Aufgabe Vererbung und Objektinteraktion - Häuser bauen

Ihr findet das Szenario „Vererbung“ vor, das schon teilweise Java-Code enthält und nun von Euch ergänzt werden soll. Implementiert das Szenario Vererbung so, dass die folgenden Aufgaben möglich sind.

Ein Stadtplaner (mit rechter Maustaste und „new Stadtplaner()“ erzeugen) legt fest, wo welche Gebäude gebaut werden. Er kann sich (bereits jetzt) mit den Pfeiltasten über den Bildschirm bewegen. Dies ist bereits programmiert durch:

```
String keyString = Greenfoot.getKey();
folgeTastaturEingaben(keyString);
```

Diese Zeilen daher bitte nicht löschen.

Aufgabe 1: Der Stadtplaner soll an seinem aktuellen Standort ein Haus errichten, wenn F1 gedrückt wird, eine Hütte, wenn F2 gedrückt wird und eine Burg, wenn F3 gedrückt wird. Dies soll in der act()-Methode des Stadtplaners programmiert werden.

Hinweis:

getWorld().addObject(new Haus(), 3, 3);

fügt ein Haus an der Position (3/3) ein.

Mit getWorld() bekommt man Zugriff auf die aktuell geladene World. Dies ist notwendig, da nur in einer World-Klasse die Befehle addObject() und removeObject() zur Verfügung stehen, denn „nur die Welt darf Objekte erschaffen und löschen“.

Aufgabe 2: Der Stadtplaner soll verschwinden, wenn F8 gedrückt wird.

Hinweis:
Wird der Befehl

getWorld().removeObject(this);

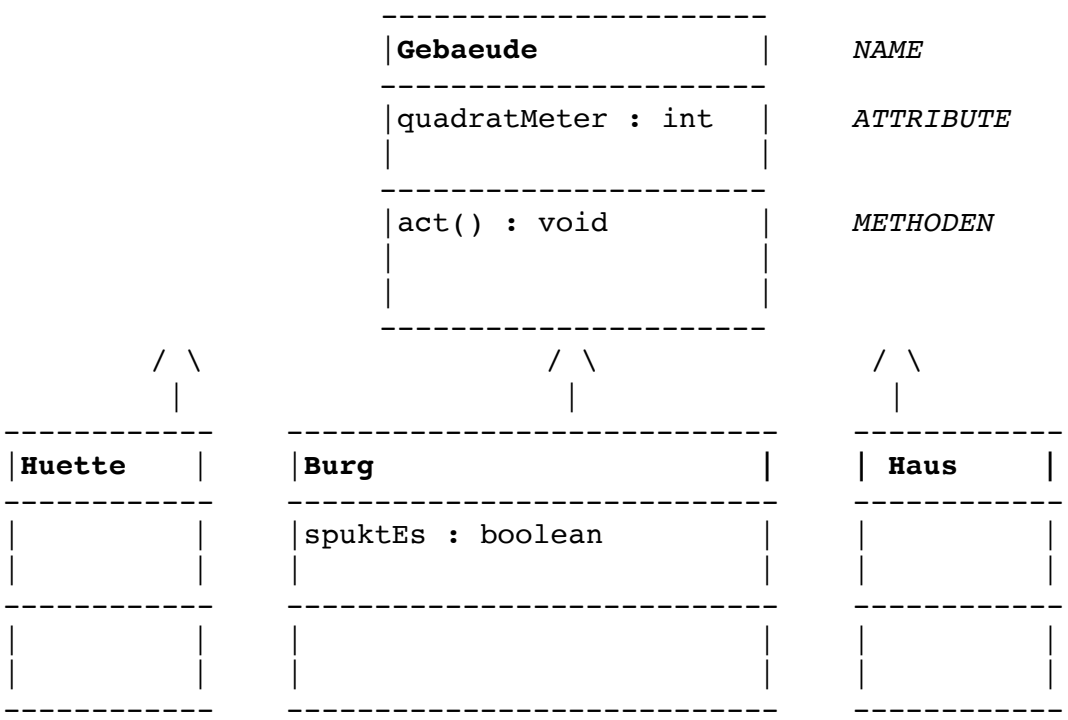
von einem Objekt aufgerufen, löscht es sich selbst vom Bildschirm

Nun kommt der Architekt ins Spiel. Er läuft ebenso in der Welt herum, wenn der Stadtplaner wieder weg ist. *Er muss ebenso mit Rechtsklick (auf das Symbol rechts) und „new Architekt()“ in die Welt gezogen werden.* Der Architekt kann für Gebäude die Attribute festlegen (z.B. wie viele Zimmer, wie viel Quadratmeter usw.).

