Mestrado Integrado em Engenharia Informática e Computação
EIC0039 - Métodos Formais de Engenharia de Software

Universidade do Porto

Faculdade de Engenharia

FEUP

Modelo de diagramas de sequência UML

Relatório Final

Nome dos elementos do grupo:

Ricardo Pinho - 090509045 - ei09045@fe.up.pt
Vitor Santos - 090509059 - ei09059@fe.up.pt

Data: 6 de Dezembro de 2012

Índice

Requisitos e principais restrições

Para que seja possível conceber um diagrama de sequência em UML, os seguintes elementos devem ser considerados:

- *Diagrams;*
- *Lifelines*, que podem ser derivadas em *Actors* ou em *Objects*;
- *Messages*;
- *Gates*;
- *Fragments*.

Um *Diagram* é a classe que representa um diagrama de sequência, que conterá todos os seus elementos característicos.

Uma *Lifeline* é uma classe abstracta, da qual a classe *Actor* e *Object* serão generalizadas. Estes são as classes que enviam ou recebem mensagens.

Message é uma classe que contêm o texto a ser transmitido de uma *Lifeline* (*Actor* ou *Object*) para outra.

Gate é onde as mensagens são inseridas. Cada *Gate* apenas recebe e envia uma mensagem.

Os *Fragments* são configurações especiais que podem ser aplicadas a uma ou mais *Lifelines* servindo de contentor de Alt (Alternativa), Loop, e outras formas de interacção.

Posto isto, podem-se considerar as seguintes restrições ao modelo, cujo programa utilizá-lo-á como base formal:

- Como pré-condições:
  - R1: Uma *lifeline* deve ser criada em primeiro lugar, antes de qualquer outra característica do *diagram*;
  - R2: Um *gate* deve ser criada após a *lifeline* de origem e de destino;
  - R3: Uma *message* deve estar contida num *gate*;
  - R4: Os parâmetros inseridos em qualquer função não podem ser nulos (nil).
- Como pós-condições:
  - R5: O número de *lifelines* deve ter, pelo menos, o mesmo número de *actors* e *objects* somado, para verificar o princípio da derivação de classes;
  - R6: Uma *lifeline* deve ter uma ligação a outra (ou a ela própria), por via de um *gate*, que irá conter *messages*;
  - R7: Um *gate* tem uma e só uma *message* que entra ou sai de uma *lifeline* para outra (pode apenas ter uma *message* de envio ou (exclusivo) uma de recepção ou as duas);
  - R8: Um *fragment* deve estar contido numa *lifeline*;
- Invariantes:

- Número máximo de *messages* por *diagram*, que serão visíveis ao utilizador;
- Número máximo de *messages* por *diagram*, determinadas por razões de performance;
- Número máximo de *actors*/*objects* por *diagram*, determinadas por razões de performance;
- Número máximo de *fragments* por *diagram*, determinadas por razões de performance;
- Número máximo de *gates* por *diagram*, determinadas por razões de performance;

Especificação das restrições

Classe *diagram:*

```
inv card actors <= 999;
inv card objects <= 999;
inv card gates <= 999;
inv card fragments <= 999;
inv card messages <= 999;


public Diagram : () ==> Diagram
...
post actors ={} and objects ={} and fragments ={} and gates ={} and messages ={};

public addActors : Actor ==> ()
...
post a in set actors;

public addObjects : Object ==> ()
...
post o in set objects;

public addFragments : Fragment ==> ()
...
pre (card actors > 0 or card objects >0)
post f in set fragments;

public addGates : Gate ==> ()
...
pre (card actors > 0 or card objects >0)
post g in set gates;

public addMessages : Message ==> ()
...
pre (card actors > 0 or card objects >0)
post m in set messages;

public remActors : Actor ==> ()
...
pre a in set actors
post a not in set actors;

public remObjects : Object ==> ()
...
pre o in set objects
post o not in set objects;

public remFragments : Fragment ==> ()
...
pre f in set fragments
post f not in set fragments;

public remGates : Gate ==> ()
...
pre g in set gates
post g not in set gates;

public remMessages : Message ==> ()
...
pre m in set messages
post m not in set messages

public addGateToLifeline:Lifeline * Gate ==> ()
```

```
...
pre (lifeline in set actors or lifeline in set objects) and newGate in set gate
post newGate in set gates and newGate in set (elems lifeline.gates);

public addFragmentToLifeline:Lifeline * Fragment ==> ()
...
pre (lifeline in set actors or lifeline in set objects) and newFragment in set fragments
              post newFragment in set fragments and newFragment in set lifeline.fragments;
```

