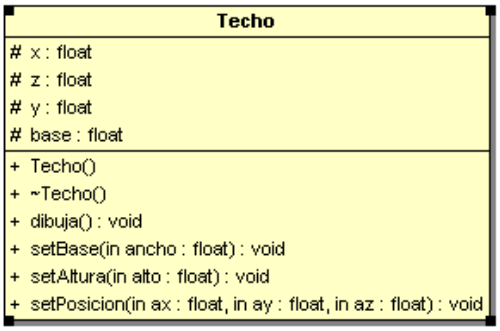


Realizar una aplicación basadas en OpenGL que permita dibujar un poblado. Las reglas de urbanización, en esta primera versión son muy simples, se creará el número de casas ordenadas en una matriz de filas y columnas. En la figura se muestra un ejemplo de un poblado de 4 filas y cinco columnas. Se pide:

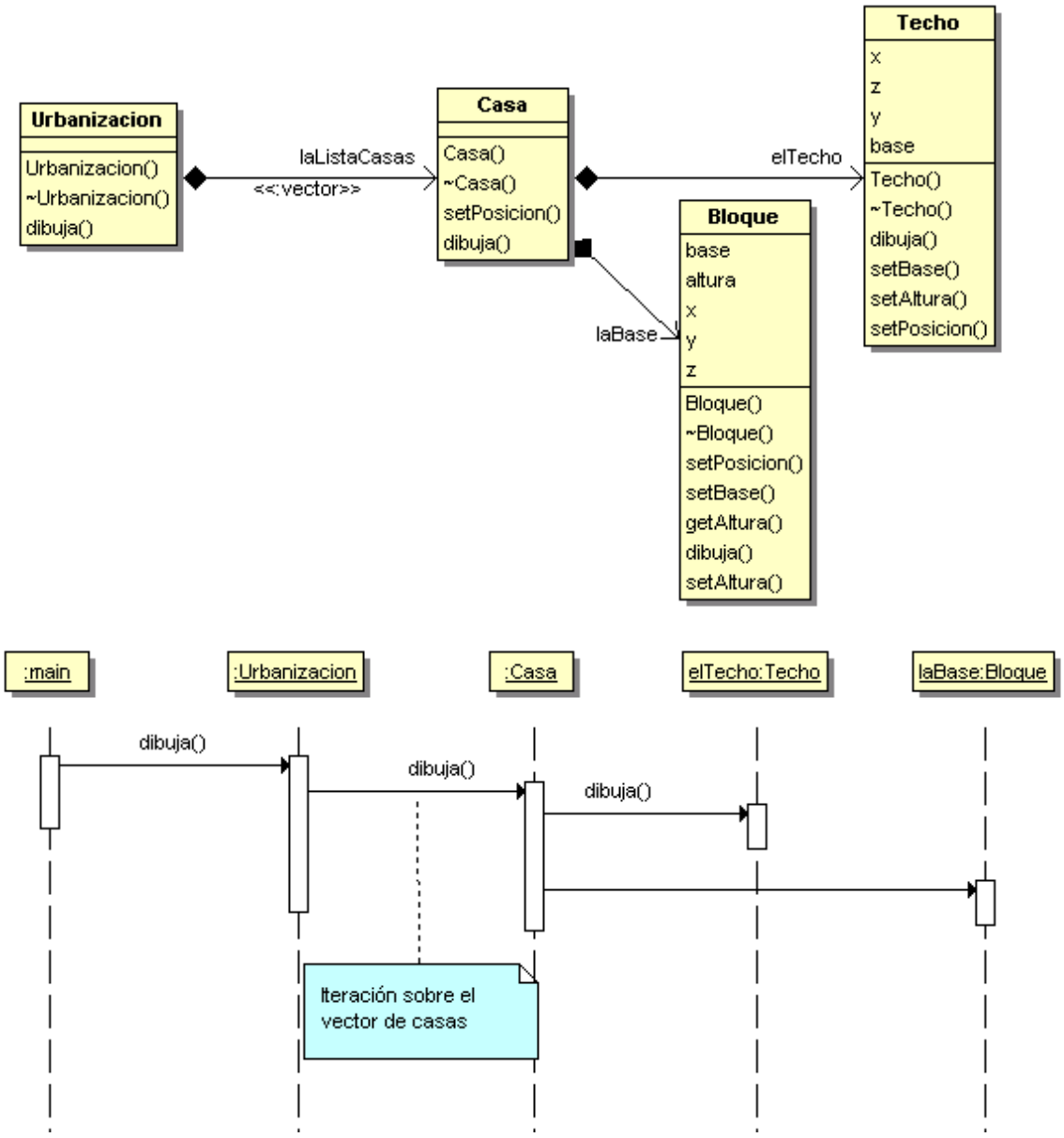
1. Representación UML de las clases Techo y Bloque.
2. Diagrama de Clases de Diseño (DCD) de la solución.
3. Diagrama de Secuencias explicativo del dibujo de la escena.
4. Implementación C++.

