Programación 4

EXAMEN DICIEMBRE 2006

SOLUCIÓN

Problema 1 (25 puntos)

a) Responda de forma breve y concisa las siguientes preguntas:

i) ¿Qué es una agregación compuesta?

Es una agregación en la que las partes son exclusivas del compuesto. Además, en general, una acción sobre el compuesto se propaga sobre las partes (ver slide 38 de "Modelado de Dominio")

ii) ¿Qué es un tipo asociativo?

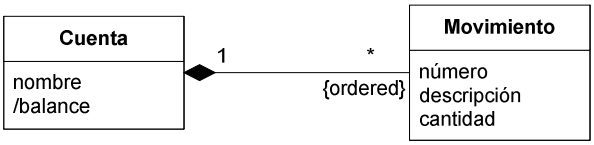
Es un elemento del que es clase y asociación al mismo tiempo. Permite agregar propiedades a las asociaciones (ver slides 40 y 41 de "Modelado de Dominio")

iii) ¿Cuándo es necesario utilizar un rol?

Los roles son necesarios para eliminar la ambigüedad sobre la asociación a la que se hace referencia cuando existe más de una asociación entre 2 clases.

b)

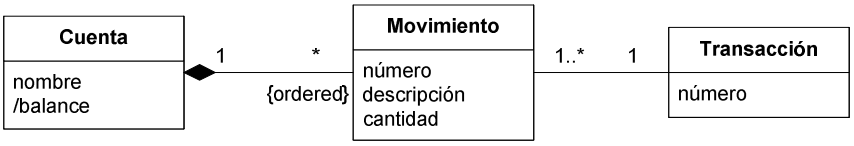
i) Fase 1: Cuentas de inventario.




```
context Cuenta
-- El nombre identifica a una cuenta.
inv: Cuenta.allInstances()->isUnique(nombre)
-- El número de movimiento es único dentro de una cuenta
inv: self.movimiento->isUnique(número)
-- El balance se corresponde con los movimientos
inv: self.balance = self.movimientos.cantidad->sum()

context Movimiento
-- La cantidad de un movimiento es distinta de 0
inv: self.cantidad <> 0
```

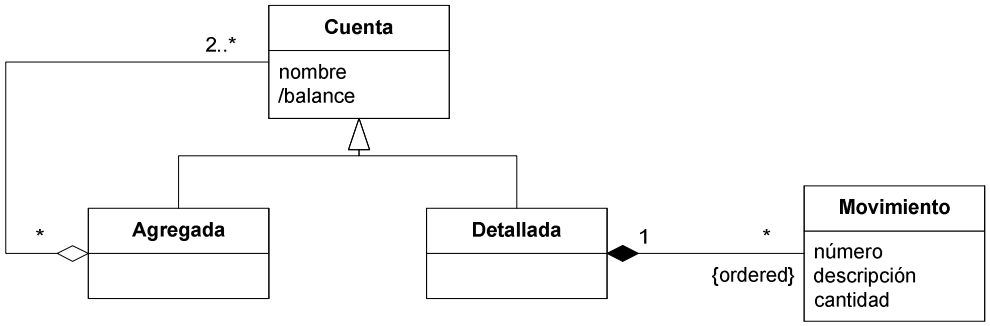
ii) Fase 2: Transacciones de inventario.




```
context Transacción
-- El número identifica a una transacción.
inv: Transacción.allInstances()->isUnique(número)
-- Todas las transacciones son del mismo tipo.
inv: self.movimiento->forAll(m | m.cantidad < 0)
     or self.movimiento->forAll(m | m.cantidad > 0)
-- Todos los movimientos son de diferentes cuentas.
```

```
inv: self.movimiento.cuenta->isUnique(nombre)
```

iii) Fase 3: Cuentas agregadas.




```
context Agregada
-- balance es la suma de los balances parciales
inv: self.balance = self.cuenta.balance->sum()
-- Una cuenta agregada no puede agregarse a si misma
inv: self.cuenta->excludes(self)

context Detallada

-- El número de movimiento es único dentro de una cuenta
-- (cambia de contexto respecto al punto i)
inv: self.movimiento->isUnique(número)
-- El balance se corresponde con los movimientos
-- (cambia de contexto respecto al punto i)
inv: self.balance = self.movimientos.cantidad->sum()
```

