# Programación 4

## EXAMEN JULIO 2009

## SOLUCIÓN

Por favor siga las siguientes indicaciones:

- Escriba con lápiz.
- Escriba las hojas de un solo lado.
- Escriba su nombre y número de documento en todas las hojas que entregue.
- Numere las hojas e indique el total de hojas en la primera de ellas.
- Recuerde entregar su número de examen junto al examen.

### Problema 1 (30 puntos)

a. ¿Existe alguna diferencia entre "Diagrama de Secuencia" y "Diagrama de Secuencia del Sistema"? Justifique brevemente su respuesta.

Si: mientras que "Diagrama de Secuencia" es un tipo de diagrama UML, "Diagrama de Secuencia del Sistema" es un artefacto de la etapa de Análisis del proceso I&I que utiliza los Diagramas de Secuencia de UML en una forma particular en el contexto de un escenario de un Caso de Uso (colocando una instancia por Actor más una instancia que representa a todo el Sistema como caja negra).

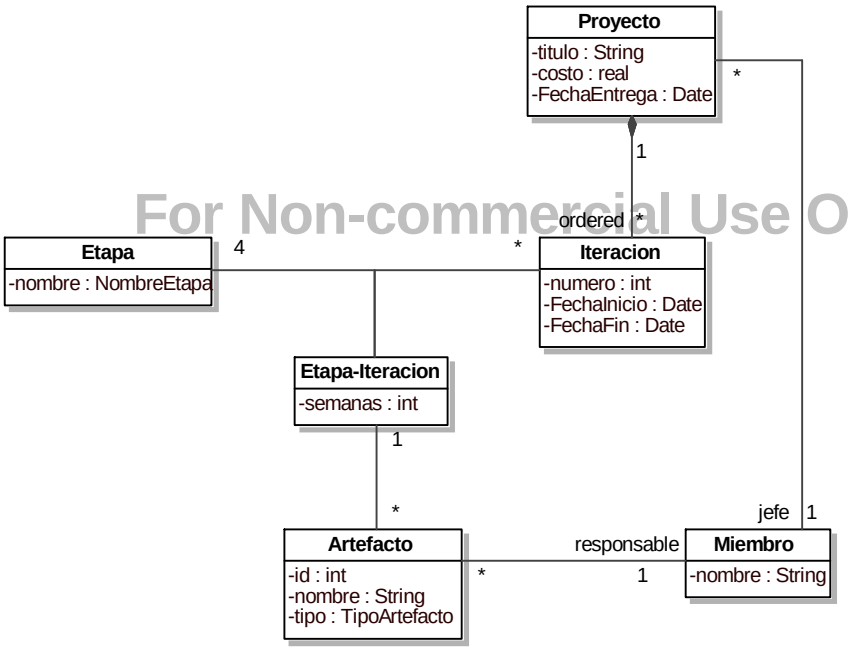
b. Mencione y explique brevemente tres ***frames*** (marcos) posibles de ser utilizados en los Diagramas de Secuencia para incluir comportamiento especial.

LOOP: permite realizar reiteradas veces aquellos mensajes incluidos dentro del ***frame*** mientras que se cumpla la condición. Análogo a la construcción FOR.
ALT: permite seleccionar una región del ***frame*** de acuerdo a la evaluación de su guarda. Se realizarán únicamente aquellos mensajes que pertenezcan a la región cuya guarda sea verdadera. Análogo a la construcción SWITCH o IF-THEN-ELSEIF...etc.
OPT: permite realizar uno o varios mensajes (aquellos contenidos dentro del ***frame***) sujeto a la validez de su guarda. Análogo a la construcción IF-THEN sin ELSE.

