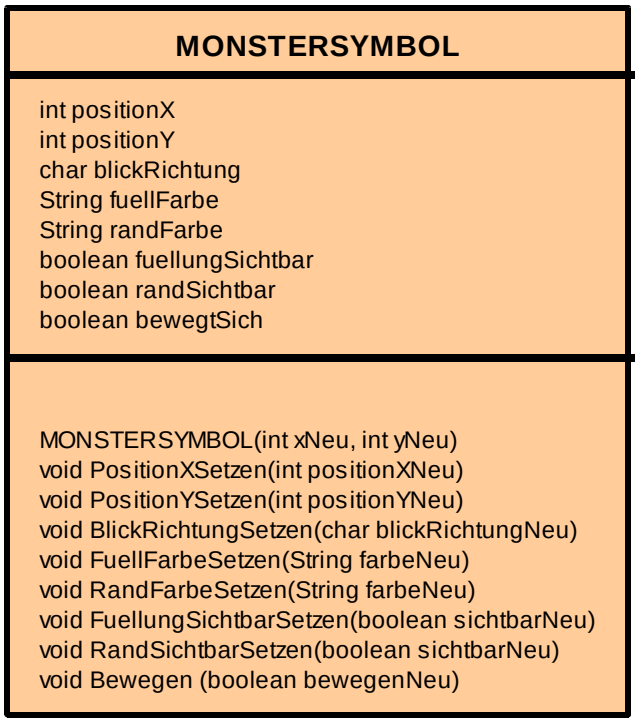
Kapitel 17 Monster als neue Spielfiguren

Lernziele:

In diesem Kapitel wird die Vererbung als zentrales Element objektorientierter Programmierung vorgestellt.

17.0 Neue Symbolklasse

Für dieses Kapitel steht die Klasse MONSTERSYMBOL.java zur Verfügung. Lade sie spätestens zur ersten Praxisaufgabe auf deinen Rechner und integriere sie in dein Projekt.



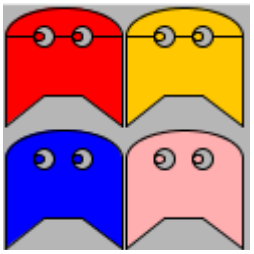


Abbildung 1: Klassendiagramm der Klasse MONSTERSYMBOL (links) und Objekte der Klasse MONSTERSYMBOL (oben)

17.1 Modellieren

Nach langer Entwicklungszeit von Krümel & Monster müssen nun endlich die Monster ins Spiel kommen. Zur Planung einer neuen Klasse sollte immer ein Klassendiagramm angefertigt werden.

Aufgabe 17.1

a) Notiere ein Klassendiagramm zur Klasse MONSTER mit allen aus deiner Sicht wichtigen Attributen und Methoden.

b) Vergleiche dein Ergebnis aus a) mit dem Klassendiagramm von der Klasse MAMPFI (Z. B. Klick mit der rechten Maustaste auf das Klassensymbol im Projekt von Kapitel 11). Beschreibe in Worten, was dir auffällt.

c) Wie könnte man das Ergebnis aus b) für die Programmierung von der Klasse MONSTER nutzen?

Im Folgenden werden die beiden Klassen MAMPFI und MONSTER gegenübergestellt:

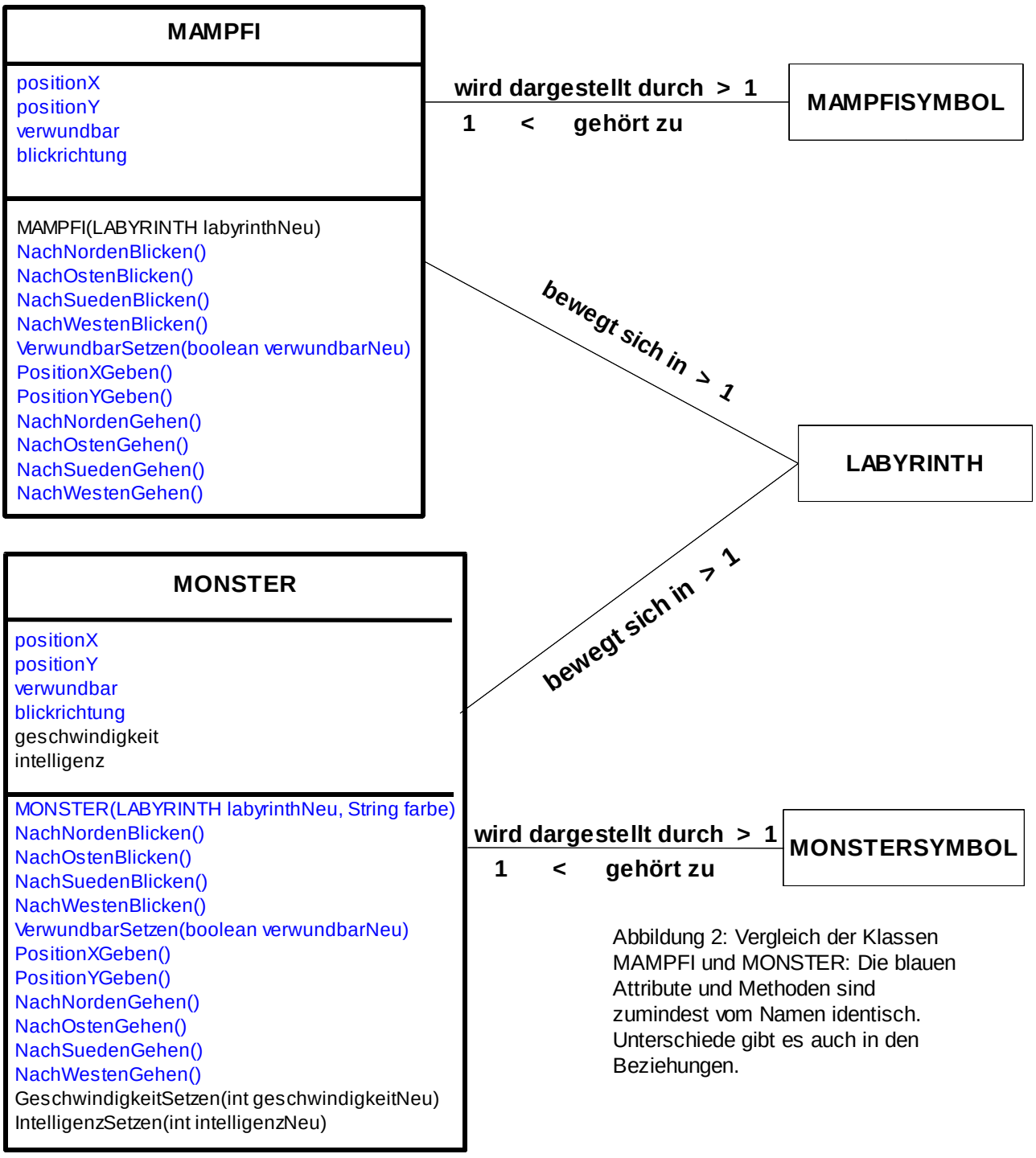


Abbildung 2: Vergleich der Klassen MAMPFI und MONSTER: Die blauen Attribute und Methoden sind zumindest vom Namen identisch. Unterschiede gibt es auch in den Beziehungen.

Auf den ersten Blick sind die beiden Klassen nahezu identisch. Auf den zweiten Blick gibt es doch ein paar kleine Unterschiede:

a) Monster könnten mit zunehmenden Level schneller und intelligenter werden, so dass es für Mampfi schwieriger wird alle Krümel zu fressen. Die dazu passenden Attribute geschwindigkeit und intelligenz der Klasse MONSTER gibt es in der Klasse MAMPFI nicht. Es wären noch weitere Attribute der Klasse MONSTER denkbar, die es in der Klasse MAMPFI nicht gibt. So könnte beispielsweise ein Attribut wurdeGefressen den Wert true haben, wenn ein Monster von Mampfi "gefressen wurde" und sein Geist zu einen Sammelpunkt wandert.