Classe *lifeline:*

```
inv len gates <= maxgates;
public Lifeline : seq of char ==> Lifeline
...
pre len newName > 0;

public addGate:Gate ==> ()
...
post newGate in set (elems gates);

public addFragment: Fragment ==> ()
...
post newFragment in set rng fragments;

public remGate: Gate ==> ()
...
pre oldGate in set (elems gates)
              post oldGate not in set (elems gates);

public remFragment: Fragment ==> ()
...
pre oldFragment in set rng fragments
              post oldFragment not in set rng fragments;
```

Classe *actor*:

Não possui quaisquer condições ou invariantes.

Classe *object*:

```
public Object : seq of char * seq of char ==> Object
...
pre len newDescription > 0;
```

Classe *message*:

```
public Message : nat ==> Message
...
pre newIdentifier > 0;

public Message : nat * seq of char ==> Message
...
pre newIdentifier > 0;
```

Classe *gate*:

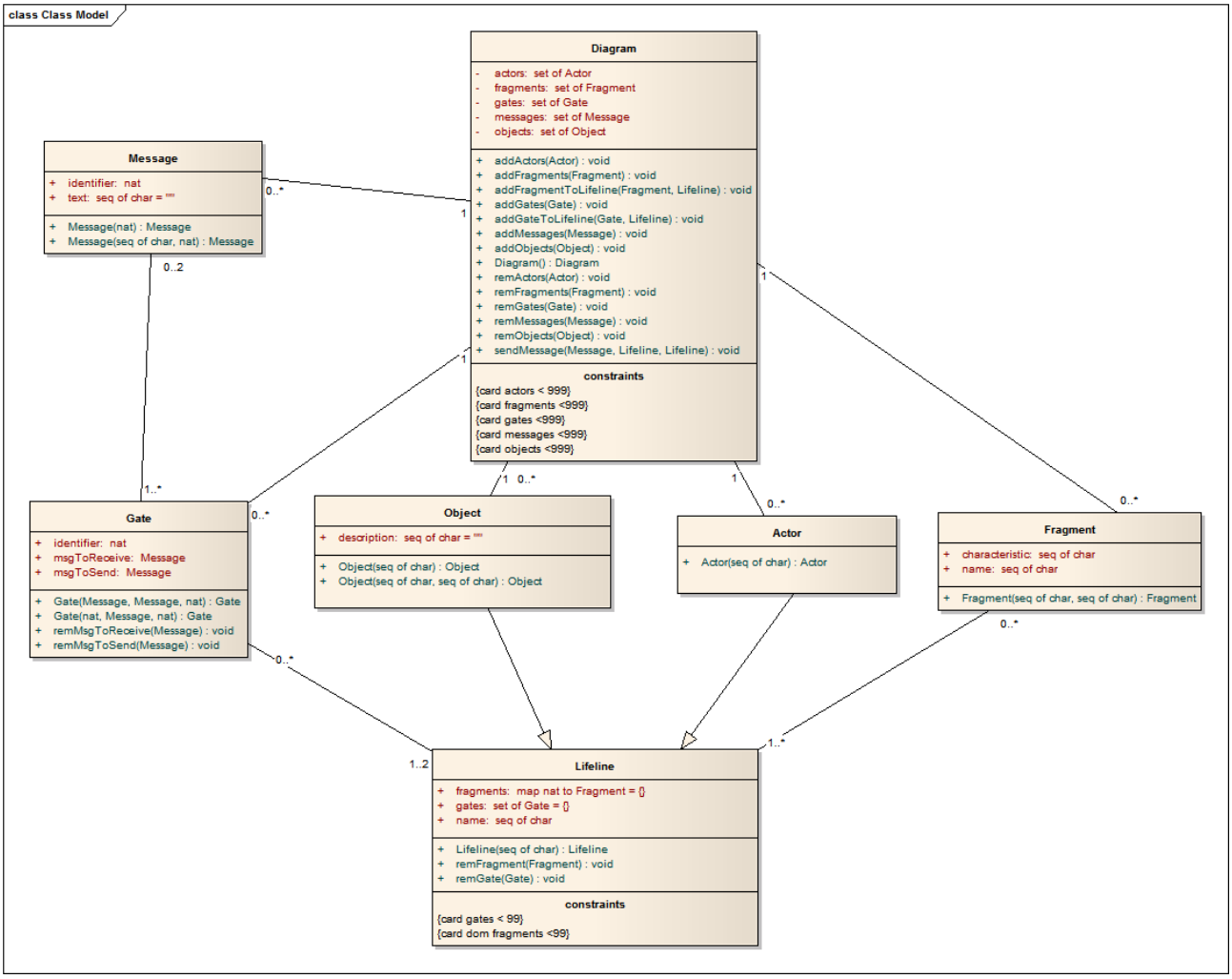
```
public Gate : nat * Message * nat ==> Gate
...
pre newIdentifier > 0 and (dir = 1 or dir = 0)
post (msgToSend = newMsg and dir = 0) or (msgToReceive = newMsg and dir = 1);

public Gate : nat * Message * Message ==> Gate
```

```
...
pre newIdentifier > 0
post (msgToSend = newMsgToSend and msgToReceive = newMsgToReceive);

public remMsgToSend : Message ==> ()
...
pre msgToSend=oldMessage
post len msgToSend.text=0;

public remMsgToReceive : Message ==> ()
...
pre msgToReceive=oldMessage
post len msgToReceive.text=0;
```

