Programación 4

SOLUCIÓN PARCIAL FINAL EDICIÓN 2006

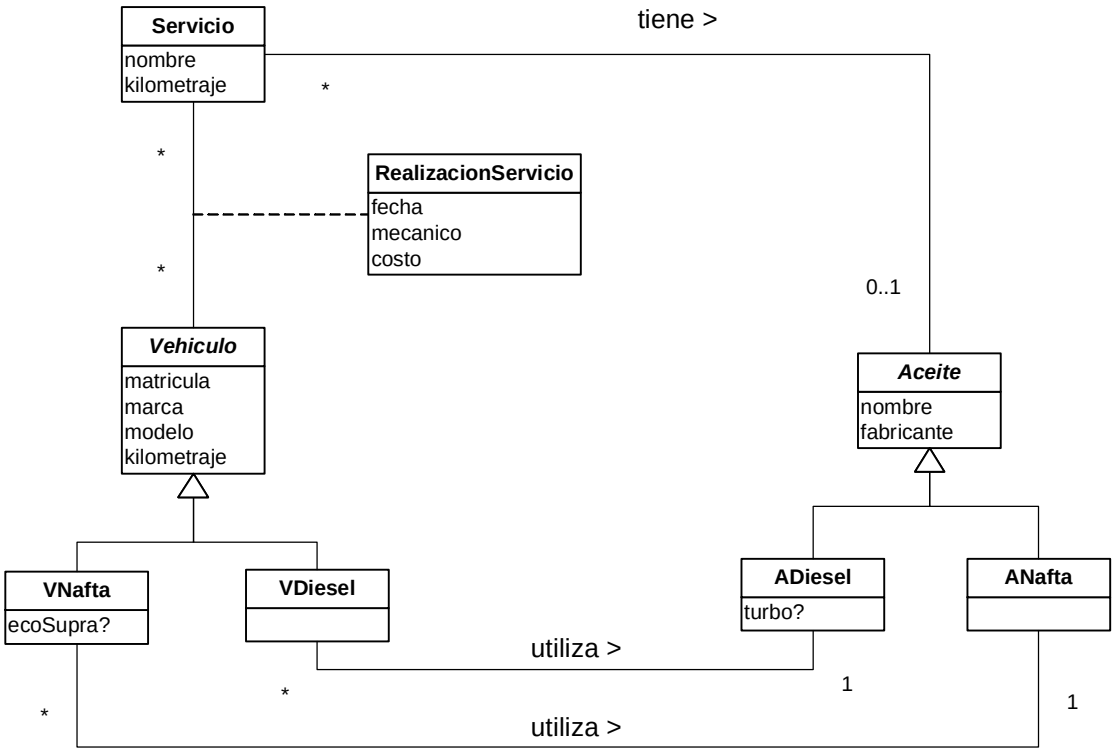
Problema 1 (25 puntos)

a) Defina "tipo asociativo" haciendo especial énfasis en las multiplicidades de éste.

Un tipo asociativo es un elemento que es tanto una asociación como un concepto. Permite conceptualizar una asociación como si fuera un concepto, pudiéndole colocar atributos y asociarlo con otros conceptos. Con respecto a las multiplicidades, existe una única instancia del tipo asociativo por cada par de instancias de los conceptos que asocia.

b) Construir el Modelo de Dominio de toda la realidad (incluyendo el caso de uso presentado) y presentarlo en un diagrama utilizando UML. Las restricciones deben ser expresadas en lenguaje natural y en OCL.