Die zu den Attributen geschwindigkeit und intelligenz gehörenden Setzen-Methoden gibt es natürlich nur in der Klasse MONSTER (schwarze Schriftfarbe), nicht in der Klasse MAMPFI.

b) Werden Objekte der Klasse MAMPFI erzeugt, so sind sie zunächst verwundbar (bis ein Powerkrümel gegessen wurde), Objekte der Klasse MONSTER sind zunächst unverwundbar. So unterscheiden sich die Konstruktoren nicht nur vom Namen, sondern es sind auch die Initialisierungen innerhalb der Konstruktoren unterschiedlich.
Auch die Eingabeparameter der Konstruktoren sind unterschiedlich, wenn gewünscht wird, dass jedes Monster durch ein Symbol unterschiedlicher Farbe dargestellt wird. Der Konstruktor *MONSTER* hat zusätzlich den Eingabeparameter farbe vom Typ String. Beim Aufruf des Konstruktors wird beispielsweise der Eingabewert "rot" verwendet, um die Farbe des Monstersymbols auf rot zu setzen. Dies geschieht ja auch im Konstruktor.

c) Objekte der Klasse MAMPFI werden durch Objekte der Klasse MAMPFISYMBOL dargestellt, Objekte der Klasse MONSTER dagegen durch Objekte der Klasse MONSTERSYMBOL. Dadurch sind die Referenzattribute, die für die Darstellung durch das entsprechende Symbol nötig sind, unterschiedlich (Abbildung 3)

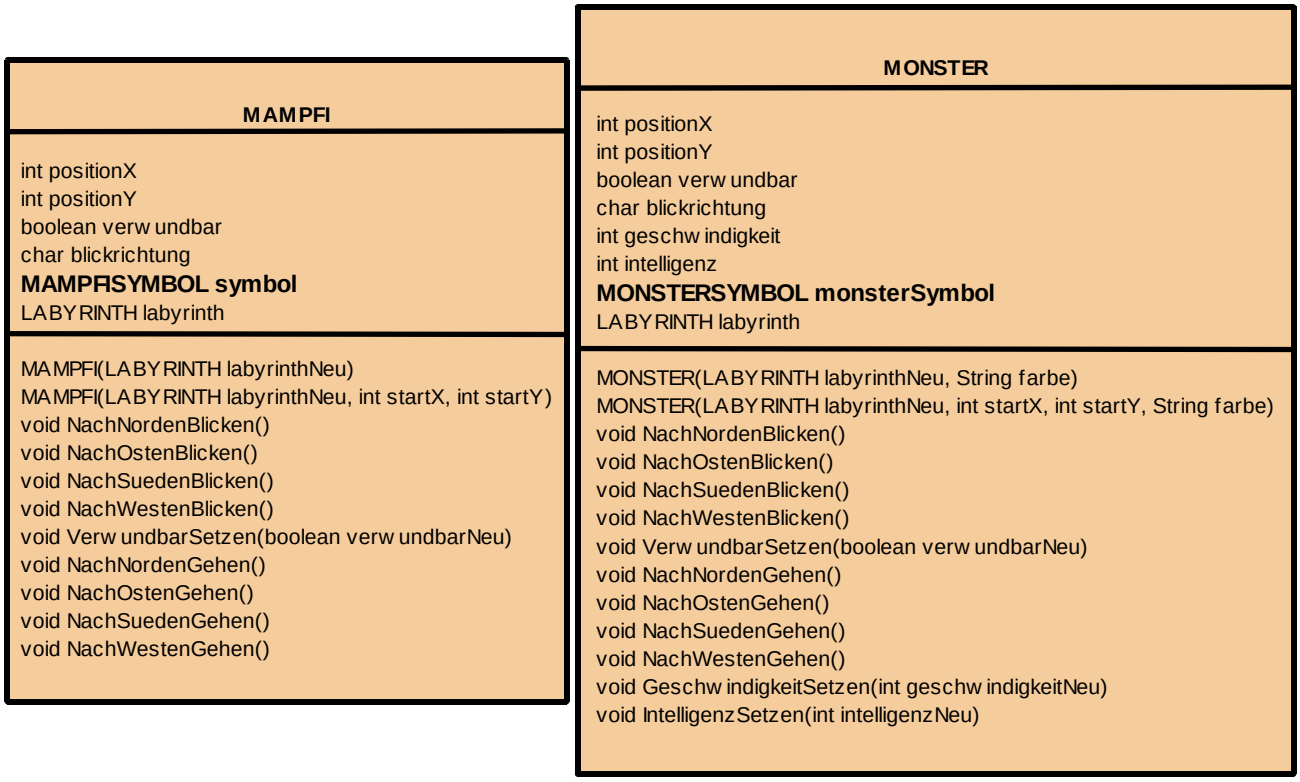


Abbildung 3: Erweiterte Diagramme der Klassen MAMPFI und MONSTER: Da die Darstellung durch Objekte unterschiedlicher Klassen erfolgt, unterschieden sich die entsprechenden Referenzattribute im Datentyp.

d) Die Methoden, die für das Gehen, die Änderung der Blickrichtung und der Verwundbarkeit zuständig sind, sind haben in den beiden Klassen zwar einen identischen Methodenkopf, aber die Methodenrümpfe unterscheiden sich. Dies liegt einerseits daran, dass die Referenzattribute unterschiedlichen Datentyps sind, d.h. der Adressat der Objektkommunikation ist unterschiedlich:
- symbol vom Typ MAMPFISYMBOL
- monsterSymbol vom Typ MONSTERSYMBOL
Andererseits unterscheiden sich die entsprechenden Symbolklassen auch in den

Schnittstellen, d.h. ihren Methoden. So ist beispielsweise innerhalb die Methode *NachNordenBlicken* beim Objekt der Klasse MAMPFISYMBOL der Methodenaufruf StartWinkelSetzen(120) nötig, beim Objekt der Klasse MONSTERSYMBOL der Methodenaufruf *BlickRichtungSetzen*('N')

Um den Punkt d) Spiegelstrich besser nachvollziehen zu können sind in Abbildung 4 die Klassendiagramme der beiden Symbolklassen gegenübergestellt.