Aufgabe 3: Erstellt zunächst ein sinnvolles Klassendiagramm der Oberklasse Gebäude mit den drei Unterklassen Huette, Burg und Haus. Erstellt dazu für jede der insgesamt vier Klassen zwei sinnvolle Attribute und zwei sinnvolle neue Methoden. Ergänzt das folgende Klassendiagramm (das Angegebene ist in den Klassen schon vorprogrammiert)!

Hinweis: Unterklassen erben automatisch alle Methoden und Attribute der Oberklasse und haben dazu noch spezielle eigene Attribute und Methoden. Gemeinsame Attribute und Methoden werden dabei soweit nach oben „gezogen“ wie möglich. Daher bitte keine gleichen Attribute und Methoden in die Unterklassen einfügen!

Modelliert also in den Unterklassen nur spezielle Attribute und Methoden.



Aufgabe 4: Implementiert nun Eurer Klassendiagramm in Java Code. Erstellt dazu die entsprechenden Methodenköpfe (die Methoden selbst müssen nicht implementiert werden). Fügt zudem auch die beiden Attribute ein und beachtet die folgenden Hinweise!

Hinweis: In den Klassen findet ihr dies bereits beispielhaft teilweise vorgefertigt. Folgt diesem Schema und programmiert Eure Ideen nun.

Neben solchen Methoden wie klappeZusammen() (wie programmiert man das eigentlich später?), solltet Ihr vor allen Dingen zusätzlich zu den beiden von euch ausgedachten Methoden ...

... get- und set-Methoden für Eure Attribute programmieren!

Lest dazu auch die folgenden Hinweise!

get- und set- Methoden, wozu?

Klassen sind wie üblich z.B. so wie im Folgenden aufgebaut:

```
public class KlasseX {

  // Attribute
  private int a;

  // Methoden
  public void setA(int wert) {
    a = wert;
  }

  public int getA() {
    return a;
  }

}
```

Das dient dazu, dass die Attribute „geschützt“ sind. Niemand kann das Attribut a von außen direkt ändern (das kennzeichnet das Schlüsselwort *private*). Der einzige Weg ist es, mit den Methoden getA() und setA(...) zu arbeiten.
getA() fragt den Wert von a ab und mit setA(...) kann man dem Attribut a einen (neuen) Wert zuweisen.

BSP:

```
KlasseX x1 = new KlasseX();
x1.setA(6);
int b = x1.getA(); // nun hat b den Wert 6
```

Aufgabe 5: Der Architekt soll nun beim Tastendruck von F1 dafür sorgen, dass es in der Burg, die unter ihm ist, spukt:

Hinweis: Der folgende Code sorgt für die Interaktion mit einem Burg-Objekt:

```
Burg aktBurg = (Burg)getOneObjectAtOffset(0, 0, Burg.class);
if (aktBurg != null) {
  aktBurg.setHatGespenst(true);
}
```

Hinweis: Die erste Zeile sorgt dafür, dass wir Zugriff auf das Burg-Objekt bekommen, das unter uns ist. Da ja ganz viele Burgen im Szenario sein könnten, brauchen wir genau dasjenige Objekt, das im Umkreis von (0/0) Feldern von uns ist (also unter uns).

aktBurg ist nun eine so genannte „Referenz" auf unser gewünschtes Burg-Objekt. Ändern wir etwas an aktBurg, dann ändern wir automatisch etwas an dem originalen Objekt auf dem Bildschirm (das im Übrigen keinen Namen hat, denn es wurde ja mit addObject(new Burg(),...) erstellt). Hätte es einen Namen, könnte man etwas einfacher darauf zugreifen.

