Programación 4

EXAMEN - DICIEMBRE 2007

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 numero de examen junto al examen

Problema 1 (25 puntos)

a)

i. Indicar y definir cuáles son las posibles relaciones entre dos conceptos de un Modelo de Dominio.

Asociación: relación que describe una relación semántica entre dos clasificadores.
Generalización: relación entre un elemento más general y un elemento más específico. El elemento más específico es consistente con el más general y puede tener información adicional.

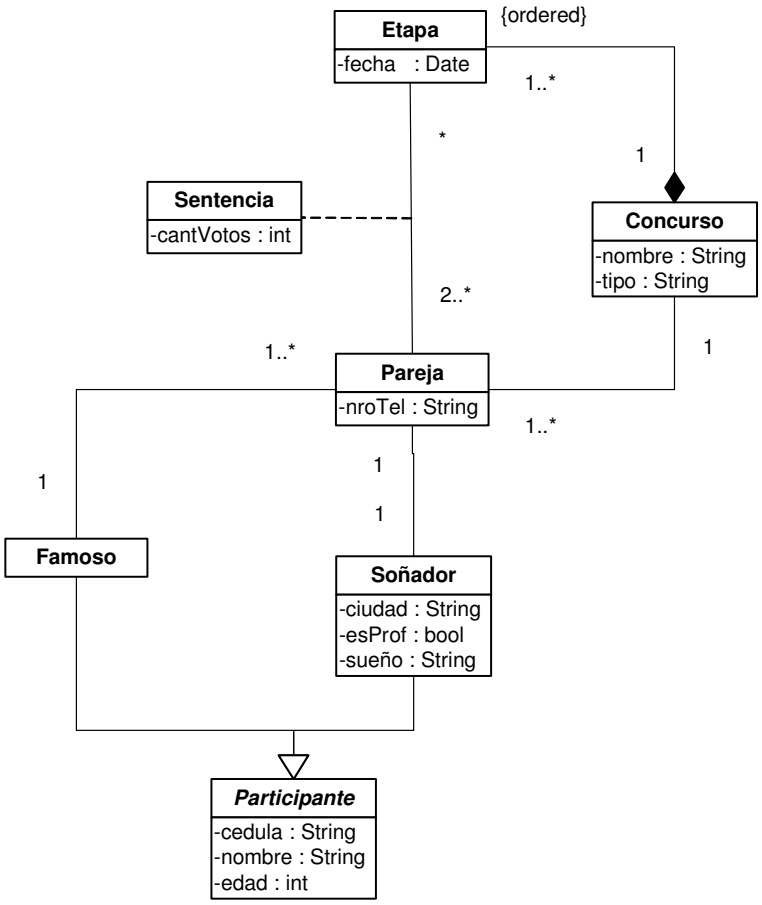
ii. Definir qué es un invariante del Modelo de Dominio e indicar tres tipos de invariantes habituales.

Invariantes habituales (teórico modelado del dominio):

- Unicidad de Atributos (Identificación de Instancias): Un atributo tiene un valor único dentro del universo de instancias de un mismo tipo (una instancia es identificada por ese valor).
- Dominio de Atributos: El valor de un atributo pertenece a cierto dominio.
- Integridad Circular: No puede existir circularidad en la navegación.
- Atributos Calculados: El valor de un atributo es calculado a partir de la información contenida en el dominio.
- Reglas de Negocio: Invariante que restringe el dominio del problema.

b)

Modelo de dominio:



Restricciones:

context Etapa **inv:**
-- En una etapa, las parejas sentenciadas deben competir en el concurso al que la etapa pertenece.

```
self.pareja->forAll(p | p.concurso =  self.concurso)
```