Problema 2 (25 puntos)

a) Las precondiciones especifican:

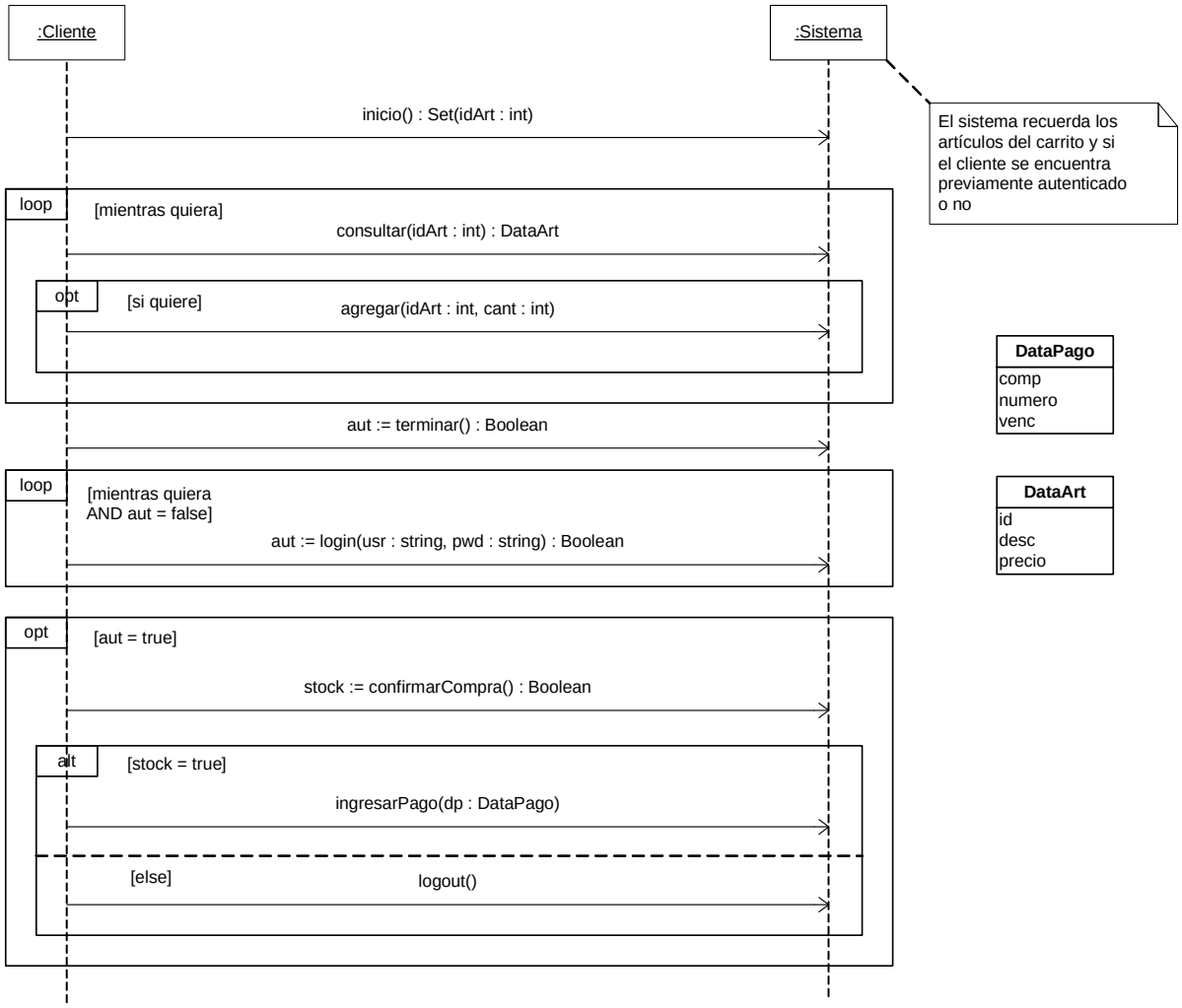
- Los valores de los parámetros de la operación
- El estado del sistema antes de ejecutar la operación, en términos de:
  - Que un objeto existe
  - Que un objeto no existe
  - Que un link existe
  - Que un link no existe
  - Propiedades sobre valores de atributos de objetos

