Software-Ontwikkeling I

Academiejaar 2009 – 2010

SOI@intec.ugent.be

Project: Behaviour-based robotics

Inleiding

Traditioneel bestaat een robot uit een centraal brein dat een interne representatie of “map” bijhoudt van de wereld. Deze representatie wordt continu bijgewerkt aan de hand van input afkomstig van de verschillende sensoren en wordt gebruikt om de volgende actie te bepalen.

Het probleem met deze benadering is echter dat het vrij rekenintensief is en dat de robot op elk moment over een accurate, up-to-date versie van de “map” moet beschikken. Dit terwijl heel wat organismen, zoals insecten, perfect gedijen zonder enige vorm van geheugen.

Behaviour-based robotics is een filosofie binnen de robotica waarbij men dit probeert te emuleren. Een behaviour-based robot kan geïmplementeerd worden aan de hand van een subsumptie architectuur waarbij het algemene gedrag van de robot gedefinieerd wordt door de opsplitsing van dit algemene gedrag in kleine onderdelen (deze onderdelen worden verderop behaviours genoemd). Door het aftasten van de omgeving, wordt bepaald welke behaviours actief kunnen zijn. Er zal dan beslist worden, aan de hand van prioriteiten, welke behaviour de controle krijgt – de bijhorende actie wordt dan uitgevoerd. Conflicten tussen behaviours kunnen nooit optreden, omdat elke behaviour een bepaalde unieke prioriteit heeft. Dus ook al kunnen er meerdere behaviours op hetzelfde moment actief zijn, toch zal het behaviour met de hoogste prioriteit steeds gekozen worden voor uitvoering. Bijvoorbeeld: bij een rondzwervende robot zal het behaviour dat zorgt voor het ontwijken van obstakels een hogere prioriteit hebben dan het behaviour dat voor een willekeurige verplaatsing zorgt.

Voor meer informatie in verband met behaviour-based robotica en subsumptie architecturen: http://baibook.epfl.ch/exercises/behaviorBasedRobotics/BBSummary.pdf

Het doel van dit project is het implementeren van een eenvoudige subsumptie architectuur in C++, enkel gebruik makend van de ANSI C++ standaard.

De opgave is opgesplitst in twee delen: een deel voorbereidend werk dat tijdens de oefeningenles van 16/11 kan uitgevoerd worden en een deel eigenlijke implementatie.

I. Voorbereidend werk

a. BinarySearchTree template klasse

Om op een efficiënte manier map-informatie te kunnen opslaan en bevragen, zal gebruik gemaakt worden van een `BinarySearchTree`. Hiertoe zal de `BinarySearchTree` die geïmplementeerd werd in practicum 5 uitgebreid worden naar een template klasse die een aantal sleutel-waarde paren bijhoudt, gesorteerd op de sleutel. Zo kunnen objecten van een willekeurig type worden opgeslagen als waarde behorend bij een bepaalde sleutel.

Compilatie van templates:

Normaalgezien worden template klassen geïmplementeerd in een header file, omdat voor de compiler zowel definitie als implementatie zichtbaar moeten zijn op het moment van instantiëring van een template klasse (hier dus in de `BinarySearchTree` klasse) en omdat template klassen doorgaans klein in omvang zijn.

Hier is dit niet het geval en wordt de implementatie opgesplitst in header en cpp bestanden. Daarom voeg je best bijgevoegd bestand `template_instantiations.cpp` aan je project toe. Dit zal de compiler forceren de code te genereren voor een `BinarySearchTree` van strings (zoals die uit practicum 5, dit als voorbeeld om de werking te illustreren).

Doe je dit niet, dan zal je linker errors krijgen voor elk van de methoden van de `BinarySearchTree` die je probeert te gebruiken. Verder hoef je dit bestand nergens te includeren, deze dient enkel om de compiler te verplichten de juiste code te genereren. Uiteraard zal je in je uiteindelijke ontwerp een `BinarySearchTree` gebruiken met een ander waarde-type dan pointers naar strings. Dan moet je ook in dit bestand een lijn toevoegen met het juiste type zodat de juiste code voor de datastructuur gegenereerd wordt.

Meer uitleg over compilatie van templates kan je vinden op:
http://www.codeproject.com/cpp/templatesourceorg.asp

Implementeer de BinarySearchTree<K, T> template klasse met K het type van de sleutel en T het type van de waarde. Voeg een extra publieke functie toe:

```
T* find(const K& key);
```

Deze functie geeft een pointer terug naar de waarde die overeenstemt met de opgegeven sleutel.

