7- Indoor Finder

Relatório Final

4º Ano do Mestrado Integrado em Engenharia Informática e Computação

Métodos Formais em Engenharia de Software

Elementos do grupo:

António Jorge Aguiar do Vale | 201404572 | up201404572@fe.up.pt

Diogo Filipe Costa | 201101988 | ei11014@fe.up.pt

04 de Janeiro de 2018

1. Descrição informal do sistema e lista de requisitos

b. Descrição informal do sistema

O modelo analisado neste relatório corresponde a uma aplicação capaz de fornecer informações de navegação dentro de edifícios. O objetivo da aplicação é ser capaz de indicar qual o caminho mais curto entre dois pontos interiores do edifício, qual o caminho com maior grau de acessibilidade entre dois pontos interiores (pensando no caso de caminhos que dêem preferência a elevadores em vez de escadas) para utilizadores com acessibilidade reduzida, bem como qual o caminho para um tipo de ponto de interesse mais próximo (por exemplo qual a máquina de café mais próxima e qual o caminho até à mesma).

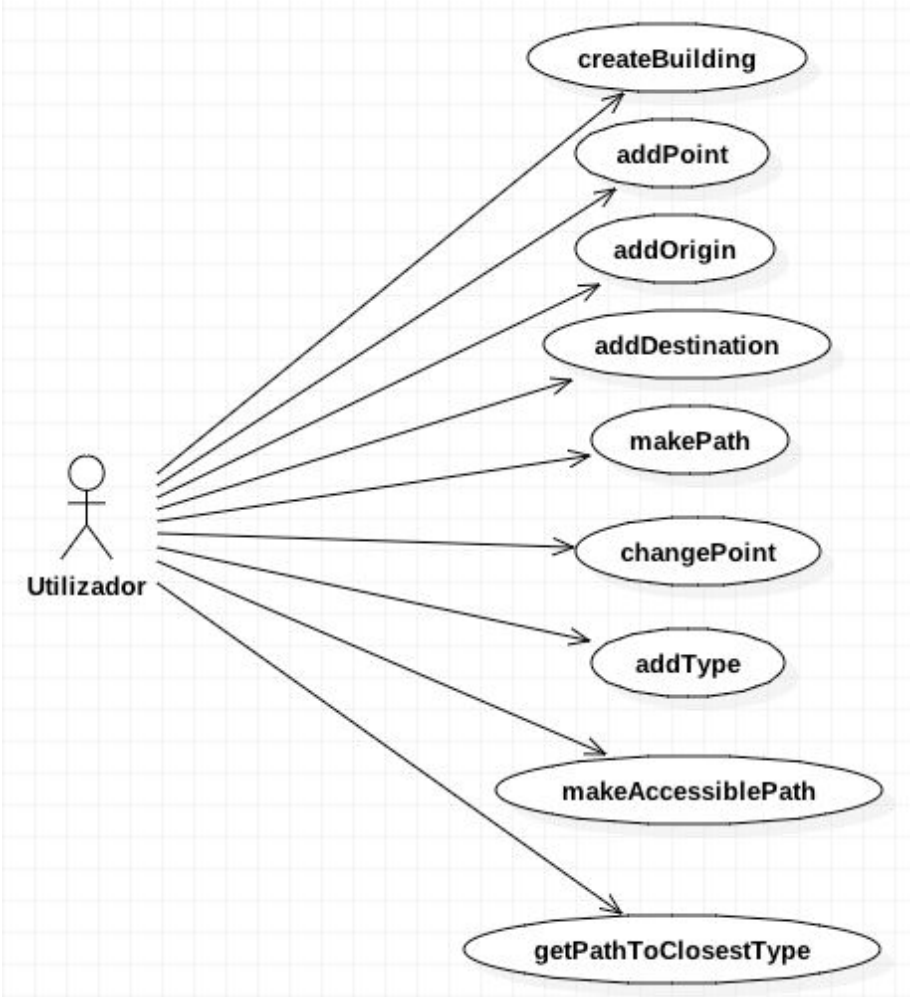
c. Lista de requisitos

Identificador	Classificação	Descrição
R1	Obrigatório	O utilizador deve poder criar o mapa com medidas,x,y e z, personalizáveis.
R2	Obrigatório	O utilizador deve poder mudar pontos existentes de um mapa já definido.
R3	Obrigatório	O utilizador deve poder marcar um ponto de um mapa como origem
R4	Obrigatório	O utilizador deve poder marcar um ponto de um mapa como destino
R5	Obrigatório	O utilizador deve poder obter o caminho mais rápido de um ponto marcado como origem até um ponto marcado como destino
R6	Obrigatório	O utilizador deve poder marcar um ponto como impassável (parede)
R7	Obrigatório	O utilizador deve poder marcar um ponto como ponto especial de interesse (casa de banho, máquina de café, escada, elevador)
R8	Obrigatório	O utilizador deve poder obter o caminho mais curto entre dois pontos do mapa
R9	Obrigatório	O utilizador deve poder obter o caminho

		mais acessível (para utilizadores com grau de incapacidade móvel) entre dois pontos do mapa
R10	Obrigatório	O utilizador deve poder saber a localização e o caminho mais curto desde um ponto até ao ponto de interesse mais próximo (casa de banho, máquina de café)

2. Modelo Visual UML

b. Casos de utilização



Os casos de uso principais são os seguintes:

Cenário	**Criar Edifício**
Descrição	Criação de um edifício de maneira a poder encontrar caminhos no mesmo
Pré-condições	Nenhumas
Pós-condições	O edifício está criado com as dimensões que o utilizador inseriu
Passos	(Não especificados)
Excepções	(Não especificados)

Cenário	**Adicionar ponto no edificio**
Descrição	Adicionar ponto no mapa de pontos do edifício para poder navegar até lá
Pré-condições	1. Não existe o ponto que se quer criar no mapa de pontos do edificio
Pós-condições	1. O ponto inserido está no mapa de pontos do edificio
Passos	1. Utilizador cria o edificio 2. Utilizador insere o ponto no mapa de pontos do edificio
Excepções	(Não especificados)