Las postcondiciones especifican:

- El valor de retorno de la operación
- El estado del sistema luego de ejecutar la operación, en términos de:
  - Que un objeto existe
  - Que un objeto no existe
  - Que un link existe
  - Que un link no existe

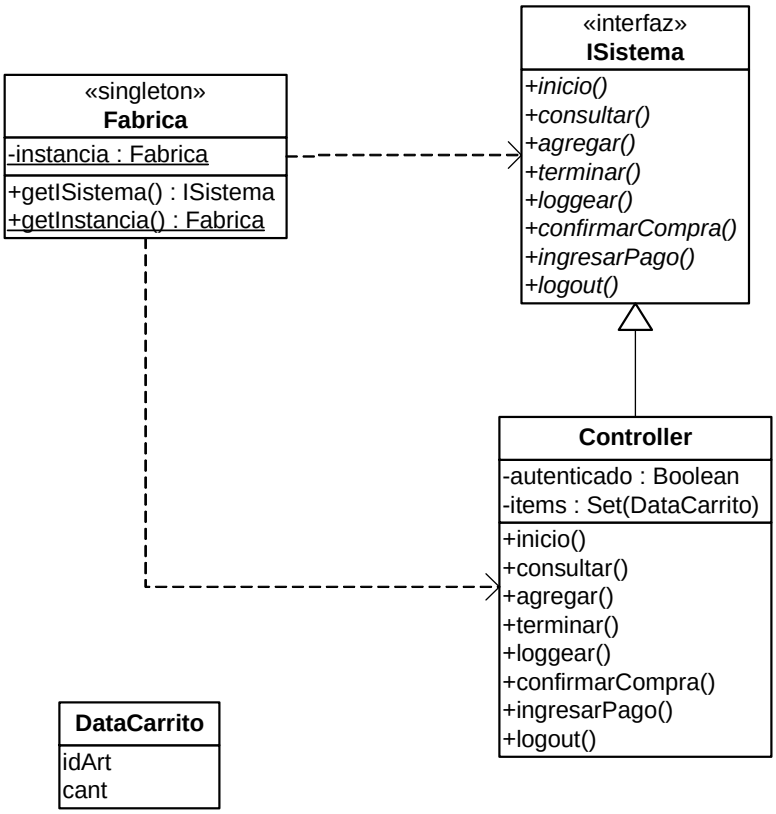
b)

Parte i:



Nota: se podría representar también el logout del cliente luego de ingresar el pago.

Parte ii:



Notas:

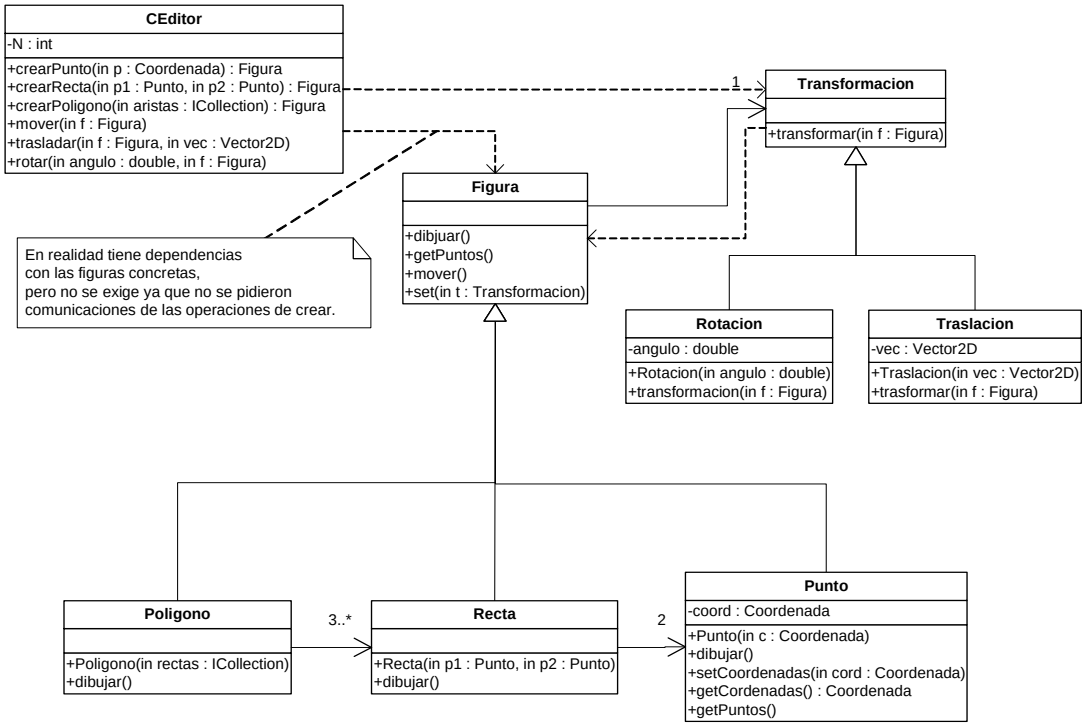
- Se omitieron, por simplicidad, los parámetros de las operaciones y los tipos de retorno.
- La forma de modelar los ítems del carrito para cada cliente puede variar. Aquí se decidió utilizar una colección de DataCarrito, siendo posible otras alternativas.
- Es importante que el Controller **no** sea Singleton. De esta forma a cada cliente conectado al sistema le será asignada (mediante la Fábrica) una nueva instancia del Controller. De lo contrario (si el Controller es Singleton), éste debe guardar la memoria para todos los usuarios del sistema (por ejemplo mediante una colección de carritos por un lado, y sabiendo para cada cliente si ya se encuentra loggeado o no).

Problema 3 (25 puntos)

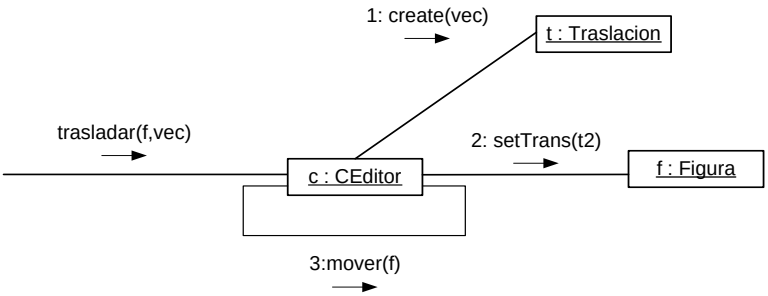
a) Defina brevemente el problema tipo que resuelven los patrones Strategy y State.

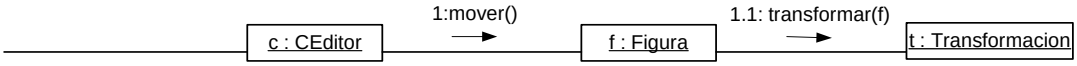
- **Strategy**: Definir una familia de algoritmos, encapsularlos y hacerlos intercambiables. Esto permite que el algoritmo varíe dependiendo del cliente que lo utiliza
- **State**: El comportamiento de un objeto depende de su estado, y debe cambiar en tiempo de ejecución dependiendo de ese estado.
  Las operaciones tienen largas sentencias condicionales con múltiples ramas que dependen del estado del objeto. El patrón State pone cada rama de la condición en una clase aparte. Esto nos permite tratar al estado del objeto como un objeto de pleno derecho que puede variar independientemente de otros objetos