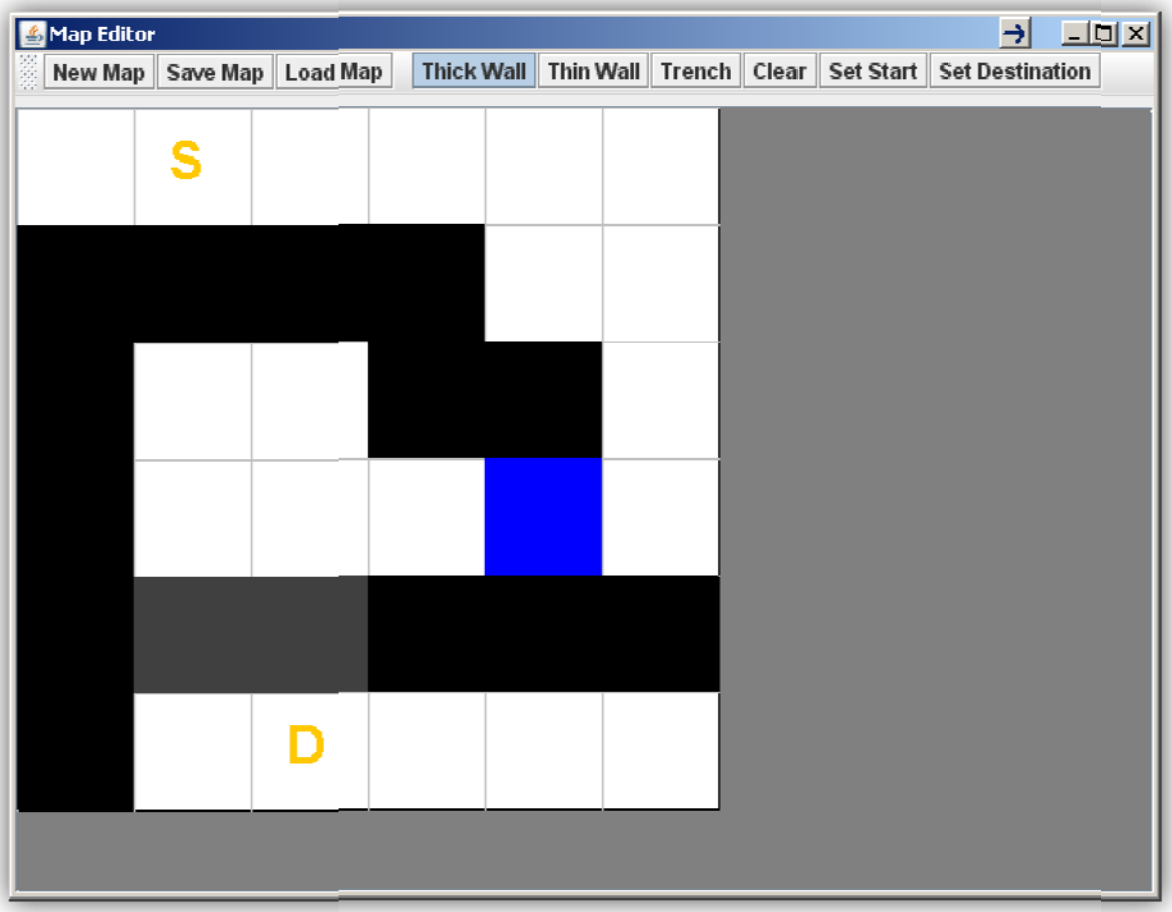
b. GUI tools

Omdat het project wegens praktische redenen niet kan getest worden op echte hardware worden er een aantal GUI tools ter beschikking gesteld zodat de robot toch gevisualiseerd kan worden. De bedoeling van dit deel is dat je kennis maakt met de mogelijkheden van deze tools.

Map Editor

`MapEditor.jar` is een eenvoudig programma waarmee bestanden met mapinformatie kunnen aangemaakt en bewerkt worden.

