Praktikum 4: Dateiverarbeitung, Algorithmen

Lernziele

- Arbeiten mit Klassen in Vererbungshierarchien
- Anwenden von Polymorphie bzw. virtuellen Methoden
- Einfache grafische Algorithmen
- Dateiein- und ausgabe

Aufgabenstellung

1 Erweiterung der `Character`-Klasse

Eine Spielfigur wird mit verschiedenen **Werten** (`int`) ausgestattet:

$Strength$ ist die physische Stärke der Spielfigur und wird verwendet um zu berechnen wie viel Schaden die Figur im Kampf austeilt.

$Stamina$ ist die Ausdauer der Spielfigur, und bestimmt unter anderem, wie viel Schaden die Figur maximal einstecken kann.

$Hitpoints$ sind die Schadenspunkte, die die Figur aktuell noch aushalten kann. Sinkt die Zahl auf 0, ist die Figur tot und wird aus dem Spiel entfernt.

Implementieren Sie ebenfalls eine Methode `getMaxHP()`, welche die maximalen Hitpoints ausrechnet. Die Formel hierfür lautet: $MaxHP = 20 + (Stamina * 5)$.

Ändern Sie den Character Konstruktor so, dass Werte für Strength und Stamina als Parameter übergeben werden können, und setzen Sie ebenfalls im Konstruktor die Hitpoints auf `getMaxHP()`.

Fügen Sie eine Methode `showInfo()` hinzu, welcher alle Werte einer Spielfigur auf der Konsole ausgibt.

2 `Item`-Klasse

Fügen Sie eine neue Klasse `Item` hinzu, welches einen benutzbaren Gegenstand im Spiel darstellen wird. `Item` ist hierbei die **abstrakte** Basisklasse für alle anderen Gegenstände.

Gegenstände modifizieren im Allgemeinen die oben genannten Attribute einer Spielfigur, also $Strength$, $Stamina$ und $Hitpoints$. Waffen beispielsweise erhöhen $Strength$ und damit die Kampfstärke, Rüstungen modifizieren üblicherweise $Stamina$ usw.
Dies geschieht entweder über einen festen Wert ($Strength + 2$) oder relativ zum Ausgangswert ($Stamina + 50\%$).

Deklarieren Sie in der `Item`-Klasse folgende virtuellen Methoden:

`int modifyStrength(int strength)` bekommt das Basisattribut $Strength$ übergeben und liefert den **Bonus** zurück, den dieser Gegenstand liefert. Wie oben beschrieben, kann dies ein fester Wert sein (in dem Fall wird der Parameter nicht benötigt), oder ein Prozentwert von `strength`.

int modifyStamina(int stamina) verhält sich genau so.

Hinweis: Die Methoden sind in `Item` virtuelle und geben jeweils 0 zurück.

Fügen Sie der `Character`-Klasse einen `vector` aus `Item*` hinzu, damit eine Spielfigur Gegenstände tragen kann. Implementieren Sie in `Character` weitere Funktionen

`addItem(Item*)` fügt der Spielfigur einen Gegenstand hinzu.
`getStrength()`
`getStamina()`

wobei die getter auf die Basisattribute $Strength$ und $Stamina$ die Boni der Gegenstände aufaddieren, anschließend den Basiswert addieren und diesen Wert zurück geben.
Wichtig: Gegenstände verändern **nicht** den eigentlichen Wert eines Attributs im Spieler!

Hinweis: Denken Sie daran, in `Character::getMaxHP()` die Methode `getStamina` zu verwenden!

Gegenstände sollen zerstört (`delete`) werden, sobald ein `Character`-Objekt zerstört wird.

Gegenstände können auf „normalen" `Floor`-Kacheln liegen. Fügen Sie in der `Floor`-Klasse also einen `Item*` hinzu. Auf dem Boden liegende Gegenstände werden durch das Symbol '*' angezeigt. Sobald ein Spieler diese Kachel betritt, bekommt er den dort liegenden Gegenstand.

Implementieren Sie die unten stehende Gegenstandliste, wobei jeder Gegenstand eine eigene Klasse ist. Die Klassen überschreiben aus `Item` die `modify`-Methoden entsprechend.

Name	Boni		Name	Boni	
	Strength	Stamina		Strength	Stamina
Arming Sword	+3	0	Gambeson	0	+1
Greatsword	+5	-1	Mail Armour	0	+3
Club	+50%	0	Shield	-1	+100%
Rapier & Dagger	+2	+1	Full Plate Armour	-2	+6

Hinweis: Da die Klassen an und für sich nicht sonderlich viel Inhalt haben, können Sie alle erwähnten Gegenstände auch in `Item.h` bzw. `Item.cpp` implementieren.

3 Implementierung von „Line-of-Sight"

Implementieren Sie ein rudimentäres **Sichtfeld**, welches Kacheln verdeckt, die vom Spieler nicht gesehen werden können. Als Vereinfachung gehen wir davon aus, dass Figuren eine 360° Rundumsicht besitzen.

