AntMe! - Antout

Sommeruni 2019

Diane Hanke, Alexander Korzec, Sönke Schmidt

25. Juli 2019

Inhaltsverzeichnis

Benötigte Programme . . . 2
Erstellung eines Ameisenvolkes . . . 3
Einbindung eines Ameisenvolkes in AntMe! . . . 4
Link zur Beschreibung von Befehlen, Eigenschaften und Ereignissen . . . 5
Neues Ameisenvolk tut nichts!? . . . 5
Einsammeln von Äpfeln . . . 5
Zuckerstraße . . . 6
Feintuning: Eigenschaften von Ameisen verändern . . . 8
Spezialisierung: Erstellung von weiteren Berufsgruppen . . . 9
Maßnahmen gegen Verhungern . . . 10
Wanzen angreifen und Steuerung in Abhängigkeit der Kastenzugehörigkeit . . . 12
Eigenschaften von Wanzen . . . 13
Rudimentäres Debuggen . . . 14
Ist das alles? Ist unser Ameisenvolk nun perfekt? . . . 14

Ein bisschen Bla Bla zu Beginn

Willkommen,

wir freuen uns, dass du dich für AntMe entschieden hast und wünschen Dir viel Freude bei diesem Kurs.

Deine Mission ist es, ein Ameisenvolk, sprich eine *KI*, für AntMe! zu programmieren. AntMe! veranstalten wir schon seit langer Zeit und diese Veranstaltung hat jedes mal für viel Unterhaltung und gute Stimmung gesorgt. Außerdem ist die Programmierung der Ameisenvölker sooo einfach, dass man dadurch wunderbar Berührungsängste gegenüber dem Programmieren ablegen kann. Das zweitrangige Ziel ist es, alle deine Gegner auf dem Schlachtfeld zu vernichten und vorzuführen. Schließlich veranstalten wir einen Wettbewerb ;). Aber, wie schon gesagt, steht vor allem der Spaß im Vordergrund.

Diese Anleitung hat nicht die Ambition dir `C#` beizubringen oder dich in die Geheimnisse der objektorientierten Programmierung einzuführen. Wir möchten dir bloß eine kleine Starthilfe

anbieten, damit du, mit gar nicht so viel Kopfschmerzen, ein gar nicht so übles Ameisenvolk erstellen kannst.

Falls du Anmerkungen, Kritik oder Verbesserungsvorschläge zu dieser Anleitung hast, dann schreib einfach eine Mail an

mentoring@mi.fu-berlin.de.

Solltest du während des Workshops Hilfe bei deinen Ameisen brauchen, dann frage irgendeinen der Autoren dieser Anleitung. Die sollten dir in der Regel sogar weiterhelfen können :).

> *Hinweis:* Falls ihr dieser Anleitung linear bis zur letzten Seite folgt, erhaltet ihr ein ziemlich passables Ameisenvolk als Ergebnis. Eure Ameisen werden zwar konkurrenzfähig sein, aber mit hoher Wahrscheinlichkeit nicht das Turnier gewinnen. Ihr könnt natürlich auch in das Inhaltsverzeichnis schauen und euch einzelne Themen aussuchen, welche euch interessieren. Wenn ihr euch noch unsicher seid oder euch in die Kategorie „*waschechte Anfänger*“ einordnet, empfehlen wir die zuerst genannte Herangehensweise.

> *Hinweis:* Ihr werdet auf den folgenden Seiten Codebeispiele sehen. Diese könnt ihr natürlich nebenbei abtippen, damit ihr einen Ausgangspunkt für die Entwicklung eures Ameisenvolkes habt.

Welche Programme benötigen wir, um Ameisenmeister zu werden?