Classe *fragment*:

```
public Fragment : seq of char * seq of char ==> Fragment
...
pre len newName > 0 and len newCharacteristic > 0;
```

Diagrama conceptual



Classes e *Scripts* de teste

Para a concepção deste projecto, houve uma necessidade de o dividir nas seguintes classes:

- Classe *Diagram:* a classe que representa *Diagrams*, que são, na prática, os diagramas de sequência que irão conter os diferentes elementos;
- Classe *Lifeline*: a classe que representa as *lifelines*, de uma forma abstracta;
- Classe *Actor*: representação de *actors*, que irá herdar as características descritas pela classe *Lifeline*;
- Classe *Object*: representação de *objects*, que irá herdar as características descritas pela classe *Lifeline*;
- Classe *Message*: a classe que representará as *messages*, que simbolarizarão as interacções entre *lifelines*;
- Classe *Gate*: classe representativa de *gates*, contentores de envio e recepção de *messages;*
- Classe *Fragment*: classe que cria *fragments*; que cria opções ou restrições para as *lifelines*, em várias tarefas.

Para comprovar a exactidão desta implementação foram criadas várias classes de teste, que permitem verificar a funcionalidade de todas as operações:

- Classe *Test*: Classe que vai fornecer uma base para todos os testes efectuados posteriormente, nomeadamente na questão dos *asserts* (funções que verificam se uma determinada expressão booleana está correcta);
- Classe *LifelineTest*: Teste que verifica se uma *lifeline* foi apropriadamente construída;
- Classe *ActorObjectTest*: Conjunto de testes que verificarão a criação de *lifelines* específicas, i.e. *actors* e *objects*; que verificarão se a comunicação entre eles, por via de uma *message*, é feita de acordo com a teoria; e a situação em que uma *lifeline* envia uma *message* para si próprio;
- Classe *FragmentsTest*: Teste que verifica a inserção correcta de um *fragment* numa *lifeline*;

Matriz de rastreabilidade

- Classe *Test*:
  - Esta classe de teste não testa nenhuma condição em especial, servindo de base para outros testes.
- Classe *LifelineTest*:
  - Testa a criação de uma lifeline e verifica a pré-condição da criação inicial de uma lifeline.
- Classe *ActorObjectTest*:
  - Testa a construção de um *actor* e de um *object*, validando a pré-condição da não inserção de parâmetros nulos;
  - Testa a comunicação entre um *actor* e um *object*, através do envio de uma *message* a partir do primeiro sujeito: durante este procedimento, a criação de um *gate* e a adição de uma *message* também é testada pelas pré-condições associadas; por fim, o gate é associado aos intervenientes, verificando se o *gate* existe no conjunto de *gates* que cada um dos sujeitos possui. O procedimento anterior é também efectuado, quando um *actor* ou *object* envia uma *message* para si próprio.
- Classe *FragmentsTest*:
  - Testa a inclusão de um *fragment* numa *lifeline*. Garante as condições da nulidade dos parâmetros e se o fragment a incluir ficou incluído na lista de *fragments*.