Suchen Sie im Internet[1] nach einem geeigneten Verfahren zur Rasterung von Linien. Die einfachste Implementierung einer Sichtlinie ist es, ausgehend von der Spielfigur eine Linie zu **jeder** Kachel des Spielfeldes zu ziehen, und „unterwegs" zu schauen, ob ein undurchsichtiges Objekt gefunden wurde.

Hinweis: Implementieren Sie in der `Tile`-Klasse eine Methode `bool isTransparent`, die zurück gibt, ob die Kachel „durchsichtig" ist. Lassen Sie die Methode am einfachsten in `Tile` `true` zurück geben, und überschreiben Sie diese in `Wall` und `Door`. Eine Wand ist **nie** durchsichtig, eine Tür nur, wenn Sie offen steht.

Implementieren Sie in der Klasse `DungeonMap` eine Methode `bool hasLineOfSight(Position from, Position to`, welche `true` liefert, falls zwischen `from` und `to` eine Sichtlinie besteht; Ansonsten `false`.

Ändern Sie die `print()`-Methode in `DungeonMap` so ab, das diese eine Position (Reihe/Spalte) übergeben bekommt. Die Karte soll **ausgehend von dieser Position** gezeichnet werden. Überprüfen Sie hier einfach für jede Kachel, ob `hasLineOfSight` für diese Kachel `true` oder `false` ist.
Falls `true` zeichnen Sie die Kachel, falls `false` geben Sie `'#'` aus.

4 Weitere `Tile`-Typen

4.1 `Lever`

Ein `Lever` funktioniert ähnlich wie ein `Switch` und kann ein `Passive`-Objekt schalten. Der Unterschied besteht darin, dass man einen `Lever` durch erneutes Betreten der Kachel wieder abschalten kann.

[1] Guter Ausgangspunkt: `https://de.wikipedia.org/wiki/Rasterung_von_Linien`

4.2 `Trap`

Diese Klasse stellt eine Falle dar. Beim Betreten der Falle werden der Spielfigur eine Anzahl an Hitpoints abgezogen. Anschließend ist die Falle entdeckt und nicht mehr wirksam.
Implementieren Sie die Klasse so, dass eine nicht ausgelöste Falle aussieht wie eine normale Bodenkachel. Beim erstmaligen Betreten durch eine Figur soll diese 20 Hitpoints abgezogen bekommen. Die Falle ist danach unschädlich und wird durch ein `'T'` dargestellt.

5 Weitere Controller

Fügen Sie einen weiteren Controller `StationaryController` hinzu, der in der `move`-Methode einfach den Wert 5 zurück liefert, gleichbedeutend mit „nicht bewegen".
Hinweis: Falls „nicht bewegen" bei Ihnen ein anderer Wert sein sollte, geben Sie diesen zurück.

6 Erweiterung des Parsers

Erweitern Sie den Parser, um mit die neuen Kacheltypen sowie Gegenstände verarbeiten zu können. Fügen Sie ebenfalls eine Unterstützung für Spielfiguren hinzu. Ihr Programm sollte danach also ausschließlich über die Parameter gesteuert werden, und nicht mehr über „hard coded values".
Beispiel:

```
vector<string> specialTiles{
  "Character @ 5 5 ConsoleController 9 9",
  "Character % 2 3 StationaryController 3 4",
  "Greatsword 1 4",
  "Door 13 9 Lever 11 9",
  "Trap 10 9"
};
```

Zeile 2 definiert eine Spielfigur mit dem Zeichen `@` mit Strength und Stamina jeweils 5, gesteuert von einem ConsoleController Objekt, startet an Reihe 9, Zeile 9.
Zeile 3 definiert eine weitere Spielfigur mit Zeichen %, Strength 2, Stamina 3, dem StationaryController an Position 3/4.
Zeile 4 legt ein `Greatsword` an Position 1/4 usw.

7 Laden von Spielständen

Implementieren Sie eine Funktion `loadFromFile(string filename)` die einen Spielstand aus einer Datei lädt. Die Datei enthält dabei die gleichen Informationen, die sonst in den beiden `string`-Vektoren gespeichert sind.
Hinweis: Sie können sich das Format der Textdatei frei aussuchen, beispielsweise kann es sinnvoll sein, die Höhe und Breite sowie die Anzahl der Spezialkacheln mit ab zu speichern.
Beispiel:

```
20 20
####################
#......#############
usw... (restliche Zeilen)
3
Character @ 5 5 ConsoleController 9 9
Character % 2 3 StationaryController 3 4
Greatsword 1 4
```

8 Spielermenü

Implementieren Sie ein Spielermenü, welches aufgerufen wird, wenn der Spieler beispielsweise die Bewegungsrichtung 0 drückt. In diesem Menü soll es möglich sein, dass Programm zu beenden, wieder zurück zur Bewegungsauswahl zu kommen und die Spielerinfos anzuzeigen.