1. Aplicando ingeniería inversa, las clases techo y bloque en UML quedan como:





2. y 3. La solución se dará con el diagrama de clases de diseño, DCD, y con los diagramas de interacción:



4. Una vez acabado el diseño, se pasará a su implementación desde las clases menos acopladas hasta alcanzar a la función *main():*

```
#include <vector>
#include "Casa.h"

class Urbanizacion
{
 unsigned filasCasa, columnasCasa;
 std::vector<Casa> laListaCasas;

public:

 Urbanizacion(unsigned, unsigned);
 virtual ~Urbanizacion();
 void dibuja();

};
```

```
#include "Bloque.h"
#include "Techo.h"

class Casa
{
        Bloque laBase;
        Techo elTecho;

public:
        Casa(float,float,float);
        virtual ~Casa();
        void setPosicion(float,float,float);
        void dibuja();

};
```

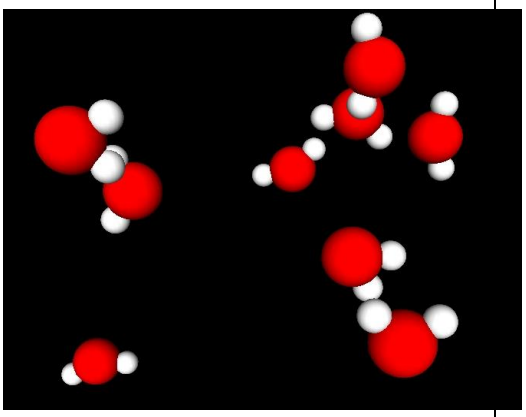
```
#include "..\..\INCLUDES\DOMINIO\Casa.h"
Casa::Casa(float ancho, float altoBase, float altoTejado)
{
        this->laBase.setBase(ancho);
        this->elTecho.setBase(ancho);

        this->laBase.setAltura(altoBase);
        this->elTecho.setAltura(altoTejado);

}

void Casa::setPosicion(float ax,float ay,float az)
{
        this->laBase.setPosicion(ax,ay,az);
        this->elTecho.setPosicion(ax,ay+(this->laBase.getAltura()),az);
}

void Casa::dibuja()
{
        this->elTecho.dibuja();
        this->laBase.dibuja();
}
```

```
#include "..\..\INCLUDES\DOMINIO\Urbanizacion.h"
#include "..\..\INCLUDES\comunes\glut.h"

#define ANCHO_BLOQUE          1.0f
#define ALTO_BLOQUE           0.5f
#define ANCHO_TECHO           1.0f
#define SEPARACION_CASA       3.0f

Urbanizacion::Urbanizacion(unsigned filas, unsigned columnas)
{
 filasCasa = filas; columnasCasa = columnas;
 for(unsigned i = 0; i<filas; i++)
    for(unsigned j = 0; j<columnas; j++) {
        this->laListaCasas.push_back(Casa(ANCHO_BLOQUE,ALTO_BLOQUE,ANCHO_TECHO));
        this->laListaCasas[(i*columnas)+j].setPosicion(SEPARACION_CASA*j,
                0,SEPARACION_CASA*i);
        }
}

void Urbanizacion::dibuja()
{
        float x_ojo=10;float y_ojo=7.5; float z_ojo=40;
        gluLookAt(x_ojo, y_ojo, z_ojo,  // posicion del ojo
                        0.0, y_ojo, 0.0,      // hacia que punto mira  (0,0,0)
                        0.0, 1.0, 0.0);      // definimos hacia arriba (eje Y)

        for (unsigned i=0;i<filasCasa*columnasCasa; i++)
                this->laListaCasas[i].dibuja();

}
```

