Programmeren2 Huiswerk

Assessment

REX: Room EXplorer

Zo... lang naar uitgekeken natuurlijk en hier ligt het dan voor je: het eindassessment. Voor dit assessment heb je 1 lesweek de tijd en maak je alleen. De beoordeling vindt plaats door een eindgesprek met een docent, dus je moet heel goed weten wat je code doet, waarom en hoe je iets snel aan kan passen. Heel veel succes en plezier!

Spelregels

Wij gaan ervan uit dat je alles weet en kan (gebruiken) wat wij hebben behandeld in het boek, de sheets, de practicum- en de huiswerkopdrachten van PROG1 en PROG2.
Je mag best overleggen met een medestudent, maar weet wel dat jij aan het eind verstand moet hebben van de code die je meeneemt naar de beoordeling. Als de docent het idee heeft dat dit niet het geval is, krijg je een onvoldoende, ook al heb je een fantastisch programma!
Tenslotte moet je je beseffen dat het niet de bedoeling is dat een docent (een gedeelte van) jouw werk gaat maken. Je kan dus best algemene vragen stellen, maar geen zaken die rechtstreeks gebonden zijn aan (de uitvoer van) deze opdracht.

Opdracht

In het kort: Maak een “Dungeon Crawler” game, waarbij een speler door middel van commando’s kamers kan ontdekken, gevuld met voorwerpen. Deze voorwerpen kan de speler oppakken, gebruiken, weggooien of in zijn rugzak doen (om bijvoorbeeld ergens anders te gebruiken). Natuurlijk moet de speler ook een einddoel hebben, met andere woorden: je moet het spel kunnen uitspelen. Het einddoel van deze game moet zijn dat je een (magisch?) voorwerp uit ‘het doolhof’ terug naar de ingang brengt.

Een aantal duidelijke regels als toevoeging op bovenstaande:

- Bij aanvang van de game zijn alle kamers, behalve de startkamer, nog verborgen.
- De speler beschikt over een rugzak waar voorwerpen ingestopt en uitgehaald kunnen worden.
- De speler moet voldoende informatie over de huidige kamer (waar hij zich nu bevindt) krijgen om te weten wat hij kan doen: eventuele uitgangen, eventuele voorwerpen, etc.
- De speler mag nooit vast komen te zitten. Er moet altijd een uitweg zijn.
- Het spel moet aan alle eisen voor een 7 voldoen die zijn beschreven bij “Eisen”.

Google “dungeon crawler”

Eisen

We stellen een aantal eisen aan jouw applicatie. Die kan je verdelen in functionele eisen (wat moet de speler straks kunnen doen?) en overige eisen (hoe moet de game geprogrammeerd zijn?). Hier onder zie je aan welke eisen jouw werk moet voldoen om maximaal een 7 te kunnen halen. Besef je goed dat het dus niet zo is dat je automatisch een 7 krijgt als je alles hier onder hebt gedaan. Het gaat namelijk om de mate waarin je aan de eisen voldoet:

01	De code is object georienteerd (volgens PROG1 en PROG2)	7
02	De code is netjes gestructureerd en voorzien van commentaar	
03	De structuur van jouw code komt overeen met het gegeven DKD (pag. 3)	
04	Het spel bevat minstens 10 kamers	
05	Het spel bevat minstens 5 voorwerpen	
06	Het spel bevat 1 speler	
07	De speler beschikt over een rugzak	
08	De speler kan zich verplaatsen door kamers dmv de opdracht **go** in de console (Voorbeeld: "go west" verplaatst de speler naar het westen.)	
09	De speler kan voorwerpen uit een kamer oppakken dmv de opdracht **get** in de console (Voorbeeld: "get stick" plaatst een stok in de rugzak.)	
10	De speler kan voorwerpen uit zijn rugzak weggooien dmv de opdracht **drop** in de console (Vb: "drop stick" haalt de stok uit de rugzak.)	
11	De speler kan voorwerpen in de huidige kamer of uit zijn rugzak gebruiken dmv de opdracht **use** in de console (Voorbeeld: "use stick" zorgt dat er iets gebeurt met de stok.)	
12	De speler krijgt dmv de opdracht **pack** een lijst met voorwerpen uit de rugzak in de console te zien	
13	De speler krijgt dmv de opdracht **help** een lijst met mogelijke opdrachten (zoals "get", enz.) in de console te zien	
14	De speler krijgt dmv de opdracht **look** een lijst in de console te zien van voorwerpen en uitgangen in de huidige kamer	
15	Kamers hebben een wisselend aantal uitgangen (Dus niet alleen maar kamers met 4 deuren bijvoorbeeld.)	
16	De speler krijgt bij binnenkomst informatie over de huidige kamer. Deze leest hij in de console en bevat info zoals een korte omschrijving en eventuele uitgangen	
17	Het spel kan je netjes afsluiten met de **quit** opdracht	
18	Het spel crashed niet en is normaal speelbaar via de console	
	Als je voor een hoger punt gaat, probeer dan (OO) het volgende extra:	
19	Implementeer een Enemy klasse en zorg ervoor dat in sommige kamers vijanden staan.	EXTRA
20	Als je een Enemy tegen komt moet je er tegen kunnen vechten. Dat mag dmv een vechtopdracht (voorbeeld: "fight ...") of door de bestaande gebruikopdracht (voorbeeld: "use ...")	
21	Als je een Enemy hebt verslagen moet je hem kunnen "looten" (voorbeeld: "loot goblin"). Hierdoor krijg je (misschien) extra voorwerpen in de kamer erbij.	
22	Er moet minstens 1 kamer (of exit?) zijn die de speler teleporteert naar een willekeurig andere kamer.	
xx	Voor een 10 zal je zelf nog meer moeten toevoegen. Denk aan een Grafische User Interface (GUI), geluid, vijanden die de speler achtervolgen, etc. Maar onthoud: bovenstaande (1 t/m 22) moet er MINSTENS in zitten en alles wat je maakt moet OO zijn en netjes.	