-- La pareja con menor cantidad de votos en una etapa no participa de una sentencia posterior

```
let eliminada : Pareja = self.sentencia->select(s1|self.sentencia->forAll(s2|s2 <> s1 implies s1.cantVotos < s2.cantVotos) )->any().pareja
in
eliminada.etapa->forAll(e|e.fecha <= self.fecha)
```

context Famoso **inv:**
-- Un famoso no puede formar parte de más de una pareja en un mismo concurso.

```
self.pareja->forAll(p1, p2 | p1 <> p2 implies p1.concurso <> p2.concurso)
```

context Concurso **inv**:
-- Un concurso se identifica por su nombre y tipo.

```
Concurso.allInstances()->forAll(c1, c2| c1 <> c2 implies c1.nombre <>
c2.nombre or c1.tipo <> c2.tipo)
```

-- Para un concurso no puede haber dos etapas con una misma fecha.
```
self.etapa->isUnique(fecha)
```

-- Para un concurso no puede haber dos parejas con el mismo número de teléfono.
```
self.pareja->isUnique(nroTel)
```

context Participante **inv**:
-- Un participante se identifica por su cédula.
```
Participante.allInstances()->isUnique(cedula)
```

Problema 2 (25 puntos)

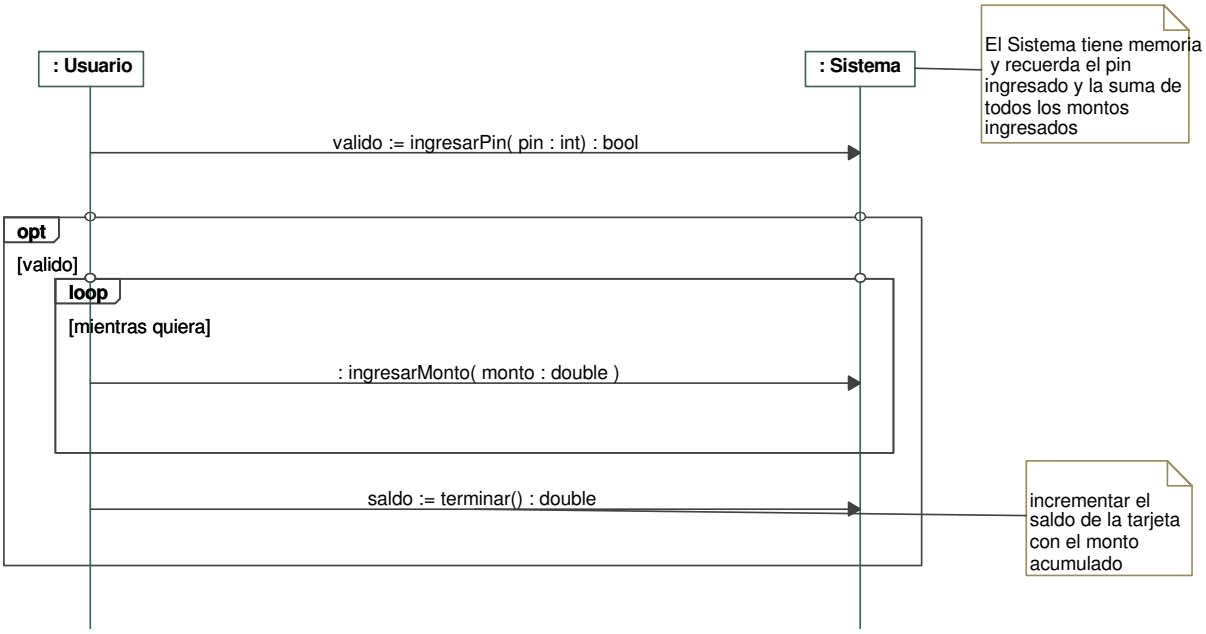
a)

Conteste brevemente que es un contrato de software y como se relaciona con los DSS.