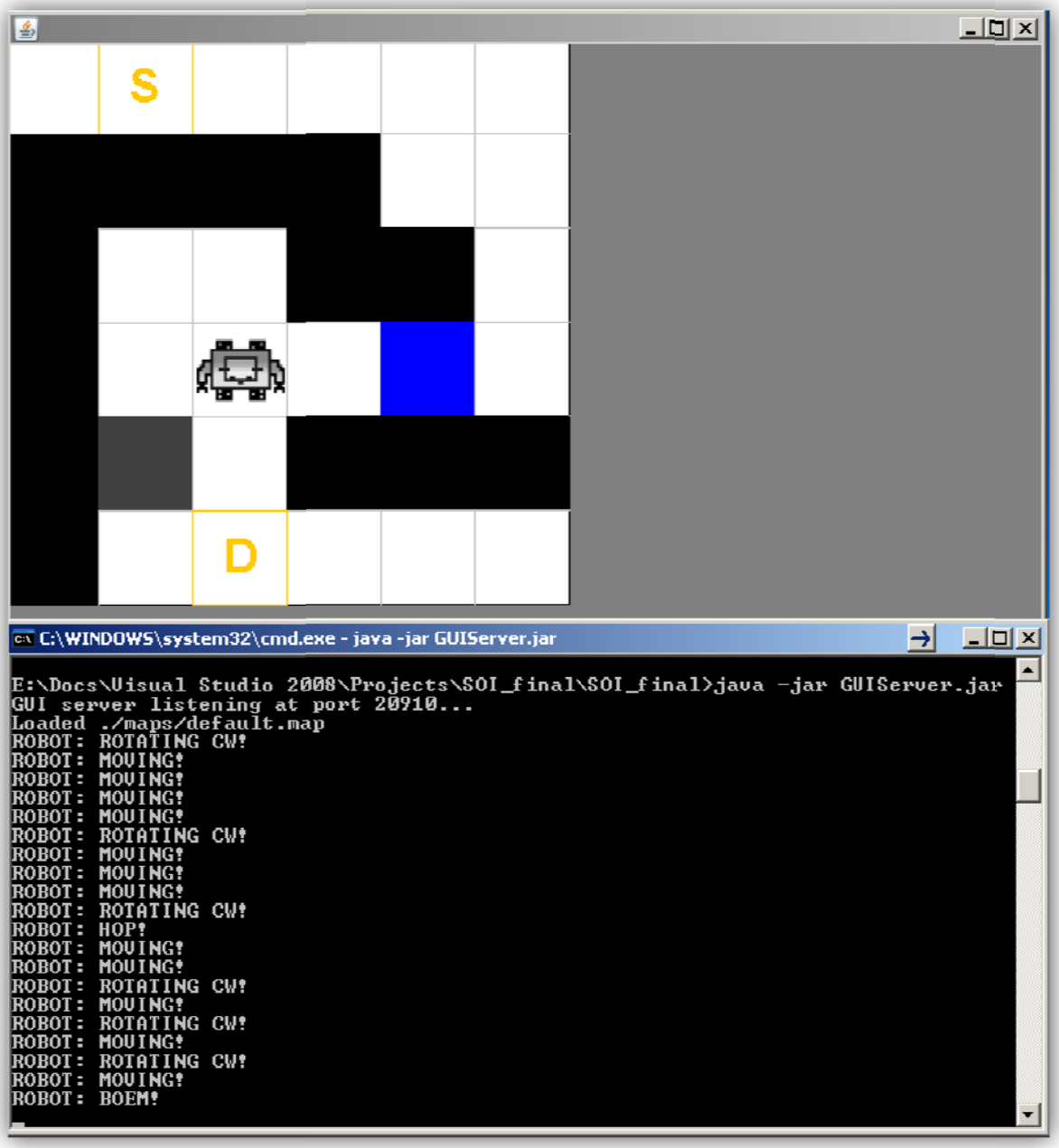
Klik op “New Map” en geef vervolgens de naam en dimensie van de map in om een nieuwe map aan te maken. Daarna kun je gebruik makend van de toolbar de verschillende objecten op de map tekenen. Klik op “Save Map” om de map dan als tekstbestand weg te schrijven naar default map-directory (zie configuratie).



Figuur 1: Map Editor

GUI Server

`GUIServer.jar` start een service die op een bepaalde netwerkpoort luistert naar inkomende instructies voor de grafische weergave. Gebruik makend van de GUI hulpklasse kunnen dan vanuit het C++ programma instructies gestuurd worden voor de visualisatie van acties. Start de service via de console voor debug-output.



Figuur 2: GUIServer

GUI.h

De `GUI` klasse abstraheert de `GUIServer` en biedt een aantal statische methoden aan voor de visualisatie van acties:

- **`void initialize(const string & file_name)`**: (her)initialiseert de `GUIServer` aan de hand van een mapbestand.
- **`void show(Command c)`**: visualiseert een bepaald commando (b.v. MOVE).

Configuratie

In het geval dat de default configuratie van `MapEditor` en `GUIServer` problemen oplevert, kan deze gewijzigd worden in het bestand `config.properties`. **Opgelet**: als je het poort nummer voor de `GUIServer` verandert, zorg dan dat je dit ook doet aan de client-zijde in `GUI.h`!

Schrijf een test methode die de GUIServer initialiseert en een aantal verschillende acties toont.

II. Eigenlijke implementatie

a. Representatie van de subsumption-architectuur

De belangrijkste klasse van de subsumption-architectuur is de `Behaviour` klasse. Deze abstracte klasse zal het gedrag van de robot modelleren.