	R1	R2	R3	R4	R5	R6	R7	R8
Test								
LifelineTest	X			X				
ActorObjectTest	X	X	X	X	X	X	X	
FragmentsTest				X				X

Definição das classes

- *Diagram:*

```
class Diagram
        instance variables
        public actors : set of Actor;
        public objects :  set of Object;
        public fragments : set of Fragment;
        public gates : set of Gate;
        public messages : set of Message;
        inv card actors <= 999;
        inv card objects <= 999;
        inv card gates <= 999;
        inv card fragments <= 999;
        inv card messages <= 999;

        operations
        public Diagram : () ==> Diagram
                Diagram() ==
                (
                        actors :={};
                        objects :={};
                        fragments :={};
                        gates :={};
                        messages :={};
                )
                post actors ={} and objects ={} and fragments ={} and gates ={} and messages ={};

        public addActors : Actor ==> ()
                addActors(a) ==
                (
                        actors:=actors union {a}
                )
                post a in set actors;

        public addObjects : Object ==> ()
                addObjects(o) ==
                (
                        objects:=objects union {o}

                )
                post o in set objects;

        public addFragments : Fragment ==> ()
                addFragments(f) ==
                (
                        fragments:=fragments union {f}
                )

                pre (card actors > 0 or card objects >0)
                post f in set fragments;


        public addGates : Gate ==> ()
                addGates(g) ==
                (
                        gates:=gates union {g}

                )
                pre (card actors > 0 or card objects >0)
                post g in set gates;

        public addMessages : Message ==> ()
```

```
        addMessages(m) ==
        (
                messages:=messages union {m}
        )
        pre (card actors > 0 or card objects >0)
        post m in set messages;

public remActors : Actor ==> ()
        remActors(a) ==
        (
                actors:=actors \ {a}
        )
        pre a in set actors
        post a not in set actors;

public remObjects : Object ==> ()
        remObjects(o) ==
        (
                objects:=objects \ {o}
        )
        pre o in set objects
        post o not in set objects;

public remFragments : Fragment ==> ()
        remFragments(f) ==
        (
                fragments:=fragments \ {f};
                for all x in set actors do(
                if f in set rng x.fragments
                 then x.remFragment(f);
                );
                for all x in set objects do(
                if f in set rng x.fragments
                 then x.remFragment(f);
                );
        )
        pre f in set fragments
        post f not in set fragments;

public remGates : Gate ==> ()
        remGates(g) ==
        (
                gates:=gates \ {g};
                for all x in set actors do(
                if g in set (elems x.gates)
                 then x.remGate(g);
                );
                for all x in set objects do(
                if g in set (elems x.gates)
                 then x.remGate(g);
                );
        )
        pre g in set gates
        post g not in set gates;


public remMessages : Message ==> ()
        remMessages(m) ==
        (
                messages:=messages \ {m};
                for all x in set gates do(
                if m = x.msgToSend
                 then x.remMsgToSend(m);
                if m = x.msgToReceive
                 then x.remMsgToReceive(m);
                );
        )
        pre m in set messages
        post m not in set messages;
```

```
		public addGateToLifeline:Lifeline * Gate ==> ()
			addGateToLifeline(lifeline,newGate) ==
			(
			self.addGates(newGate);
			lifeline.gates:= lifeline.gates ^ [newGate]
			)
			pre (lifeline in set actors or lifeline in set objects) and newGate in set gate
			post newGate in set gates and newGate in set (elems lifeline.gates);

		public addFragmentToLifeline:Lifeline * Fragment ==> ()
			addFragmentToLifeline(lifeline,newFragment) ==
			(
			self.addFragments(newFragment);
			lifeline.fragments := lifeline.fragments union {newFragment};
			)
			pre (lifeline in set actors or lifeline in set objects) and newFragment in set
fragments
				post newFragment in set fragments and newFragment in set
	lifeline.fragments;


		public sendMessage:Lifeline * Lifeline * Message ==> ()
			sendMessage(sourceLifeline,destinationLifeline,message) ==
			(
			dcl newGate : Gate := new Gate(card gates, message,0);
			dcl newGate2 : Gate := new Gate(card gates, message,1);
			self.addGateToLifeline(sourceLifeline , newGate);
			self.addGateToLifeline(destinationLifeline ,newGate2);
			);

	end Diagram
```