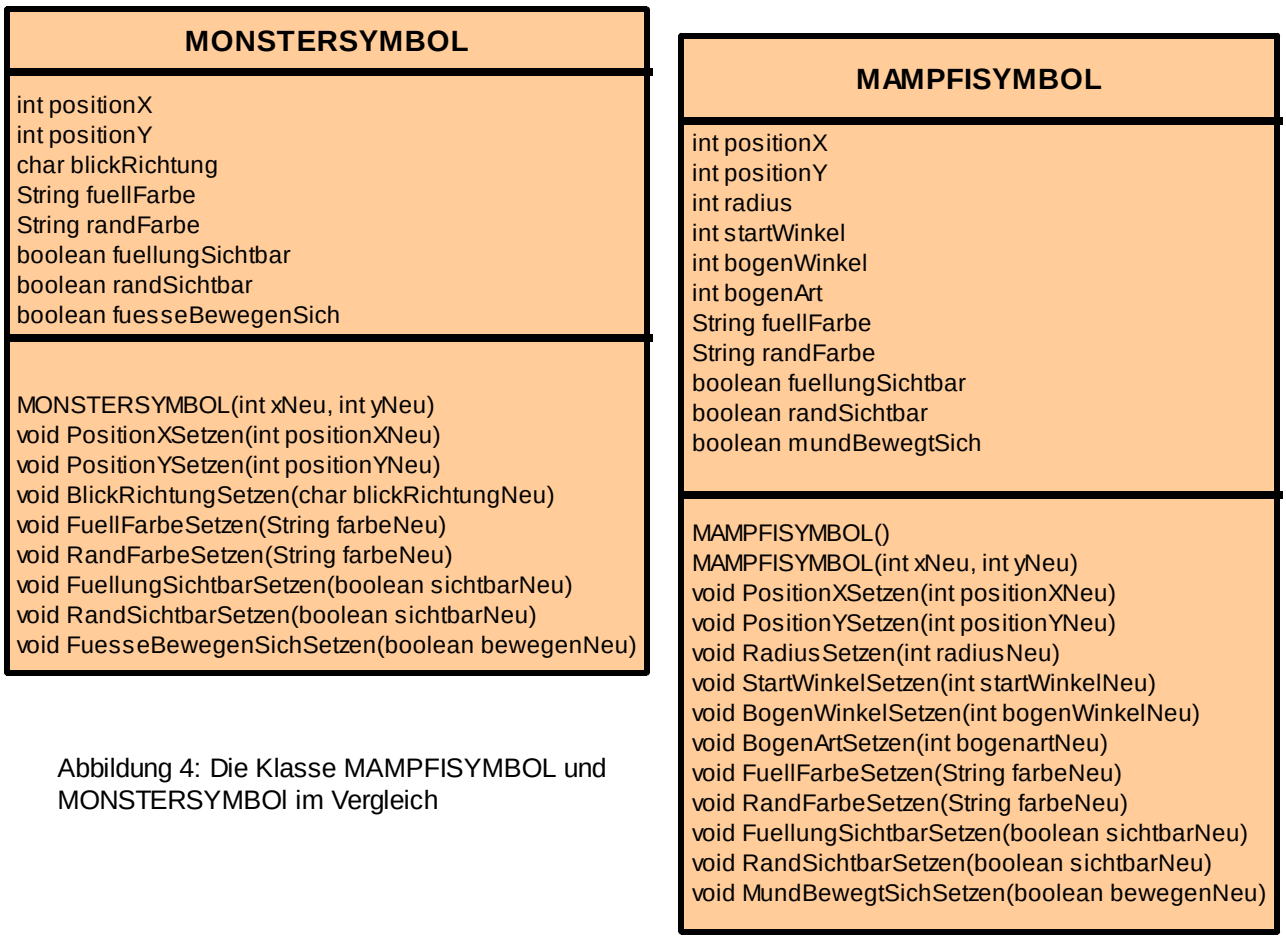


Abbildung 4: Die Klasse MAMPFISYMBOL und MONSTERSYMBOl im Vergleich

Es gibt eine Möglichkeit hinsichtlich des Punktes d) noch mehr Gemeinsamkeiten zwischen den Klassen MONSTER und MAMPFI herzustellen.

Aufgabe 17.2
Überlege dir, wie man durch das Ergänzen einer neuen Methode in den Klassen MAMPFI und MONSTER, bei allen Methoden die für das Gehen, die Änderung der Blickrichtung und der Verwundbarkeit zuständig sind, identische Methodenrümpfe herstellen kann.

Eine Möglichkeit noch mehr Gemeinsamkeiten zwischen den Klassen MONSTER und MAMPFI zu schaffen ist, eine neue Methode *SymbolAktualisieren* zu ergänzen. Sie wird am Ende aller Methoden, die für das Gehen, die Änderung der Blickrichtung und der Verwundbarkeit zuständig sind, aufgerufen.
Durch diese Änderung werden nur in der Methode *SymbolAktualisieren* (und dem Konstruktor) Methoden des Objekts der Symbolklasse aufgerufen. Alle anderen Methoden (*NachNordenBlicken*, *NachOstenBlicken*, ..., *VerwundbarSetzen, NachNordenGehen, NachOstenGehen,* ...) sind in der Klasse MAMPFI und der Klasse MONSTER identisch.

Aufgabe 17.3
Ergänze in der Klasse MAMPFI eine Methode *SymbolAktualisieren*, die oben beschriebenes leistet. Teste nach jeder Änderung, ob die Darstellung von Mampfi weiterhin korrekt ist.
Gehe bei den Quelltextänderungen der Klasse MAMPFI wie folgt vor:
Teil 1:
Beginne mit der Methode *VerwundbarSetzen*: Lagere alles, was mit der Kommunikation mit dem Mampfisymbol zu tun hat in die Methode *SymbolAktualisieren* aus und ergänze am Ende der Methode *VerwundbarSetzen* den Aufruf der Methode *SymbolAktualisieren*.
Teil 2:
Verfahre analog mit den "Bewegen-Methoden". Beachte dabei, dass die Aufrufe der Methoden des Mampfisymbols nicht immer die gleichen sind! Mit zwei Methodenaufrufen kann man jedoch immer alle vier Fälle abdecken.
Teil 3:
Verfahre analog mit den "Blicken-Methoden". Beachte dabei, dass bei jeder dieser Methoden der Aufruf der Methode *StartWinkelSetzen* einen unterschiedlichen Eingabewert hat. So müssen in der neuen Methode *SymbolAktualisieren* vier unterschiedliche Fälle untersucht werden. Die kürzeste Formulierung erfolgt mit der Mehrfachauswahl.

Solltest du Hilfe benötigen, findest du im Anhang ein Struktogramm.

Bist du sicher, dass du alle Stellen an denen eine Änderung notwendig ist gefunden hast? Suche über den Menüpunkt "Bearbeiten-->Suchen" alle Stellen an denen eine Botschaft an das Symbol geschickt wird. Suche die Zeichenkette "symbol.". Findest du Stellen außerhalb des Konstruktors und der Methode *SymbolAktualisieren,* musst du noch weitere Änderungen vornehmen.

17.2 Vererbung

Durch die Einführung der Methode *SymbolAktualisieren* wurde im letzten Kapitel erreicht, dass die Klassen MAMPFI und MONSTER (siehe Klassendiagramm auf S. 2) in den mit blauer Schriftfarbe dargestellten Methoden und Attributen exakt übereinstimmen.
Eine einfache Vorgehensweise wäre nun, den Quellcode der Klasse MAMPFI zu kopieren, die Methode *SymbolAktualisieren* und den Konstruktor umzuschreiben, und dann die zusätzlichen Attribute und Methoden der Klasse MONSTER zu ergänzen.

Aufgabe 17.4
Welche Nachteile können sich aus dieser Vorgehensweise ergeben?

Bisher war es immer so, dass die Klassen im Rahmen der Spielentwicklung ergänzt bzw. optimiert werden müssen. Hätte man den identischen Quellcode in zwei Klassen besteht der Aufwand diesen an zwei Stellen ändern zu müssen und die Gefahr dass es zu Inkonsistenzen kommt, wenn man bei Änderungen nicht sorgfältig ist. Die objektorientierte Modellierung bietet ein Konzept an, dass die beiden geschilderten Nachteile vermeidet: Generalisierung und Spezialisierung.
Mit den blau dargestellten Attributen und Methoden im Klassendiagramm auf Seite 2 wurden Eigenschaften und Fähigkeiten herausgearbeitet, die generell für alle Spielfiguren gelten sollen. Fasst man diese Attribute und Methoden in einer Klasse SPIELFIGUR zusammen, dann ist dies das Ergebnis einer Verallgemeinerung (**Generalisierung**). In die andere Richtung gesehen sind Objekte der Klassen MAMPFI und MONSTER **Spezialisierungen** der Klasse SPIELFIGUR, denn neben den Gemeinsamkeiten gibt es Unterschiede in Attributen und Methoden. In UML wird eine Generalisierungs- Spezialisierungsbeziehung mit einem nicht ausgefüllten dreieckigen Pfeil in Richtung der Spezialisierung dargestellt. In dem Klassendiagramm auf der folgenden Seite würde man also lesen „Die Klasse MONSTER ist eine Spezialisierung der Klasse SPIELFIGUR.“
Die Klasse SPIELFIGUR ist in diesem Kontext eine **Oberklasse**, die Klassen MAMPFI und MONSTER sind **Unterklassen**.