Behaviour

`Behaviour` is een abstracte klasse met slechts 2 methodes:

- **`bool isActive(void)`:** deze methode bepaalt of het `Behaviour` actief is.
- **`void action(void)`:** wanneer deze `Behaviour` de hoogste prioriteit heeft, wordt deze actie uitgevoerd.

Main.cpp

In de `run` methode zal je achtereenvolgens alle objecten moeten initialiseren, deze objecten op de correcte plaats registreren en daarna wordt het subsumption algoritme uitgevoerd (weergegeven in Figuur 3). Een meer getailleerde beschrijving van deze gebruikte klassen volgt hieronder.

Het subsumption algoritme werkt als volgt. Eerst wordt de robot op de hoogte gebracht van zijn onmiddellijke omgeving (lijn 2). Vervolgens zal de actieve Behaviour met de hoogste prioriteit opgezocht worden (lijn 4). Van deze Behaviour wordt dan de actie uitgevoerd. Het subsumption algoritme zal herhaald worden tot de robot zijn eindbestemming heeft bereikt.

```
1 while(!robot.getDestinationReached()) {
2     map.refresh();
3
4     Behaviour* b = robot.getFirstActiveBehaviour();
5
6     b->action();
7 }
```

Figuur 3: het subsumption algoritme

b. Representatie van de omgeving

Map

Eerst en vooral hebben we het `Map` object dat de omgeving zal weergeven (let op: dit is niet de STL map klasse). Het is de bedoeling dat `Map`-objecten kunnen geconstrueerd worden aan de hand van tekstbestanden die gegenereerd worden door de `MapEditor` (zie I.b) . Het `Map` object zal dan ook een publieke constructor hebben met als parameter de bestandsnaam van het tekstbestand dat de `Map` voorstelt:

- **`Map(const string & file_name)`**

```
1  origin 1 0
2  destination 2 5
3  dimension 6 6
4
5  obstacle thin_wall 2 4
6  obstacle thin_wall 1 4
7  obstacle thick_wall 4 2
8  obstacle thick_wall 0 4
9  obstacle thick_wall 3 4
10 obstacle trench 4 3
11 obstacle thick_wall 0 3
12 obstacle thick_wall 2 1
13 obstacle thick_wall 1 1
14 obstacle thick_wall 4 4
15 obstacle thick_wall 5 4
16 obstacle thick_wall 0 2
17 obstacle thick_wall 3 2
18 obstacle thick_wall 0 1
19 obstacle thick_wall 3 1
20 obstacle thick_wall 0 5
```

Figuur 4: Voorbeeld van een map-bestand

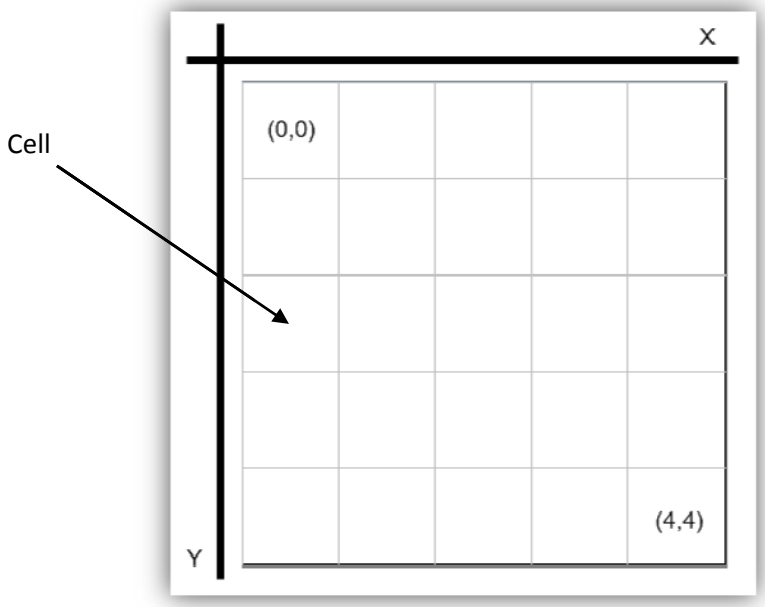
Zoals je kan zien in Figuur 4 zal een `Map` object een bepaalde dimensie hebben: de breedte en hoogte van de `Map`. De twee attributen, `origin` en `destination`, geven respectievelijk de startlocatie en de eindlocatie van de robot weer. Beide attributen zijn objecten van het type `Cell`. De `Cell` klasse stelt een enkele cel binnen de map voor. Het gebruikte coördinatensysteem is equivalent aan dat van Java Swing, waarbij de x- coördinaat toeneemt naar rechts en de y- coördinaat toeneemt naar beneden. Voor de `Cell` klasse dienen volgende methoden geïmplementeerd te worden:

- **`Cell(int x=0, int y=0)`**: de constructor van de `Cell` klasse (de x- en y-coördinaat staan default op 0)
- **`int getX(void)`**: geeft de waarde van de x-coördinaat terug
- **`int getY(void)`**: geeft de waarde van de y-coördinaat terug

- **`bool isAccessible(void)`**: `true` als de robot over deze cel kan rijden, anders wordt `false` teruggegeven

Voor de `Cell` klasse moeten er ook enkele operators geherdefinieerd worden:

- **`bool operator==(const Cell &other) const`**
- **`bool operator>(const Cell &other) const`**
- **`bool operator<(const Cell &other) const`**



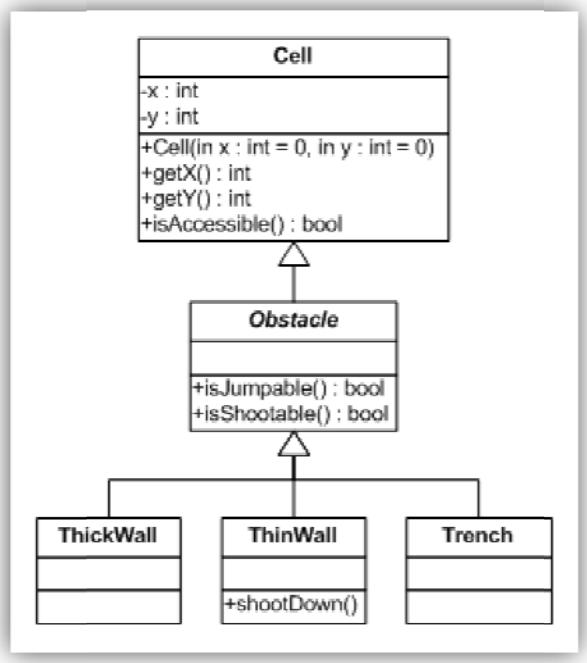
Figuur 5: Het coördinatensysteem

Uiteraard zijn niet alle cellen van de `Map` toegankelijk (`isAccessible()`). Sommige cellen worden geblokkeerd door een obstakel. Om deze speciale cellen weer te geven, gebruiken we de abstracte klasse `Obstacle`, een afgeleide klasse van `Cell`. Een `Obstacle` voegt twee nieuwe methoden toe aan de `Cell` klasse:

- **`bool isJumpable(void)`:** geeft `true` terug als de robot over dit obstakel kan springen, anders wordt `false` teruggegeven.
- **`bool isShootable(void)`:** geeft `true` terug als de robot dit obstakel kan verniettigen, anders wordt `false` teruggegeven.

Onze `Map` kan drie verschillende soorten obstakels bevatten (allen afgeleide klassen van `Obstacle`; Figuur 6 geeft de overervingstructuur weer voor de obstakels):

- `ThickWall`: Een stevige muur die niet kan vernietigd worden.
- `ThinWall`: Een minder stevige variant van bovenstaande muur die kan neergeschoten worden door de robot.
- `Trench`: Een gracht met water waar de robot over kan springen.



Figuur 6: overervingstructuur voor de obstakels

De verschillende obstakels zullen bijgehouden worden in de `Map` klasse. Hiervoor zullen we gebruik maken van de `BinarySearchTree` template klasse. De `key` die gebruikt wordt om een obstakel toe te voegen aan de `BinarySearchTree` en om een obstakel te zoeken, wordt gevormd aan de hand van de x- en y-coordinaten van het obstakel en de breedte van de `Map`:

```
key = obstacle.getY()*width + obstacle.getX().
```

Naast de vorige attributen zal een `Map` object ook een referentie bijhouden naar de robot. Hiervoor wordt de volgende methode gebruikt:

- `void setRobot(Robot* robot);`

Events en EventListeners

Normaal gezien zouden de sensoren op de robot de omgeving zelf waarnemen. Maar in deze vereenvoudigde versie zal de `Map` de robot op de hoogte brengen van veranderingen in zijn onmiddellijke omgeving. Dit gebeurt aan de hand van `Events`. De onmiddellijke omgeving van de robot moet zeer eng geïnterpreteerd worden: dit zijn enkel het vakje waar de robot zich op bevindt en het vakje waar de robot momenteel naar kijkt. In onze omgeving kunnen er twee verschillende `Events` optreden, die beiden overerven van een gemeenschappelijke abstracte klasse `Event`:

- `ObstacleEvent`: deze `Event` zal optreden als de volgende cel een obstakel bevat.

- `DestinationEvent`: wanneer de cel waarop de robot zich momenteel bevindt de eindbestemming (`destination`) is, zal dit soort `Event` optreden.