- *Lifeline*:

```
	class Lifeline
		instance variables
		public name : seq of char;
		public gates :  seq of Gate :=[];
		public fragments : set of Fragment:={};
		private maxgates : nat :=99;
		inv len gates <= maxgates;

		operations
		public Lifeline : seq of char ==> Lifeline
			Lifeline(newName) ==
			(name := newName)
			pre len newName > 0;

		public addGate:Gate ==> ()
			addGate(newGate) ==
			(
			gates:= gates ^ [newGate];
			)
			post newGate in set (elems gates);

		public addFragment: Fragment ==> ()
			addFragment (newFragment) ==
			(
			fragments := fragments munion {(card dom fragments)|-> newFragment};
			)
				post newFragment in set rng fragments;

		public remGate: Gate ==> ()
			remGate(oldGate) ==
			(
				dcl momgates : seq of Gate := [];
				for all x in set elems gates do
				(
					if x <> oldGate
```

```
                                then momgates := momgates ^ [x];
                        );

                        gates := momgates
                )
                                pre oldGate in set (elems gates)
                                post oldGate not in set (elems gates);

        public remFragment: Fragment ==> ()
                remFragment (oldFragment) ==
                (
                fragments := fragments:->{oldFragment};
                )
                                pre oldFragment in set rng fragments
                                post oldFragment not in set rng fragments;

end Lifeline
```

- *Actor*:

```
class Actor is subclass of Lifeline
        operations
        public Actor : seq of char ==> Actor
                Actor(newName) == (Lifeline(newName));

end Actor
```

- *Object*:

```
class Object is sublcass of Lifeline
        instance variables
        public description : seq of char

        operations
        public Object : seq of char ==> Object
                Object(newName) == (Lifeline (newName));

        public Object : seq of char * seq of char ==> Object
                Object(newName, newDescription) ==
                (
                description := newDescription;
                Lifeline (newName)
                )
                pre len newDescription > 0;

end Object
```

- *Message*:

```
class Message
        instance variables
        public identifier : nat;
        public text : seq of char;

        operations
        public Message : nat ==> Message
                Message(newIdentifier) ==
                (identifier := newIdentifier)
                pre newIdentifier > 0;

        public Message : nat * seq of char ==> Message
                Message(newIdentifier, newText) ==
                (
                text := newText;
                identifier := newIdentifier
                )
                pre newIdentifier > 0;

end Message
```

- *Gate*:

```
class Gate
        instance variables
        public identifier : nat;
        public msgToSend : Message;
        public msgToReceive: Message;

        operations
        public Gate : nat * Message * nat ==> Gate
                Gate(newIdentifier, newMsg, dir) ==
                (
                identifier := newIdentifier;
                if dir=0
                then msgToSend := newMsg
                else msgToReceive := newMsg
                )
                pre newIdentifier > 0 and (dir = 1 or dir = 0)
                post (msgToSend = newMsg and dir = 0) or (msgToReceive = newMsg and dir = 1);

        public Gate : nat * Message * Message ==> Gate
                Gate(newIdentifier, newMsgToSend, newMsgToReceive) ==
                (
                identifier := newIdentifier;
                msgToSend := newMsgToSend;
                msgToReceive := newMsgToReceive
                )
                pre newIdentifier > 0
                post (msgToSend = newMsgToSend and msgToReceive = newMsgToReceive);

        public remMsgToSend : Message ==> ()
                remMsgToSend(oldMessage) ==
                (
                msgToSend:=new Message(oldMessage.identifier);
                )
                pre msgToSend=oldMessage
                post len msgToSend.text=0;

        public remMsgToReceive : Message ==> ()
                remMsgToReceive(oldMessage) ==
                (
                msgToReceive:= new Message(oldMessage.identifier);
                )
                pre msgToReceive=oldMessage
                post len msgToReceive.text=0;

end Gate
```

- *Fragment*:

```
class Fragment
        instance variables
        public name : seq of char;
        public characteristic : seq of char;

        operations
        public Fragment : seq of char * seq of char ==> Fragment
                Fragment(newName, newCharacteristic) ==
                (name := newName;
                characteristic := newCharacteristic)
                pre len newName > 0 and len newCharacteristic > 0;

end Fragment
```

- *Test*

```
class Test
        operations
        public Assert : bool ==> ()
                Assert(expression) == return
                pre expression;

end Test
```