Modelo de Dominio



Restricciones

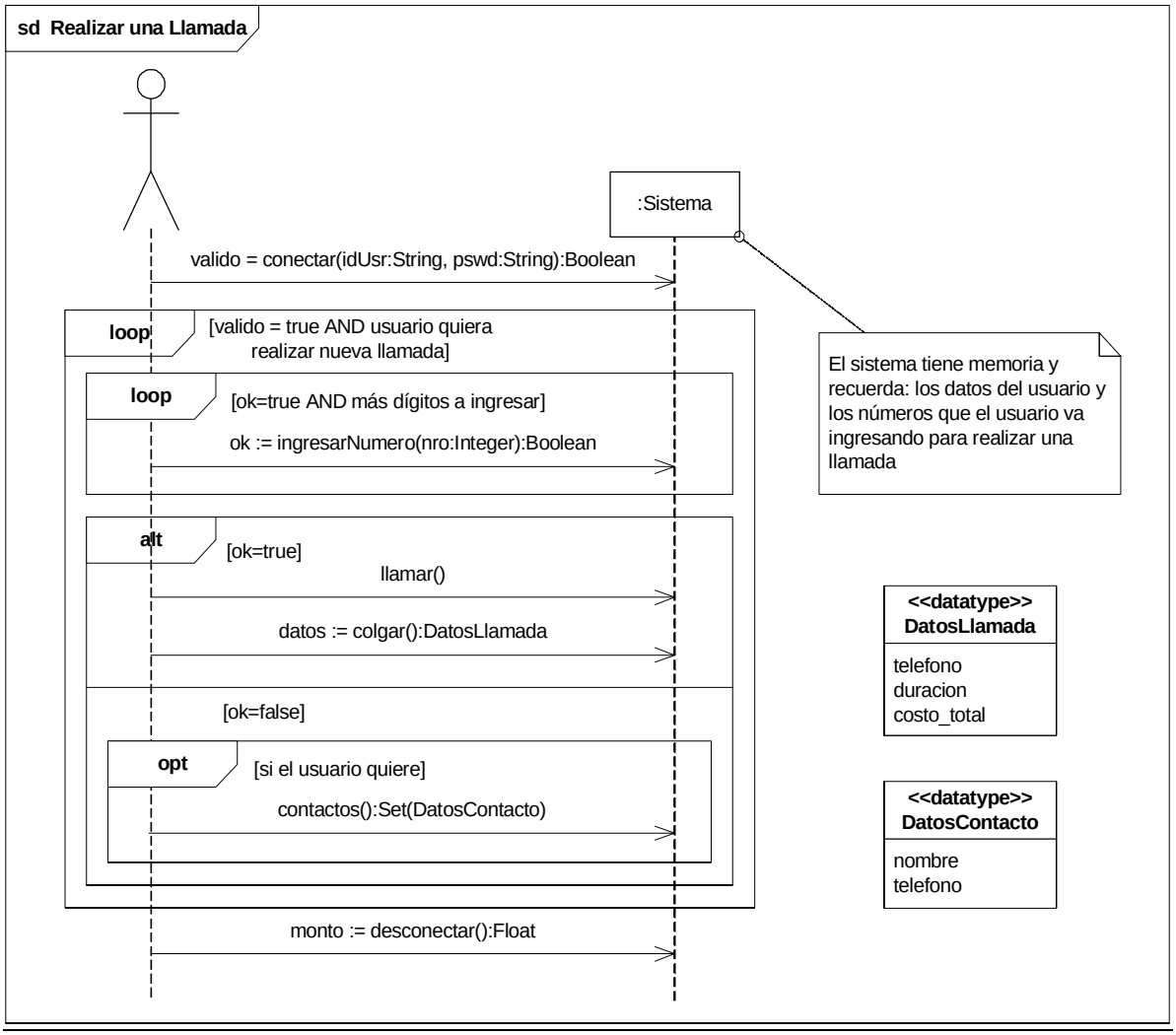
```
-- el nombre identifica al servicio (Restricción de Unicidad)
context Servicio inv:
       Servicio.allInstances() à isUnique(nombre)

-- la matrícula identifica al vehículo (Restricción de Unicidad)
context Vehiculo inv:
       Vehiculo.allInstances() à isUnique(matricula)

-- el nombre identifica al aceite  (Restricción de Unicidad)
context Aceite inv:
       Aceite.allInstances() à isUnique(nombre)

-- el kilometraje del vehículo al momento de hacer un servicio debe ser mayor o igual
-- que el kilometraje definido por el servicio (Restricción de Regla de Negocio)
context RealizacionServicio inv:
       self.vehiculo.kilometraje >= self.servicio.kilometraje

-- los servicios que utilicen un aceite del fabricante "Shell" tendrán un costo superior
-- a 2.500 pesos  (Restricción de Regla de Negocio)
context Aceite inv:
       self.servicio à forAll ( s:Servicio | self.fabricante="Shell" implies
                                  s.realizacionServicio.costo > 2500 )

-- el aceite utilizado en un servicio a un vehículo debe ser el mismo aceite que
-- ese vehículo utiliza (Restricción Circular)
context VNafta inv:
       self.servicio à forAll ( s:Servicio | s.aceite à size()=1 implies
                                  s.aceite.oclIsTypeOf(ANafta) and
                                  s.aceite.oclAsType(ANafta) = self.aNafta )
context VDiesel inv:
       self.servicio à forAll ( s:Servicio | s.aceite à size()=1 implies
                                  s.aceite.oclIsTypeOf(ADiesel) and
                                  s.aceite.oclAsType(ADiesel) = self.aDiesel )
```

Problema 2 (25 puntos)

a) Conteste brevemente qué especifica y cómo está estructurado un Contrato.