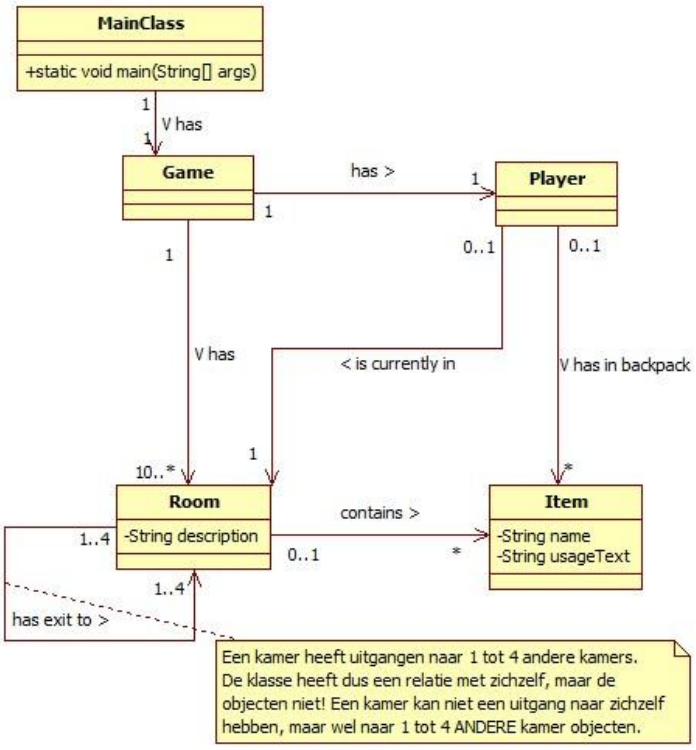
En dan nog dit

Op de dag van het assessment is weinig tijd. Zorg er voor dat je op tijd je werk hebt gesubmit in Blackboard, dat je je goed hebt voorbereid en dat het programma zonder crashes runt.

Extra hulp!

Domein Klasse Diagram

Hier onder zie je het klassediagram van REX, uitgaande van een 7. Als je voor de 7 gaat (en dat zou ten minste jouw uitgangspunt moeten zijn) dan moet de stuctuur van jouw programma er als volgt uit zien:



Bij sommige klassen zijn wat instance variables (middelste vak) of methods (onderste vak) toegevoegd. Deze moet jij minstens ook in jouw programma hebben. Daarnaast komt er natuurlijk nog wel iets bij, maar dat moet je zelf bedenken.

We doen er één voor: Kijk naar de relatie (lijntje) tussen Game en Player. Die geeft aan dat in de Game klasse gebruik wordt gemaakt van de Player klasse (want daar wijst de pijl naar toe). Je ziet ook dat de relatie 1 op 1 is: Eén Game object heeft dus precies één Player object. Dit kan je al programmeren:

```
Class Game {
  private Player m_Player;
  ...
  // Constructor van Game:
  public Game() {
    // misschien heeft jouw Player
    // constructor wel parameters…
    m_Player = new Player();
    ...
  }
  ...
}
```

Dus de klasse waar de pijl naar toe wijst, wordt gebruikt in de klasse waar de pijl vandaan komt. In de Game klasse programmeer je een Player object (zie voorbeeld), in de Player klasse programmeer je een verzameling (ArrayList? Array?) van Item objecten, enz. enz.