- *ActorObjectTest*

```
class ActorObjectTest is subclass of Test
        operations
        public objectTest : () ==> ()
                objectTest() == (
                dcl d : Diagram := new Diagram();
                dcl o : Object := new Object ("anObject");
                dcl o2 : Object := new Object ("anObject2", "this is an object");
                dcl m : Message := new Message (1);
                d.addObjects(o);
                d.addObjects(o2);
                d.addMessages(m);
                d.sendMessage(o,o2,m);
                Assert(o.name = "anObject" and o2.name = "anObject2" and m.identifier = 1 and o
in set d.objects and o2 in set d.objects and m in set d.messages);
                d.remObjects(o);
                d.remObjects(o2);
                Assert(o not in set d.objects and o2 not in set d.objects)
                );

        public actorTest : () ==> ()
                actorTest() == (
                dcl d : Diagram := new Diagram();
                dcl a : Actor := new Actor ("anActor");
                dcl a2 : Actor := new Actor ("anActor2");
                dcl m : Message := new Message (1);
                d.addActors(a);
                d.addActors(a2);
                d.addMessages(m);
                d.sendMessage(a,a2,m);
                Assert(a.name = "anActor" and a2.name = "anActor2" and m.identifier = 1 and a in
set d.actors and a2 in set d.actors and m in set d.messages);
                d.remActors(a);
                d.remActors(a2);
                Assert(a not in set d.actors and a2 not in set d.actors)
                );

        public actorObjectCommunicating : () ==> ()
                actorObjectCommunicating() == (
                dcl d : Diagram := new Diagram();
                dcl a : Actor := new Actor ("anActor");
                dcl o : Object := new Object ("anObject", "this is an object");
                dcl m : Message := new Message (1);
                d.addActors(a);
                d.addObjects(o);
                d.addMessages(m);
                d.sendMessage(o,a,m);
                Assert(a.name = "anActor" and o.name = "anObject" and m.identifier = 1 and a in
set d.actors and o in set d.objects and m in set d.messages);
                d.remMessages(m);
                Assert(a in set d.actors and o in set d.objects and  m not in set d.messages)
                );

        public actorObjectCommunicatingAndSomeoneAlone : () ==> ()
                actorObjectCommunicatingAndSomeoneAlone() == (
                dcl d : Diagram := new Diagram();
                dcl a : Actor := new Actor ("anActor");
                dcl o : Object := new Object ("anObject");
                dcl someoneAlone : Actor := new Actor ("someoneAlone");
```

```
                        dcl m : Message := new Message (1,"this is a message and I hope you get it");
                        dcl g : Gate := new Gate(3, m,1);
                        d.addActors(a);
                        d.addObjects(o);
                        d.addMessages(m);
                        d.addGates(g);
                        d.addGateToLifeline(o, g);
                        d.sendMessage(a,o,m);
                        Assert(a.name = "anActor" and o.name = "anObject" and m.identifier = 1 and
someoneAlone.name="someoneAlone");
                        d.remMessages(m);
                        Assert(a in set d.actors and o in set d.objects and  m not in set d.messages and
g in set d.gates and g in set (elems o.gates));
                        d.remGates(g);
                        Assert(o in set d.objects and  g not in set d.gates and g not in set (elems
o.gates))
                        );

                public crazyActorThatspeaksToHimself : () ==> ()
                        crazyActorThatspeaksToHimself() == (
                        dcl d : Diagram := new Diagram();
                        dcl a : Actor := new Actor ("anActor");
                        dcl m : Message := new Message (1);
                        dcl g : Gate := new Gate(3, m,1);
                        d.addActors(a);
                        d.addMessages(m);
                        d.addGates(g);
                        d.addGateToLifeline(a, g);
                        d.sendMessageToSelf(a,m);
                        Assert(a.name = "anActor" and m.identifier = 1);
                        d.remMessages(m);
                        Assert(a in set d.actors and  m not in set d.messages and g.msgToSend<>m and
g.msgToReceive<>m);
                        Assert(a in set d.actors and  g in set d.gates and g in set (elems a.gates));
                        d.remGates(g);
                        Assert(a in set d.actors and  g not in set d.gates and g not in set (elems
a.gates))
                        );


                public actorSpeakingToNoone : () ==> ()
                        actorSpeakingToNoone() == (
                        dcl d : Diagram := new Diagram();
                        dcl a : Actor := new Actor ("anActor");
                        dcl m : Message := new Message (1,"this is a message and I hope you get it");
                        dcl g : Gate := new Gate(3, m,1);
                        d.addActors(a);
                        d.addObjects(o);
                        d.addMessages(m);
                        d.addGates(g);
                        d.addGateToLifeline(o, g);
                        d.sendMessage(a, nil,m);
                        );

                public lostInCommunication : () ==> ()
                        lostInCommunication() == (
                        dcl d : Diagram := new Diagram();
                        dcl a : Actor := new Actor ("anActor");
                        dcl g : Gate := new Gate(3, nil, nil);
                        );

                public corruptedMessage : () ==> ()
                        corruptedMessage() == (
                        dcl d : Diagram := new Diagram();
                        dcl a : Actor := new Actor ("anActor");
                        dcl m : Message := new Message (1,nil);
                        );

        end ActorObjectTest
```