Un contrato de software especifica el comportamiento o efecto de una operación. Se estructura especificando un nombre para la operación, las responsabilidades de la misma, el tipo del cual la operación es propiedad, las referencias cruzadas (caso(s) de uso a los que pertenece la operación), notas generales, salida de la operación, las pre- y post-condiciones, y opcionalmente, snapshots que ejemplifiquen el estado de la instancia a la que se le aplicó la invocación, previo y posterior a la invocación.

b.i) Realice un único Diagrama de Secuencia de Sistema para el caso de uso anterior, incluyendo toda la información contenida en el mismo.



b.ii) Especifique las pre- y post-condiciones de la operación ***altaContacto()***.

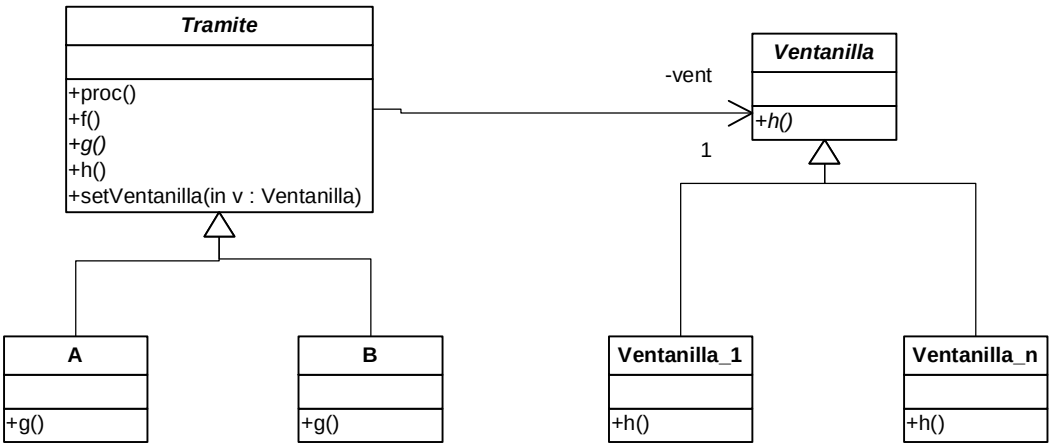
Operación	altaContacto(ID:String, nomC:String, numC:Integer):Boolean
Pre- y poscondiciones	
pre: existe un objeto de la clase Usuario con idUsr = ID (esta pre puede obviarse si se considera que la sesión está abierta, i.e., el usuario existe) post: si el objeto Contacto con nombre = nomC no existe, se crea una instancia de Contacto con nombre = nomC y numero = numC; y se genera un link entre esa instancia y la instancia de Usuario con idUsr = ID post: si la instancia de Contacto con nombre = nomC ya existe entonces se devuelve false, en caso contrario se devuelve true	

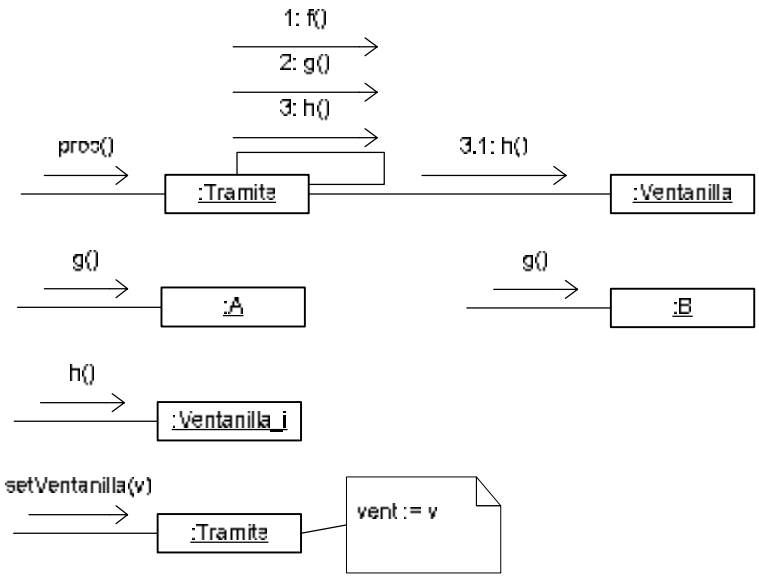
Problema 3 (25 puntos)

a) Nombrar tres criterios GRASP y explicar brevemente en qué consiste cada uno.

