Exam 2012: Dungeon Crawl

Object-Oriented Programming and Design B2-2011
Department of Computer Science at the University of Copenhagen (DIKU)

Deadline: 2012-01-20, 23:55 CET

1 Preface

1.1 Exercises & Time

The exam consists solely of coding exercises. You will be handing in a hierarchy of Java source files. The exam is designed to take roughly 20 hours to solve. We've provided some rough estimates for the major sections of the assignment.

1.2 What we expect of your code

We expect of you to follow the rough code conventions outlined in "Refactoring in a nutshell" under the folder "Undervisningsmateriale" in Absalon. Wherever this convention is thin, we refer you to the Code Conventions for the Java Programming Language[1]. Deviation from the conventions does not necessarily lead to a reduced grade, but the conventions are in place to ensure that we can concentrate on grading your code rather than interpreting it.

Additionally, make sure to follow the explicit code specifications in the assignment text as your code might be subject to automated tests. Such tests usually rely on various classes, fields, methods, etc. to be named in the way specified by the assignment text. Last but not least, make sure to structure your code in an intuitive way, and document your source code and interfaces wherever appropriate.

1.3 Deadline

The submission deadline for the exam assignment is **Friday the 20th of January, 2012, 23:55 CET**. Re-submissions are *not* possible, and the deadline *cannot* be extended. In the unfortunate case that the KU systems are down, send your solution directly to `kasper.steenstrup@gmail.com`. The aforementioned deadline holds regardless.

1.4 Submission

We would like to ensure that we can identify every file that you hand in as yours. We therefore ask you to place your alumni id at the top of *every* file that you hand in. You use your alumni id to log into KU Net. For instance, if your alumni id is abc123, we would like to see *every* file starting with the line:

```
// abc123
```

You hand in your solution in digital form, under the"Exam" folder in Absalon. You should hand in a single zip archive. The actual structure of the desired archive is explicitly stated at the end of the assignment text. The name of the archive should be your alumni id, i.e. `abc123.zip`.

Please make sure that you hand in your *source code* and *not* java bytecode. It is always a good idea to check that all of your files have been uploaded correctly by downloading them again and browsing them yourself.

[1] `http://www.oracle.com/technetwork/java/codeconv-138413.html`

1.5 Collaboration and help

Unlike during the course, you are *not* allowed to discuss your solutions with your peers. However, you *are* allowed to discuss the assignment text and seek clarifications from your peers and the instructors. You can ask your questions on the discussion forum on Absalon, or catch an instructor on duty in the DIKU canteen. An overview of when each instructor will be on duty in the canteen will be made available under the "Exam" folder in Absalon before the exam starts.

The instructors are free to moderate the Absalon forum to ensure that the questions you ask are not too specific to the assignment. Indeed, the nature of the assignment allows for certain questions to be so specific that they're not specific to the assignment at all. For instance, the question "How do I draw a line with a width of 20 pixels in Swing?", is a very specific question, but not specific enough to be deemed inappropriate since no exercise in the exam assignment explicitly asks you to do so. The instructors will also be monitoring other popular internet fora to ensure that you're not cheating.

Last but not least, we expect you not to share explicit code, nor to take credit for other peoples' work. All in all, we expect you to honor each other's, and your own work.

1.6 Feedback

There is officially *no feedback*, other than your final grade. If you desire further feedback, you are advised to contact the course responsible, Knud Henriksen, first.

1.7 Handed out files

```
./oopd-exam-2012.pdf
./dungeon.map
./dungeon-crawler.jar
```

2 Assignment

Your exam assignment is to create a two-dimensional dungeon crawler game. A dungeon crawler is a fantasy game scenario, where a hero navigates a labyrinth, battling various monsters, and looting any treasures he may find. There is no handed out code, so the structure is entirely up to you. However, if you lack inspiration, we suggest you seek it among the weekly assignments you've completed throughout the course.

The assignment can be said to consist of five parts:

1. Create a model for the game.
2. Create the GUI elements for the game.
3. Implement game play.
4. Couple everything together.
5. Implement one of the suggested extensions.