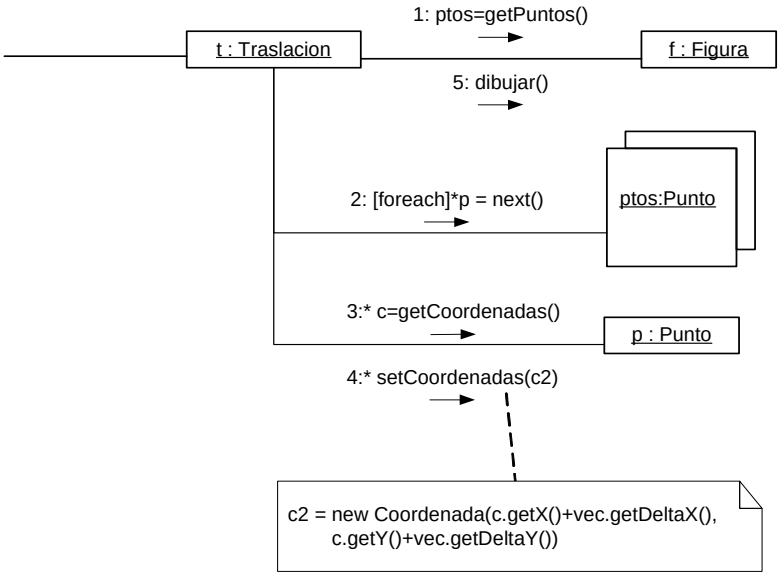
b.i) Realice el DCD completo correspondiente a las figuras y transformaciones.



b.ii) Realice un diagrama de comunicación que muestre todas las interacciones que ocurren cuando se ejecuta la operación `trasladar()` en la Clase `CEditor`.

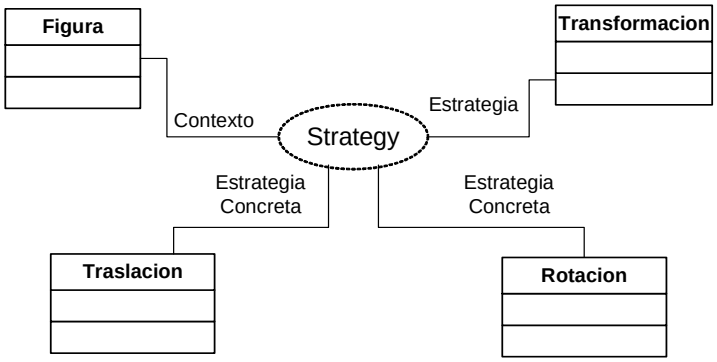




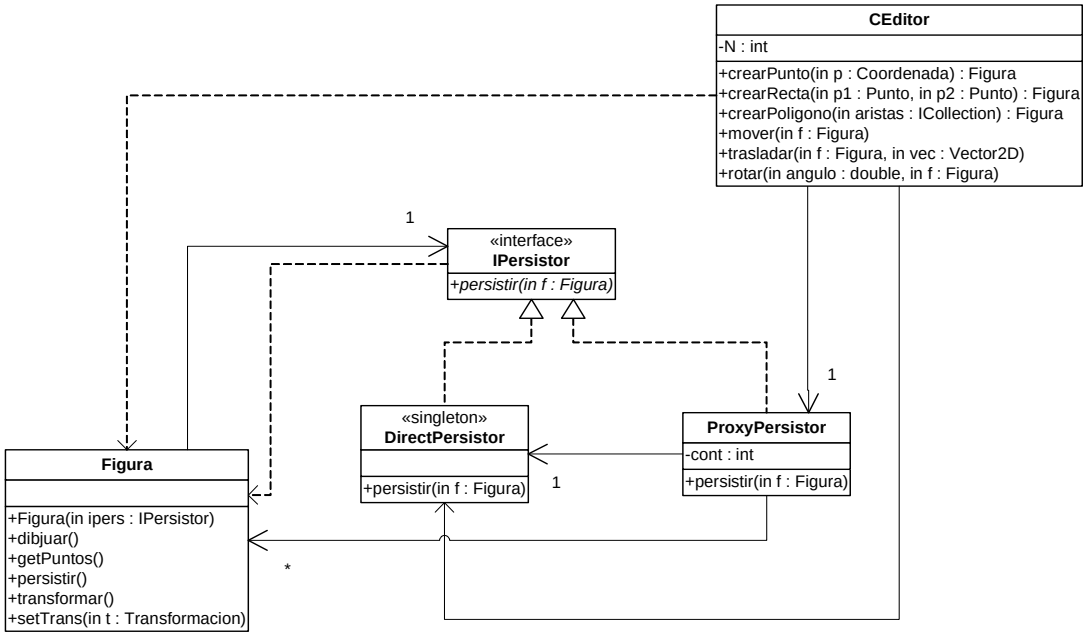


b.iii) Explique qué patrón(es) de diseño utilizó indicando (para cada patrón) las clases participantes y sus roles.