Ver material de teórico.

b.i) Diseñar la operación `proc()` (Diagrama de Clases de Diseño y Diagrama de Comunicación) teniendo en cuenta que se busca minimizar la duplicación de código, respetando las consideraciones de diseño dadas.





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

Para el diseño de la clase Tramite y sus clases derivadas se utilizó el patrón Template Method, donde Tramite es la clase abstracta (proc() es el template method), A y B son las clases concretas (g() es una operación primitiva). Para el diseño de la operación h() se utilizó el patrón Strategy, donde Tramite es el contexto Tramite::h() es la interface del contexto), y Ventanilla es la estrategia (Ventanilla::h() es la interface del algoritmo).

Problema 4 (25 puntos)

a) Defina brevemente cómo se mapean a C++ las relaciones de generalización, realización y dependencia.

Ver diapositivas de teórico 13, 14 y 22 del juego "generación de código".

b) Implementar en C++ completamente la colaboración salvo la clase `Cambios`. Incluir el código relativo al patrón Singleton en la clase `ControladorDocumentos`. Del datatype `DataDocumento` es necesario incluir únicamente su módulo de definición, es decir su `.h`.

```
// DataDocumento.hh
class DataDocumento : public ICollectible
{
private:
      String nombre;
      String descripcion;
      int cantidadVersiones;

public:
      DataDocumento(String nom, String desc, int cantVersiones);
      void setNombre(String nom);
      void setDescripcion(String desc);
      void setCantidadVersiones(int cantVersiones);
      String getNombre();
      String getDescripcion();
      int getCantidadVersiones();
};


// Documento.hh
class Documento : public ICollectible
{
private:
      String nombre;
      String descripcion;

public:
      virtual DataDocumento getDataDocumento();
      virtual ~Documento();
};


// Documento.cc
DataDocumento Documento::getDataDocumento()
{
      return DataDocumento(this->nombre, this->descripcion, 1);
}

Documento::~Documento(){}


// Binario.hh
class Binario : public Documento
{
};
```

```
// TextoPlano.hh
class TextoPlano : public Documento
{
private:
	ICollection * cambios;

public:
	DataDocumento getDataDocumento();
	~TextoPlano();
};


// TextoPlano.cc
DataDocumento TextoPlano::getDataDocumento()
{
	DataDocumento aux = Documento::getDataDocumento();
	aux.setCantidadVersiones(this->cambios->size() + 1);

	return aux;
}

TextoPlano::~TextoPlano()
{
	IIterator * i = this->cambios->getIterator();

	while (i->hasCurrent())
	{
		delete i->removeCurrent();
	}

	delete i;
	delete this->cambios;
}


// ControladorDocumentos.hh
class ControladorDocumentos
{
private:
	static ControladorDocumentos * instance;
	IStringDictionary * documentos;

	ControladorDocumentos();

public:
	static ControladorDocumentos * getInstance();

	ICollection * listarDocumentos();
	void borrarDocumento(String nombre);
};


// ControladorDocumentos.cc
ControladorDocumentos * ControladorDocumentos::instance = NULL;

ControladorDocumentos::ControladorDocumentos()
{
	this->documentos = new List();
}
```

```
ControladorDocumentos * ControladorDocumentos::getInstance()
{
	if (ControladorDocumentos::instance == NULL)
	{
		ControladorDocumentos::instance =
			new ControladorDocumentos();
	}

	return ControladorDocumentos::instance;
}

ICollection * ControladorDocumentos::listarDocumentos()
{
	IIterator * i = this->documentos->getIterator();
	List * resultado = new List();

	while (i->hasCurrent())
	{
		// Se crea una copia para devolver
		resultado->add(new DataDocumento(
			((Documento*)i->getCurrent())->getDataDocumento()
		));

		i->next();
	}

	delete i;

	return resultado;
}

void ControladorDocumentos::borrarDocumento(String nombre)
{
	delete this->documentos->remove(nombre);
}
```