We recommend reading the entire assignment text first, so as to get an idea of what you should end up with. What's more we recommend running the runnable jar file `dungeon-crawler.jar`, handed out alongside the assignment text, to get a look and feel of what it is that we would like for you to end up with.

E.1 Create a model for the game (ca. 6 hours)

A dungeon is a series of connected rooms. All rooms are rectangular and rooms are connected by doors. There is a path through the dungeon from every room to every other room, and no doors lead out of the dungeon. For now, you don't have to worry about how dungeons are generated, rather, we would like you to start by modelling the dungeon itself.

This is a two-dimensional game, and hence, any room can be represented as a point pair, where one is the top-left corner and the other is the bottom-right corner. Furthermore, since rooms are rectangular and connected, the dungeon itself can be covered by a rectangle in two-dimensional space, encompassing all the rooms of the dungeon. Therefore, we let the top-left corner of the dungeon be the point of origin.

For instance, given a dungeon of size 200 × 200, $(0,0)$ is the top-left corner and $(199,199)$ is the bottom-right corner. We illustrate a scaled down dungeon in Figure 1 (page 3).

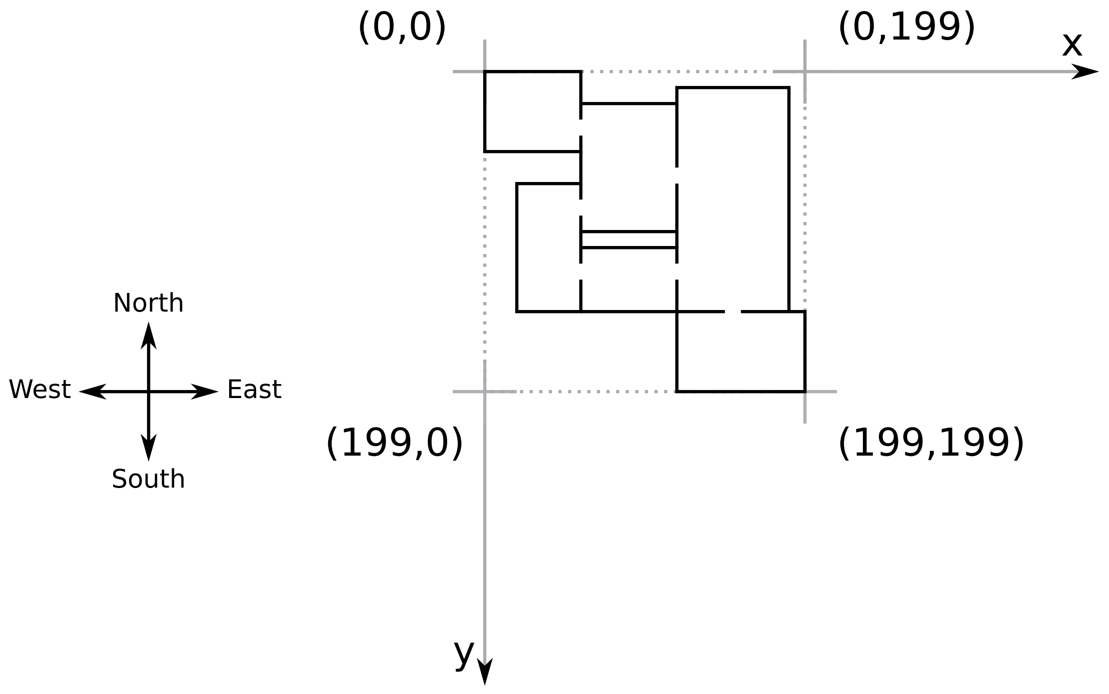


Figure 1: The layout of a dungeon of size 200 × 200, along with an axis and compass of the dungeon.

E.1.1 Points

1. Define a class `Point` to represent a point in two-dimensional space.

Note: You can assume for the dungeon to always be at most 300 points in both width and height.