- *LifelineTest*

```
class LifelineTest is subclass of Test
        operations
        public oneLifeline : () ==> ()
                oneLifeline() == (
                dcl l : Lifeline := new Lifeline ("aLifeline");
                Assert( l.name = "aLifeline")
                );

end LifelineTest
```

- *FragmentsTest*

```
class FragmentsTest is subclass of Test
        operations
        public actorWithFragments : () ==> ()
                actorWithFragments() == (
                dcl d : Diagram := new Diagram();
                dcl a : Actor := new Actor ("anActor");
                dcl f : Fragment := new Fragment ("aName","aCharacteristic");
                dcl f2 : Fragment := new Fragment ("aName2","aCharacteristic2");
                d.addActors(a);
                d.addFragments(f);
                d.addFragmentToLifeline(a,f);
                d.addFragments(f2);
                d.addFragmentToLifeline(a,f2);
                Assert(a.name = "anActor" and  f.name = "aName" and
f.characteristic="aCharacteristic" and f in set rng a.fragments and a in set d.actors and f in set
d.fragments);
                d.remFragments(f);
                Assert(f not in set rng a.fragments and f not in set d.fragments and f2 in set
rng a.fragments and f2 in set d.fragments)
                );

        public objectWithFragments : () ==> ()
                objectWithFragments() == (
                dcl d : Diagram := new Diagram();
                dcl o : Object := new Object ("anObject");
                dcl f : Fragment := new Fragment ("aName","aCharacteristic");
                d.addObjects(o);
                d.addFragments(f);
                d.addFragmentToLifeline(o,f);
                Assert(o.name = "anObject" and  f.name = "aName" and
f.characteristic="aCharacteristic" and f in set rng o.fragments and o in set d.objects and f in set
d.fragments);
                d.remFragments(f);
                Assert(f not in set rng o.fragments and f not in set d.fragments)
                );

end FragmentsTest
```

Cobertura dos testes

Em primeiro lugar, será adequado mostrar que todos os testes que foram elaborados foram corridos com sucesso. Assim:

- Classe *ActorObjectTest*

name	***#calls***	***coverage***
ActorObjectTest`actorTest	1	100%
ActorObjectTest`objectTest	1	100%
ActorObjectTest`corruptedMessage	1	100%
ActorObjectTest`lostInCommunication	1	100%
ActorObjectTest`actorObjectCommunicating	1	100%
ActorObjectTest`actorSpeakingWithoutMessage	1	100%
ActorObjectTest`crazyActorThatspeaksToHimself	1	100%
ActorObjectTest`actorObjectCommunicatingAndSomeoneAlone	1	100%
total		***100%***

- Classe *FragmentsTest*

name	***#calls***	***coverage***
FragmentsTest`actorWithFragments	1	100%
FragmentsTest`objectWithFragments	1	100%
total		***100%***

- Classe *LifelineTest*

name	***#calls***	***coverage***
LifelineTest`oneLifeline	1	100%
total		***100%***

- Classe *Test*

name	***#calls***	***coverage***
Test`Assert	18	100%
total		***100%***

Após mostrar que todos os testes correram de forma apropriada, é necessário mostrar se todas as funções que compõem este programa foram executadas. Assim sendo:

- Classe *Diagram*

name	***#calls***	***coverage***
Diagram`Diagram	10	100%
Diagram`addGates	12	100%
Diagram`remGates	3	100%
Diagram`addActors	8	100%
Diagram`remActors	2	100%
Diagram`addObjects	5	100%
Diagram`remObjects	2	100%
Diagram`addMessages	5	100%
Diagram`remMessages	4	100%
Diagram`sendMessage	4	100%
Diagram`addFragments	3	100%
Diagram`remFragments	2	100%
Diagram`addGateToLifeline	11	100%
Diagram`sendMessageToSelf	2	100%
Diagram`addFragmentToLifeline	3	100%
total		***100%***

