Praktikum 2: Dungeon Crawler - Erweiterung

Lernziele

- Arbeiten mit Klassen in Vererbungshierarchien
- Mehrfachvererbung ohne gemeinsame Basisklasse
- Anwenden von `typeid` und `dynamic_cast`

Aufgabenstellung

In Praktikum 2 werden Sie das Programm aus dem letzten Praktikum noch weiter erweitern und verfeinern.

Inhaltlich kommen damit vor allen Dingen neue Kacheltypen (abgeleitet von Klasse `Tile`) hinzu, sowie die Einführung einer `Controller`-Klasse, die eine Spielfigur steuert. Weiterhin werden Kacheltypen eingeführt, die mit anderen Objekten bzw. Kacheln interagieren können.

Am Ende jeder Aufgabe fassen wir kurz zusammen, welchen Sinn die jeweiligen Änderungen für das große Gesamtbild haben. Uns ist wichtig, dass Sie immer das große Ganze verstehen, und nicht nur einfach implementieren, was die Aufgabenstellung gerade erfordert.

1 Aktive und passive Objekttypen

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]

Die beiden Klassen stellen zunächst keine konkreten Objekte dar, sondern implementieren lediglich ein **Konzept**. Im weiteren Verlauf des Praktikums wird dieses Konzept auf Kacheltypen und auf Gegenstände angewendet.

Passive Kacheln besitzen eine **rein virtuelle** Methode `notify(Active* source)`. Diese soll aufgerufen werden, wenn für dieses `Passive`-Objekt eine Aktion ausgelöst werden soll. Das `Active` Objekt übergibt dabei einen Zeiger auf sich selbst. Dies kann in von `Passive` Objekten verwendet werden um zu ermitteln, durch welches Objekt die Aktion ausgelöst wurde.

Aktive Kacheln besitzen eine Liste von Zeigern auf `Passive`-Objekte. Die Liste soll die `Passive`-Objekte enthalten, die durch dieses `Active`-Objekt aktiviert werden. Weiterhin soll die Klasse über die Methoden `attach(Passive*)` und `detach(Passive*)` verfügen, mit denen `Passive`-Zeiger der Liste hinzugefügt bzw. entfernt werden können.

Wichtig: Achten Sie bei der Implementierung von `attach` darauf, dass der `Passiv*` maximal einmal in der Liste vorkommen darf.

Implementieren Sie weiterhin eine Methode `activate()`, welche die `notify()` Methode **aller** verbundenen `Passive`-Objekte aufruft.

Hinweis: „Liste“ meint eine beliebige Datenstruktur, nicht zwangsläufig eine `std::list`.

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

1.1 Schalter und Türen

Um das eben implementierte Rahmenwerk zu testen implementieren Sie zwei neue Kacheltypen: Eine Art Bodenplatte, die als Schalter funktioniert sowie eine Tür, welche durch einen Schalter geöffnet und geschlossen werden kann.

[1] In Anlehnung an das „Observer Pattern“: `https://en.wikipedia.org/wiki/Observer_pattern`

Implementieren Sie eine Klasse `Switch` als Mehrfachvererbung von `Tile` und `Active`. `Switch` erbt damit sowohl die Eigenschaft, eine Kachel zu sein, als auch die Eigenschaft ein aktives Steuerelement zu sein!

Ein Switch soll dabei eine Kachel sein, die eine Art Bodenschalter besitzt, der durch Betreten der Kachel ausgelöst wird. Ein Switch wird auf dem Terminal durch ein `'?'` dargestellt.

Implementieren Sie die Klasse `Door` als Mehrfachvererbung von `Tile` und `Passive`.

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 `'/'`.

Hinweis: Auch denkbar wäre es, `Switch` und `Door` von `Floor` anstatt von `Tile` erben zu lassen. Eine weitere Variante ist, `Door` von `Floor`, `Wall` und `Passive` erben zu lassen. Im Prinzip verhält sich eine Tür wie eine Wand, so lange sie geschlossen ist und wie eine normale Bodenkachel, wenn Sie geöffnet sind. Sie können hier frei wählen.

Ziel und Sinn: Testen des oben genannten Rahmenwerks.

Implementieren Sie die beiden Klassen und erweitern Sie Ihren Level Konstruktor so, dass Sie die Klassen testen können.

2 Neue Kacheltypen: Pit and Ramp

Implementieren Sie zwei neue Kacheltypen als Klassen `Pit` und `Ramp`, die von `Tile` erben.

Ein Pit ist eine Vertiefung im Boden, die von beliebigen anderen Kacheln aus betreten werden kann, allerdings kann ein Pit nicht ohne weiteres verlassen werden. Von einer Pit Kachel aus kann nur eine andere Pit Kachel oder aber eine Rampe (Klasse `Ramp`) betreten werden.

Eine Rampe wiederum kann normal betreten oder verlassen werden. Die Rampe dient also nur dazu, einen tiefer liegenden Bereich des Levels wieder verlassen zu können.

Implementieren Sie die beiden Klassen und erweitern Sie Ihren Level Konstruktor so, dass Sie die Klassen testen können.

3 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()`.

Die Funktionalität in der `move()` Methode aus `Character` soll nun folgerichtig in die Klasse `TerminalUI` verschoben werden, da ja dort die Ein- und Ausgaben zentralisiert werden sollen. `TerminalUI` erbt demnach zusätzlich zu `AbstractUI` von `Controller` und implementiert die `move()` Methode entsprechend.

`TerminalUI` erbt dann mehrfach von `AbstractUI` und von `Controller`!

Ziel und Sinn: Schaffen der Möglichkeit, Spielfiguren flexibel durch verschiedene Klassen steuern zu können. Neben der Möglichkeit, Figuren durch die Tastatur zu steuern (via `TerminalUI`) schafft dies Flexibilität für weitere Klassen. In späteren Praktika wird dies verwendet, um NPCs zu implementieren, die sich durch festgelegte Regeln bewegen. Diese Regeln sind dann wiederum in eigenen Controllern implementiert.

4 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. Im späteren Verlauf kommen „Computergegner“ in Form von verschiedenen Controllern zum Einsatz.

5 Änderungen an `Level`

Erweitern Sie die `Level` Klasse um einen **Kopierkonstruktor** sowie einen **Zuweisungsoperator**. Diese sollen eine eigenständige Kopie des übergebenen `Level` Objekts erstellen.

Hierzu ist es notwendig, die genauen Objekttypen der `Tile*` zu ermitteln. Hierzu sollte der `typeid` Operator verwendet werden.