E.1.2 Players

There are two types of players in the game, heroes and monsters. Any instance of the game has exactly one hero and a finite set of monsters. The two have some properties in common. In particular,

(a) All players be represented as points in the plane. This means that every player has a position in the plane and a width and height of exactly one point.

(b) Players can hit, and eventually kill each other. Hence, every player has some level of health, and can incur some level of damage on another player. The amount of damage a player can incur, depends on the state of the player instance, and will be discussed in further exercises. A player dies when he runs out of health. Players take turns hitting each other, so no two players can die simultaneously.

(c) In each turn, a player mildly heals himself, a concept we'll refer to as regeneration. The amount of regeneration per turn is constant throughout the game.

(d) All players initially have health level 100.

(e) All players can only move either north, south, east or west, and only one position at a time.

1. Define an abstract class `Player` to represent a player in the game. It should at least have the following methods:

```java
/**
 * @true if player's health <= 0.
 * @false if player's health > 0.
 */
public boolean isDead() { }

/**
 * Performs an amount of regeneration for a single turn.
 * A player can at most regenerate to health level 100.
 * A dead player cannot regenerate.
 */
public void regenerate() { }

/**
 * @ensure return value >= 0.
 * @return the level of damage that the player can incur on another player.
 */
public abstract int getDamageLevel() { }

/**
 * Damages the player with the given amount.
 */
public abstract void takeDamage(int amount);
```

Note: `getDamageLevel` and `takeDamage` are abstract methods. This is because we will allow for heroes to carry weapons and armor.

E.1.3 Items

Any instance of the game has a hero, and that hero can pick up different items on his adventure. Items are scattered throughout the dungeon, and alike players, have a position, as well as a width and hight of at exactly one point. A hero picks up an item by stepping on it. There are two kinds of items he can pick up, armor and weapons. Armor has has varying resistance and weapons have varying damage.

1. Define a class `Weapon`, which should have a field `damage`, for the amount of damage a weapon causes to a hero.

2. Define a class `Armor`, which should have a field `resistance`, for the amount of resistance the piece of armor yields for a hero.

E.1.4 Heroes

The hero is our story's protagonist, he is a solitary champion seeking fame and glory in the deepest and darkest dungeons. A hero is a player that can pick up armor and weapons. A hero initially has no armor and has only his bare hands to fight with.

The armor the hero picks up has the effect that whenever hit, he does not lose health unless he's out of armor. In particular, a hit decreases armor until it is exhausted, and only then is the health of the hero be decreased. For instance, the player starts with health level 100, and armor level 0. He picks up armor with defence level 20. He is hit with damage amount 40. The net result is that the hero is deprived of his armor, and health level is decreased to 80. The armor level is simply the sum of the resistances of all the armor items that have been looted.

At any given point in the game, the hero carries exactly one weapon. Whenever the hero picks up another weapon, he uses a greedy strategy, and keeps the weapon with the most damage. What happens to the old weapon when it is picked up is undefined.

A weapon has some initial damage expressed in terms of a positive integer. The hero has a *damage magnifier*, as in, a level of skill at using weapons in general. This is also expressed in terms of a positive integer. The amount of damage level of the hero is hence damage of the weapon *multiplied* by his damage magnifier. The damage magnifier for any given hero stays constant throughout the game.

The game will have multiple types of heroes, all with varying damage magnitudes and regeneration levels.

1. Define an abstract class `Hero` with initial armor level 0, and his bear hands as his weapon, which has damage level 5. The class should retain the following properties:

   (a) The name of the hero.

   (b) The weapon the hero is carrying.

2. Define the following subclasses of `Hero`, to represent hero types:

   (a) A class `Warrior` that has regeneration 5 and a damage magnitude 3.

   (b) A class `Cleric` that has regeneration 20 and damage magnitude 1.

   (c) A class `Mage` that has regeneration 10 and damage magnitude 2.

E.1.5 Monsters

The monsters are the enemies in the game. They are able to move autonomously and attack the hero. Monsters, unlike heroes, cannot pickup armor or weapons, hence they always lose health whenever hit, and they retain some exact level of damage throughout the game. We would like for multiple types of monsters, in particular, orcs and goblins.