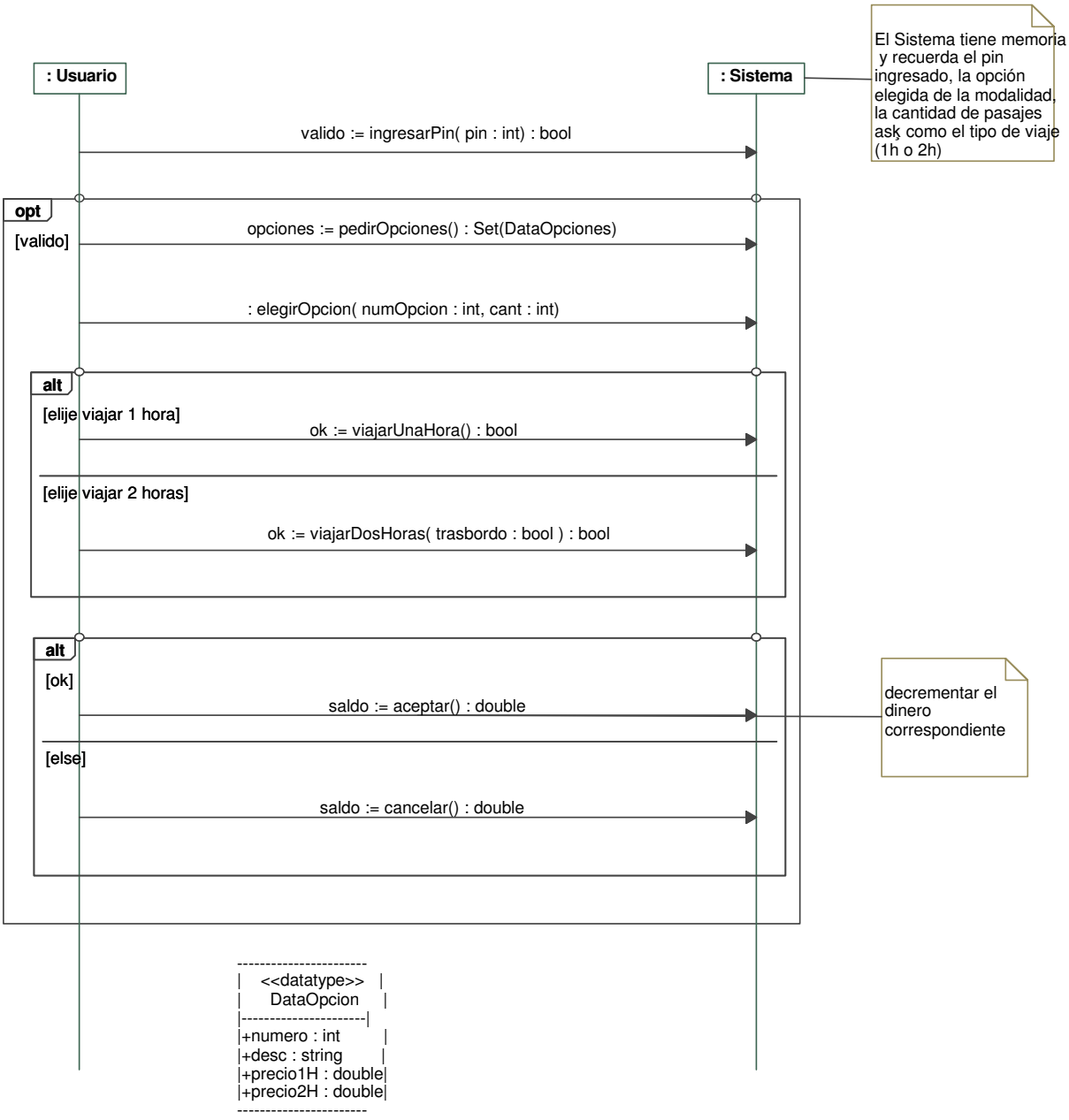
Un contrato de software especifica (declarativamente) el comportamiento o efecto de una operación. Existe un contrato de software por cada operación del DSS.

b)

DSS correspondiente al **CU Recarga de Tarjeta**:

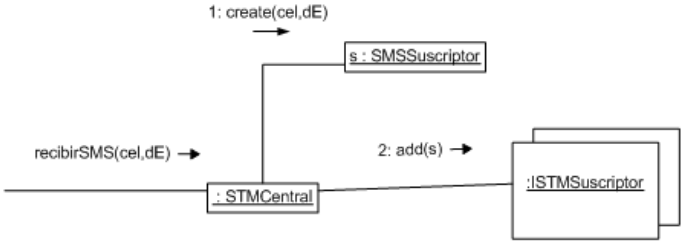


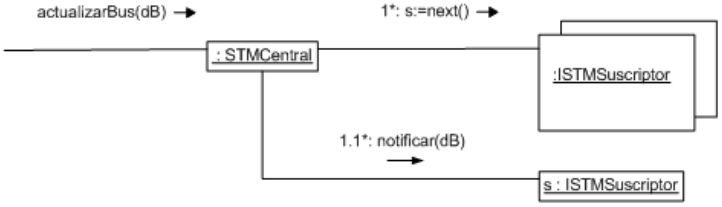
DSS correspondiente al **CU Uso de Tarjeta:**

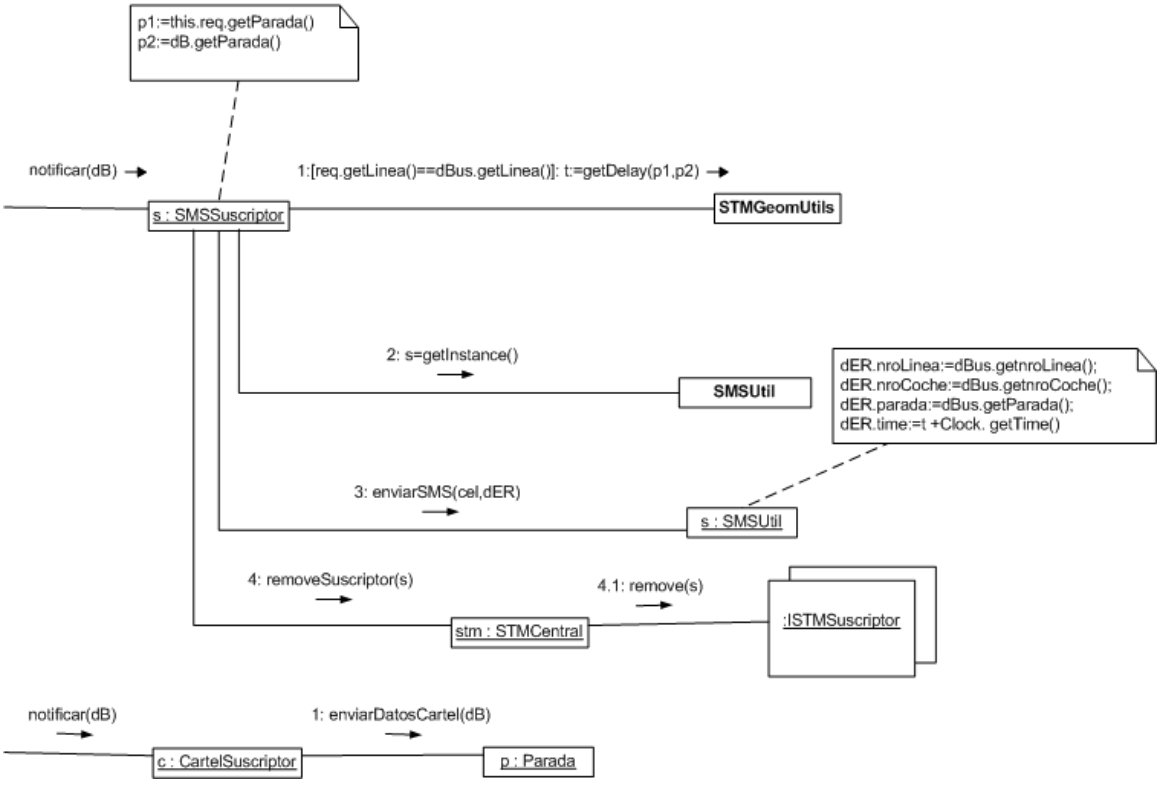


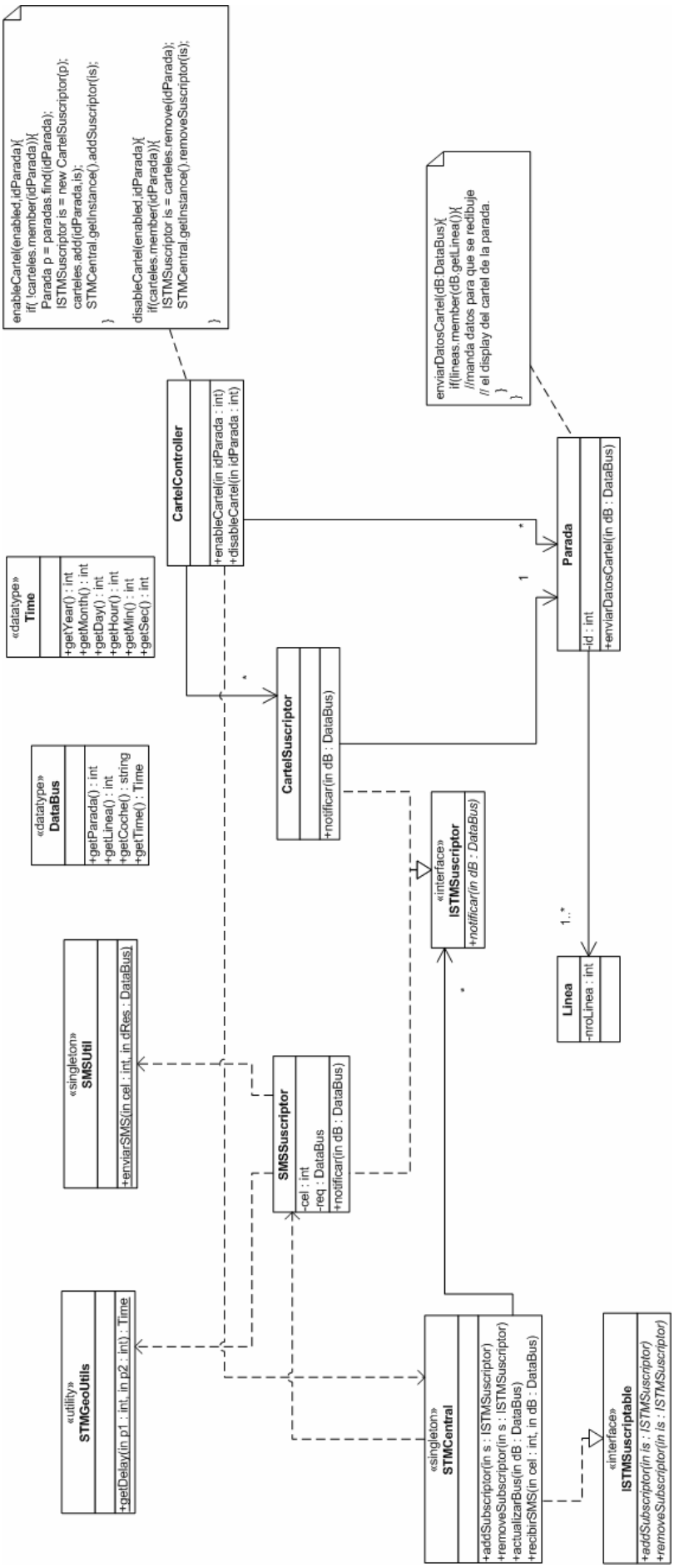
Problema 3 (25 puntos)