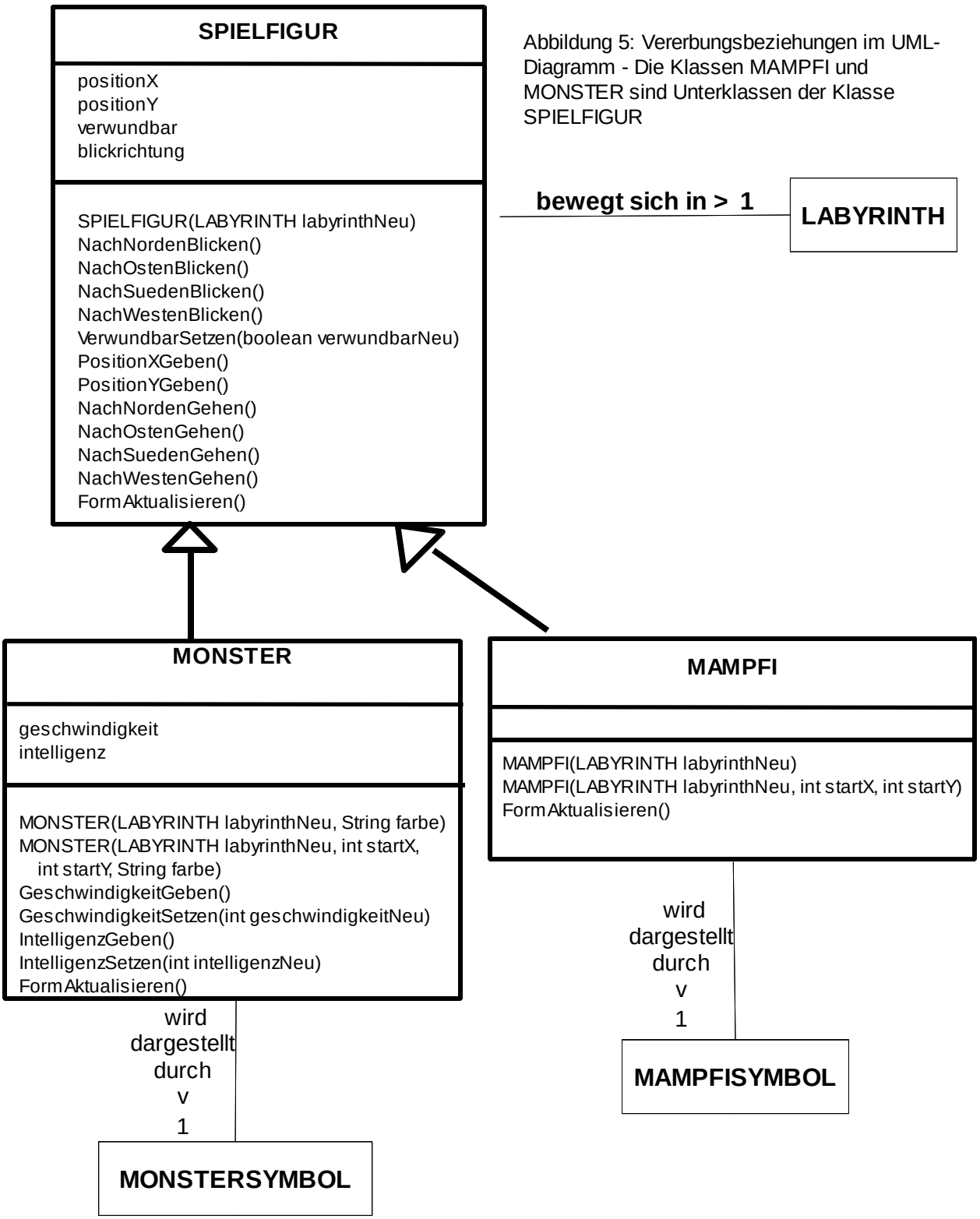


Abbildung 5: Vererbungsbeziehungen im UML-Diagramm - Die Klassen MAMPFI und MONSTER sind Unterklassen der Klasse SPIELFIGUR

Wichtige Beschaffenheiten von Unterklassen:
Die in der Oberklasse deklarierten Attribute und Methoden werden bei den Unterklassen nicht aufgelistet. Dennoch haben alle Objekte der Klassen MAMPFI und MONSTER die Attribute positionX, positionY usw.
Man nennt die Generalisierungs- Spezialisierungsbeziehung auch **Vererbung**sbeziehung. Die Objekte der Klassen MAMPFI und MONSTER erben alle Attribute und Methoden der Klasse SPIELFIGUR.
In den Unterklassen können neue Attribute und Methoden hinzugefügt werden.

Aufgabe 17.5

Warum muss die Methode *SymbolAktualisieren* in der Klasse SPIELFIGUR stehen, obwohl sie in den Unterklassen unterschiedliche Methodenrümpfe hat?

Da die Methode *SymbolAktualisieren* von vielen anderen Methoden der Klasse SPIELFIGUR aufgerufen wird, muss sie auch dort vereinbart (deklariert) werden. Das Vererbungskonzept bietet die Möglichkeit, dass in Unterklassen Methoden neu definiert **(überschrieben)** werden können. Somit ist es möglich in den Klassen MONSTER und MAMPFI jeweils einen passenden Methodenrumpf zu *SymbolAktualisieren* zu schreiben. Der Methodenkopf muss jedoch in der Ober- und Unterklasse identisch sein.

Aufgabe 17.6

In Abbildung 6 siehst du ein weiteres Beispiel für eine Vererbungsbeziehung außerhalb des Projekts Krümel & Monster. Überlege dir selbst zwei weitere Beispiele. Zeichne das UML-Diagramm. Die Klassen sollten mindestens zwei gemeinsame Attribute und eine gemeinsame Methode haben.

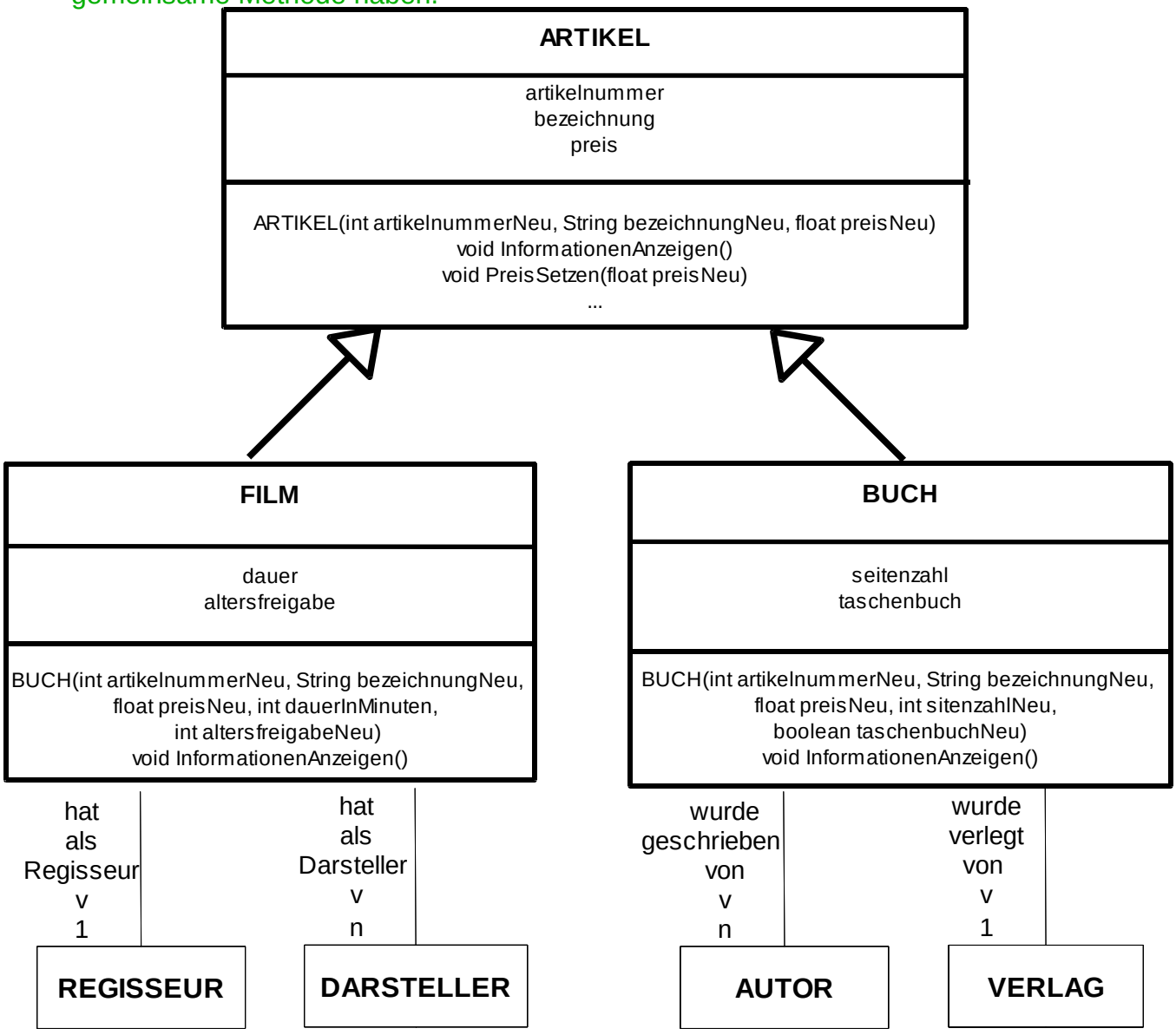


Abbildung 6: Vererbungsbeziehungen am Beispiel eines Online-Versands, der Bücher und Filme vertreibt. Die Methode InformationAnzeigen zeigt unterschiedliche Informationen an, je nachdem es sich um ein Buch oder um einen Film handelt.