1. Define an abstract class `Monster`, and its two subclasses:

   (a) A class `Goblin` that has regeneration level 2 and damage level 5.

   (b) A class `Orc` that has regeneration level 10 and damage level 20.

E.1.6 Rooms

The dungeon consists of a series of rooms connected by doors. The monsters, as well as the hero, can move freely around rooms and through doors, but can't walk through walls.

1. Define the class `Room`, to represent the rooms of the dungeon. Remember, that a room can be represented with two points indicating the top-left and bottom-right corners of the room, respectively. The class should retain the following properties:

   (a) A collection of doors in the room. A door should map its position in the room to the room it leads to. A door is always located on one of the walls of the room, but never in a corner. A door is exactly wide enough to let a player pass through, i.e. it is one point in width and height. Doors do not overlap.

   (b) A collection of the monsters in the room.

(c) A collection of items in the room.

Hint: Should you choose to use `java.util.HashMap` for either of these collections, e.g. doors could be represented by an `HashMap<Point, Room>`, keep in mind that you must override the `public int hashCode()` method of the class that you're using as the key in the hash map, i.e. `Point`. In overriding that method it may prove beneficial to know that a dungeon is at most 300 points in both width and height.

Should you choose any other collection, e.g. `ArrayList<Door>` you should keep in mind that it probably makes heavy use of the `public boolean equals(Object other)` method for the collected class, i.e. `Door`.

In general, however, you should always override the `equals` method whenever you override the `hashCode` method and vice versa.

2. Implement the appropriate accessor and mutator methods to the data in the collections above.

3. Implement the following methods:

```java
/**
 *  Removes the given player from the room, if present.
 */
private void removePlayer(Player player) { }

/**
 * @return true if a given position is inside the bounds of the room.
 */
public boolean isInside(Point point) { }
```

E.1.7 Dungeon

1. Define the class `Dungeon` to represent the entire map of the game. The class should retain the following fields:

   (a) A collection of the rooms in the dungeon.

   (b) A textual description of the dungeon.

   (c) The hero in the dungeon.

   (d) The point scale factor.

   Note: When we draw the dungeon on screen, it's not particularly playable if every point is represented by exactly one pixel. We therefore make use of a point scale factor defined in terms of an integer strictly greater than 0.

E.1.8 Loading the board from disk

We would like to load the setup of the dungeon from the file `~/dungeon.map`, where ~ is the current user's home directory[2]. We've provided a sample `dungeon.map` alongside the assignment text.

The file is specified in ASCII and has the following format:

```
<scale>
room: <x1> <y1> <x2> <y2>
door: <x1> <y1> <x2> <y2>
monster: <x> <y> <type>
item: <x> <y> <type> <argument>
```

[2] You can get the path to the current user's home directory in Java using `System.getProperty("user.home")`.

That is, the file starts by specifying the scale factor for the dungeon, followed by a number of rooms, doors, monsters and items, in that order. A bit more about each one of these:

- Rooms are specified in terms of their top-left corner, i.e. `<x1>` and `<y1>`, and their bottom-right corner, i.e. `<x2>` and `<y2>`.
- Doors are specified in terms of two adjacent points, each of which can be assumed to lie in different rooms. It is intentionally up to you to figure out which.
- Monsters are specified in terms of their position and type, which is either 'O' for Orc, or 'G' for Goblin.
- Items are specified in terms of their position, type and a certain argument. The type is either 'A' for Armor, or 'W' for Weapon. The argument is an integral number in the range $[1, 100]$, specifying either the resistance or damage of the armor or weapon, respectively.

You can assume the configuration file to always be valid. That is, you can assume that all doors refer to existing rooms, rooms do not overlap, etc. However, your program should support a variable number of spaces between the variables on a single line. For instance, the following rooms are equivalent:

```
room: 0 0 4 5
room:    0  0    4   5
```

It is intentional that the initial position of the player is not specified in the configuration file. This is because the player should spawn at a random position, albeit within some room of the dungeon and not in a position already occupied by a monster.