`Events` worden opgevangen door `EventListeners`. In ons programma hebben we ook twee `EventListeners`, één voor elk type `Event`:

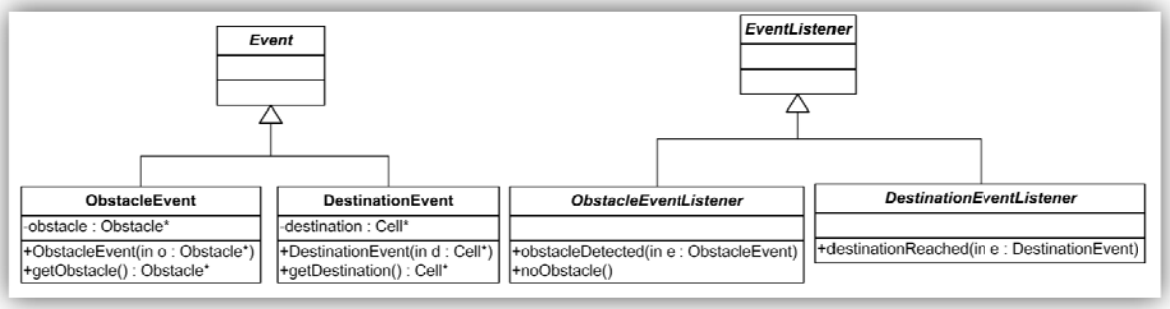
- `ObstacleEventListener`: deze `EventListener` zal op de hoogte gebracht worden wanneer er obstakel ontdekt wordt (d.m.v. een `ObstacleEvent`) en ook wanneer de volgende cel vrij is.
- `DestinationEventListener`: dit type `EventListener` zal op de hoogte gebracht worden van `DestinationEvents`.

`EventListeners` zijn steeds abstracte klassen, waarvan de objecten die geïnteresseerd zijn in dat type `Events`, zulllen overerven. M.a.w. objecten die geïnteresseerd zijn in `ObstacleEvents`, zullen overerven van de klasse `ObstacleEventListener`. Denk goed na over welk(e) object(en) op de hoogte gebracht dienen te worden van deze `Events`.

Al de `EventListeners` zullen door de `Map` bijgehouden worden in een lijst (een STL klasse), want het is de `Map` zelf die deze objecten op de hoogte zal brengen van eventuele veranderingen in de omgeving van de robot. Hierbij moet je twee belangrijke methoden implementeren:

- **`void registerListener(EventListener* listener)`**: zal de `listener` registreren bij de `Map`.
- **`void refresh(void)`**: deze methode van de `Map` klasse zal ervoor zorgen dat al de `EventListeners` op de hoogte gebracht zullen worden van eventueel nieuwe `Events`.

Al de te implementeren publieke methoden en de overervingstructuur worden getoond in Figuur 7.



Figuur 7: overervingstructuur voor het eventmodel

De bewegingen van de robot

Het enige wat nog niet beschreven is, zijn de bewegingen van de robot. Onderstaande methoden zullen de bewegingen van de robot duidelijk maken aan de omgeving – de `Map` dus. Dit moet op de `Map` gebeuren omdat enkel de `Map` op de hoogte is van alle obstakels.

- **`void move(void)`**: als de robot voorwaarts rijdt, zal deze methode er voor zorgen dat deze beweging opgenomen wordt door de `Map`. Let wel: het is niet mogelijk om over een obstakel te rijden (tenzij het neergeschoten is).
- **`void jump(void)`** : deze methode zal ervoor zorgen dat de `Map` op de hoogte is van de sprong van de robot. Let ook hier op: de robot kan niet landen op een obstakel.

c. Representatie van de robot

De `Robot` klasse zal de behaviour-gebaseerde robot voorstellen. Deze klasse heeft drie belangrijke attributen. Voor elk attribuut moet ook een bijhorende "get" en "set" methode geïmplementeerd worden:

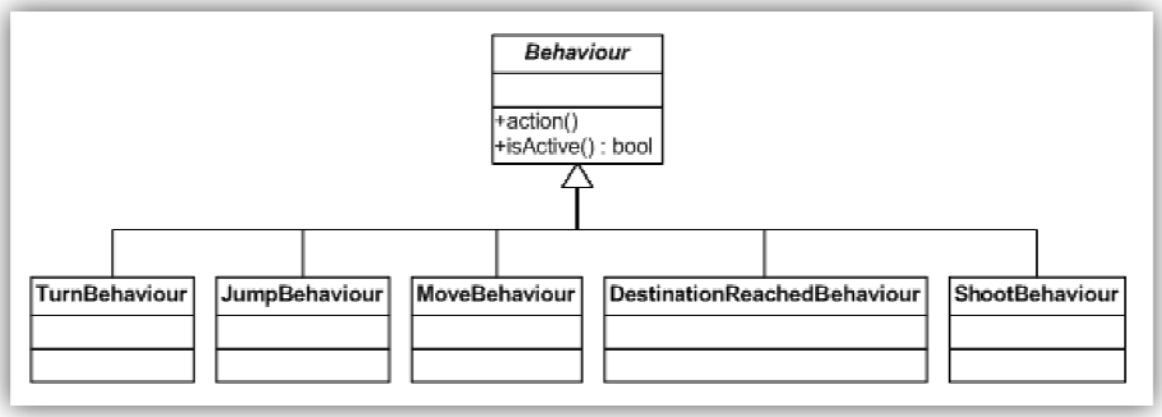
- **`bool destinationReached`**: dit attribuut zal aanduiden of de robot zijn eindbestemming heeft bereikt. Deze waarde wordt uiteraard geïnitialiseerd op `false`.
  - **`bool isDestinationReached(void)`**
  - **`void setDestinationReached(bool d)`**
- **`int speed`**: geeft de snelheid van de robot weer. De snelheid van de robot vertaalt zich in het aantal cellen hij tegelijkertijd zal kunnen vooruitgaan. De defaultwaarde is `1`, wat betekent dat de robot één cel tegelijkertijd zal vooruit bewegen.
  - **`int getSpeed(void)`**
  - **`void setSpeed(int s)`**
- **`Orientation orientation`**: dit attribuut duidt de richting van de robot aan. Hij zal steeds volgens deze richting bewegen. Om de richting eenvoudig weer te geven gebruiken we de `Orientation` enumeration (zie Figuur 8).
  - **`Orientation getOrientation(void)`**
  - **`void setOrientation(Orientation o)`**

```
1 enum orientation {
2     NORTH, EAST, SOUTH, WEST
3 };
```

Figuur 8: Orientation enumeration

We zullen de `Behaviours` van de robot op een eenvoudige manier bijhouden: in een gewone lijst (voorgesteld via een STL klasse). De plaats in deze lijst bepaalt de prioriteit van de `Behaviour`: het eerste element zal de hoogste prioriteit hebben en bijgevolg heeft het laatste element uit de lijst de laagste prioriteit. Om deze `Behaviours` te beheren zullen we twee methoden implementeren:

- **`void registerBehaviour(Behaviour* b)`**: via deze methode zullen we `Behaviour` b aan het einde van de lijst toevoegen.
- **`Behaviour* getFirstActiveBehaviour(void)`**: deze methode geeft de eerste `Behaviour` terug, die de controle wil hebben (`isActive()` == true). Van deze behaviour zal uiteindelijk de actie uitgevoerd worden.



Figuur 9: overervingsstructuur voor de verschillende Behaviours

De volgende `Behaviours` dienen geïmplementeerd te worden (de overervingstructuur wordt getoond in Figuur 9). De volgorde waarin deze geregistreerd worden bij de robot zal een belangrijke rol spelen bij het gedrag van de robot.

- `MoveBehaviour`: verplaatst de robot een aantal cellen afhankelijk van zijn oriëntatie en zijn snelheid.
- `TurnBehaviour`: draait de robot een kwartslag met de klok mee. De robot zal moeten draaien wanneer hij bij een obstakel komt dat hij niet kan stukschieten of waar hij niet over kan springen.
- `JumpBehaviour`: laat de robot een sprong uitvoeren over het obstakel recht voor zich. Let op, de robot zal niet over elk obstakel kunnen springen.
- `ShootBehaviour`: laat de robot een obstakel vernietigen. Let ook hier op dat niet alles kan stukgeschoten worden.
- `DestinationBehaviour`: dit gedrag zal ervoor zorgen dat bij het bereiken van de eindbestemming, het attribuut `destinationReached` op `true` staat.

Objecten die voor bepaalde `Behaviours` van belang zijn kan je best meegeven via de diverse constructoren.

Vergeet zeker niet aan de hand van de bijgeleverde klasse `GUI`, de `GUIServer` te updaten. Let wel op, dit zal in bepaalde gevallen door de `Behaviours` (in de `action` methode) moeten gebeuren en in andere gevallen door de `Map` (in de methodes `move`, `jump`).

We vragen jullie ook om nog twee extra methoden te implementeren:

- **`void addNextMove(Cell cell)`**: het afgelegde pad van de robot wordt bijgehouden in een lijst en via deze methode wordt de volledige verplaatsing toegevoegd aan deze lijst. Let wel, als de snelheid van de robot groter is dan 1, wordt enkel de laatste cel van deze beweging bijgehouden. Dus per beweging wordt maar één `cell`, namelijk de plaats waar de robot zich na de beweging bevindt, toegevoegd aan het pad van de robot.
- **`Cell* getCurrentPosition(void)`**: deze methode zal de huidige locatie van de robot teruggeven.