Cenário	**Marcar ponto como origem**
Descrição	Marcar um ponto do mapa de pontos como origem para cálculo dos caminhos
Pré-condições	1. Mapa de pontos ainda não tem origem 2. Origem a ser marcada é um ponto válido
Pós-condições	1. Mapa de pontos tem origem
Passos	1. Utilizador cria o edificio

	2. Utilizador insere diversos pontos no mapa de pontos do edificio 3. Utilizador marca um dos pontos como origem
Excepções	(Não especificados)

Cenário	**Marcar ponto como destino**
Descrição	Marcar um ponto do mapa de pontos como destino para cálculo dos caminhos
Pré-condições	3. Mapa de pontos ainda não tem destino 4. Destino a ser marcado é um ponto válido
Pós-condições	2. Mapa de pontos tem destino
Passos	1. Utilizador cria o edificio 2. Utilizador insere diversos pontos no mapa de pontos do edificio 3. Utilizador marca um dos pontos como destino
Excepções	(Não especificados)

Cenário	**Alterar ponto**
Descrição	Alterar um ponto do mapa mudando as suas coordenadas ou o seu tipo
Pré-condições	1. Ponto a ser alterado está no mapa de pontos do edifício
Pós-condições	3. Ponto foi alterado com as novas coordenadas e/ou tipo
Passos	1. Utilizador cria o edificio 2. Utilizador insere diversos pontos no mapa de pontos do edificio 3. Utilizador altera um dos pontos do mapa

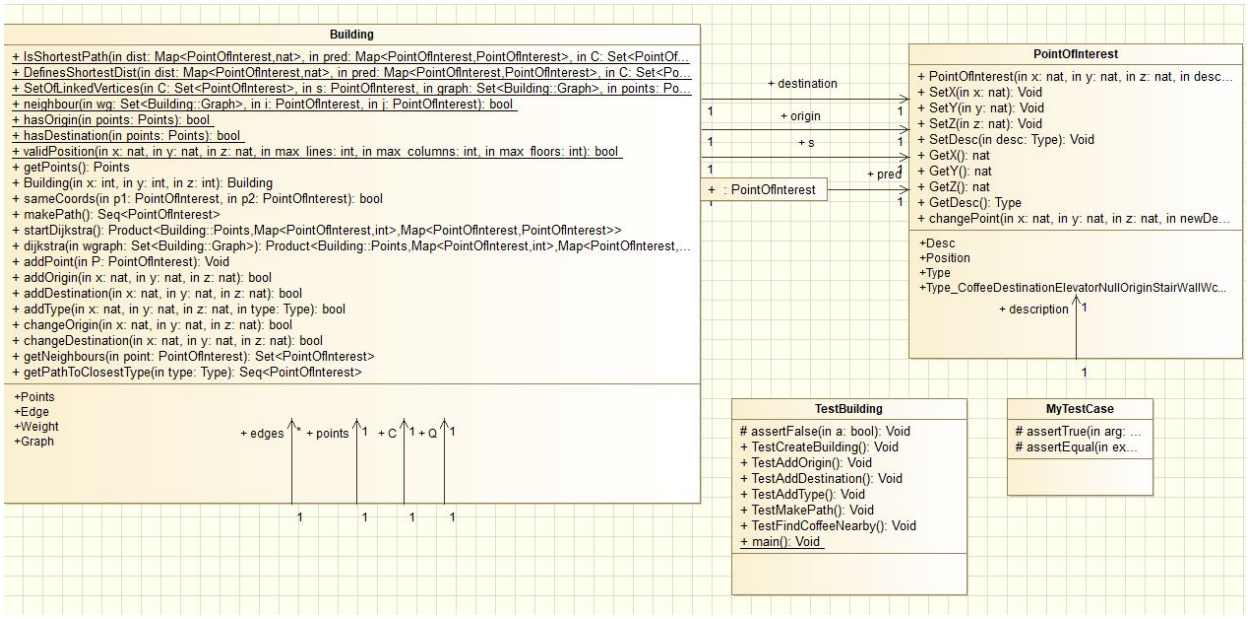
Excepções	(Não especificados)

Cenário	**Caminho mais curto entre dois pontos**
Descrição	Utilizador pretende obter a sequência de pontos que o leva da origem ao destino
Pré-condições	1. Os conjuntos gerados pelo algoritmo dijkstra de distâncias a partir de um ponto para outros é diferente do conjunto vazio 2. Os conjuntos gerados pelo algoritmo dijkstra de caminho para chegar até um qualquer outro ponto é diferente do conjunto vazio 3. Mapa de pontos tem pelo menos um ponto marcado como origem 4. Mapa de pontos tem pelo menos um ponto marcado como destino 5. Origem e Destino não podem ser o mesmo ponto
Pós-condições	Nenhumas
Passos	1. Utilizador cria o edificio 2. Utilizador insere diversos pontos no mapa de pontos do edificio 3. Marca um dos pontos como origem 4. Marca um dos pontos como destino 5. Pede o caminho mais curto entre os dois pontos
Excepções	(Não especificados)

Cenário	**Ponto de interesse mais próximo e caminho**
Descrição	Utilizador pretende saber onde se encontra um certo tipo de ponto de interesse mais próximo de uma origem, e

	qual a sequência de passos para lá chegar
Pré-condições	1. Os conjuntos gerados pelo algoritmo dijkstra de distâncias a partir de um ponto para outros é diferente do conjunto vazio 2. Os conjuntos gerados pelo algoritmo dijkstra de caminho para chegar até um qualquer outro ponto é diferente do conjunto vazio 3. Mapa de pontos tem pelo menos um ponto marcado como origem 4. Mapa de pontos tem pelo menos um ponto marcado como destino 5. Origem e Destino não podem ser o mesmo ponto
Pós-condições	Nenhumas
Passos	1. Utilizador cria o edificio 2. Utilizador insere diversos pontos no mapa de pontos do edificio 3. Marca um dos pontos como origem 4. Marca alguns dos pontos como pontos de interesse do tipo que pretende procurar 5. Pede qual o ponto mais próximo do tipo que pretende e qual o caminho
Excepções	(Não especificados)

c. Modelo UML de Classes



O modelo de classes vai dentro do ficheiro zip, para poder ser melhor visualizado, em generated