Room is nog bijzonder, die heeft een relatie met zichzelf. Dit betekent dus dat je in de Room klasse een verzameling moet programmeren met weer andere Room objecten. Dat zijn natuurlijk de deuren, je moet immers van de ene kamer naar de andere kunnen lopen. Omdat je bij de eisen (pag. 2) al hebt gelezen dat dit met een tweede commando kan (“go west”) moet je je twee dingen beseffen:

- Ik moet de tekst “west”, “north”, etc. kunnen *mappen* naar een uitgang. Dan is het dus verstandig om hiervoor een HashMap<String, Room> te gebruiken met de naam “exits” of iets dergelijks. Die kan kan bijvoorbeeld de string “east” koppelen aan het Room object ten oosten van dit huidige Room object.
- Je moet de ingevoerde tekst “go west” kunnen ophakken in “go” en “west”, omdat je die commando’s los moet kunnen afhandelen (eerst “go” en vervolgens kijk je welke kant dan). Gebruik hiervoor je kennis over String Manipulation uit de eerste paar weken van PROG2.

Code Snippets!

Omdat we beseffen dat één week niet al te veel tijd is, geven we als laatste nog twee steuntjes in de rug.

String Switch

In het programma kom je best veel Strings tegen, zoals bijvoorbeeld commando's en namen van Items. Het valt te verwachten dat je nog wel eens zal moeten kijken welk commando je nu uitvoert, of dat je iets wil doen afhankelijk van het Item dat je gebruikt. In plaats van een heleboel if-statements, kan je beter een switch statement gebruiken. Omdat we tijdens het practicum niet heel veel aandacht hieraan besteed hebben, nog even de syntax.

```
String s = "Jasper";

switch(s)
{
case "Jasper":
  <insert your Jasper code here>;
  break;
case "Bob":
  <insert your Bob code here>;
  break;
default:
  <insert your default code here, in
   case s is neither Jasper nor Bob.>;
  break;
}
```

Game Command Flow

Hoe je omgaat met commando's is best complex. Als laatste hulp bij deze de (geavanaceerde) pseudo-code voor een deel van de meest complexe class van de applicatie, namelijk de Game class. We geven je enkel de pseudo-code voor het omgaan met de commando's. Het zou dus goed kunnen zijn dat je in deze pseudo-code of aan de class als geheel nog dingen moet toevoegen.

```
public class Game {
  // Get yourself all the instance variables you need.

  /**
   * @param none
   *
   *
   */
  public Game() {
    // Initialize everything you need:
    // the player, the rooms, items you want in the rooms, everything
    // then call the run() command.
  }

  /**
   * @param none
   *
   *            Run the game
   */
  private void run() {
    try {
      // As long as the command isn't to quit:
      // get the next input line and handle it. (With handleCommand.)
    } catch (Exception e) {
      // Something went terribly wrong. Inform the user.
    }
  }
```

```
/**
   * @param userInput (This is the entire input string from the user.)
   *
   *            (Tell others to) Perform the task which belongs to the given
   *            command.
   */
  private void handleCommand(String userInput) {
    // Split the user input string.
    // The first word is a command. The rest is extra information

    // Check if the command is to travel between rooms. If so, handle
    // the room travelling using the method: checkRoomTravel
    // This one is explained later.

    // If there isn't any room travel, then check all other command
    // command options. (Oh, look: this might be a great place for
    // a switch over the command string.)
    // Depending on the command, you might also need the extra information.
    // e.g. "use stick", has "use" as command and "stick" as extra
    // information.
    // To make things easy, we created private methods to handle the commands.
    // They are presented below.
  }

  private void handleUseCommand(String itemName) {
    // Check if the player has the item in his backpack.
    // If in the backpack: use it.
    // If not in the backpack: check if the item is in the room.
    // If the item is in the room: use it.
    // If no item with that name is present: tell the user he's
    // trying to use something that isn't there.
  }

  private void handleDropCommand(String itemName) {
    // Check if the item is in the backpack.
    // If so: remove the item from the backpack and put it
    // in the room.
    // If not: tell the player he can't drop that.
  }

  // Please note that this method looks A LOT like
  // Game.handleDropCommand, only the other way arround!
  private void handleGetCommand(String itemName) {
    // Check if the item is in the room.
    // If so: remove the item from the room and put it
    // in the backpack.
    // If not: tell the player he can't get that.
  }

  /**
   * @param command
   *
   *            Check if the command can take us to another room. If so: do
   *            it! Let the caller know if we actually traveled.
   */
  private boolean checkRoomTravel(String command) {
    // Get the currentRoom from the player and check if this room
    // has an exit in the direction of command. (East, south, north, west.)
    // If there is an exit in that direction, ask the currentRoom to get that
    // that room.
    // Tell the player to travel to the destination room.
    // If there is no exit in that direction, tell the player.
    // If travel was successful, return true. If not, return false.
  }
}
```