17.3 Vererbung in Java umsetzen

Für die Oberklasse gibt es keine Besonderheiten zu beachten.

Aufgabe 17.7
Schreibe eine Klasse SPIELFIGUR entsprechend dem Klassendiagramm oben. Folgende effektive Vorgehensweise bietet sich an.

a) Erzeuge eine Klasse SPIELFIGUR in BlueJ.
b) Kopiere den gesamten Quelltext der Klasse MAMPFI und füge ihn als Quelltext der Klasse SPIELFIGUR ein.
c) Ersetze den Klassennamen MAMPFI durch SPIELFIGUR.
d) Ersetze den Bezeichner der Konstruktoren durch SPIELFIGUR.
e) Lösche die Deklaration des Referenzattributs symbol vom Datentyp MAMPFISYMBOL.
f) Die Initialisierund der Attribute verwundbar und blickrichtung kann als voreingestellter Wert bestehen bleiben. Wichtig ist jedoch später in den Unterklassen Mampfi und den Monstern unterschiedliche Startwerte beim Attribut verwundbar zu geben.
g) Lösche im Konstruktor die Zeile, in der das Mampfisymbol erzeugt und dem Referenzattribut symbol zugewiesen wird.
   Lösche alle Zeilen, in denen Methodenaufrufe an die Referenz symbol gesendet werden.
h) Lösche den gesamten Rumpf der Methode *SymbolAktualisieren*. (Das Paar mit den geschweiften Klammer muss stehen bleiben!)
i) Prüfe durch Übersetzen, ob deine Klasse Fehler enthält.
j) Verständnisfrage: Warum siehst du in der Anzeige keine Spielfigur, wenn du ein Objekt der Klasse SPIELFIGUR erzeugst?

Hinweis:
Solltest du Schwierigkeiten haben oder unsicher sein, kannst du im Anhang den Anfang des Quellcodes der Klasse SPIELFIGUR finden.

In der Unterklasse wird die Vererbung durch das Schlüsselwort **extends**[1] umgesetzt. So lautet die erste Zeile der Klasse MAMPFI

```
class MAMPFI extends SPIELFIGUR
```

mit der Bedeutung „Die Klasse MAMPFI spezialisiert (erweitert) die Klasse SPIELFIGUR.“ oder Die Klasse MAMPFI erbt von der Klasse SPIELFIGUR.“

An Attributen gibt es nur ein Referenzattribut, das zur Darstellung von Mampfi in der Anzeige dient. Neben der Methode *SymbolAktualisieren*, die in Kapitel 17.1 ausführlich behandelt wurde, und von dort übernommen werden kann gibt es noch eine Besonderheit beim Konstruktor:

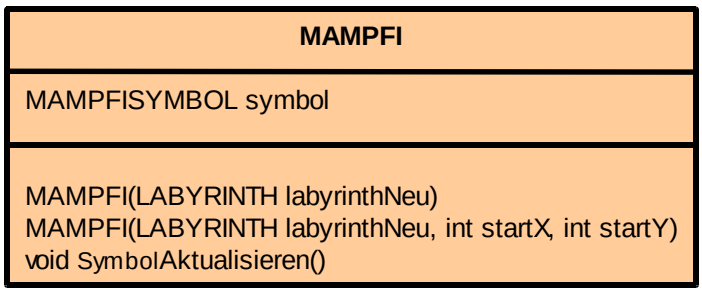


Abbildung 7: Erweitertes Diagramm der Klasse MAMPFI - auf Grund der Vererbung gibt es hier nur ein Referenzattribut und drei Methoden

1 engl. erweitert

Der Konstruktor dient zum Initialisieren[2] der Attribute. Es müssen alle Attribute, auch jene, die bereits in der Oberklasse festgelegt sind, initialisiert werden. Dies geschieht dadurch, dass man den Konstruktor der Oberklasse mit dem Schlüsselwort ***super*** aufruft. Die folgende Abbildung zeigt den Quelltext der Klasse MAMPFI.

```
public MAMPFI(LABYRINTH labyrinthNeu, int startX, int startY)
 {
      super(labyrinthNeu, startX, startY);
      verwundbar = true;

      symbol = new MAMPFISYMBOL();
      symbol.RadiusSetzen(25);
      symbol.StartWinkelSetzen(120);
      symbol.BogenWinkelSetzen(300);
      symbol.BogenArtSetzen(2);
      symbol.FuellFarbeSetzen("gelb");
      symbol.FuellungSichtbarSetzen(true);
      symbol.RandFarbeSetzen("gelb");
      symbol.RandSichtbarSetzen(false);
      symbol.MundBewegtSichSetzen(true);
      symbol.PositionXSetzen(positionX);
      symbol.PositionYSetzen(positionY);
 }
```

Aufruf des Konstruktors der Oberklasse mit Hilfe des Schlüsselworts "super"

Fehlermeldung:
verwundbar bzw. positionX
bzw. positionY
has **private access**
in SPIELFIGUR

Abbildung 8: Quelltext des Konstruktors MAMPFI(LABYRINTH labyrinthNeu, int startX, int startY)

Aufgabe 17.8

Passe nun die Klasse MAMPFI an. Folgende effektive Vorgehensweise bietet sich an.

a) Lösche in der Klasse MAMPFI alle Attribute und Methoden, die nicht im Klassendiagramm in Abbildung 7 stehen (denn sie sind schon in der Oberklasse SPIELFIGUR)

b) Ist der Konstruktor *MAMPFI(LABYRINTH labyrinthNeu)* auf Basis des anderen Konstruktors implementiert, dann müssen hier keine Änderungen vorgenommen werden (Quelltext diese Konstruktors siehe Anhang, Erklärungen zum dort verwendeten Schlüsselwort "this" siehe Kapitel 14 Anhang B). Da dieser Konstruktor im weiteren Verlauf nicht mehr verwendet wird, kann er auch gelöscht werden.

c) Verkürze den Konstruktor entsprechend Abbildung 8: Der Konstruktor der Oberklasse SPIELFIGUR wird mit *super* aufgerufen und alle Initialisierungen, die im Konstruktor der Oberklasse durchgeführt werden, werden gelöscht.

d) Übersetze und überprüfe, ob du die gleiche Fehlermeldung wie in Abbildung 8 dargestellt erhältst

Aufgabe 17.9 – Verständnisfragen

a) Mit dem Schlüsselwort super wird der Konstruktor der Oberklasse aufgerufen. Warum muss man den Namen der Oberklasse nicht angeben?

b) Was bedeutet die Fehlermeldung? An welcher Stelle muss man Änderungen vornehmen, um den Fehler zu beheben.

2 Setzen der Werte auf einen (sinnvollen) Startwert

Ein neues Zugriffsrecht

Wir haben allen Attributen das Zugriffsrecht private gegeben, damit von außen nicht direkt darauf zugegriffen werden kann. Nun sagt jedoch die Fehlermeldung aus Abbildung 8, dass die Unterklasse (MAMPFI) nicht auf das Attribut zugreifen kann, das es von der Oberklasse (SPIELFIGUR) geerbt hat. Das ist eine zu starke Einschränkung! Für diesen Fall gibt es das Zugriffsrecht **protected**. Es erlaubt allen Unterklassen den Zugriff auf mit protected geschützte Attribute (und Methoden) der Oberklasse. Abbildung 9 gibt einen Überblick über die Zugriffsrechte.