3. Modelo Formal VDM++

b. Building

```
class Building
types
        public Points = set of PointOfInterest;
        public Edge = set of PointOfInterest
       inv e == card(e)=2;
  public Weight = int;
        public Graph :: e : Edge
                                                    w : Weight;

values
        public min_lines : int = 3;
        public min_columns : int = 3;
        public min_floors : int = 1;

instance variables
        public points : Points := {};
        public origin : PointOfInterest;
        public destination : PointOfInterest;
        public s : PointOfInterest;
```

```
	public C: Points := {};
	public Q: Points := {};
	public dist: map PointOfInterest to nat := {|->};
	public pred: map PointOfInterest to PointOfInterest := {|->};
	public edges : set of Points := {};
	public weights : set of nat := {1};
	public graph : set of Graph := {};
	public max_lines: int;
	public max_columns : int;
	public max_floors : int;

functions
	/* CHECK if we're only moving 1 position at a time, ie one move on the x left or right, one
move on y up or down and one move on z up or down*/
	public moveIsValid : nat * nat * nat -> bool
		moveIsValid(deltaX, deltaY, deltaZ) == (
			(deltaX = 0 and deltaY = 0 and deltaZ = 1) or
			(deltaX = 0 and deltaY = 0 and deltaZ = -1) or
			(deltaX = 1 and deltaY = 0 and deltaZ = 0) or
			(deltaX = -1 and deltaY = 0 and deltaZ = 0) or
			(deltaX = 0 and deltaY = 1 and deltaZ = 0) or
			(deltaX = 0 and deltaY = -1 and deltaZ = 0)
		);

	/* CHECK If result is the Shortest Path*/
	public IsShortestPath: map PointOfInterest to nat * map PointOfInterest to
PointOfInterest * set of PointOfInterest * PointOfInterest * set of Graph * Points -> bool
	 IsShortestPath(dist, pred, C, s, graph,points) ==
		DefinesShortestDist(dist, pred, C, s, graph) and
SetOfLinkedVertices(C,s,graph, points);

	public DefinesShortestDist: map PointOfInterest to nat * map PointOfInterest to
PointOfInterest * set of PointOfInterest * PointOfInterest * set of Graph -> bool
	DefinesShortestDist(dist, pred, C, s, graph) ==
	(dist(s)=0 and
	forall u in set C\{s} & (exists v in set C & (
		pred(u)=v and neighbour(graph, u, v) and
		let tup in set graph be st tup.e={u,v} in (dist(u)=dist(v)+tup.w)))
	and
	forall u1, v in set C & (neighbour(graph, u1, v) =>
			let tup in set graph be st tup.e={u1,v} in (dist(u1)<=dist(v)+tup.w))
 );
```

```
 public SetOfLinkedVertices: set of PointOfInterest * PointOfInterest * set of Graph * Points ->
bool
	SetOfLinkedVertices(C,s,graph,points) ==
	(forall u in set C & (forall v in set points & neighbour(graph, u, v) => v in set C) and
	forall u1 in set C\{s} & (exists v in set C & neighbour(graph, u1, v))
	);

	public neighbour: set of Graph * PointOfInterest * PointOfInterest   -> bool
	neighbour(wg, i, j) ==
	exists tup in set wg & (tup.e={i,j} and tup.w>0 and j.description <> <Wall>);

	/* CHECK if building has origin defined*/
	public hasOrigin : Points -> bool
		hasOrigin(points) == (
			exists p in set points & p.description = <Origin>
		);

		-- CHECK if building has destination defined
	public hasDestination : Points -> bool
		hasDestination(points) == (
			exists p in set points & p.description = <Destination>
		);


	/*CHECK if x,y,z is valid position (its on points)*/
	public validPosition : nat * nat * nat * int * int * int-> bool
	validPosition(x,y,z,max_lines,max_columns,max_floors) == (
		(x >= 0 and x <= max_lines and y >= 0 and y <= max_columns and z >= 0 and z
<= max_floors)
		);
operations

	public getPoints:() ==> Points
	getPoints() == (
		return points;
	);

	/*CREATE a building*/
	public Building: int*int*int ==> Building
	Building(x,y,z) == (
		max_lines := x-1;
		max_columns := y-1;
		max_floors := z-1;
```

```
	for i = 0 to (x - 1) do
	(
		for j = 0 to (y - 1) do
		(
			for k = 0 to (z - 1) do
			(
				dcl point: PointOfInterest := new PointOfInterest(i,j,k,<Null>);
				addPoint(point);
			)
		)
	);
	return self;
	);



	/*CHECK if a point has the same coords as another point*/
	public sameCoords: PointOfInterest * PointOfInterest ==> bool
	sameCoords(p1, p2) == (
		if(p1.GetX() = p2.GetX() and p1.GetY() = p2.GetY() and p1.GetZ() = p2.GetZ())
then (
			return true;
		) else (
			return false;
		)
	) pre p1 in set points and p2 in set points;

	/*MAKE path from origin to dest*/
	public makePath: () ==> seq of PointOfInterest
	makePath() == (
		dcl pathToDest: seq of PointOfInterest := [];
		dcl arrivedOrigin: bool := false;
		dcl tempPoint: PointOfInterest :=  destination;
		dcl count : int := 0;
		dcl maxIterations : int := card(dom pred);
		pathToDest := pathToDest ^ [tempPoint]; --add destination as starter
		while(not arrivedOrigin and count < maxIterations) do (
			for all p1 in set dom pred do (
			if(sameCoords(tempPoint, p1)) then (
				if(sameCoords(p1, origin)) then (
					arrivedOrigin := true;
				) else (
```

```
                                        pathToDest := pathToDest ^ [pred(p1)]; --when we find the
one that matches the tempPoint then we add the value that it maps to on the map to the set of
path
                                        tempPoint := pred(p1);
                                )
                        )
                        );
                        count := count + 1;
                );
                if(count = maxIterations and not arrivedOrigin) then (
                        return [];
                ) else (
                        return pathToDest;
                );
        ) pre dist <> {|->} and pred <> {|->} and hasOrigin(points) and hasDestination(points);


        public startDijkstra: () ==> Building`Points * map PointOfInterest to int * map
PointOfInterest to PointOfInterest
                startDijkstra() == (
                for all p in set points do(
                        let tedges = {{p,e2}| e2 in set getNeighbours(p) & e2.GetDesc() <>
<Wall>} in(

                        edges := edges union tedges;
                        )
                );

                let tedges = {e | e in set edges} in(
                        let tgraph = {mk_Graph(e,w) | e in set edges, w in set weights} in (
                        graph := tgraph;
                        )
                );

                dist := {|->};
                pred := {|->};
                C := {};
                Q := {};

                return dijkstra(graph);

                )
                                pre hasOrigin(points);
```

```