a. i ver teórico
b. i





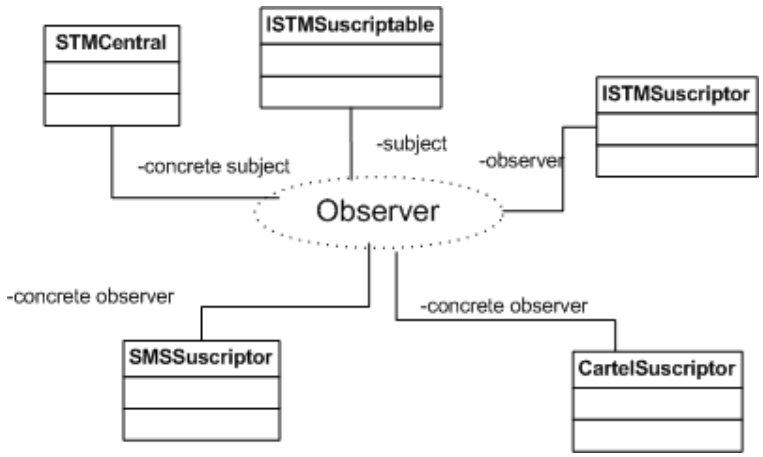



enableCartel(enabled,idParada){
if(!carteles.member(idParada)){
Parada p = paradas.find(idParada);
ISTMSuscriptor is = new CartelSuscriptor(p);
carteles.add(idParada,is);
STMCentral.getInstance().addSuscriptor(is);
}
disableCartel(enabled,idParada){
if(carteles.member(idParada)){
ISTMSuscriptor is = carteles.remove(idParada);
STMCentral.getInstance().removeSuscriptor(is);
}
}
enviarDatosCartel(dB:DataBus){
if(lineas.member(dB.getLinea()){
//manda datos para que se redibuje
// el display del cartel de la parada.
}
}
CartelController
+enableCartel(in idParada : int)
+disableCartel(in idParada : int)
Parada
-id : int
+enviarDatosCartel(in dB : DataBus)
«datatype»
Time
+getYear() : int
+getMonth() : int
+getDay() : int
+getHour() : int
+getMin() : int
+getSec() : int
«datatype»
DataBus
+getParada() : int
+getLinea() : int
+getCoche() : string
+getTime() : Time
CartelSuscriptor
+notificar(in dB : DataBus)
«interface»
ISTMSuscriptor
+notificar(in dB : DataBus)
«singleton»
SMSUtil
+enviarSMS(in cel : int, in dRes : DataBus)
SMSSuscriptor
-cel : int
-req : DataBus
+notificar(in dB : DataBus)
Linea
-nroLinea : int
«utility»
STMGeoUtils
+getDelay(in p1 : int, in p2 : int) : Time
«singleton»
STMCentral
+addSubscriptor(in s : ISTMSuscriptor)
+removeSubscriptor(in s : ISTMSuscriptor)
+actualizarBus(in dB : DataBus)
+recibirSMS(in cel : int, in dB : DataBus)
«interface»
ISTMSuscriptable
+addSubscriptor(in is : ISTMSuscriptor)
+removeSubscriptor(in is : ISTMSuscriptor)
1
*
1..*

Nota:

- Se asume que el sistema se inicializa con la colecciones de paradas que se cargan en el Cartel Controller con sus lineas correspondientes.
- Tambien que existe una clase Clock que devuelve la fecha y hora actual con la operación getTime().

iii) El patrón de diseño utilizado fue el Observer, los roles y participantes se detallan en el siguiente diagrama.



Problema 4 (25 puntos)

a) Ver teórico.

b)

// ManejadorArchivos.hh

```
class ManejadorArchivos {
private:
	static ManejadorArchivos* instance;
	Elemento* raiz;
	ManejadorArchivos();
public:
	static ManejadorArchivos* getInstance();
	bool agregarArchivo(const DataRuta&, const String&);
	bool borrar(const DataRuta&);
};
```

// ManejadorArchivos.cc

```
ManejadorArchivos* ManejadorArchivos::instance = NULL;

ManejadorArchivos::ManejadorArchivos() {
	this->raiz = new Directorio("/");
}

ManejadorArchivos* ManejadorArchivos::getInstance() {
	if (ManejadorArchivos::instance == NULL)
		ManejadorArchivos::instance = new ManejadorArchivos();

	return ManejadorArchivos::instance;
}

bool ManejadorArchivos::agregarArchivo(const DataRuta& ruta, const String& contenido) {
	return this->raiz->agregarArchivo(ruta, contenido, 1);
}

bool ManejadorArchivos::borrar(const DataRuta& ruta) {
	// No se puede borrar la raíz
	if (ruta.getCantidadPartes() <= 0)
		return false;

	return this->raiz->borrar(ruta, 1);
}
```