Zugriffsrecht	Zugriff aus gleicher Klasse	Zugriff von einer Subklasse	Zugriff einer beliebigen Klasse
public	ja	ja	ja
protected	ja	ja	nein
private	ja	nein	nein

Abbildung 9: Überblick der Bedeutung der Zugriffsrechte public, protected und private

Aufgabe 17.10

a) Ändere die Zugriffsrechte aller Attribute der Klasse SPIELFIGUR auf protected und übersetze diese Klasse.

b) Übersetze erneut die Klasse MAMPFI. Die vorher aufgetretenen Fehlermeldungen müssten verschwunden sein.

c) Gehe beim Testen diesmal wie folgt vor, um ein besseres Verständnis für die Vererbung zu erhalten:
- Erzeuge ein Objekt der Klasse LABYRINTH mit dem Namen labyrinth
- Erzeuge ein Objekt der Klasse MAMPFI mit dem Namen mampfi, der sich im soeben erzeugten Labyrinth bewegt
- Öffne den Objektinspektor vom Objekt mampfi (Abbildung 10). Du kannst sehen, dass er sechs Attribute hat, obwohl in der Klasse MAMPFI nur eines deklariert ist. Die anderen fünf Attribute hat Mampfi von der Oberklasse SPIELFIGUR geerbt.

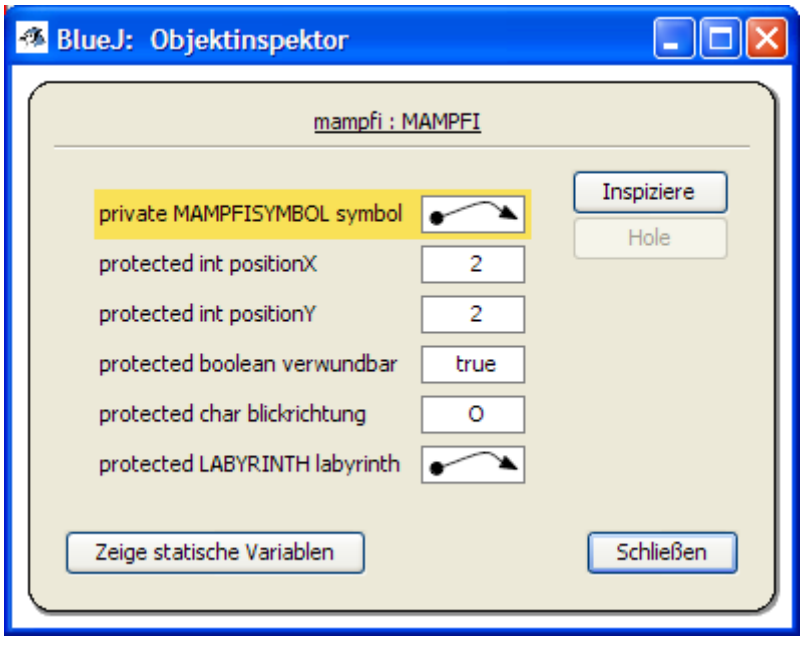


Abbildung 10: Objektinspektor von Mampfi – Von den sechs angezeigten Attributen sind fünf von der Oberklasse SPIELFIGUR geerbt.

- Gehe nun mit der rechten Maustaste auf die Objektkarte von Mampfi um Methoden aufzurufen. Du hast einerseits die in der Klasse MAMPFI implementierte Methode *SymbolAktualisieren* zur Verfügung und andererseits über "geerbt von SPIELFIGUR" alle Methoden der Klasse SPIELFIGUR (Abbildung 11).

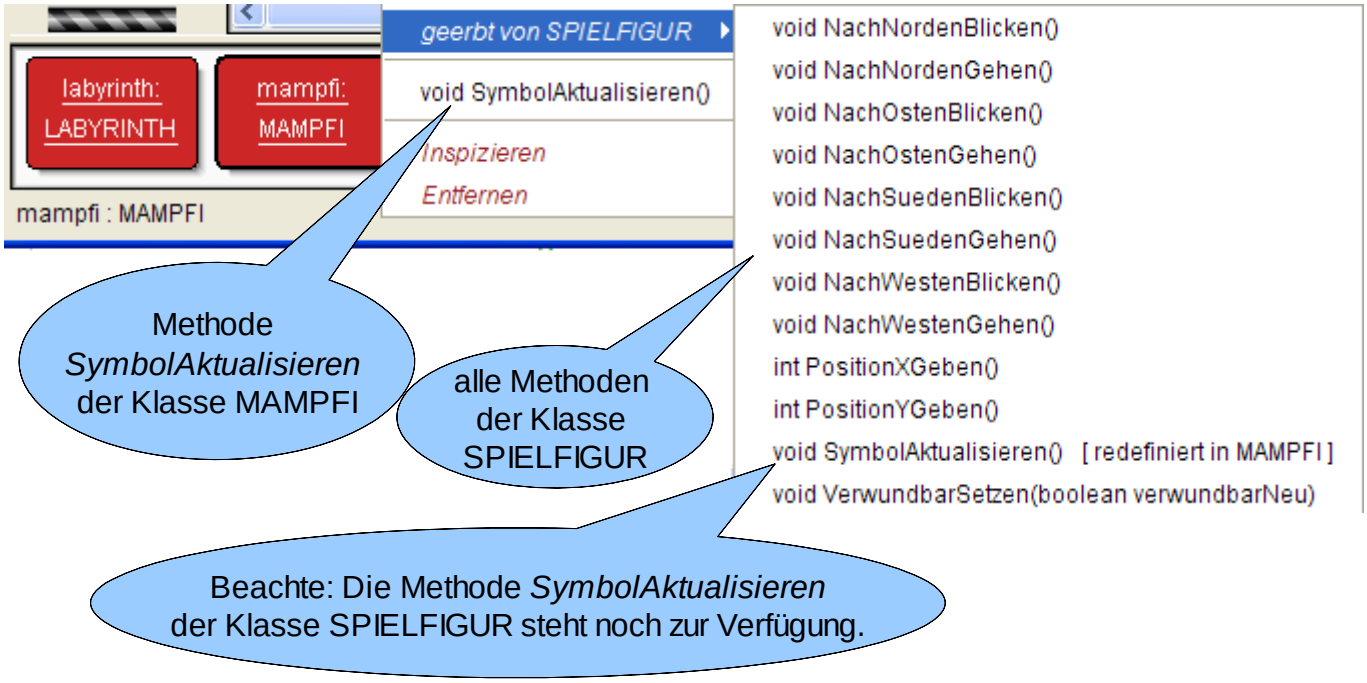


Abbildung 11:
Kontextmenü des Objekts Mampfi mit der eigenen Methode und allen Methoden der Oberklasse.

Aufgabe 17.11
Erstelle ein erweitertes Klassendiagramm der Klasse MONSTER, das die Vererbung von der Klasse SPIELFIGUR ausnutzt.

Aufgabe 17.12

Stelle zunächst sicher, dass du die Klasse MONSTERSYMBOL bereits in dein BlueJ-Projekt integriert hast.

Wenn du dich sicher im programmieren fühlst, dann erzeuge eine Klasse MONSTER in deinem BlueJ-Projekt entsprechend deiner Planung aus Aufgabe 17.11.

Falls du Hilfestellungen benötigst, enthalten folgende Teilaufgaben kleinschrittige Anweisungen.

a) Erzeuge eine Klasse MONSTER in deinem BlueJ-Projekt.

b) Sorge dafür, dass die Klasse MONSTER eine Unterklasse der Klasse SPIELFIGUR ist.

c) Deklariere die Attribute entsprechend dem Klassendiagramm in Abbildung 12.