/*Dijkstra Algorithm*/

	public dijkstra: set of Graph ==> Building`Points * map PointOfInterest to int * map
PointOfInterest to PointOfInterest

	dijkstra(wgraph) == (

		s := origin;
		Q := points;
		C := C union {s};
		dist := dist ++ {s|->0};
		pred := dist ++ {s|->s};

		while(C inter Q <> {}) do (
			let u in set C inter Q be st (forall u1 in set C inter Q & dist(u)<=dist(u1)) in
(
			Q := Q\{u};

				for all v in set Q do(
				if(v.GetDesc() <> <Wall>) then (
					if(u in set getNeighbours(v)) then(
					let uv in set wgraph be st uv.e={u,v} in(
						if uv.w > 0 then (
							let length=dist(u)+uv.w in(
							if v in set C then(
								if length < dist(v) then(
								dist := dist ++ {v|->length};
								pred := pred ++ {v|-> u};
								)
							)
							else (
								dist := dist ++ {v|->length};
								pred := pred ++ {v|->u};
								C := C union {v};
							)
							)
						)
					)
					)
				)
			);
			)
		);
```

```
        return mk_(C,dist,pred);
)
pre dist = {|->} and   pred = {|->} and C={} and Q={}
post IsShortestPath(dist, pred, C, s, graph,points);

/* ADD a point to the building*/
public addPoint: PointOfInterest ==> ()
addPoint(P) == (
                points := points union {P};
        return;
)
pre P not in set points
post P in set points;

/* ADD an origin point to the building*/
public addOrigin: nat * nat * nat ==> bool
addOrigin(x,y,z) == (
        for all p in set points do (
                if (p.GetX() = x and p.GetY() = y and p.GetZ() = z)
                then (
                p.changePoint(x,y, z, <Origin>);
                origin := p;
                return true;
                )
        );
        return false;
)
pre  not hasOrigin(points) and validPosition(x,y,z, max_lines,max_columns,max_floors)
post hasOrigin(points);

/* ADD a destination point to the building */
public addDestination: nat * nat * nat ==> bool
addDestination(x,y,z) == (
        for all p in set points do (
                if (p.GetX() = x and p.GetY() = y and p.GetZ() = z)
                then (
                p.changePoint(x,y, z, <Destination>);
                destination := p;
                return true;
                )
        );
        return false;
)
```

```
	pre  not hasDestination(points) and
validPosition(x,y,z,max_lines,max_columns,max_floors)
	post hasDestination(points);

	/* ADD a type to a point in the building*/
	public addType: nat * nat * nat * PointOfInterest`Type==> bool
	addType(x,y,z,type) == (
		for all p in set points do (
			if (p.GetX() = x and p.GetY() = y and p.GetZ() = z)
			then (
			p.changePoint(x,y,z, type);
			return true;
			)
		);
		return false;
	) pre validPosition(x,y,z,max_lines,max_columns,max_floors);

		-- CHANGE Origin point to the building
	public changeOrigin: nat * nat * nat ==> bool
		changeOrigin(x,y,z) == (
		dcl result : bool := false;
			for all p in set points do (
				if (p.GetX() = x and p.GetY() = y and p.GetZ() = z)
				then (
					p.changePoint(x,y, z, <Origin>);
					origin := p;
					result := true;
				)
			);
			if(result) then (
			for all p in set points do (
				if (p.GetDesc() = <Origin> and p.GetX() <> x and p.GetY() <> y
and p.GetZ() <> z)
				then (
					p.changePoint(x,y, z, <Null>);
				)
			);
			);
			return result;
		)
		pre  hasOrigin(points) and
validPosition(x,y,z,max_lines,max_columns,max_floors)
		post hasOrigin(points);
```

```
		-- CHANGE Destination point to the building
	public changeDestination: nat * nat * nat ==> bool
		changeDestination(x,y,z) == (
		dcl result : bool := false;
			for all p in set points do (
				if (p.GetX() = x and p.GetY() = y and p.GetZ() = z)
				then (
					p.changePoint(x,y, z, <Destination>);
					destination := p;
					result := true;
				)
			);
			if(result) then (
			for all p in set points do (
				if (p.GetDesc() = <Destination> and p.GetX() <> x and p.GetY()
<> y and p.GetZ() <> z)
				then (
					p.changePoint(x,y, z, <Null>);
				)
			);
			);
			return result;
		)
		pre  hasDestination(points) and
validPosition(x,y,z,max_lines,max_columns,max_floors)
		post hasDestination(points);


	/* GET all positions next to a Point*/
	public getNeighbours : PointOfInterest ==> set of PointOfInterest
	getNeighbours(point) == (
		dcl neighboursSet : set of PointOfInterest := {};
		for all p1 in set points do (
			if(p1.GetX() = (point.GetX() - 1) and p1.GetY() = point.GetY() and
p1.GetZ() = point.GetZ()) then
			(
				neighboursSet := neighboursSet union {p1};
			) else if (p1.GetX() = (point.GetX() + 1) and p1.GetY() = point.GetY() and
p1.GetZ() = point.GetZ()) then
			(
				neighboursSet := neighboursSet union {p1};
```

```
                        ) else if (p1.GetX() = point.GetX() and p1.GetY() = (point.GetY() - 1) and
p1.GetZ() = point.GetZ()) then
                        (
                                neighboursSet := neighboursSet union {p1};
                        ) else if (p1.GetX() = point.GetX() and p1.GetY() = (point.GetY() + 1) and
p1.GetZ() = point.GetZ()) then
                        (
                                neighboursSet := neighboursSet union {p1};
                        );
                        if(point.GetDesc() = <Stair> or point.GetDesc() = <Elevator>) then(
                                if (p1.GetX() = point.GetX() and p1.GetY() = point.GetY() and
p1.GetZ() = (point.GetZ()+1)) then
                                (
                                        neighboursSet := neighboursSet union {p1};
                                ) else if (p1.GetX() = point.GetX() and p1.GetY() = point.GetY()
and p1.GetZ() = (point.GetZ()-1)) then
                                (
                                        neighboursSet := neighboursSet union {p1};
                                );
                        )
                );
                return neighboursSet;
        )
        pre points <> {};


        /*GET closer type*/
        public getPathToClosestType : PointOfInterest`Type ==> seq of PointOfInterest
                getPathToClosestType(type) == (
                        for all p1 in set dom dist do (
                        if(p1.GetDesc() = type) then (
                                destination := p1;
                                return makePath();
                        )
                        );
                        return [];
                )
                pre dist <> {|->} and hasOrigin(points) and exists p in set points & (p.description =
type);

end Building
```