1. Implement a class `DungeonParser` that loads a dungeon configuration file from `~/dungeon.map` and constructs a corresponding instance of `Dungeon`.

   Hint: It can be advised to use the `java.util.Scanner`, which you can initialize with a `java.io.File`. That way, you can read the file in much the same way you've read from the command line in the weekly assignments throughout the course.

E.2 Create the GUI elements for the game (ca. 5 hours)

A dungeon crawler without some sort of a graphical user interface is not particularly fun to play. Therefore, we make a GUI for it. The GUI is separated into two parts. The dungeon and the player creation. As already mentioned, we've provided a compiled dungeon crawler game as the runnable jar file `dungeon-crawler.jar`. We suggest trying this out to get a look and feel of what sort of interface it is that we would like for you to build.

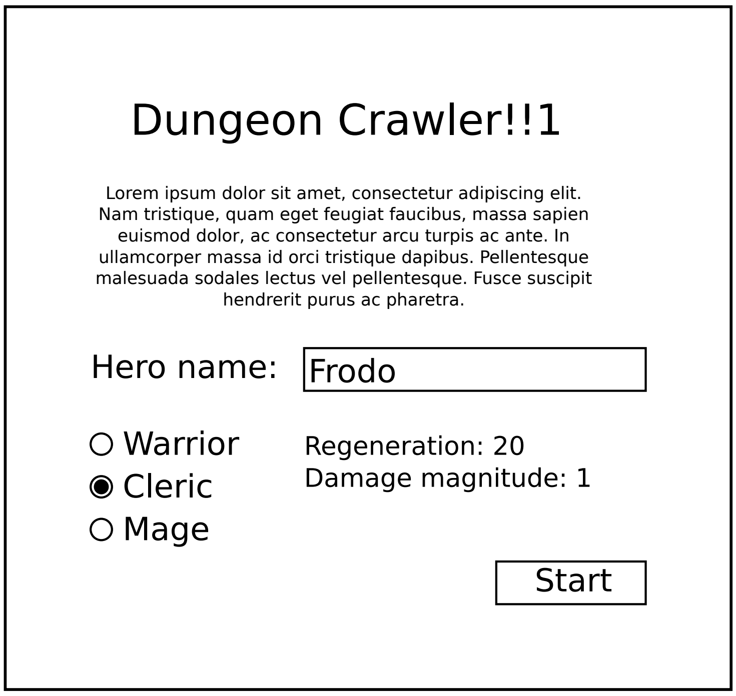
E.2.1 The main frame

A `JFrame` in Swing jargon is equitable to a window in your desktop environment. We want for the game to have a single main window, which you may call `MainFrame`. The content of the frame will be two-fold, the welcome screen and the game itself. These two types of content can be represented by different `JPanel`'s, which we'll define further down.

1. Define the class `MainFrame`. Let it extend `JFrame`, and have a `main` method that renders the frame visible when the program starts.

E.2.2 The welcome panel

A `JPanel` in Swing jargon is a rectangular area that contains other components. We provide a wireframe[3] in Figure 2 (page 8) showing you what sort of functionality we would like in the welcome panel. The welcome screen should introduce the game and provide the user with the ability to pick a type of hero to play with, name him, and start the game. You are welcome to extend this interface if you should find it appropriate.


Dungeon Crawler!!1

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam tristique, quam eget feugiat faucibus, massa sapien euismod dolor, ac consectetur arcu turpis ac ante. In ullamcorper massa id orci tristique dapibus. Pellentesque malesuada sodales lectus vel pellentesque. Fusce suscipit hendrerit purus ac pharetra.

Hero name: Frodo

○ Warrior
● Cleric
○ Mage

Regeneration: 20
Damage magnitude: 1

Figure 2: A wireframe of the desired welcome panel. The text starting with "Lorem ipsum" should provide a description of the game and game play.

Hint: You can seek some inspiration for the sort of components that you need at `http://docs.oracle.com/javase/tutorial/ui/features/components.html`.