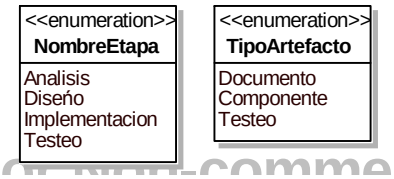
c. Un proyecto de software que siga el modelo de proceso Iterativo e Incremental como el visto en el curso constará de un conjunto de iteraciones (típicamente 2 o más pues de lo contrario no sería iterativo). Cada iteración involucra realizar las etapas de Análisis, Diseño, Implementación y Testeo (obviamente en ese orden). En cada una de estas etapas (para cada iteración) se realizan un conjunto de artefactos (como ser documentos, componentes de software, testeos, etc.) identificados por un nombre y cuyo responsable es un miembro del equipo de desarrollo. Asimismo, el proyecto contará con un jefe de proyecto, tendrá un título que lo diferencia de otros proyectos, un costo estimado y una fecha de entrega. Cada iteración contará con un número, una fecha de inicio y de finalización (por ejemplo, la primera iteración comienza el 1º de Junio de 2010 y finaliza el 30 de Julio del mismo año) y dentro de ella se realizan las 4 etapas ya mencionadas. También interesa conocer la duración de cada una de las etapas dentro de cada iteración (por ejemplo, la etapa de Análisis de la primera iteración llevará 2 semanas, la de Diseño 3 semanas, la de Implementación 2 semanas y la de Testeo 1 semana). No es necesario representar ni la etapa previa a las iteraciones (Relevamiento) ni las posteriores (Liberación y Mantenimiento).

Se pide:

i. Diagrama de dominio de la realidad planteada con restricciones en lenguaje natural.








Restricciones en lenguaje natural:

- El título identifica al proyecto
- El Id identifica al artefacto
- Las 4 etapas de cada iteración deben ser una de cada tipo (Análisis, Diseño, Implementación y Testeo)
- Las semanas de todas las etapas de una iteración no puede superar al tiempo de duración de dicha iteración (determinado como FechaFin – FechaInicio)
- La fecha de entrega del proyecto no puede ser inferior a la fecha de fin de la última iteración
- La fecha de inicio de una iteración debe ser mayor o igual a la fecha de fin de la iteración anterior
- El número de iteración es único para cada proyecto

ii. Especificar en OCL <u>únicamente</u> las dos siguientes restricciones (las cuales no se desprenden de la realidad planteada):

- que el jefe de proyecto sea responsable de algún artefacto de ese proyecto

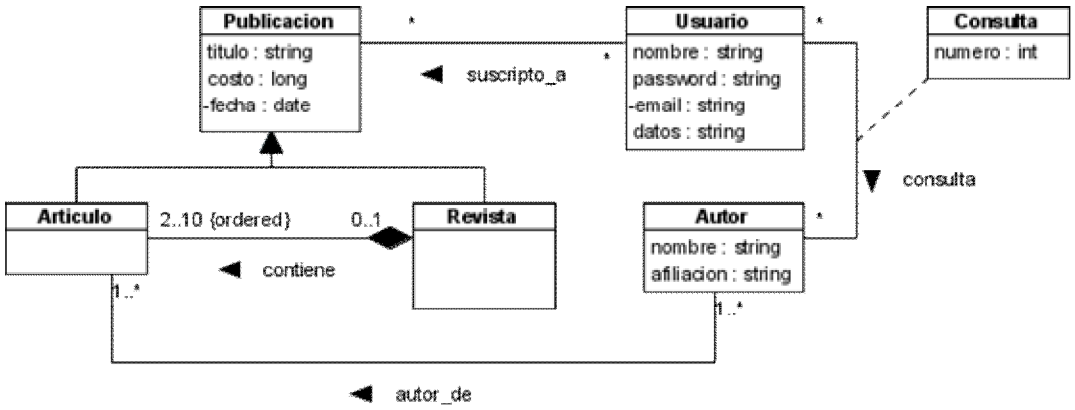
```
Context Proyecto inv:
   self.iteracion.etapa-Iteracion.artefacto.responsable
        -> includes(self.jefe)
```