c. PointOfInterest

```
class PointOfInterest
types
	public Desc = seq of char;
	public Position :: x: nat
                                                    y: nat
                                                    z: nat;

	public Type = <Destination> | <Origin> | <Null> | <Wc> | <Coffee> | <Stair> | <Elevator>
| <Wall>
instance variables
	public position : Position;
	public description : Type;
operations
	/* Create point of interest*/
	public PointOfInterest : nat * nat * nat * PointOfInterest`Type  ==> PointOfInterest
	PointOfInterest(x,y,z,desc) == (
	position := mk_Position(x,y,z);
	  description := desc;
	);
	public SetX: nat ==> ()
	SetX(x) == (position.x := x;);
	public SetY : nat ==> ()
	SetY(y) == (position.y := y;);
	public SetZ : nat ==> ()
	SetZ(z) == (position.z := z;);
	public SetDesc : PointOfInterest`Type ==> ()
	SetDesc(desc) == (description := desc;);
	public GetX : () ==> nat
	GetX() == (return position.x;);
	public GetY : () ==> nat
	GetY() == (return position.y;);
	public GetZ : () ==> nat
	GetZ() == (return position.z;);
	public GetDesc : () ==> PointOfInterest`Type
	GetDesc() == (return description;);

	/* Change position and Description of point of interest */
	public changePoint : nat * nat * nat * PointOfInterest`Type ==> ()
	changePoint(x, y, z, newDesc) == (
		position.x := x;
```

```
            position.y := y;
            position.z := z;
            description := newDesc;
        );

end PointOfInterest
```

4. Validação do Modelo

Apresenta-se de seguida o código de teste do modelo bem como a cobertura dos código que pode ser melhor verifica na pasta de do projeto “coverage”.