Ejercicio 3

Se pretende desarrollar un programa de simulación de un vapor de agua, para el que se ha desarrollado ya la siguiente clase, que funciona correctamente y no es necesario modificar:

```
class Atomo
{
public:
      Atomo(int num);
       virtual ~Atomo();
       void Enlaza(Atomo* a);
       void CalculaPosicion();
       void Dibuja();
protected:
       float x;
       float y;
       float z;
       int numero_atomico;

       Atomo* enlace;
};
```



Ejemplo del resultado final

Supóngase que el método CalculaPosicion() es capaz de calcular la posición del átomo en el espacio teniendo en cuenta todas las interacciones físicas, incluyendo la posibilidad que dicho átomo haya establecido un enlace con otro átomo. El código para dibujar una molécula de agua utilizando la clase Atomo sería el siguiente:

```
void main()
{
Atomo hidrogeno1(1);
Atomo hidrogeno2(1);
Atomo oxigeno(16);

       hidrogeno1.Enlaza(&oxigeno);
       hidrogeno2.Enlaza(&oxigeno);

       oxigeno.CalculaPosicion();
       hidrogeno1.CalculaPosicion();
       hidrogeno2.CalculaPosicion();

       hidrogeno1.Dibuja();
       hidrogeno2.Dibuja();
       oxigeno.Dibuja();
}
```

Donde es importante el orden de cálculo de las posiciones, es decir, primero se calcula la posición del oxigeno, y después la de los átomos de hidrogeno, que dependen del átomo de oxigeno. Cuando el programa este completado, el código del main() debe quedar como:

```
void main()
      {
      VaporAgua vapor(30);      //30 moleculas de agua
      vapor.CalculaPosicion();
      vapor.Dibuja();
      }
```