- los artefactos generados en la etapa de Análisis (para todas las iteraciones) sean únicamente documentos

```
Context Artefacto inv:
   self.etapa-Iteracion.etapa.nombre = NombreEtapa::Análisis
      implies self.tipo = TipoArtefacto::Documento
```

Problema 2 (40 puntos)

Se está desarrollando un sitio web con un sistema de suscripción a publicaciones electrónicas. El modelo de dominio realizado se puede ver en la figura siguiente. El sitio tiene disponible un conjunto de publicaciones electrónicas que son revistas y artículos científicos, que pueden o no formar parte de una revista. Existen usuarios que se registran en el sitio para acceder a las publicaciones electrónicas. Además, se les permite realizar consultas sobre las publicaciones, llevándose registro de la cantidad de consultas que cada usuario realizó para un autor determinado.



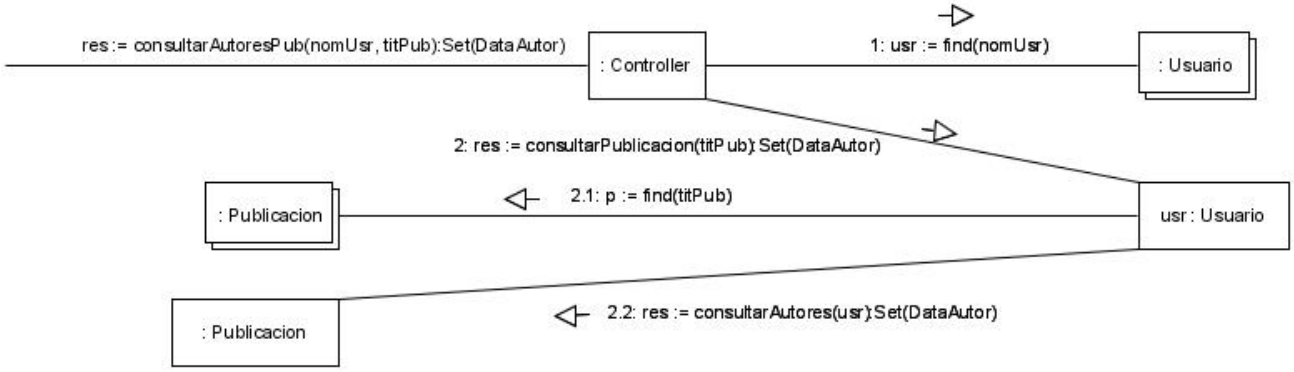

Además, se definieron las siguientes operaciones de sistema.

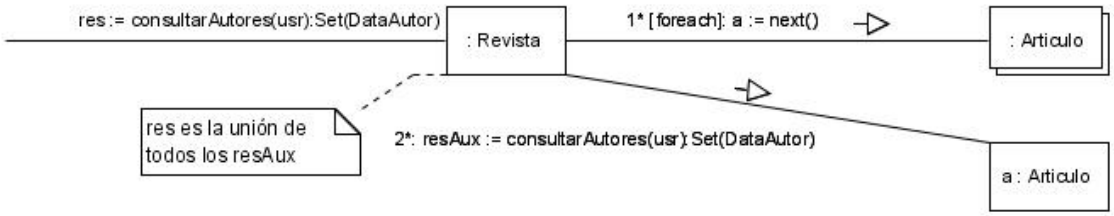
| Operación | consultarAutoresPub(nomUsr:String, titPub:String):Set(DataAutor) |
|---|---|
| Pre- y poscondiciones | |
| def: Definimos **colAutores** como la colección de todas las instancias de Autor asociadss a la Publicación con título = titPub. En el caso de una instancia de Revista, **colAutores** es la unión de los autores de todas las instancias de Artículo linkeadas con la Revista.<br>pre: Existe una Publicación con título = titPub<br>pre: Existe un Usuario con nombre = nomUsr con un link a la Publicacion anterior<br>post: Para cada instancia de Autor en **colAutores**, si existía una instancia de Consulta entre la instancia de Autor y la de Usuario con nombre = nomUsr, entonces se suma 1 a la cantidad de consultas realizadas entre ambos (numero = numero + 1); sino, se crea una instancia de Consulta entre ambas instancias y se inicializa el atributo numero en 1.<br>post: Se retorna una colección de datavalues DataAutor. Cada DataAutor se corresponde con una instancia de Autor existente en **colAutores** y contiene el nombre del Autor y el número de Consulta que ha realizado el Usuario con nombre = nomUsr para ese autor. | |