a. Cobertura Building

```
class Building
types
   public Points = set of PointOfInterest;
   public Edge = set of PointOfInterest
  inv e == card(e)=2;
   public Weight = int;
   public Graph :: e : Edge
                   w : Weight;

values
  public min_lines : int = 3;
  public min_columns : int = 3;
  public min_floors : int = 1;

instance variables
  public points : Points := {};
  public origin : PointOfInterest;
  public destination : PointOfInterest;
  public s : PointOfInterest;
  public C: Points := {};
  public Q: Points := {};
  public dist: map PointOfInterest to nat := {|->};
  public pred: map PointOfInterest to PointOfInterest := {|->};
  public edges : set of Points := {};
  public weights : set of nat := {1};
  public graph : set of Graph := {};
  public max_lines: int;
  public max_columns : int;
  public max_floors : int;

functions

  -- CHECK If result is the Shortest Path
  public IsShortestPath: map PointOfInterest to nat * map PointOfInterest to PointOfInterest * set of PointOfInterest * PointOfInterest * set of Graph * Points -> bool
   IsShortestPath(dist, pred, C, s, graph,points) ==
    DefinesShortestDist(dist, pred, C, s, graph) and SetOfLinkedVertices(C,s,graph, points);

  public DefinesShortestDist: map PointOfInterest to nat * map PointOfInterest to PointOfInterest * set of PointOfInterest * PointOfInterest * set of Graph -> bool
  DefinesShortestDist(dist, pred, C, s, graph) ==
  (dist(s)=0 and
    forall u in set C\{s} & (exists v in set C & (
      pred(u)=v and neighbour(graph, u, v) and
      let tup in set graph be st tup.e={u,v} in (dist(u)=dist(v)+tup.w)))
    and
    forall u1, v in set C & (neighbour(graph, u1, v) =>
        let tup in set graph be st tup.e={u1,v} in (dist(u1)<=dist(v)+tup.w))
);

public SetOfLinkedVertices: set of PointOfInterest * PointOfInterest * set of Graph * Points -> bool
  SetOfLinkedVertices(C,s,graph,points) ==
  (forall u in set C & (forall v in set points & neighbour(graph, u, v) => v in set C) and
    forall u1 in set C\{s} & (exists v in set C & neighbour(graph, u1, v))
  );

  public neighbour: set of Graph * PointOfInterest * PointOfInterest  -> bool
    neighbour(wg, i, j) ==
    exists tup in set wg & (tup.e={i,j} and tup.w>0 and j.description <> <Wall>);

    -- CHECK if building has origin defined
    public hasOrigin : Points -> bool
      hasOrigin(points) == (
        exists p in set points & p.description = <Origin>
      );
```

```
64
65      -- CHECK if building has destination defined
66    public hasDestination : Points -> bool
67      hasDestination(points) == (
68        exists p in set points & p.description = <Destination>
69      );
70
71
72    --CHECK if x,y,z is valid position (its on points)
73    public validPosition : nat * nat * nat * int * int * int-> bool
74    validPosition(x,y,z,max_lines,max_columns,max_floors) == (
75      (x >= 0 and x <= max_lines and y >= 0 and y <= max_columns and z >= 0 and z <= max_floors)
76      );
77 operations
78
79  public getPoints:() ==> Points
80    getPoints() == (
81      return points;
82    );
83
84  --CREATE a building
85  public Building: int*int*int ==> Building
86  Building(x,y,z) == (
87      max_lines := x-1;
88      max_columns := y-1;
89      max_floors := z-1;
90
91    for i = 0 to (x - 1) do
92    (
93      for j = 0 to (y - 1) do
94      (
95          for k = 0 to (z - 1) do
96          (
97            dcl point: PointOfInterest := new PointOfInterest(i,j,k,<Null>);
98            addPoint(point);
99          )
100     )
101   );
102   return self;
103   );
104
105 --CHECK if a point has the same coords as another point
106 public sameCoords: PointOfInterest * PointOfInterest ==> bool
107   sameCoords(p1, p2) == (
108     if(p1.GetX() = p2.GetX() and p1.GetY() = p2.GetY() and p1.GetZ() = p2.GetZ()) then (
109       return true;
110     ) else (
111       return false;
112     )
113   ) pre p1 in set points and p2 in set points;
114
115 --MAKE path from origin to dest
116 public makePath: () ==> seq of PointOfInterest
117   makePath() == (
118     dcl pathToDest: seq of PointOfInterest := [];
119     dcl arrivedOrigin: bool := false;
120     dcl tempPoint: PointOfInterest :=  destination;
121     dcl count : int := 0;
122     dcl maxIterations : int := card(dom pred);
123     pathToDest := pathToDest ^ [tempPoint]; --add destination as starter
124     while(not arrivedOrigin and count < maxIterations) do (
```

```
      for all p1 in set dom pred do (
        if(sameCoords(tempPoint, p1)) then (
          if(sameCoords(p1, origin)) then (
              arrivedOrigin := true;
          ) else (
            pathToDest := pathToDest ^ [pred(p1)]; --when we find the one that matches the tempPoint then we add the value that it maps to on the map to the set of path
            tempPoint := pred(p1);
          )
        )
      );
      count := count + 1;
    );
    if(count = maxIterations and not arrivedOrigin) then (
      return [];
    ) else (
      return pathToDest;
    );
  ) pre dist <> {|->} and pred <> {|->} and hasOrigin(points) and hasDestination(points);


  public startDijkstra: () ==> Building`Points * map PointOfInterest to int * map PointOfInterest to PointOfInterest
    startDijkstra() == (
    for all p in set points do(
      let tedges = {{p,e2}| e2 in set getNeighbours(p) & e2.GetDesc() <> <Wall>} in(

        edges := edges union tedges;
      )
    );

    let tedges = {e | e in set edges} in(
      let tgraph = {mk_Graph(e,w) | e in set edges, w in set weights} in (
        graph := tgraph;
      )
    );

    dist := {|->};
    pred := {|->};
    C := {};
    Q := {};

    return dijkstra(graph);

    )
          pre hasOrigin(points);

public dijkstra: set of Graph ==> Building`Points * map PointOfInterest to int * map PointOfInterest to PointOfInterest

  dijkstra(wgraph) == (

    s := origin;
    Q := points;
    C := C union {s};
    dist := dist ++ {s|->0};
    pred := dist ++ {s|->s};

    while(C inter Q <> {}) do (
      let u in set C inter Q be st (forall u1 in set C inter Q & dist(u)<=dist(u1)) in (
        Q := Q\{u};

        for all v in set Q do(
          if(v.GetDesc() <> <Wall>) then (
            if(u in set getNeighbours(v)) then(
```

```
187                 let uv in set wgraph be st uv.e={u,v} in(
188                   if uv.w > 0 then (
189                     let length=dist(u)+uv.w in(
190                       if v in set C then(
191                         if length < dist(v) then(
192                           dist := dist ++ {v|->length};
193                           pred := pred ++ {v|-> u};
194                         )
195                       )
196                       else (
197                         dist := dist ++ {v|->length};
198                         pred := pred ++ {v|->u};
199                         C := C union {v};
200                       )
201                     )
202                   )
203                 )
204               )
205             )
206           );
207         )
208       );
209       return mk_(C,dist,pred);
210   )
211   pre dist = {|->} and pred = {|->} and C={} and Q={}
212   post IsShortestPath(dist, pred, C, s, graph,points);
213
214   -- ADD a point to the building
215   public addPoint: PointOfInterest ==> ()
216     addPoint(P) == (
217         points := points union {P};
218       return;
219     )
220     pre P not in set points
221     post P in set points;
222
223   -- ADD an origin point to the building
224   public addOrigin: nat * nat * nat ==> bool
225     addOrigin(x,y,z) == (
226       for all p in set points do (
227         if (p.GetX() = x and p.GetY() = y and p.GetZ() = z)
228         then (
229           p.changePoint(x,y, z, <Origin>);
230           origin := p;
231           return true;
232         )
233       );
234       return false;
235     )
236     pre  not hasOrigin(points) and validPosition(x,y,z, max_lines,max_columns,max_floors)
237     post hasOrigin(points);
238
239   -- ADD a destination point to the building
240   public addDestination: nat * nat * nat ==> bool
241     addDestination(x,y,z) == (
242       for all p in set points do (
243         if (p.GetX() = x and p.GetY() = y and p.GetZ() = z)
244         then (
245           p.changePoint(x,y, z, <Destination>);
246           destination := p;
247           return true;
248         )
```

```
249       );
250       return false;
251     )
252     pre  not hasDestination(points) and validPosition(x,y,z,max_lines,max_columns,max_floors)
253     post hasDestination(points);
254
255   -- ADD a type to a point in the building
256     public addType: nat * nat * nat * PointOfInterest`Type==> bool
257     addType(x,y,z,type) == (
258       for all p in set points do (
259         if (p.GetX() = x and p.GetY() = y and p.GetZ() = z)
260         then (
261           p.changePoint(x,y,z, type);
262           return true;
263         )
264       );
265       return false;
266     ) pre validPosition(x,y,z,max_lines,max_columns,max_floors);
267
268   -- CHANGE Origin point to the building
269   public changeOrigin: nat * nat * nat ==> bool
270     changeOrigin(x,y,z) == (
271     dcl result : bool := false;
272       for all p in set points do (
273         if (p.GetX() = x and p.GetY() = y and p.GetZ() = z)
274         then (
275           p.changePoint(x,y, z, <Origin>);
276           origin := p;
277           result := true;
278         )
279       );
280       if(result) then (
281       for all p in set points do (
282         if (p.GetDesc() = <Origin> and p.GetX() <> x and p.GetY() <> y and p.GetZ() <> z)
283         then (
284           p.changePoint(x,y, z, <Null>);
285         )
286       );
287       );
288       return result;
289     )
290     pre  hasOrigin(points) and validPosition(x,y,z,max_lines,max_columns,max_floors)
291     post hasOrigin(points);
292
293   -- CHANGE Destination point to the building
294   public changeDestination: nat * nat * nat ==> bool
295     changeDestination(x,y,z) == (
296     dcl result : bool := false;
297       for all p in set points do (
298         if (p.GetX() = x and p.GetY() = y and p.GetZ() = z)
299         then (
300           p.changePoint(x,y, z, <Destination>);
301           destination := p;
302           result := true;
303         )
304       );
305       if(result) then (
306       for all p in set points do (
307         if (p.GetDesc() = <Destination> and p.GetX() <> x and p.GetY() <> y and p.GetZ() <> z)
308         then (
309           p.changePoint(x,y, z, <Null>);
```

```
311       );
312       );
313       return result;
314     )
315     pre  hasDestination(points) and validPosition(x,y,z,max_lines,max_columns,max_floors)
316     post hasDestination(points);
317
318
319   -- GET all positions next to a Point
320   public getNeighbours : PointOfInterest ==> set of PointOfInterest
321     getNeighbours(point) == (
322       dcl neighboursSet : set of PointOfInterest := {};
323       for all p1 in set points do (
324           if(p1.GetX() = (point.GetX() - 1) and p1.GetY() = point.GetY() and p1.GetZ() = point.GetZ()) then
325           (
326             neighboursSet := neighboursSet union {p1};
327           ) else if (p1.GetX() = (point.GetX() + 1) and p1.GetY() = point.GetY() and p1.GetZ() = point.GetZ()) then
328           (
329             neighboursSet := neighboursSet union {p1};
330           ) else if (p1.GetX() = point.GetX() and p1.GetY() = (point.GetY() - 1) and p1.GetZ() = point.GetZ()) then
331           (
332             neighboursSet := neighboursSet union {p1};
333           ) else if (p1.GetX() = point.GetX() and p1.GetY() = (point.GetY() + 1) and p1.GetZ() = point.GetZ()) then
334           (
335             neighboursSet := neighboursSet union {p1};
336           );
337           if(point.GetDesc() = <Stair> or point.GetDesc() = <Elevator>) then(
338             if (p1.GetX() = point.GetX() and p1.GetY() = point.GetY() and p1.GetZ() = (point.GetZ()+1)) then
339             (
340               neighboursSet := neighboursSet union {p1};
341             ) else if (p1.GetX() = point.GetX() and p1.GetY() = point.GetY() and p1.GetZ() = (point.GetZ()-1)) then
342             (
343               neighboursSet := neighboursSet union {p1};
344             );
345           )
346       );
347       return neighboursSet;
348     )
349     pre points <> {};
350
351
352     --GET closer type
353     public getPathToClosestType : PointOfInterest`Type ==> seq of PointOfInterest
354       getPathToClosestType(type) == (
355         for all p1 in set dom dist do (
356           if(p1.GetDesc() = type) then (
357             destination := p1;
358             return makePath();
359           )
360         );
361         return [];
362       )
363       pre dist <> {|->} and hasOrigin(points) and exists p in set points & (p.description = type);
364
365 end Building
```