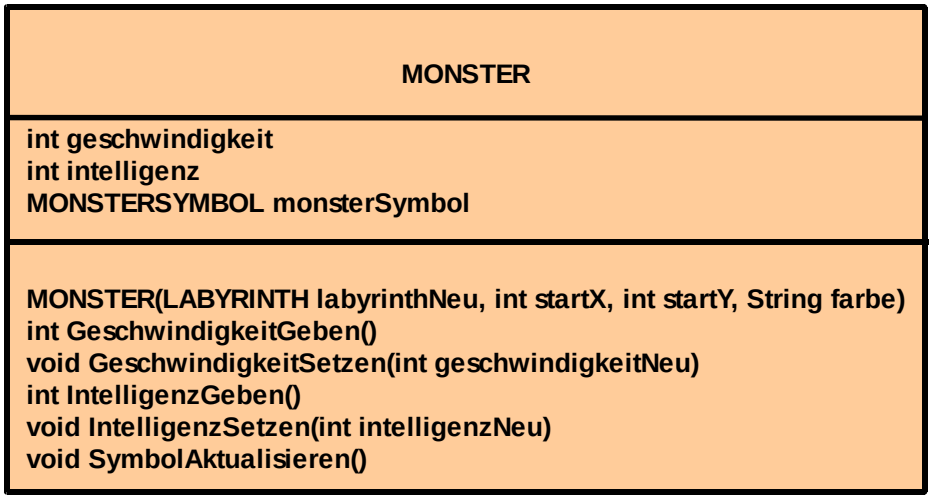


Abbildung 12: Attribute und Methoden der Klasse MONSTER, die von der SPIELFIGUR erbt.

d) Schreibe den Konstruktor. Wichtige Punkte sind hierbei:
- Der Konstruktor der Oberklasse muss aufgerufen werden.
- Das Attribut verwundbar muss auf false initialisiert werden.
- Die Attribute intelligenz und geschwindigkeit werden erst an späterer Stelle näher betrachtet. Sinnvolle Anfangswerte sind 0 für die Intelligenz (d.h. keine Intelligenz) und 5 für die Geschwindigkeit (d.h. eine mittlere Geschwindigkeit auf einer Skala von 1-10: Geschwindigkeitsangaben kleiner als 5 wären langsamer als Mampfi und größer als 5 wären schneller als Mampfi)
- Ein Objekt der Klasse MONSTERSYMBOL muss erzeugt werden. Beachte dabei die Eingangsparameter des Konstruktors der Klasse MONSTERSYMBOL (Klassendiagramm in Abbildung 1).
- Das Monstersymbol muss entsprechend der Farbe (Eingabewert des Konstruktors MONSTER) und der Blickrichtung angepasst werden. Wie die Methoden heißen und welche Eingangsparameter sie haben kann man dem Klassendiagramm in Abbildung 1 entnehmen.

e) Schreibe nun die Methode *SymbolAktualisieren*, damit ein Testen leichter möglich ist. Wie bei Mampfi hat diese Methode 3 Teile:
- Teil 1: Die Monster sollten unterschiedlich dargestellt werden, je nachdem ob sie verwundbar sind oder nicht. Wie du dies machst, kannst du frei entscheiden. Eine Möglichkeit wäre beispielsweise durch eine unterschiedliche Randfarbe**.**
- Teil 2: Das Aktualisieren der Position nach einer Bewegung unterscheidet sich in der Klasse MONSTER nicht von dem in der Klasse MAMPFI, außer natürlich dem anderen Referenzattribut.
- Teil 3: Das Ändern des Monstersymbols bei einer Änderung der Blickrichtung unterscheidet sich stark von dem Vorgehen in der Klasse MAMPFI, da die Klasse

MONSTERSYMBOL nicht die Methode *StartWinkelSetzen* anbietet. Welche Methode der Abbildung 1 ist passend? Warum ist eine Mehrfachauswahl nicht mehr nötig?

f) Teste ausführlich, indem du ein Labyrinth und mindestens zwei Monster erzeugst und dann bei den Monstern alle möglichen Methodenaufrufe testest.

g) Die verbleibenden vier Methoden entsprechend Abbildung 12 werden vorerst noch nicht in Erscheinung treten. Entscheide selbst, ob du Zeit hast sie schon jetzt zu implementieren. Falls ja, sorge jeweils bei den Setzen Methoden, dass nur Eingabewerte in einem begrenzten Bereich zulässig sind, z.B. bei der Geschwindigkeit Werte zwischen 1 und 9: Vier Geschwindigkeitsstufen bei denen die Monster langsamer sind und vier bei sie schneller als Mampfi sind.

17.4 Einmal ändern, mehrfach nutzen

Um den Vorteil des Vererbungskonzepts zu verdeutlichen soll das BlueJ-Projekt so ergänzt werden, dass sowohl die Monster als auch Mampfi die Methode *BlickrichtungGeben* anbieten.

Aufgabe 17.13

a) Wo muss die Methode BlickrichtungGeben ergänzt werden? Es wird die Lösung mit dem minimalen Aufwand gesucht!

b) Ergänze die Methode in deinem BlueJ-Projekt und teste sie, indem du ein Labyrinth, einen Mampfi und ein Monster erzeugst und die Methode bei verschiedenen Objekten mit verschiedenen Werten des Attributs blickrichtung aufrufst.

17.4 Zusammenfassung

Für die Oberklasse gibt es keine Besonderheiten zu beachten.

Aufgabe 17.14

In den Kapiteln 12.1 bis 12.4 wurde das Konzept der Vererbung vorgestellt und am Beispiel der Klasse SPIELFIGUR in Java umgesetzt. Erstelle eine Zusammenfassung dieser für dich neuen Inhalte. Die Zusammenfassung sollte mindestens folgende Bestandteile beinhalten:

a) Erkläre die Begriffe Generalisierung, Spezialisierung, Vererbung, Oberklasse, Unterklasse, Überschreiben von Methoden

b) Erkläre die Bedeutung der beiden neuen Schlüsselwörter, die du bei der Umsetzung der Vererbung in Java verwendet hast.

c) Erkläre den Sinn bzw. Nutzen des Vererbungskonzeptes.

d) Erkläre die Wirkungsweise des Zugriffsrechts `protected`.

Aufgabe 17.15

In der Fachliteratur ist stets der Ausdruck zu finden, dass durch die Deklaration einer Methode in der Unterklasse, diejenige mit dem gleichen Methodenkopf in der Oberklasse **überschrieben** wird. Führe die in den Teilaufgaben a) bis d) beschriebenen Anweisungen aus und erläutere danach, warum der Begriff überdecken statt überschreiben besser geeignet wäre.

Hinweis:

Führe die folgenden Änderungen bei einer Kopie deines aktuellen Projekts durch. Die folgenden Teilaufgaben führen nicht das Projekt Krümel & Monster fort, sondern dienen dem Verständnis des Vererbungskonzepts.

a) Ergänze in der Klasse SPIELFIGUR im Rumpf der Methode *SymbolAktualisieren* die Zeile:

```
System.out.println("Die Methode SymbolAktualisieren der Klasse
      SPIELSTEURUNG wurde aufgerufen");
```

b) Ergänze in der Klasse MAMPFI folgende Methode:

```
void Test()
  {
      super.SymbolAktualisieren();
  }
```

c) Übersetze die beiden veränderten Klassen, erzeuge ein Objekt der Klasse LABYRINTH und ein Objekt der Klasse MAMPFI.

d) Hole das Fenster der Standardausgabe – es ist beschriftet mit BlueJ-Konsole und öffnet sich bei der ersten Objekterzeugung - in den Vordergrund und rufe dann die Methode Test auf. Beachte die Änderung in der Standardausgabe.

Aufgabe 17.16

Solltest du das Vererbungskonzept vertieft üben wollen, bieten sich im Buch "Informatik II", Oldenbourg Verlag die Aufgaben S. 128/8 und 9 an.