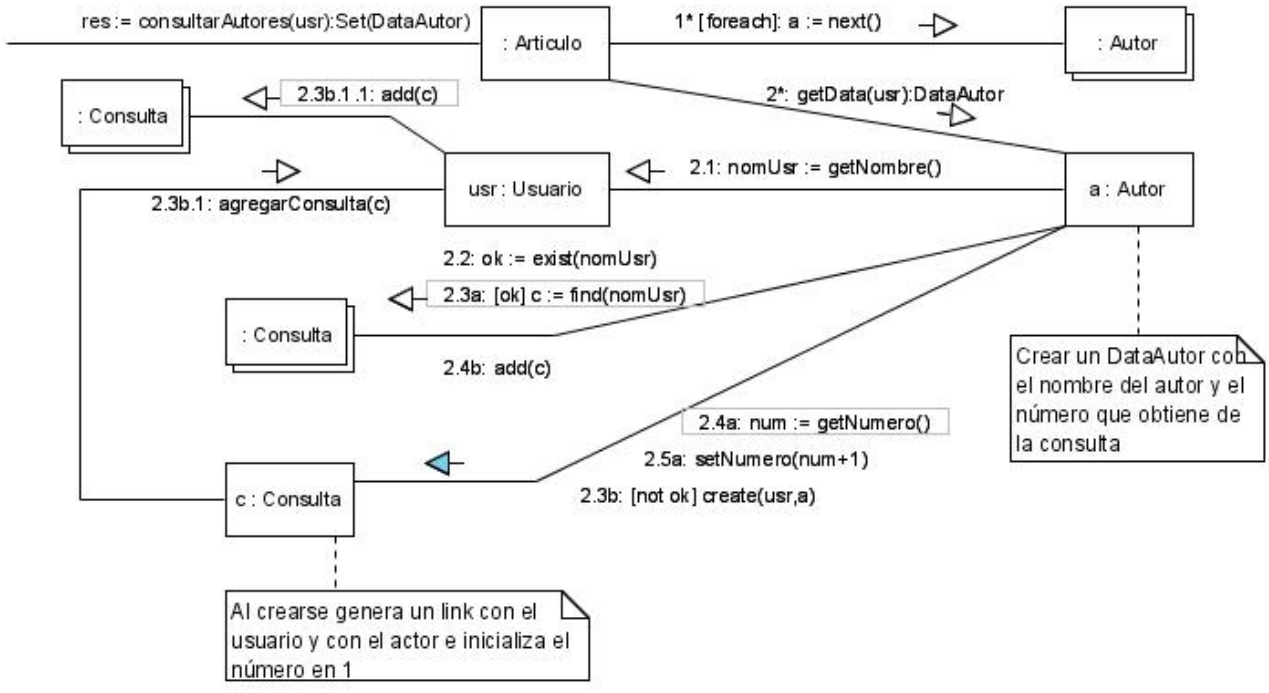
| Operación | bajaUsuario(nomUsr:String) |
|---|---|
| Pre- y poscondiciones | |
| pre: Existe un Usuario con nombre = nomUsr<br>post: Se eliminó la instancia de Usuario con nombre = nomUsr y los links de la instancia de Usuario a las instancias de Publicacion<br>post: Se eliminaron las instancias de Consulta linkeadas con la instancia de usuario | |