b. Cobertura PointOfInterest

```
class PointOfInterest
types
    public Desc = seq of char;
    public Position :: x: nat
                       y: nat
                       z: nat;

    public Type = <Destination> | <Origin> | <Null> | <Wc> | <Coffee> | <Stair> | <Elevator> | <Wall> ;

instance variables
    public position : Position;
    public description : Type := <Null>;
operations
  -- Create point of interest
  public PointOfInterest : nat * nat * nat * PointOfInterest`Type  ==> PointOfInterest
  PointOfInterest(x,y,z,desc) == (
    position := mk_Position(x,y,z);
    description := desc;
  );
  public SetX: nat ==> ()
    SetX(x) == (position.x := x;);
  public SetY : nat ==> ()
    SetY(y) == (position.y := y;);
  public SetZ : nat ==> ()
    SetZ(z) == (position.z := z;);
  public SetDesc : PointOfInterest`Type ==> ()
    SetDesc(desc) == (description := desc;);
  public GetX : () ==> nat
    GetX() == (return position.x;);
  public GetY : () ==> nat
    GetY() == (return position.y;);
  public GetZ : () ==> nat
    GetZ() == (return position.z;);
  public GetDesc : () ==> PointOfInterest`Type
    GetDesc() == (return description;);

  -- Change position and Description of point of interest
  public changePoint : nat * nat * nat * PointOfInterest`Type ==> ()
    changePoint(x, y, z, newDesc) == (
      position.x := x;
      position.y := y;
      position.z := z;
      description := newDesc;
    );

end PointOfInterest
```