Strategy para las transformaciones y las figuras.

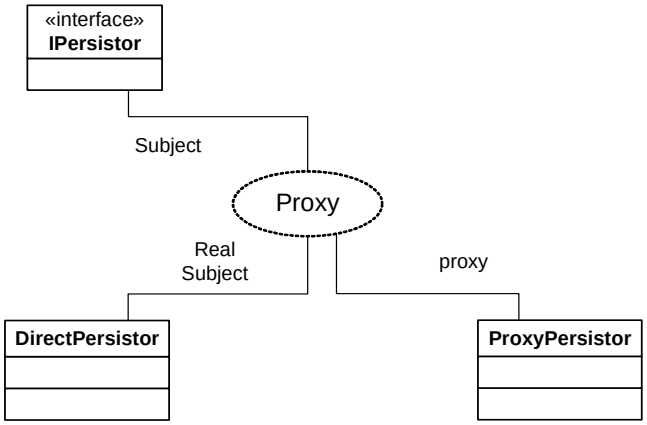


ci) Realice el DCD completo correspondiente al requerimiento de persistencia, incluyendo solo lo necesario del DCD de la parte anterior.



cii) Explique qué patrón(es) de diseño utilizó indicando (para cada patrón) las clases participantes y sus roles.

Se utilizó el patrón proxy para resolver el requerimiento de persistencia.



Otra solución posible es que la clase ProxyPersistor sea singleton en lugar de que haya una instancia como atributo de CEditor.

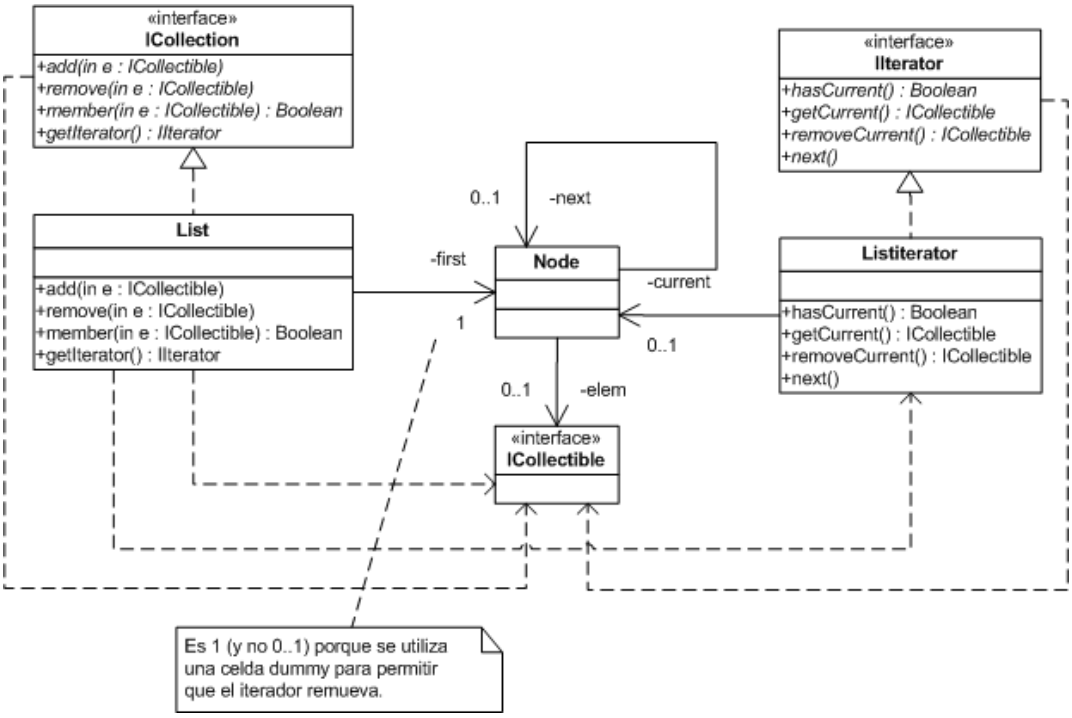
Problema 4 (25 puntos)

a) Explique brevemente que es una colección genérica y una colección concreta (o tipada). Indique cómo se relacionan ambos conceptos a nivel de diseño.