Ihr braucht auf jeden Fall:

- `AntMe!`

  https://service.antme.net/

- *und* `Microsoft .NET Framework 4`

  http://www.microsoft.com/de-de/download/details.aspx?id=17718

- *und* `Microsoft XNA Framework Redistributable 4.0 Refresh`

  http://www.microsoft.com/en-us/download/details.aspx?id=27598

Darüber hinaus benötigt ihr noch eine *Entwicklungsumgebung*. Wählt am besten eine der folgenden Entwicklungsumgebungen:

- *Entweder* SharpDevelop

  http://www.icsharpcode.net/opensource/sd/download/Default.aspx

  Diese Entwicklungsumgebung reicht für unsere Zwecke vollkommen aus und ist, im Gegensatz zu Visual Studio, relativ schnell installiert. Solltet ihr euch für SharpDevelop entscheiden, dann müsst ihr zusätzlich das *Framework* Microsoft Build Tools 2013 installieren, welches ihr hier findet:

  https://www.microsoft.com/en-us/download/details.aspx?id=40760

- *oder* Visual Studio

  https://www.visualstudio.com/de

Wie wird ein Ameisenvolk erstellt?

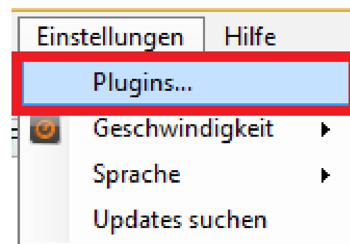
Folgt der Anleitung weiter unten. Eventuell könnt ihr die Schritte 2 und 3 überspringen, wenn nach dem Start der Software der Reiter „Freies Spiel“ verfügbar ist. Bei einer frischen AntMe! Installation müsst ihr in der Regel diesen Modus erst freischalten, indem ihr das *Plugin* „Freies Spiel“ aktiviert.

In *Schritt 7* müsst ihr eventuell einen Dateipfad angeben (grüner Kasten). Prinzipiell ist es egal, welchen ihr konkret angebt. Hauptsache, ihr findet euer Projekt wieder. Wenn ihr mit Visual Studio arbeitet, dann lohnt es sich, den Projektordner von Visual Studio anzugeben.

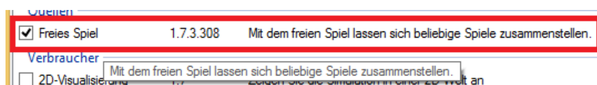
Solltet ihr SharpDevelop nutzen, dann wählt „*Nein*“ in *Schritt 8* aus (grüner Kasten) und öffnet das Projekt mit SharpDevelop manuell.

Schritt 1:

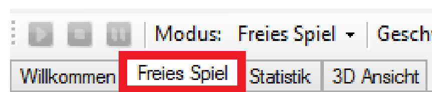
Schritt 2:



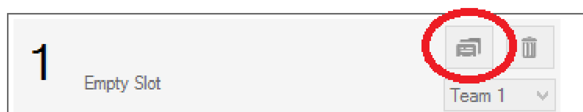
Schritt 3:



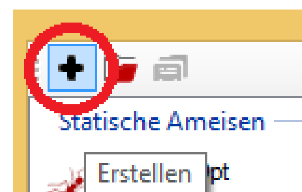
Schritt 4:



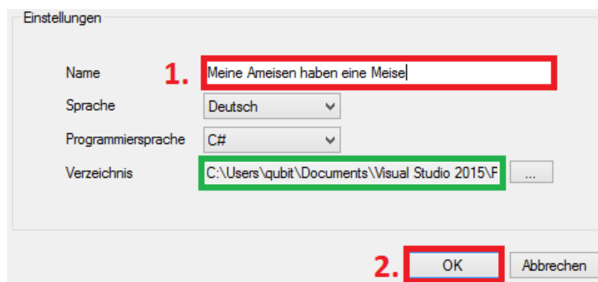
Schritt 5:



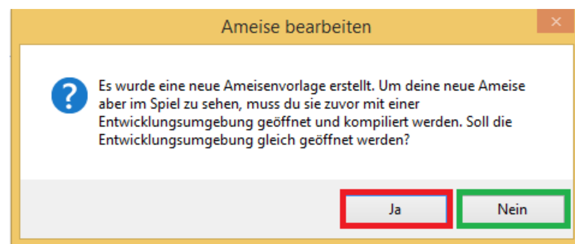
Schritt 6:



Schritt 7:



Schritt 8:

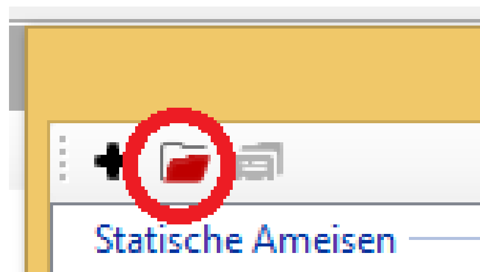


Wie binden wir unser Ameisenvolk in AntMe! ein?