// Elemento.hh

```
class Elemento : public ICollectible {
private:
	String nombre;
public:
	Elemento(const String&);
	String getNombre();
	virtual bool agregarArchivo(const DataRuta&, const String&, int);
	virtual bool borrar(const DataRuta&, int);
	virtual ~Elemento();
};
```

// Elemento.cc

```
Elemento::~Elemento() {}

String Elemento::getNombre() {
	return this->nombre;
}

bool Elemento::agregarArchivo(const DataRuta& ruta, const String& contenido, int i) {
	// Implementación vacía por defecto para el caso de Archivo
	return false;
}

bool Elemento::borrar(const DataRuta& ruta, int i) {
	// Implementación vacía por defecto para el caso de Archivo
	return false;
```

```
}

// Archivo.hh

class Archivo : public Elemento {
private:
	String contenido;
public:
	Archivo(const String&, const String&);
	~Archivo();
};

// Archivo.cc
Archivo::Archivo(const String& n, const String& c) : Elemento(n), contenido(c) {}

Archivo::~Archivo() {
	UtilidadesIO::borrar(this);
}

// Directorio.hh
class Directorio : public Elemento {
private:
	IDictionary* hijos;
public:
	Directorio(const String&);
	bool agregarArchivo(const DataRuta&, const String&, int);
	bool borrar(const DataRuta&, int);
	~Directorio();
};

// Directorio.cc
Directorio::Directorio(const String& n) : Elemento(n), hijos(new List()) {}

bool Directorio::agregarArchivo(const DataRuta& ruta, const String& contenido, int i) {
	StringKey k(StringKey(ruta.getParte(i)));
	Elemento* e = (Elemento*)this->hijos->find(&k);
	bool hijoDirecto = ruta.getCantidadPartes() == i;

	if (e == NULL) {
		// Se crea el directorio o archivo
		String parte = ruta.getParte(i);

		IKey* key = new StringKey(parte);

		if (hijoDirecto) {
			// Archivo

			Archivo* a = new Archivo(parte, contenido);
			this->hijos->add(key, a);
			UtilidadesIO::crear(a);
		}
		else {
			// Directorio

			Directorio* d = new Directorio(parte);
			this->hijos->add(key, d);
			UtilidadesIO::crear(d);

			// Esta invocación nunca va a fallar dado que el directorio está
vacío
			d->agregarArchivo(ruta, contenido, i + 1);
		}

		return true;
	}
	else {
		if (hijoDirecto) {
			// Ya existía un elemento con ese nombre
			return false;
		}
		else {
			// Invocación recursiva
			return e->agregarArchivo(ruta, contenido, i + 1);
		}
	}
}
```

```
bool Directorio::borrar(const DataRuta& ruta, int i)
{
	StringKey key(ruta.getParte(i));
	Elemento* e = (Elemento*)this->hijos->find(&key);

	// Ruta inválida
	if (e == NULL)
		return false;

	if (ruta.getCantidadPartes() == i) {
		// Se borra un hijo directo
		delete this->hijos->remove(&key);
		return true;
	}
	else {
		// Se borra un hijo indirecto, invocación recursiva
		return e->borrar(ruta, i + 1);
	}
}

Directorio::~Directorio() {
	// Primero se eliminan los hijos para que no queden descolgados
	// al momento de borrarlos de disco, además no se pueden borrar
	// directorio no vacíos
	IIterator* it;

	for (it = this->hijos->getIterator(); it->hasCurrent(); it->next())
		delete it->getCurrent();

	delete it;
	delete this->hijos;

	// Se elmina físicamente este directorio que ahora está vacío
	UtilidadesIO::borrar(this);
}
```