1. Define the class `WelcomePanel`.
2. Let `MainFrame` initialize with the `WelcomePanel`.

E.2.3 The Dungeon

We provide no wireframe for the dungeon panel, as Figure 1 (page 3) and the handed out compiled jar file should be specific enough.

1. Define the class `DungeonPanel` that constructs a panel as desired, given an instance of `Dungeon` as constructor parameter.

   Hint: You can perform custom painting of the panel and its components, in particular rooms, monsters, items, etc. For this purpose we recommend overriding the `public void paintComponent (Graphics graphics)` of the `DungeonPanel` class. This method is called every time the panel is "painted" on the screen, i.e. whenever the view has to be changed due to a change in the model, or if the window is resized, moved, etc.

[3] A rough sketch of the desired interface.

The `java.awt.Graphics` class provides various methods for drawing elements within the `graphics` instance. We recommend browsing the class documentation at `http://docs.oracle.com/javase/6/docs/api/java/awt/Graphics.html`. Furthermore, if you lack an introduction to custom painting we suggest `http://docs.oracle.com/javase/tutorial/uiswing/painting/index.html`.

2. Implement switching of the `MainFrame` from the `WelcomePanel` to the `GamePanel`.
3. When the user presses "Start", the `MainFrame` should switch from showing the `WelcomePanel` to showing the `GamePanel`, i.e. the game should start. Implement this transition.
4. Whenever the user switches to the game `GamePanel`, the `MainFrame` should add action listeners to the arrow keys on the user's keyboard, so that the user can use the arrow keys to navigate the hero through the dungeon.

E.3 Implement game play (ca. 3 hours)

The game play is where we make the hero and the monster move around the dungeon, and fight each other. We start by implementing the monsters, fighting, and movement.

All players take turns to make a move. A hero is the first one to make a move, followed by all the monsters, in some particular order. A hero makes a move when the user presses an arrow key, and only if the direction is not blocked by a wall, in that case the user gets to try again, without the monsters moving.

A hero can hit any monster, as any monster can hit a hero. A "hit" happens in the case one player tries to move into the position of another player. Clearly, he cannot do so, as no two players are allowed in the same position. You can think of "hitting" as players jumping on and back off one another player until some player dies, in which case the attacker doesn't need to jump back to his original position, and stays in the dead player's position.

The game is over when the hero dies.

E.3.1 Player movement

Heroes and Monsters move alike. In particular, every player moves exactly one position in some direction in every turn, if possible. A move may fail due to a wall or another player being in the desired final position. If the move fails, the player stays in the same place, and unless it is a hero, yields his turn to the next player.

1. Define the enum type `Direction`, having the values `NORTH`, `SOUTH`, `EAST` and `WEST`. Refer to Figure 1 (page 3) if you're in doubt which way is which.
2. Implement the following method in the `Player` class:

```java
/**
 * Attempts to move the player one point in the desired direction.
 * The move fails if there is a wall or another player in the target
 * position. In the latter case, the player in the target position is
 * damaged. If the other player dies due to damage, the move succeeds
 * anyways.
 * @return true if move succeeded, false otherwise.
 */
public boolean tryMove(Direction direction) { }
```

Hint: It might be a good idea to let the `Player` class retain a reference to the `Dungeon` that it is in.

E.3.2 Monster movement

All monsters inherently want to kill the hero. Initially, they have a simple strategy – move towards the hero and hit him. Here, "towards" means towards a reduced Manhattan distance. Given a player in position (x_1, y_1), and another player in position (x_2, y_2), the Manhattan distance between them is $|x_1 - x_2| + |y_1 - y_2|$. This should imply to you that the monsters disregard the placements of the doors and just blindly "follow the scent of the hero".

We define the following restrictions on monster movement in general:

(a) If a monster fails to move in a given direction, he gets no extra turn.

(b) A monster does not hit another monster.