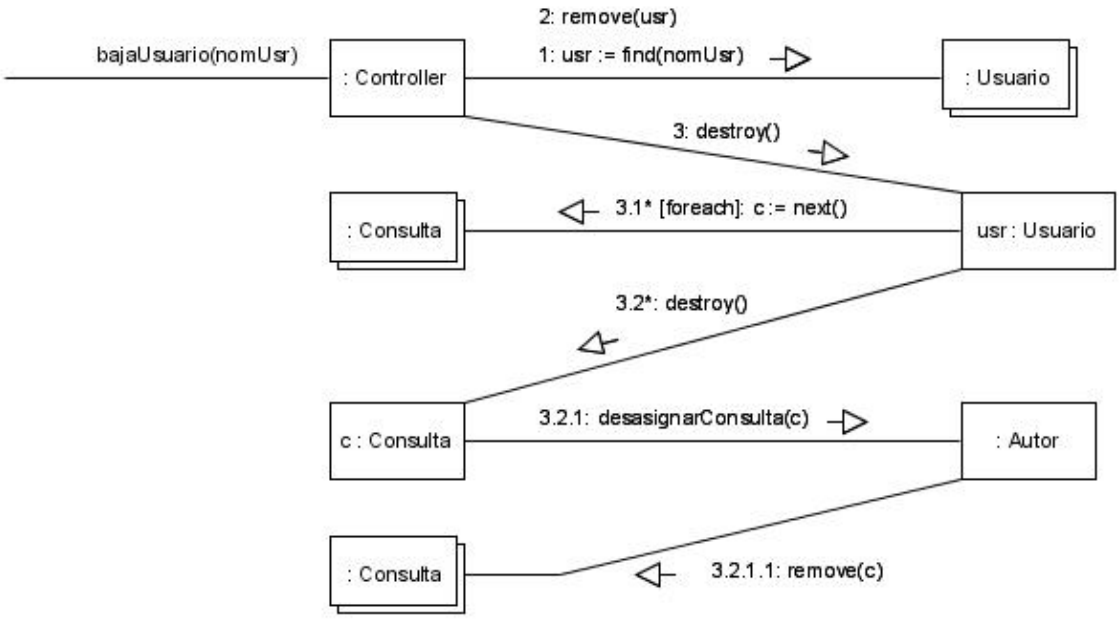
Se pide:

i) Realice los Diagramas de Comunicación para las operaciones especificadas en los contratos. No es necesario indicar las visibilidades.