Nun denn...
... falls wir nicht über einem Burg-Objekt stehen sollten und trotzdem F1 drücken, soll nichts passieren, ansonsten wird von unserem Burg-Objekt die Methoden setHatGespenst(...) mit dem Wert true aufgerufen. In der Burg spukt es also ab jetzt, da das Attribut „spuktEs" in der Klasse Burg nun TRUE ist.

Weiterführender Hinweis: Die Methode **getOneObjectAtOffset(...)** gibt uns eine Referenz auf ein **Actor** – Objekt.
Wir benötigen aber eine Referenz auf ein **Burg** – Objekt, damit wir auch die Methode **setHatGespenst(...)** aufrufen können.

Dazu wird ein so genannter **Cast** durchgeführt: Quasi wird dazu ein Actor-Objekt in ein Burg-Objekt umgewandelt. Dies passiert, indem man den Namen der Klasse in Klammern ( ) voranstellt: (Burg).

Pausiere (auch bei den folgenden Aufgaben) das Szenario und kontrolliere manuell mittels rechter Maustaste, ob alles so funktioniert hat, wie geplant!

Aufgabe 5a: Der Architekt soll nun beim Tastendruck von F1 dafür sorgen,
- **dass es in der Burg die unter ihm ist, spukt.**
- **dass ein Haus einen Kamin hat.**
- **dass eine Huette ein Plumpsklo hat.**

Aufgabe 5b: Der Architekt soll nun beim Tastendruck von F2 dafür sorgen,
- **dass es in der Burg die unter ihm ist, nicht mehr spukt.**
- **dass ein Haus keinen Kamin hat.**
- **dass eine Huette kein Plumpsklo hat.**

Aufgabe 6: Der Architekt soll nun beim Tastendruck von F3 dafür sorgen,
- **dass ein Haus unter ihm 150 qm groß wird und 5 Zimmer hat.**
- **dass eine Huette unter ihm 10qm groß ist und 2 Zimmer hat.**
- **dass eine Burg unter ihm 666qm groß ist und 26 Zimmer hat.**

Aufgabe 7: Der Architekt soll nun beim Tastendruck von F4 dafür sorgen,
- **dass ein Haus unter ihm 170 qm groß wird und 7 Zimmer hat.**
- **dass eine Huette unter ihm 8qm groß ist und 1 Zimmer hat.**
- **dass eine Burg unter ihm 400qm groß ist und 20 Zimmer hat.**

Aufgabe 8: Setzt Eure eigenen Ideen aus dem Klassendiagramm mit den Tasten F5 bis F7 um.

Aufgabe 9: Spinnt das Szenario weiter. Mit F8 soll der Architekt wieder verschwinden. Ein Kaeufer kommt hinzu und kauft ...

a) **mit F1 nur Gebäude, die mindestens 6 Zimmer haben**

b) **mit F2 nur Gebäude, die mindestens 160 qm haben und höchstens 10 Zimmer haben**

c) **mit F3 Burgen, in denen es nicht spukt**

d) **oder eben mit F4 bis F7 Gebäude, die den von Euch festgelegten Kriterien entsprechen**

e) **usw. usw.**

Aufgabe 10: Hat ein Käufer ein Haus gekauft, soll dies entsprechend auf dem Bildschirm kenntlich gemacht werden. Es soll für das jeweilige Gebäude ein Schild ein neues Bild geladen werden mit der Aufschrift „VERKAUFT“.

Hinweis: Mit dem Befehl
setImage("images\paid.png")
kann in einer Actor-Klasse ein neues Bild für den Actor geladen werden. Schaut im Unterordner „images“ nach, wie die Bilder heißen, die Euch zur Verfügung stehen. Denkt auch genau darüber nach, wie Ihr diese Funktionalität implementieren solltet.

Aufgabe 11: Erweitert das Szenario mit kreativen und sinnvollen Ideen (bitte mit gewissem Anspruch!)

Aufgabe 12: Versucht zu verstehen, was das Schlüsselwort „abstract“ in den Klassen Gebaeude und Mensch bewirkt!

Aufgabe 13: Warum und wie funktioniert die Tastatursteuerung? In der Klasse Architekt werden z.B. gar keine Tasten abgefragt, trotzdem kann der Architekt sich bewegen. Wie funktioniert dies?

VIEL ERFOLG!