1. Implement the method `public void makeAutonomousMove()` in the `Monster` class.

E.3.3 Items & Looting

All players are allowed to move into a position where an item is located. If a monster steps on an item, nothing happens, if a player steps on an item, he picks it up. There is at most one item in any given position.

1. Implement the following method in the `Dungeon` class:

```java
/**
 * @return the item at the given position, if any.
 * @null if there is nothing to gather.
 * Removes the returned item from the appropriate room.
 */
public Item loot(Point position)
```

E.4 Couple everything together (ca. 1 hour)

E.4.1 Bind keys to actions

1. Implement the actions for the arrow key action listeners so that the hero can be moved about the dungeon by the user. Here's a rough overview of what should happen on *every* move:

   (a) The hero moves (if he can't, the user gets to try again without the monsters moving).

   (b) The monsters move.

   (c) All players regenerate.

   (d) The dungeon is repainted.

By now, you should be able to freely play the game.

E.5 Implement one of the suggested extensions (ca. 5 hours)

We suggest three extensions to the game. We *do not* ask you to implement all of them, although you may if you wish. We *do* ask you to implement *at least* one of them. To make sure that we know what to grade, we ask you to submit a file named `extension.txt` where on the first line you write either 1, 2 or 3, indicating which of the below extensions you've implemented. You may add additional comments in the file, if you deem it required to explain your implementation of the extension.

Note: The handed out compiled jar file does not have any of the extensions implemented.

E.5.1 Monster strategy

Make the monsters smarter. In particular,

1. If the hero is in the same room as the monster, the monster moves as before, towards the hero.
2. If the hero is in a room next to the room the monster is in, the monster moves towards the door and through it.
3. If none of the above conditions are satisfied, the monster picks a door at random, and moves towards and through it. The monster should not pick the door it just went through, unless it is the only door in the room.

E.5.2 Auto-generated dungeon

Instead of loading the dungeon from disk, auto-generate a dungeon. The auto-generated dungeon should follow all the properties of dungeons already mentioned. Here's a brief summary:

1. Every room has at least one door.
2. Rooms are rectangular and rooms connected by doors share a wall. This implies that any two rooms share at most one wall.
3. There should be a path from every room to every other room in the dungeon.
4. Doors are located on the walls of the room, but never in a corner.
5. No rooms overlap. No two monsters are in the same position. No two items are in the same position. Monsters and items are allowed to be placed in the same position.
6. No items or monsters are placed on walls or outside of the dungeon.

E.5.3 Hero inventory

Although the hero only uses the most powerful weapon he has picked up, we would like to retain a list of items picked up by the hero. This allows us to show an inventory overview when the user presses the 'i' key on the keyboard during the game (the game should be paused). The inventory overview should show the following:

1. Name and type of the hero.
2. The list of items picked up by the hero.
3. The current weapon, armor and health of the hero.

6 Files expected to be handed in

Although the structure is largely in your hands, some of the assignments have asked you to implement classes with specific names, and hence include files with certain specific names. The list below is the list of those files with an explanation of what purpose that file serves. The files may be nested in directories to follow some package hierarchy you may choose to employ.

`Point.java`	Represents a point-pair in two-dimensional space.
`Player.java`	The base class for all players.
`Hero.java`	The hero base class.
`Warrior.java`	The warrior type of hero.

`Mage.java`	The mage type of hero.
`Cleric.java`	The cleric type of hero.
`Monster.java`	The monster base class.
`Orc.java`	The orc type of monster.
`Goblin.java`	The goblin type of monster.
`Dungeon.java`	The model for the dungeon itself.
`Room.java`	The model for a room of the dungeon.
`Weapon.java`	The weapon type of item.
`Armor.java`	The armor type of item.
`MainFrame.java`	The main frame of the GUI, contains the `main` method.
`WelcomePanel.java`	The panel shown when the game is started.
`GamePanel.java`	The panel that shows the main game.
`Direction.java`	An enum type for represented movement directions.
`extension.txt`	A file explaining what extension was implemented.
`overview.txt`	It might prove beneficial for you to provide a similar overview of all the files in your submission. This is the place to put it.