1. Diagrama de Clases de Diseño (DCD de la solución), que incluya orientación a objetos para cada molécula de agua.

2. Un diagrama de secuencias del cálculo de la posición y el dibujo del vapor, a partir de la función main().

3. Implementación de la solución en C++

Ejercicio 4

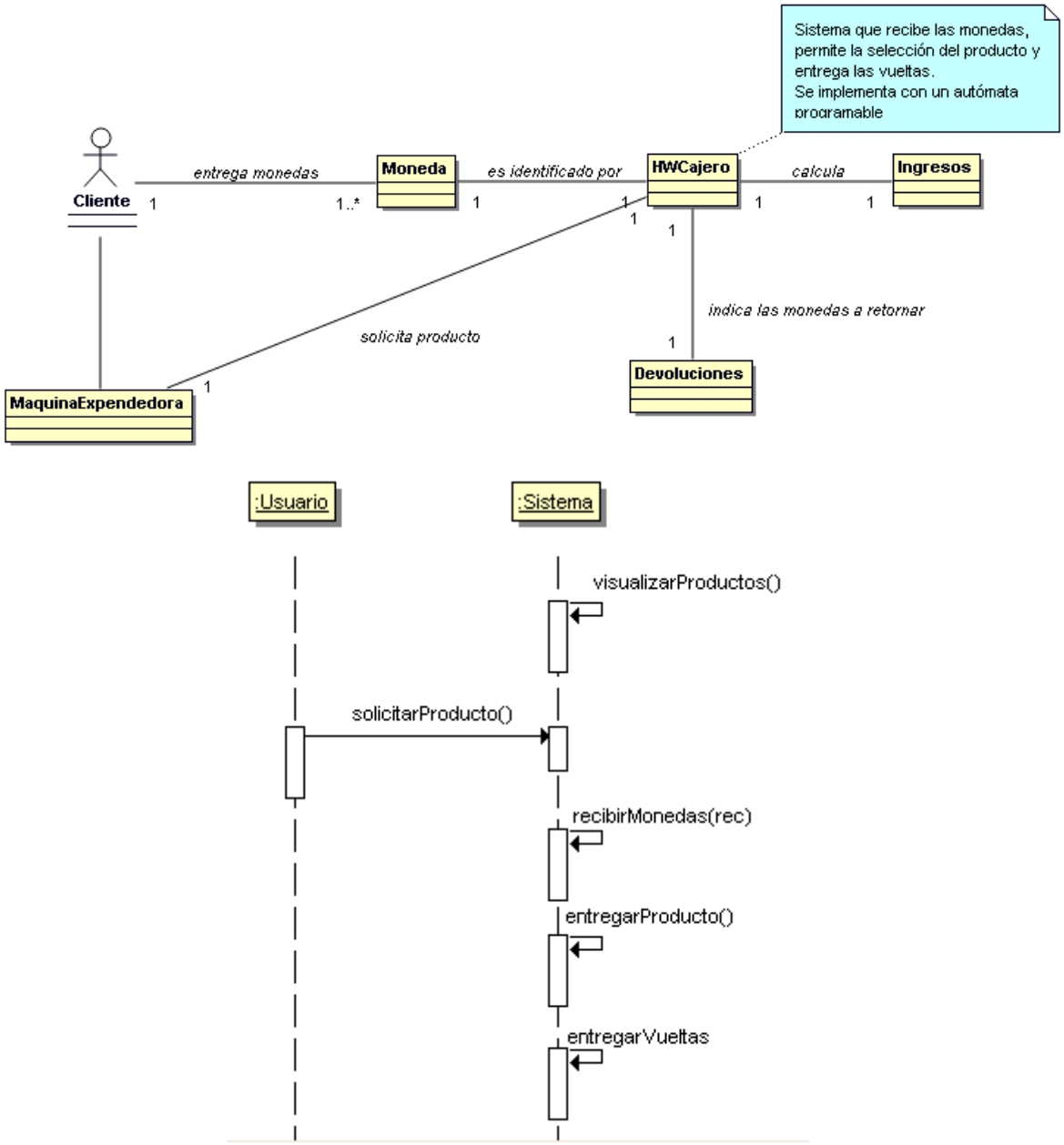
Diseñar el programa de una máquina expendedora de bebidas, de manera que recibe el producto seleccionado y las monedas que entran en el cajero desde un autómata programable. La aplicación debe de retornar la lista de monedas mínimas a entregar al usuario.

Primero se realizará la jerarquía a dos niveles de las principales características:

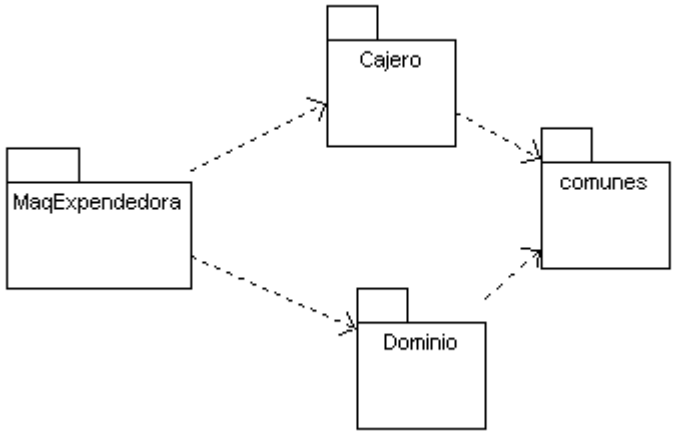
1. El sistema debe de mostrar los productos que ofrece y sus precios
    1.1 El usuario puede elegir uno de los productos ofertados
2. El sistema debe de reconocer las monedas que se le entrega
    2.1 Debe de evaluar el crédito del cliente.
3. El sistema entregará el producto cuando el cliente tenga suficiente crédito.
    3.1 El sistema devolverá las vueltas cuando hubiera exceso de crédito.