- Classe *Lifeline*

name	***#calls***	***coverage***
Lifeline`addGate	11	100%
Lifeline`remGate	2	100%
Lifeline`Lifeline	16	100%
Lifeline`addFragment	3	100%
Lifeline`remFragment	2	100%
total		***100%***

- Classe *Actor*

name	***#calls***	***coverage***
Actor`Actor	10	100%
total		***100%***

- Classe *Object*

name	***#calls***	***coverage***
Object`Object	3	100%
Object`Object	2	100%
total		***100%***

- Classe *Message*

name	***#calls***	***coverage***
Message`Message	41	100%
Message`Message	1	100%
total		***100%***

- Classe *Gate*

name	***#calls***	***coverage***
Gate`remMsgToSend	3	100%
Gate`remMsgToReceive	5	100%
Gate`Gate	11	100%
Gate`Gate	1	100%
total		***100%***

- Classe *Fragment*

name	***#calls***	***coverage***
Fragment`Fragment	3	100%
total		***100%***

Análise da consistência

A consistência da criação de todos estes elementos já vem sintetizada pelas condições impostas nos contractos. Sendo assim, pode-se afirmar que:

- Antes de uma qualquer execução, as funções construtoras dos elementos do diagrama verificam sempre se os parâmetros nunca são nulos;
- Um *diagram* terá sempre um conjunto de *lifelines* concretos: *actors* e *objects*. Assim, poder-se-á verificar que o número de *lifelines* criados durante a execução, corresponde exactamente ao número somado do número de instâncias das classes derivadas;
- Quando existe a adição de um *gate* ao conjunto do género pertencente a um *diagram*, a função correspondente necessita de um *gate* como parâmetro: nesse caso, o elemento parametrizado deve passar nas condições que estão impostas no seu construtor. Por exemplo, o *gate* continua a ter a obrigação de ter pelo menos uma instância das classes derivadas de *lifeline*, para que possa ser criado;
- Uma das consequências de criar um *gate* é a da criação de *messages*: essas *messages* também terão condições que impedirão a sua criação, mal sejam violadas.

Para além desta análise das condições, seria apropriado observar a ferramenta de "integrity checker", fornecida pelo VDMTools (o programa usado para validação de código e geração de código Java) que fornece alguns exemplos do género, tais como:

- `ActorObjectTest`pre_Assert(a.name = "anActor" and o.name = "anObject" and m.identifier = 1 and a in set d.actors and o in set d.objects and m in set d.messages)` na classe *ActorObjectTest* verifica se os *actors* e os *objects* estão contidos no *diagram* e se possuem os nomes especificados em parâmetros;
- `isofclass(Actor,lifeline)` é produto da classe *diagram*, que deduz se um *actor* pertence ao conjunto respectivo do diagrama, quer dizer que é uma instância da classe *Actor*;
- Na função `sendMessageToSelf` da classe *Diagram*, na criação de um *gate* verifica-se uma propriedade que deriva da pré-condição da criação de *gates*: `self.Diagram`pre_addGates(newGate)`;
- Na remoção de um *gate* a um determinado *actor* pertencente a um *diagram*, terá também a verificação da pré-condição inerente à remoção de um *gate*: saber se ele, de facto, existe no conjunto.

Distribuição do tempo de trabalho

1ª reunião: Elaboração das pré-condições, pós-condições, invariantes, situações de teste, classes e início da elaboração das classes: 1 hora e 30 minutos.

2ª reunião: Elaboração das classes: 1 hora.

Elaboração das classes: 2 horas.

Verificação dos contractos: 1 hora.

Criação do diagrama de uml: 15 minutos.

Criação dos testes iniciais: 30 minutos.

Criação dos rtfs : 30 minutos.

Início da elaboração do relatório: 1 hora.

Continuação da elaboração do relatório: 30 minutos.

Concepção de uma classe adicional, correcção de pequenos problemas e continuação da elaboração do relatório: 1 hora e 30 minutos.

Actualização do relatório e inserção do diagrama conceptual: 1 hora.

Últimos testes: 2 horas.

Últimos pormenores do relatório e do trabalho em geral: 30 minutos.