Ver slides 8, 9, 13 y 14 de teórico del juego "Implementación – Colecciones".

b)

i. Realice el DCD de una realización basada en una lista encadenada de las interfaces `ICollection` e `IIterator`. Detallar el comportamiento de cada una de las operaciones de la interfaz `IIterator`.



`IIterator`:

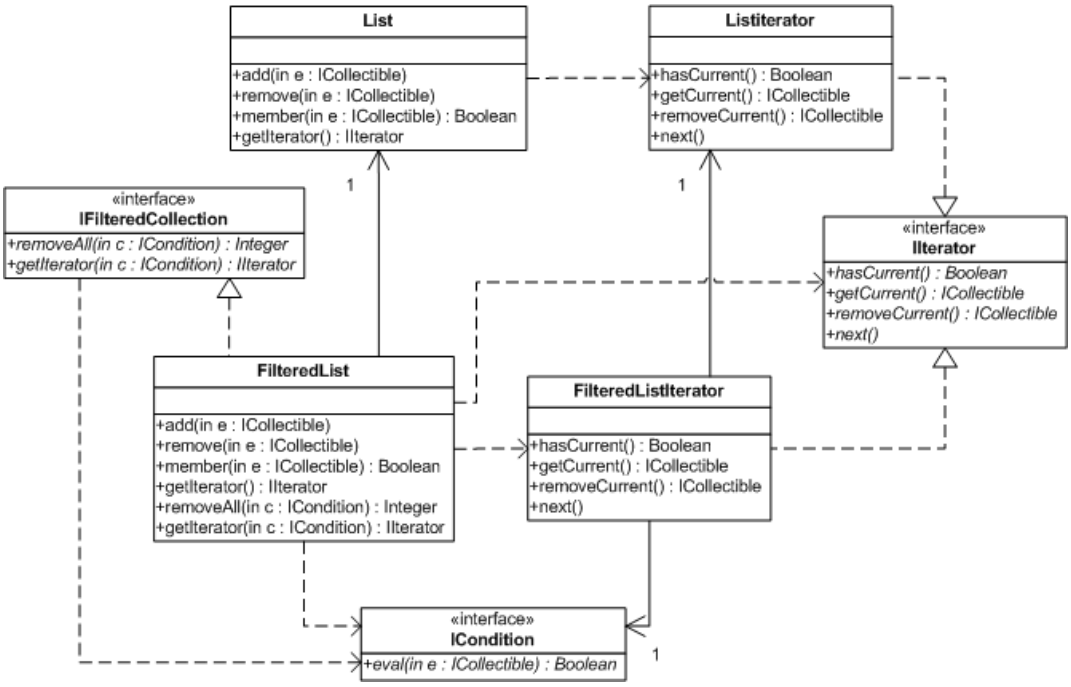
`hasCurrent`: devuelve true si el iterador tienen un elemento actual.

`getCurrent`: devuelve el elemento actualmente apuntado por el iterador. Tiene como precondición que la operación `hasCurrent` devuelva true.

`removeCurrent`: quita el elemento actual de la colección avanzando el iterador y devolviendo el elemento eliminado.

`next`: avanza el iterador. Tiene como precondición que la operación `hasCurrent` devuelva true.

ii. Diseñe (sólo DCD) e implemente en C++ esta extensión (incluyendo `ICondition`) sin modificar ninguna de las clases definidas en la parte anterior. No incluya el código de la interfaz `IFilteredCollection` ni de las clases o interfaces definidas en la parte anterior.



