Praktikum 3: Klassen, `const` & Kopierkonstruktoren

Lernziele

- Arbeiten mit Klassen in Vererbungshierarchien
- Anwenden von Polymorphie bzw. virtuellen Methoden
- Einfache Stringverarbeitung

Aufgabenstellung

In Praktikum 3 werden Sie das Programm aus dem letzten Praktikum noch weiter erweitern und verfeinern. Ein Hauptaugenmerk liegt dabei auf der Einführung von **Vererbung** und **Polymorphie**.

Inhaltlich kommen damit vor allen Dingen neue Kacheltypen (Klasse `Tile`) hinzu, sowie die Einführung einer `Controller`-Klasse, die eine Spielfigur steuert.

1 Erweiterung der `Tile`-Klasse

Aktuell ist der Ist-Zustand, dass Kacheltypen durch ein Attribut eines `enum`-Datentyps unterschieden werden. Dies soll durch die Einführung einer **Vererbungshierarchie** abgelöst werden.

Implementieren Sie hierzu die beiden Klassen `Floor` und `Wall` als Unterklassen der Klasse `Tile`. Sinnvollerweise setzen Sie hierbei die beiden Methoden `onEnter` sowie `onLeave` auf `virtual` zu setzen.

Implementieren Sie die beiden Methoden so, dass diese eine „normale" Bewegung darstellen, also genau so funktionieren wie die Bewegung auf eine Bodenkachel. Die Idee ist, das von einer normalen Bewegung abweichender Kacheltyp die jeweiligen Methoden einfach überschreibt.

Hinweis: In Hinsicht auf die zukünftige Implementierung ist es sinnvoll, die Logik das Wände nicht betretbar sind, in der Methode `Wall::onEnter` zu implementieren. Sinnvoll ist es, dass eine Wandkachel die Spielfigur quasi „auf das Ursprungsfeld" zurück schickt. Hierzu können Sie den Zeiger `fromTile` verwenden.

Wenn Sie alles korrekt implementieren, dann benötigen Sie innerhalb der `Tile`-Klasse den `enum`-Wert nicht mehr! Führen Sie ihn dennoch weiter, vermutlich benötigen Sie ihn bei der Ausgabe (`DungeonMap::print`) oder beim Erzeugen der Kacheln. Falls Sie sicher sind, den Datentyp jedoch nicht mehr zu benötigen, dann dürfen Sie ihn gerne entfernen.

Ziel und Sinn: Entfernen der Notwendigkeit, Objekte anhand des `enum`s zu identifizieren und hin zu Aufrufen von virtuellen Methoden.

1.1 Aktive und passive Kacheltypen

Implementieren Sie zwei **abstrakte** Klassen `Active` und `Passive`. Diese sollen Kategorien von Objekten darstellen, die Aktionen auslösen können (Active) bzw. auf Aktionen reagieren können (Passive).[1]

Passive Kacheln erben von `Tile` und besitzen darüber hinaus einen Status (`bool`) mit einem dazugehörigen setter (ggfs. auch einem getter)

Aktive Kacheln erben ebenfalls von `Tile` und besitzen eine Information, ob sie bereits betätigt wurden oder nicht (`bool`) sowie ggfs. setter und getter. Ebenfalls besitzen Sie einen Zeiger auf ein `Passive`-Objekt, nämlich das passive Objekt, welches durch das aktive Objekt geschaltet werden soll. Für den Zeiger sollte ebenfalls ein setter existieren.

Ziel und Sinn: Schaffen eines Rahmenwerks für verschiedene Komponenten des Spiels, die miteinander interagieren können.

[1] In Anlehnung an das Entwurfsmuster „Beobachter": `https://de.wikipedia.org/wiki/Beobachter_(Entwurfsmuster)`

1.2 Schalter und Türen

Implementieren Sie eine Klasse `Switch`, welche von `Active` erbt sowie eine Klassen `Door`, welche von `Passive` erbt.