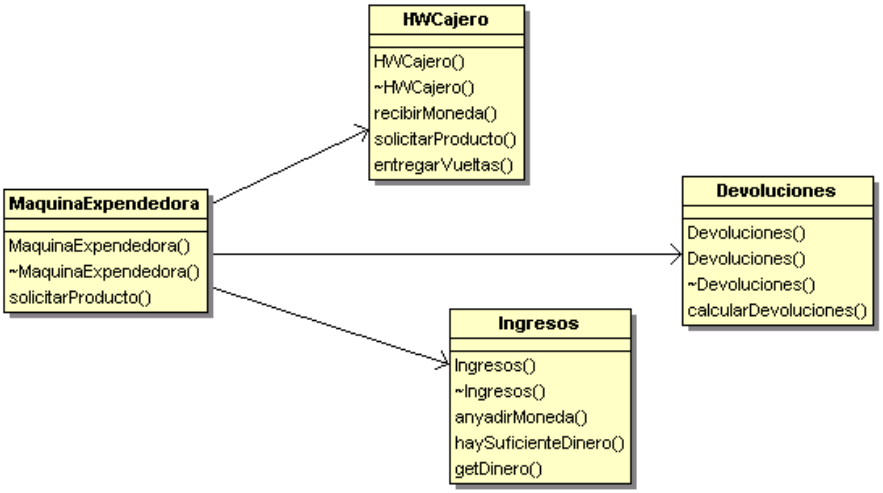
Seguidamente se procederá a rellenar los documentos de Visión y Alcance, glosario y el caso de uso EBP: *SolicitarProducto*.

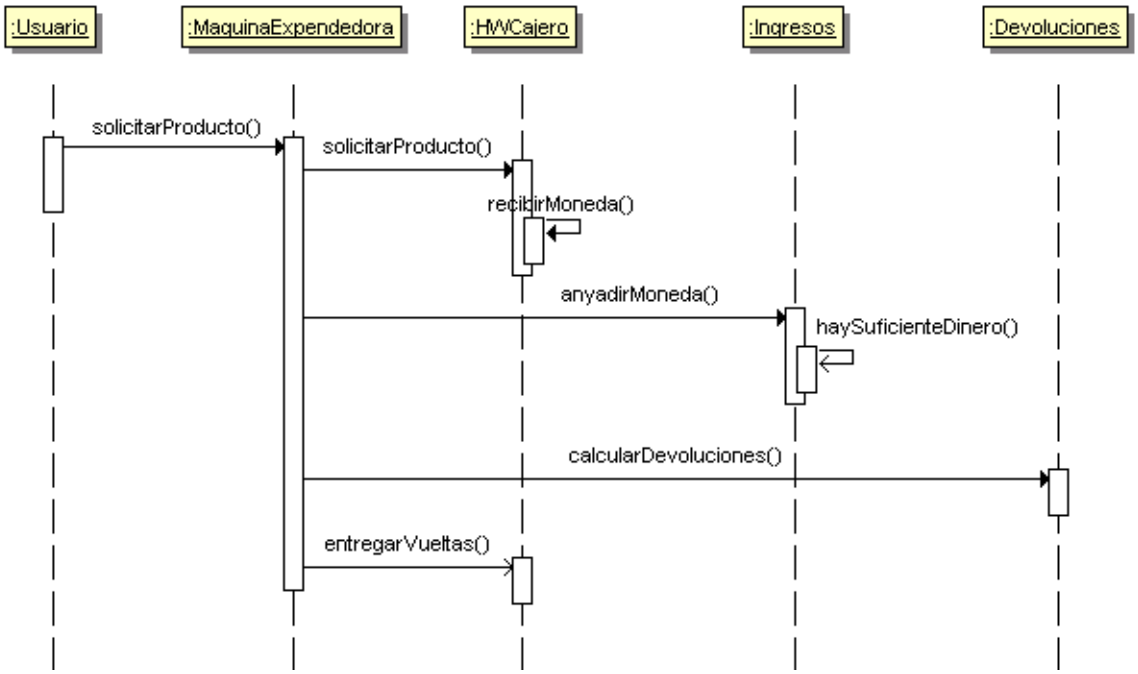
En el AOO se realizará el modelo del dominio y el DSS:



Durante DOO se planteará la solución lógico a través de la vista de gestión, diagrama de clases de diseño, DCD, y los diagramas de interacción:


Cajero
comunes
MaqExpendedora
Dominio


HWCajero
HWCajero()
~HWCajero()
recibirMoneda()
solicitarProducto()
entregarVueltas()
MaquinaExpendedora
MaquinaExpendedora()
~MaquinaExpendedora()
solicitarProducto()
Devoluciones
Devoluciones()
Devoluciones()
~Devoluciones()
calcularDevoluciones()
Ingresos
Ingresos()
~Ingresos()
anyadirMoneda()
haySuficienteDinero()
getDinero()


:Usuario
:MaquinaExpendedora
:HWCajero
:Ingresos
:Devoluciones
solicitarProducto()
solicitarProducto()
recibirMoneda()
anyadirMoneda()
haySuficienteDinero()
calcularDevoluciones()
entregarVueltas()

El código de test sería:

```
#include "../Cajero/HWCajero.h"
#include "../Dominio/Ingresos.h"
#include "../Dominio/Devoluciones.h"

class MaquinaExpendedora
{
public:
	MaquinaExpendedora();
	virtual ~MaquinaExpendedora();
	void solicitarProducto();
};
////////////////////////////////
void MaquinaExpendedora::solicitarProducto()
{
	HWCajero miCajero;
	Dinero elPrecioProducto = miCajero.solicitarProducto();
	Ingresos miDinero;
	while(miDinero.haySuficienteDinero(elPrecioProducto) == false)
		miDinero.anyadirMoneda(miCajero.recibirMoneda());

	Devoluciones elDineroEntregar(miDinero.getDinero(),elPrecioProducto);

	std::vector<Moneda> laListaMonedas;
	elDineroEntregar.calcularDevoluciones(laListaMonedas);
	miCajero.entregarVueltas(laListaMonedas);
	miCajero.entregarProducto();
}
////////////////////////////
#include <iostream>
int main()
{
	MaquinaExpendedora laMaquinaExpendedora;
	bool continuar = true;char opcion;
	while(continuar){
		laMaquinaExpendedora.solicitarProducto();
		std::cout<<"Nuevo producto (s/n): ";
		std::cin>>opcion;
		continuar = (opcion == 'n') || (opcion == 'N') ? false :true;
	}
	return 0;
}
```