17.5 Abstrakte Klassen: Vertiefung zur Vererbung

Neue Klasse in Kapitel 17 war SPIELFIGUR. Obwohl sie innerhalb des Projekts zur Reduzierung und leichteren Wartbarkeit des Quelltextes beiträgt, wird nie ein Objekt dieser Klasse (sondern nur ihrer Unterklassen) erzeugt.
Es ist auch nicht sinnvoll, dass ein Objekt der Klasse SPIELFIGUR erzeugt wird, da der Bauplan unvollständig ist. So gibt es in der Klasse SPIELFIGUR keinen sinnvollen Rumpf zur Methode *SymbolAktualisieren*(). Klassen, von denen keine Objekte erzeugt werden sollen nennt man **abstrakte Klassen**. Sie haben ausschließlich den Zweck Superklasse für andere Klassen zu sein.
Die Methode *SymbolAktualisieren*() in der Klasse SPIELFIGUR dient nur dazu, den Methodenkopf festzulegen, damit diese Methode innerhalb anderer Methoden der Klasse SPIELFIGUR aufgerufen werden kann. Da der Methodenrumpf mit zweckdienlichen Anweisungen erst in den Unterklassen festgelegt wird, wäre es sinnvoll diesen in der Klasse SPIELFIGUR wegzulassen. Solche Methoden werden als abstrakte Methoden bezeichnet. In jeder Unterklassen muss dann jeweils die abstrakte Methode der Oberklasse mit Methodenrumpf ausgearbeitet werden. Abstrakte Methode dürfen nur in abstrakten Klassen festgelegt werden. In den UML Diagrammen werden abstrakte Klassen und Methoden mit dem Zusatz {abstract} versehen (siehe blaue Hervorhebungen in der Abbildung 13).

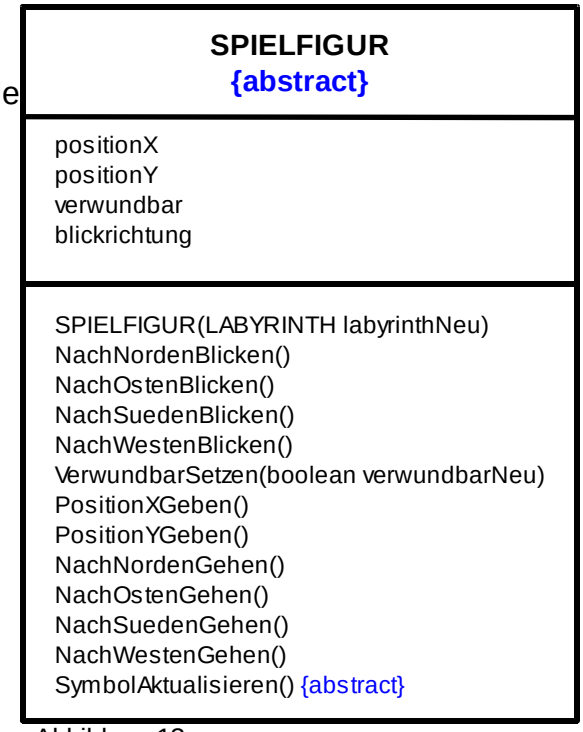


Abbildung 13:
Die Klasse SPIELFIGUR als abstrakte Klasse

In Java ist das Umsetzen abstakter Klassen bzw. Methoden durch das Voranstellen des Schlüsselwortes `abstract` vor das Schlüsselwort `class` bzw. vor dem Datentyp des Ausgabewertes möglich. Bei abstrakten Methoden muss der Methodenkopf durch einen Strichpunkt abgeschlossen werden, der Methodenrumpf entfällt. Die Änderungen in der Klasse Spielfigur können dem folgenden Quelltextauszug entnommen werden:

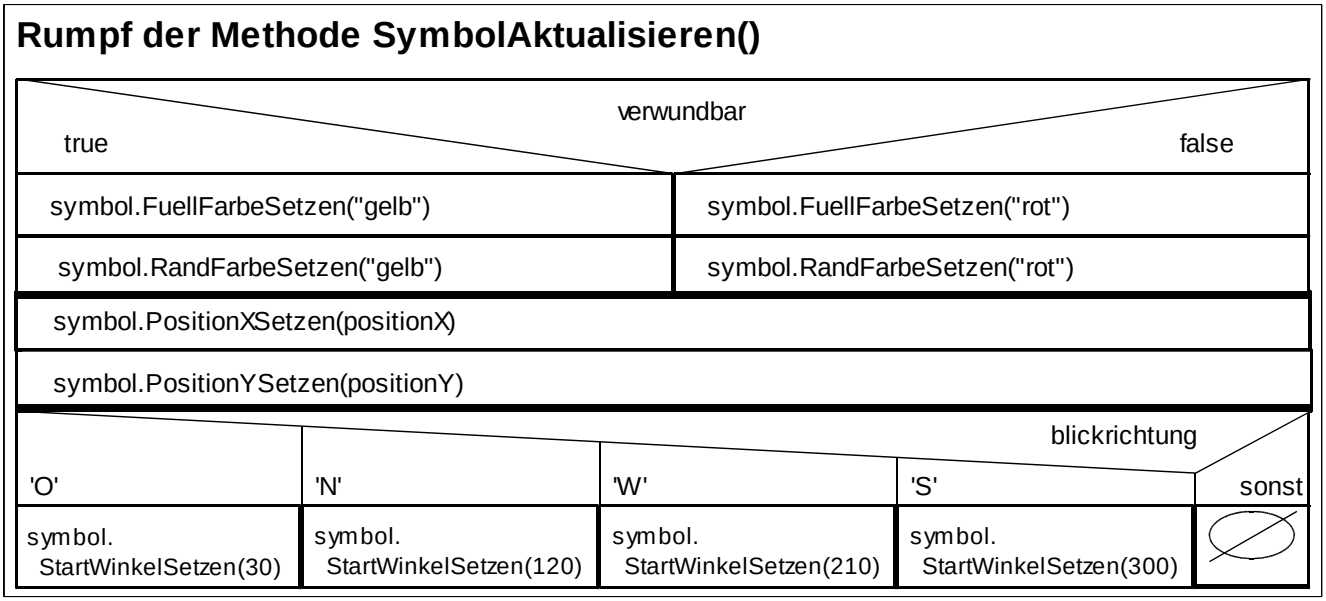
```
abstract class SPIELFIGUR
{
    // Attribute
    ...
    // Referenzattribute
    ...
    // Konstruktor
    ...
    // Methoden
     abstract void SymbolAktualisieren();

    void NachNordenBlicken()
    {
        blickrichtung = 'N';
        SymbolAktualisieren();
    }
    ...
}
```

Aufgabe 17.17
Ändere die Klasse SPIELFIGUR zu einer abstrakten Klasse um. Dazu musst du wie oben gezeigt das Schlüsselwort `abstract` an den richtigen Stellen ergänzen und den Rumpf der Methode *SymbolAktualisieren* löschen.
Übersetze und teste. Suche auch Antworten zu folgenden Fragen:
a) Wie werden im Klassendiagramm von BlueJ abstrakte Klassen dargestellt?
b) Wie lautet die Fehlermeldung, wenn eine abstrakte Klasse in einer Unterklasse nicht ausgearbeitet ist. Kommentiere dazu die *SymbolAktualisieren* in der Klasse MAMPFI aus.

Anhang A: Tipps zu den Aufgaben

Aufgabe 17.1:
Struktogramm zur Methode SymbolAktualisieren der Klasse MAMPFI:



Aufgabe 17.8b
Quelltext des Konstruktors MAMPFI(LABYRINTH labyrinthNeu)

```
public MAMPFI(LABYRINTH labyrinthNeu)
{
    this(labyrinthNeu, 2, 2);
}
```

Aufgabe 17.12e
Methode SymbolAktualisieren der Klasse MONSTER

```
void SymbolAktualisieren()
{
  //Teil 1: notwendig in der Methode VerwundbarSetzen
  if (verwundbar == true)
  {
     monsterSymbol.RandFarbeSetzen("gelb");
  }
  else
  {
     monsterSymbol.RandFarbeSetzen("schwarz");
  }
  //Teil 2: notwendig in den "Gehen-"Methode
  monsterSymbol.PositionXSetzen(positionX);
  monsterSymbol.PositionYSetzen(positionY);

  // Teil 3: notwendig in den "Blicken-"Methode
  monsterSymbol.BlickRichtungSetzen(blickrichtung);
 }
```

Anhang B: Farbwerte

Folgende Farbwerte sind innerhalb des Projekts Krümel & Monster möglich:
"blau", "dunkelblau"
"gelb", "dunkelgelb"
"gruen", dunkelgruen"
"rot", "dunkelrot"
"orange", "dunkelorange“
"pink", "dunkelpink"
"magenta", "dunkelmagenta"
"grau", "dunkelgrau "
"weiss", "schwarz", "zufall"