Ein Switch soll dabei eine Kachel sein, die eine Art Bodenschalter besitzt, der durch Betreten der Kachel ausgelöst wird. Ein Switch kann nur einmal betätigt werden. Ein nicht-gedrückter Schalter soll durch `'?'`, ein gedrückter durch `'!'` bei der Ausgabe dargestellt werden.

Eine Tür (Door) ist ein passives Element, welches im geschlossenen Zustand wie eine Wand reagiert, also das Betreten durch eine Spielfigur verbietet. Im geöffneten Zustand verhält sich eine Tür wie eine normale Bodenkachel. Eine geschlossene Tür wird durch `'X'` dargestellt, eine offene durch `'/'`.

Ziel und Sinn: Testen des oben genannten Rahmenwerks.

2 Abstrakte Klasse `Controller`

Fügen Sie eine neue, abstrakte Klasse `Controller` hinzu. Ein Controller Objekt ist zuständig für die Steuerung einer Spielfigur. Von dieser abstrakten Klasse erben alle weiteren Controller.

Die Klasse enthält als **rein virtuelle** Methode `int move()` und soll darüber hinaus die kontrollierte Spielfigur kennen, also als Attribut einen `Character*` enthalten. Sehen Sie nach Bedarf Konstruktoren, getter und setter vor.

Implementieren Sie weiterhin eine Klasse `ConsoleController`, welche von `Controller` erbt und die `move`-Methode implementiert. Diese soll genau so funktionieren wie die `move` Methode aus der Klasse `Character` aus dem letzten Praktikum.

Ziel und Sinn: Schaffen der Möglichkeit, Spielfiguren flexibel durch verschiedene Klassen steuern zu können. Neben der Möglichkeit, Figuren durch die Tastatur zu steuern (`ConsoleController`) schafft dies Flexibilität für weitere Klassen. Denkbar ist zu einem späteren Zeitpunkt beispielsweise eine kleine KI, die sich auf den Spieler zu bewegt.

3 Erweiterung der `Character`-Klasse

Erweitern Sie die `Character`-Klasse, so dass diese einen `Controller*` enthält. Die `move`-Methode aus der Klasse `Character` soll nun nichts weiter tun, als die `move`-Methode des Controllers aufzurufen.

Ziel und Sinn: Entkopplung von Spielfigur und Steuerung.

4 Erweiterung der `GameEngine`-Klasse

Die Klasse `GameEngine` muss erweitert werden, um die neuen Kacheltypen `Switch` und `Door` unterstützen zu können. Zwei Objekte der jeweiligen Klassen sind über einen Zeiger verbunden, allerdings fehlt aktuell in der Spielwelt (dem `vector<string>` aus der `main`) noch die Möglichkeit, dort Schalter und dazugehörige Tür anzugeben.

Um so flexibel wie möglich zu bleiben, sollen Sie einen zweiten `vector<string>` übergeben, der Positionen von zusammen gehörenden Schaltern und Türen enthält. Überlegen Sie sich eine **Syntax** für diese Information und schreiben Sie im Konstruktor von `GameEngine` einen entsprechenden Interpreter. Denkbar ist beispielsweise:

```
"6 5 Door 2 5 Switch"
```

was eine Tür an Reihe 6 Spalte 5 definiert und den dazugehörigen Schalter an Position $2/5$. Sie können natürlich, wenn das für Ihre Implementierung sinnvoll ist, in der Spielwelt dennoch Zeichen für Tür- bzw. Schalterobjekte vorsehen.

Ziel und Sinn: Schaffen eines Interpreters, um beliebig viele spezielle Kacheln in der Spielwelt zu definieren. Bei alleiniger Verwendung von Zeichen in der Spielwelt (`'D'` bzw. `'S'` für Türen bzw. Schalter) fehlt die Information, welche Tür mit welchem Schalter verbunden ist.