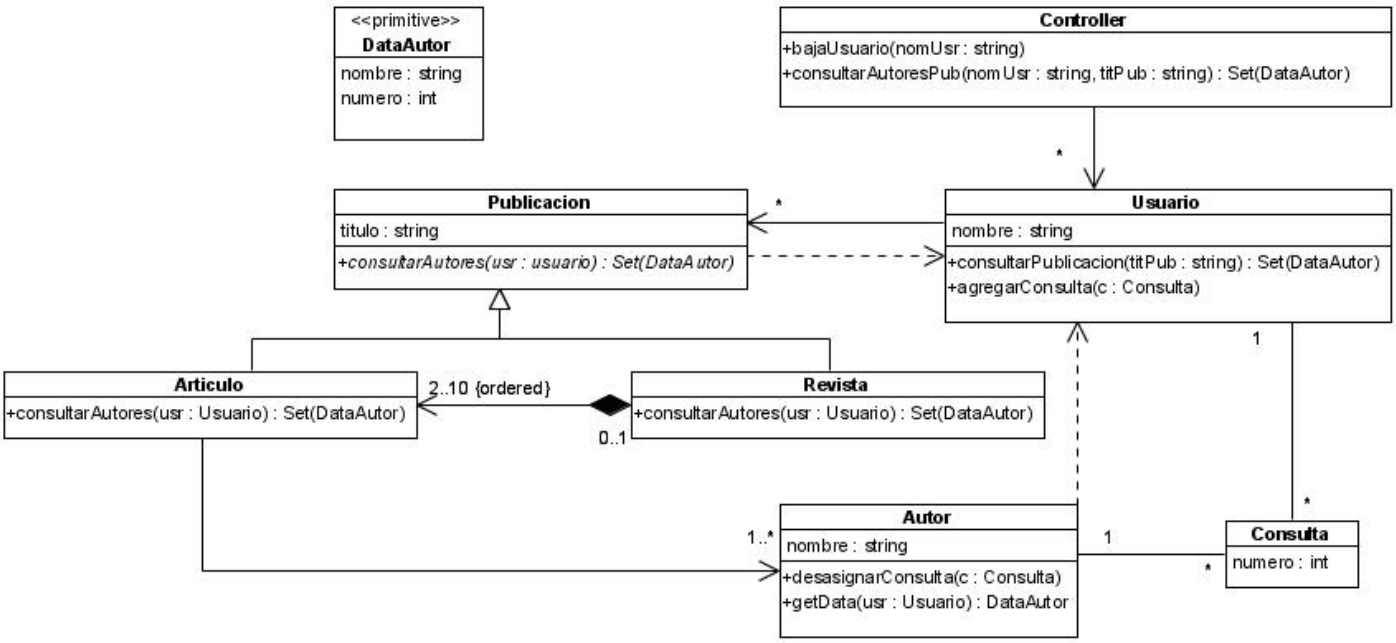


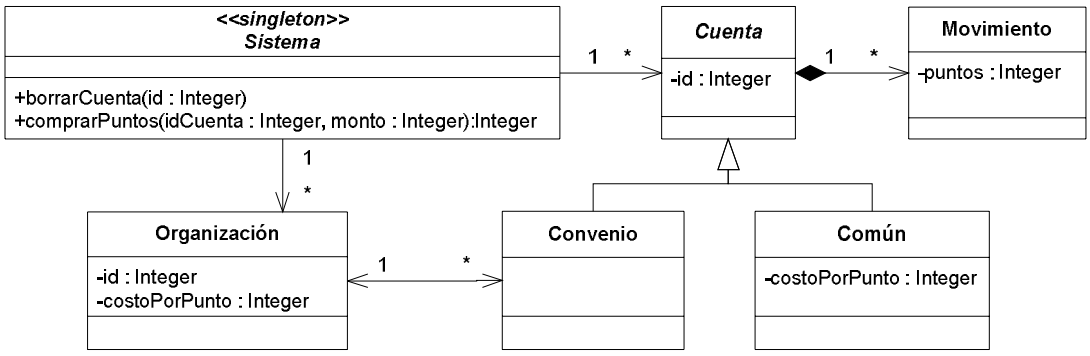



ii) Realice el Diagrama de Clases de Diseño completo de la solución.



Problema 3 (30 puntos)

a) Se le pidió que implemente un sistema para la gestión de las tarjetas de puntos de un supermercado. El sistema deberá registrar las cuentas de puntos de los clientes, los movimientos de puntos de cada cuenta (ganancia y uso de los puntos) y las organizaciones con la que el supermercado tiene convenios y promociones especiales. Se le ha entregado el siguiente diseño parcial que usted deberá respetar:




Se le ha entregado también las siguientes especificaciones para las operaciones pedidas:

| Operación | Sistema::borrarCuenta(idCuenta : Integer) |
|---|---|
| Descripción | Elimina la instancia de Cuenta cuyo atributo id = idCuenta. Elimina también todas las instancias Movimiento asociadas. Si es una cuenta Convenio, desasocia a la cuenta de la Organización correspondiente. |

| Operación | Sistema::comprarPuntos(idCuenta : Integer, monto : Integer):Integer |
|---|---|
| Descripción | Crea una nueva instancia de Movimiento y la asocia a la cuenta cuyo atributo id = idCuenta. El valor del atributo puntos se calcula como el monto dividido el costo por punto de la cuenta. Si es una cuenta Común, el costo por punto de la cuenta es el valor del atributo costoPorPunto. Si es una cuenta Convenio, es el valor del atributo con el mismo nombre en la instancia de Organización asociada. El valor retornado es el valor del atributo puntos de la nueva instancia. Si el monto es menor que el costo por punto, no se crea el movimiento y se lanza una excepción. |

Se pide:

i) Implementar los .h de las clases Sistema, Cuenta, Común y Convenio.

```
class Sistema {
      public:
            static Sistema* getInstancia();
            ~Sistema();
            void borrarCuenta(int id);
            int comprarPuntos(int idCuenta, int monto);

      private:
            Sistema();
            static Sistema* instancia;
            IDictionary* cuentas;
            IDictionary* organizaciones;
};
```

```
class Organizacion: public ICollectible {
	public:
		Organizacion(int id, int costoPorPunto);
		~Organizacion();
		void quitarConvenio(Convenio* convenio);
		int getCostoPorPunto();
	private:
		int id;
		int costoPorPunto;
		IDictionary* convenios;
};

class Cuenta: public ICollectible {
	public:
		Cuenta(int id);
		~Cuenta();
		int comprarPuntos(int monto);
		virtual int getCostoPorPunto() = 0;
	private:
		int id;
		ICollection* movimientos;
};

class Convenio: public Cuenta {
	public:
		Convenio(int id, Organizacion* org);
		~Convenio();
		int getCostoPorPunto();
	private:
		Organizacion* organizacion;
};

class Comun: public Cuenta {
	public:
		Comun(int id, int costo);
		int getCostoPorPunto();
	private:
		int costoPorPunto;
};
```

ii) Implementar en C++ las operaciones especificadas anteriormente y todas aquellas operaciones necesarias para su implementación que pertenezcan a las clases Sistema, Cuenta, Común y Convenio.

Operación borrarCuenta.

```
void Sistema::borrarCuenta(int id) {
	KeyInteger* key = new KeyInteger(id);
	ICollectible* cuenta = cuentas->find(key);

	if (cuenta == NULL) {
		delete key;
		throw invalid_argument("No existe la cuenta");
	}
	cuentas->remove(key);
	delete cuenta;
	delete key;
}
```

```
Cuenta::~Cuenta() {
	IIterator* it = movimientos->getIterator();
	ICollectible* elemento;

	while(it->hasCurrent()) {
		elemento = it->getCurrent();
		it->remove();
		delete elemento;
	}

	delete it;
	delete movimientos;
}

Convenio::~Convenio() {
	organizacion->quitarConvenio(this);
}
```

Operación comprarPuntos.

```
int Sistema::comprarPuntos(int idCuenta, int monto) {
	KeyInteger* key = new KeyInteger(idCuenta);
	Cuenta* cuenta = (Cuenta*) cuentas->find(key);
	delete key;

	if (cuenta != NULL) {
		return cuenta->comprarPuntos(monto);
	} else {
		throw invalid_argument("No existe la cuenta");
	}
}

int Cuenta::comprarPuntos(int monto) {
	int costoPorPunto = getCostoPorPunto();
	if (monto < costoPorPunto)
		throw out_of_range("monto insuficiente");

	int puntos = monto / costoPorPunto;
	Movimiento* mov = new Movimiento(puntos);
	movimientos->add(mov);
	return puntos;
}

int Convenio::getCostoPorPunto() {
	return organizacion->getCostoPorPunto();
}

int Comun::getCostoPorPunto() {
	return costoPorPunto;
}

Movimiento::Movimiento(int puntos) : puntos(puntos) {

}
```