De intelligentie van de robot zal afhangen van de gedefinieerde `Behaviours` en hun prioriteit. Het is voldoende als de bijgevoegde mappen 1 t.e.m. 3 kunnen opgelost worden door een implementatie van de basis `Behaviours` (`Move`, `Turn`, `Jump`, `Shoot`, `Destination`). Let wel, er mogen geen extra publieke methoden toegevoegd worden om deze mappen op te lossen.

Als je van een uitdaging houdt en als je 1 of 2 punten extra wil scoren, dan kan je proberen extra `Behaviours` toe te voegen zodat map 4, 5 en 6 ook oplosbaar worden. Hierbij is het mogelijk dat er extra publieke methoden nodig zijn. Dit is toegelaten, maar enkel de nieuwe `Behaviours` mogen gebruik maken van deze extra publieke methoden. Vermeld deze extra publieke methoden en hun nut in het verslag.

Er moet echter nog een kanttekening gemaakt worden bij de nieuwe `Behaviours`: blijf getrouw aan de basis architectuur; d.w.z. behoud de scheiding tussen de omgeving (`Map`) en de `Robot`!

Hints en opmerkingen

- Respecteer de signatuur van de opgave. Het aanpassen van de bestaande inhoud van de header-bestanden is NIET toegestaan, evenals het aanpassen van de publieke interface van de klassen (bijvoorbeeld door het toevoegen van publieke methodes). Voeg de benodigde attributen en methodes toe in de private scope.
- De GUIServer gaat ervan uit dat de oriëntatie van de robot bij aanvang altijd NORTH is. **Hou hier dus rekening mee in je programma!**
- Begin niet onmiddellijk te programmeren. Lees eerst aandachtig de opgave en denk na hoe je de verschillende problemen zou oplossen (vb. welke klassen ga je gebruiken, wat houden ze bij, …). Teken indien dat helpt een blauwdruk van je klassenschema uit, en begin pas te programmeren nadat je een goed overzicht hebt!
- Maak gebruik van de Standard Template Library (STL) waar je dat nuttig acht (uitgezonderd de gevallen waarvoor je een `BinarySearchTree` kan gebruiken). Volgende link kan daarbij van pas komen: http://www.cppreference.com/wiki/stl/start
- Alle opmerkingen over de practica gelden ook hier: wees zuinig met geheugen, let op voor dangling pointers, geef alles wat gealloceerd wordt ook weer vrij, controleer of je

open bestanden wel correct geopend zijn en gesloten worden, bescherm je header files…

- Voor parsing kan je handig gebruik maken van stringstreams, zie ook http://www.cppreference.com/cppsstream/index.html. Alternatief kan je de strtok methode uit de (C) string library gebruiken.
- Als referentiecompiler wordt Microsoft Visual C++ Express Edition gebruikt. Zorg er dus voor dat je project daarmee compileert en uitvoerbaar is!
- Probeer zoveel mogelijk compiler warnings te vermijden; deze duiden meestal op fouten die Visual Studio C++ EE voor jou zal oplossen, maar die je evengoed kan vermijden door je code aan te passen.
- Vanzelfsprekend zijn er zaken waar deze opgave je vrij in laat; deze mag je zelf kiezen naar eigen inzicht. Verduidelijk in alle geval je broncode met commentaar waar dat nuttig is!

Indienen

- Het project wordt gemaakt in groepen van 2 studenten. Het mag eventueel ook alleen gemaakt worden. Het is verplicht hiervoor een groep op minerva te kiezen.
- Schrijf een kort verslag (max. 2 bladzijden) met daarin:
    - De naam, voornaam en richting van de groepsleden + groepsnummer van minerva
    - Een (informele) figuur waarin je de samenwerking van je klassen toont (hoeft geen UML te zijn, zolang het maar duidelijk is)
    - Een extra woordje uitleg bij je ontwerpbeslissingen (max. ½ bladzijde)
    - De taakverdeling: wie heeft wat gedaan?
- Zip je broncode bestanden (header files en implementatiebestanden) en je verslag in een bestand naam_voornaam_project_nr.zip. De naam en voornaam zijn die van degene die het project indient. Het nummer is je groepsnummer van op minerva.
- Gebruik een standaard zip formaat, geen rar of 7zip.
- Zet in je bronbestanden bovenaan in commentaar de namen van de groepsleden en de naam van het bestand + groepsnummer.
- Stuur je oplossing door middel van de dropbox op minerva door naar **Frederic Iterbeke**, ten laatste op **maandag 14 december**, **11u59 pm**.

Veel succes!