La implementación de las clases de la menos acoplada a la más acoplada:

```
typedef enum{EURO,DOLAR,LIBRA} tipoDinero;
typedef enum{UN_CENT,DOS_CENT,CINCO_CENT,DIEZ_CENT,VEINTE_CENT,CINCUENTA_CENT,
	UNO,DOS} tipoMoneda;
#define FACTOR_DINERO 100
class Dinero
{
	tipoDinero elTipoDinero;
	unsigned cantidad;
public:
	Dinero() {cantidad=0;elTipoDinero=EURO;}
	Dinero(unsigned valor)  {cantidad=valor;elTipoDinero=EURO;}
	unsigned getCantidad(){return cantidad;}
	tipoDinero getTipoDinero() {return elTipoDinero;}
	void operator+=(const Dinero masDinero)
	{if(this->elTipoDinero == masDinero.elTipoDinero)
		this->cantidad+=masDinero.cantidad;}
	void operator-=(const unsigned valor)
	{this->cantidad-=valor;}
	bool operator>=(const Dinero elPedido)
	{if(this->elTipoDinero == elPedido.elTipoDinero) return(
		this->cantidad>=elPedido.cantidad?true:false);}
	Dinero operator-(const Dinero elPedido)
	{if(this->elTipoDinero == elPedido.elTipoDinero) return(Dinero(
		this->cantidad-elPedido.cantidad));}
};
```

```
class Moneda: public Dinero
{
	tipoMoneda elTipoMoneda;
public:
	Moneda(unsigned);
	virtual ~Moneda();
};
////////////////////////////////////////////////
Moneda::Moneda(float cantidad):Dinero(cantidad)
{
	if(cantidad == 1) this->elTipoMoneda = UN_CENT;
	else if (cantidad == 2) this->elTipoMoneda = DOS_CENT;
	else if (cantidad == 5) this->elTipoMoneda = CINCO_CENT;
	else if(cantidad == 10)this->elTipoMoneda = DIEZ_CENT;
	else if (cantidad == 20) this->elTipoMoneda = VEINTE_CENT;
	else if (cantidad == 50)this->elTipoMoneda = CINCUENTA_CENT;
	else if (cantidad == 100)this->elTipoMoneda = UNO;
	else if (cantidad == 200)this->elTipoMoneda = DOS;
	else
		; //Error

}
```

```
#include <vector>
#include "../Comunes/Moneda.h"
class HWCajero
{
public:
	Moneda recibirMoneda();
	Dinero solicitarProducto();
	void entregarVueltas(std::vector<Moneda> &);
	void entregarProducto();
};

////////////////////////////////////
Moneda HWCajero::recibirMoneda()
{
	float cantidad;
	std::cin >> std::setprecision(2)>> cantidad;
	return(Moneda((unsigned)(cantidad*FACTOR_DINERO)));
}

Dinero HWCajero::solicitarProducto()
{
	std::cout<<"Elegir tipo producto:"<<std::endl;
	std::cout<<"====================="<<std::endl;
	std::cout<<"1: Cafe  40 centimos"<<std::endl;
	std::cout<<"2: Te    35 centimos"<<std::endl;
	std::cout<<"3: Limon 50 centimos"<<std::endl;
	unsigned opcion;unsigned valor;
	std::cin>>opcion;
	switch(opcion) {
	case 1: valor = 40;break;
	case 2: valor = 35;break;
	default: valor = 50;
	}
	std::cout<<"Introducir monedas:"<<std::endl;
	std::cout<<"==================="<<std::endl;
	return(Dinero(valor));
}
void HWCajero::entregarVueltas(std::vector<Moneda> &laListaMonedas)
{
	std::cout<<"Lista de monedas a devolver"<<std::endl;
	for(unsigned i=0;i<laListaMonedas.size();i++)
	std::cout<<((float)laListaMonedas[i].getCantidad())/FACTOR_DINERO
		<<std::setprecision(2)
		<<std::endl;
}
void HWCajero::entregarProducto()
{
	std::cout<<"Recoja el producto seleccionado"<<std::endl;
}
```

```
#include "../Comunes/Moneda.h"

class Ingresos
{
        Dinero elDineroIngresado;
public:
        Ingresos();
        virtual ~Ingresos();
        void anyadirMoneda(Dinero nuevaMoneda)
        { elDineroIngresado += nuevaMoneda;}
        bool haySuficienteDinero(Dinero elPrecioPedido)
        { return (elDineroIngresado>= elPrecioPedido ? true : false); }
        Dinero getDinero(){return elDineroIngresado;}

};
```

```
#include <vector>
#include "../Comunes/Moneda.h"

class Devoluciones
{
        Dinero elDineroAdevolver;
public:
        Devoluciones();
        Devoluciones(Dinero elIngreso,Dinero elPedido)
        {elDineroAdevolver = elIngreso-elPedido;}
        virtual ~Devoluciones()
        void calcularDevoluciones(std::vector<Moneda> &);
};
////////////////////////////
void Devoluciones::calcularDevoluciones(std::vector<Moneda> &laListaMonedas)
{
        if(this->elDineroAdevolver.getTipoDinero() == EURO){
                while(this->elDineroAdevolver.getCantidad()>= 200){
                        laListaMonedas.push_back(Moneda(200));
                        this->elDineroAdevolver-=200;
                }
                while(this->elDineroAdevolver.getCantidad()>= 100){
                        laListaMonedas.push_back(Moneda(100));
                        this->elDineroAdevolver-=100;
                }
                //Repetición con el resto de monedas
                //........
        }
}
```