c. Classe TestBuilding

```
class TestBuilding is subclass of MyTestCase
operations
    protected assertFalse : bool ==> ()
      assertFalse(a) == return
        pre a = false;

    -- tests the creation of a building, and if it has an origin and a destination at the beginning which should be false since it doesn't have it!
    public TestCreateBuilding : () ==> ()
      TestCreateBuilding() == (
          dcl b : Building := new Building(5,5,2);
          assertFalse(b.hasOrigin(b.points));
          assertFalse(b.hasDestination(b.points));
    );

    -- tests adding an origin to the building and checking if the building has an origin after
    public TestAddOrigin : () ==> ()
      TestAddOrigin() == (
          dcl b : Building := new Building(5,5,2);
          dcl result : bool;
          assertFalse(b.hasOrigin(b.points));
          result := b.addOrigin(1,0,0);
          assertTrue(b.hasOrigin(b.points));
      );

    -- tests adding a destination to the building and checking if the building has a destination after
    public TestAddDestination : () ==> ()
      TestAddDestination() == (
          dcl b : Building := new Building(5,5,2);
          dcl result : bool;
          assertFalse(b.hasDestination(b.points));
          result := b.addDestination(1,0,0);
          assertTrue(b.hasDestination(b.points));
      );

    -- tests changing a point type from NULL to Coffee and checking if it was changed
    public TestAddType : () ==> ()
      TestAddType() == (
          dcl b : Building := new Building(5,5,2);
          dcl result : bool;
          for all p1 in set b.points do (
            if(p1.GetX() = 1 and p1.GetY() = 1 and p1.GetZ() = 1) then (
              assertEqual(p1.GetDesc(), <Null>);
              result := b.addType(p1.GetX(),p1.GetY(),p1.GetZ(),<Coffee>);
              assertEqual(p1.GetDesc(), <Coffee>);
            );
          );
      );

    -- tests finding a path between an origin (1,0,0) and a destination (3,0,0) with a wall between
    -- and test changeOrigin and changeDestination
    public TestMakePath : () ==> ()
      TestMakePath() == (
        dcl b: Building := new Building(5,5,2);
        dcl result : bool;
        dcl path : seq of PointOfInterest;
        dcl result2 : (Building`Points * map PointOfInterest to int * map PointOfInterest to PointOfInterest);
        result := b.addOrigin(1,0,0);
        result := b.addDestination(3,0,0);
        result := b.addType(2,0,0,<Wall>);
        result2 := b.startDijkstra();
        path := b.makePath();
        assertEqual(len(path),5);
        result:= b.changeOrigin(4,4,0);
```

```
        result:= b.changeDestination(3,3,0);
        result2:= b.startDijkstra();
        path:= b.makePath();
        assertEqual(len(path),3);
      );

    -- tests if it can find a coffee nearby an origin after adding a coffee spot
    public TestFindCoffeeNearby : () ==> ()
      TestFindCoffeeNearby() == (
        dcl b: Building := new Building(5,5,2);
        dcl result : bool;
        dcl path : seq of PointOfInterest;
        dcl result2 : (Building`Points * map PointOfInterest to int * map PointOfInterest to PointOfInterest);
        result := b.addOrigin(1,0,0);
        result := b.addDestination(3,0,0);
        result2 := b.startDijkstra();

        for all p1 in set b.points do (
          if(p1.GetX() = 2 and p1.GetY() = 2 and p1.GetZ() = 0) then (
            assertEqual(p1.GetDesc(), <Null>);
            result := b.addType(p1.GetX(),p1.GetY(),p1.GetZ(),<Coffee>);
            assertEqual(p1.GetDesc(), <Coffee>);
          );
        );

        path := b.getPathToClosestType(<Coffee>);
        assertEqual(len(path),4);

      );

    public FailingTestMakePath : () ==> ()
      FailingTestMakePath() == (
        dcl b: Building := new Building(5,5,2);
        dcl result : bool;
        dcl path : seq of PointOfInterest;
        dcl result2 : (Building`Points * map PointOfInterest to int * map PointOfInterest to PointOfInterest);
        result2 := b.startDijkstra(); -- breaks pre-condition
        path := b.makePath();
      );

    public FailingTestFindCoffeeNearby : () ==> ()
      FailingTestFindCoffeeNearby() == (
        dcl b: Building := new Building(5,5,2);
        dcl result : bool;
        dcl path : seq of PointOfInterest;
        dcl result2 : (Building`Points * map PointOfInterest to int * map PointOfInterest to PointOfInterest);
        result := b.addOrigin(1,0,0);
        result := b.addDestination(3,0,0);
        result2 := b.startDijkstra();

        path := b.getPathToClosestType(<Coffee>); -- breaks pre-condition
        assertEqual(len(path),4);
      );

    public static main: () ==> ()
      main() ==
      (
        new TestBuilding().TestCreateBuilding();
        new TestBuilding().TestAddOrigin();
        new TestBuilding().TestAddDestination();
        new TestBuilding().TestAddType();
          new TestBuilding().TestMakePath();
          new TestBuilding().TestFindCoffeeNearby();
      );
functions
-- TODO Define functiones here
traces
-- TODO Define Combinatorial Test Traces here
end TestBuilding
```

5. Geração de código JAVA

Com o uso do VDM foi possível converter facilmente o código feito para Java. Todas as classes ficaram numa pasta, /generated/java. No entanto, tivemos que fazer algumas alterações, em primeiro lugar tivemos que importar as classes do package “Building.quotes” em todas as classes e depois, também nas três classes, temos que eliminar “Building.quotes.” em todos os sítios onde aparece por exemplo “Building.quotes.NullQuote.getInstance();” e apenas deixar, neste caso, “NullQuote.getInstance()”. De seguida, criamos um “main” para

conseguirmos executar o código em java e aí é quase igual ao método testAll() da class TestBuilding.

6. Conclusões

Relativamente ao trabalho desenvolvido, o grupo cumpriu 2 dos 3 requisitos pedidos na especificação sendo que ficou em falta o requisito de encontrar o caminho mais acessível (por exemplo para utilizadores com cadeira de rodas, que dá prioridade a elevadores em vez de escadas). Foram efetuados alguns testes, mas se o tempo o permitisse teriam sido efetuados mais de forma a garantir cobertura de código e cenários possíveis totais.

Relativamente a aspetos a melhorar, deveria ter sido implementado o requisito em falta, bem como na procura de um caminho entre dois pontos a mudança na coordenada dos Z (em altura) utilizando os pontos definidos como <Stair> ou <Elevator>.

Em relação à distribuição do trabalho entre os elementos do grupo, os elementos concordaram na distribuição da seguinte forma:

António Jorge Vale - 60%

Diogo Filipe Costa - 40%

7. Referências

[1] Formally Modeling and Analyzing Mathematical Algorithms with Software Specification Languages & Tools. Daniela Ritirc, BsC - Master Thesis available at URL: https://www.risc.jku.at/publications/download/risc_5224/Master_Thesis.pdf (acedido a 03 de Janeiro de 2018)

[2] Overture tool web site, URL: http://overturetool.org/ (acedido a 03 de Janeiro de 2018)

[3] Overture Quick Start Exercise, Available on the course's Moodle page

[4] TR-004: Tutorial for Overture/VDM++, P. G. Larsen, J. S. Fitzgerald, S. Wolff, N. Battle, K. G. Lausdahl, A. Ribeiro, K. G. Pierce, and S. Riddle, URL: current version (acedido a 03 de Janeiro de 2018)