// ICondition.hh

```
class ICondition
{
public:
      virtual bool eval(ICollectible*) = 0;
      virtual ~ICondition();
};
```

// ICondition.cc

```
ICondition::~ICondition()
{
}
```

// FilteredList.hh

```
class FilteredList : public IFilteredCollection
{
private:
      List* list;
public:
      FilteredList();
      void add(ICollectible*);
      void remove(ICollectible*);
      bool member(ICollectible*);
      int removeAll(ICondition*);
      IIterator* getIterator();
      IIterator* getIterator(ICondition*);
      ~FilteredList();
};
```

// FilteredList.cc

```
FilteredList::FilteredList() : list(new List())
{
```

```
}

void FilteredList::add(ICollectible* c)
{
      this->list->add(c);
}

void FilteredList::remove(ICollectible* c)
{
      this->list->remove(c);
}

bool FilteredList::member(ICollectible* c)
{
      return this->list->member(c);
}

IIterator* FilteredList::getIterator()
{
      return this->list->getIterator();
}

int FilteredList::removeAll(ICondition* c)
{
      IIterator* i = this->list->getIterator();
      int result = 0;

      while (i->hasCurrent())
      {
            if (c->eval(i->getCurrent()))
            {
                  i->removeCurrent();
                  result++;
            }
            else
                  i->next();
      }

      delete i;

      return result;
}

IIterator* FilteredList::getIterator(ICondition* c)
{
      return new FilteredListIterator(this->list->getIterator(), c);
}

FilteredList::~FilteredList()
{
      delete this->list;
}

// FilteredListIterator.hh

class FilteredListIterator : public IIterator
{
private:
      IIterator* innerIterator;
      ICondition* condition;
public:
```

```
      FilteredListIterator(IIterator*, ICondition*);
      ICollectible* getCurrent();
      void next();
      bool hasCurrent();
      ICollectible* removeCurrent();
      ~FilteredListIterator();
};

// FilteredListIterator.cc

FilteredListIterator::FilteredListIterator(IIterator* i, ICondition*
c) : innerIterator(i), condition(c)
{
      if (i->hasCurrent() && !c->eval(i->getCurrent()))
            this->next();
}

ICollectible* FilteredListIterator::getCurrent()
{
      return this->innerIterator->getCurrent();
}

void FilteredListIterator::next()
{
      this->innerIterator->next();

      while (this->innerIterator->hasCurrent() && !this->condition-
>eval(this->innerIterator->getCurrent()))
            this->innerIterator->next();
}

bool FilteredListIterator::hasCurrent()
{
      return this->innerIterator->hasCurrent();
}

ICollectible* FilteredListIterator::removeCurrent()
{
      ICollectible* result = this->innerIterator->removeCurrent();

      if (this-innerIterator->hasCurrent() && !this->condition-
>eval(this->innerIterator->getCurrent()))
            this->next();

      return result;
}

FilteredListIterator::~FilteredListIterator()
{
      delete this->innerIterator;
      delete this->condition;
}
```

iii. Implementar en C++ la operación `envioPublicidad` incluyendo el código de la o las clases auxiliares utilizadas. No incluya las clases `ManejadorClientes`, `Cliente`, `String` ni ninguna de las clases o interfaces definidas en las partes anteriores.

```
// CondicionCliente.hh

class CondicionCliente : public ICondition
```

```
{
private:
	int minEdad;
	int maxEdad;
	String departamento;
public:
	CondicionCliente(int minEdad, int maxEdad, String departamento);
	bool eval(ICollectible*);
};

// CondicionCliente.cc

CondicionCliente::CondicionCliente(int minEdad, int maxEdad, String
departamento) : minEdad(minEdad), maxEdad(maxEdad),
departamento(departamento)
{
}

bool CondicionCliente::eval(ICollectible* ic)
{
	bool result = false;
	Cliente* c = dynamic_cast<Cliente*>(ic);

	if (c != NULL)
	{
		if ((c->getEdad() >= this->minEdad) && (c->getEdad() <=
this->maxEdad))
		{
			if ((this->departamento.length() == 0) || (this-
>departamento.equals(c->getDepartamento())))
				result = true;
		}
	}

	return result;
}

// envioPublicidad

void ControladorClientes::envioPublicidad(int edadMinima, int
edadMaxima, String departamento, String texto)
{
	CondicionCliente* c = new CondicionCliente(edadMinima,
edadMaxima, departamento);
	IIterator* i = ManejadorClientes::getInstance()->getClientes(c);

	while (i->hasCurrent())
	{
		((Cliente*)i->getCurrent())->mandarSms(texto);
		i->next();
	}

	delete i;
}
```