Annahme: Ihr habt den Modus „Freies Spiel“ in der vorherigen Anleitung freigeschaltet.

Schritt 1: Befolgt die Schritte 1, 4 und 5 der Anleitung zum Thema „Wie wird ein Ameisenvolk erstellt?“.

Schritt 2:



Schritt 3: Es öffnet sich ein Dialogfenster, in dem ihr eine Datei auswählen sollt. Ihr müsst eine Datei mit dem Dateinamen `AntMe.Player.<Volkname>.dll` finden. In den Poolräumen befindet sich diese Datei vermutlich in folgendem Verzeichnis:

```
C:\Users\<Benutzername>\My Documents\Visual Studio
20XX\Projects\<Volkname>\bin
```

Anderenfalls müsst ihr nachsehen, wo euer Projektverzeichnis liegt und dort findet ihr im Ordner `\bin` die erwähnte Datei.

> *Hinweis:* Falls die genannte Datei im Projektordner nicht existiert, dann versucht euer Ameisenvolk in eurer Entwicklungsumgebung zu *kompilieren* (F5-Taste in `Visual Studio` drücken).

Schritt 4: Ab jetzt könnt ihr euer Ameisenvolk auswählen.

Wo findet man eine Beschreibung zu Befehlen, Eigenschaften und Ereignissen?

Schaut mal hier: https://wiki.antme.net/de/API1:Befehlsliste

Wir haben ein neues Ameisenvolk erstellt, aber unsere Ameisen stehen doof am Ameisenbau herum!

Ihr müsst zuerst das Verhalten eurer Ameisen programmieren! Ein neu erstelltes Ameisenvolk verhält sich nicht viel lebhafter, als ein Haufen Steine ;). Schreibt mal den Befehl `GeheGeradeaus()` in das Ereignis `Wartet()` innerhalb der geschweiften Klammern. Vergesst nicht das Semikolon!

Auf dem Spielfeld gibt es ja noch Äpfel. Wie können wir diese in unseren Ameisenbau transportieren?

Relevant für euer Problem sind die Ereignisse `Sieht(Obst obst)` und `ZielErreicht(Obst obst)`. Dort müsst ihr das gewünschte Verhalten eurer Ameisen beschreiben. Wir sollten außerdem beachten, dass wir einen Apfel nur dann effizient transportieren können, wenn wir genügend Ameisen zum Apfel schicken. Es bringt uns jedoch nichts, wenn wir zum Apfel Ameisen hinschicken, die aus Platzmangel beim Tragen nicht mehr mithelfen können.

Bevor wir einen Lösungsvorschlag präsentieren, möchten wir eine Kleinigkeit vorwegnehmen, die im Vortrag manchmal untergeht und einigen Anfängern Steine in den Weg legt auf dem Weg zur Weltherrschaft. Jede Ameise kommt in der Simulation irgendwann zum Zuge und versucht irgendwelche, mehr oder weniger sinnvollen, Ziele zu verwirklichen. Dabei muss sich die Ameise zwischen den Zügen merken, was sie damals für eine Aufgabe erhalten hat. Würde sie ihre Aufgabe zwischendurch wieder vergessen, dann hätten wir einen Haufen dementer Ameisen und viel Frust.

Die Zustandseigenschaft `Ziel` entspricht dem Gedächtnis einer Ameise. Dort merkt sich die Ameise, ob sie zu ihrem Ameisenbau zurück, eine Wanze angreifen oder gar ein Nickerchen machen wollte. Das `Ziel` einer Ameise wird vor allem durch *Befehle* beeinflusst. Wichtig ist an dieser Stelle, dass man das `Ziel` einer Ameise mit `==` erfragen kann. Eine arbeitslose Ameise erkennt man mit der *Abfrage* `Ziel == null`. Dabei steht `null` in diesem Kontext für Ziellosigkeit und der ganze Ausdruck kann in normales Deutsch als „Hast du, liebe Ameise, etwas zu tun?“ übersetzt werden. Wir werden gleich ein Beispiel sehen dazu, wie wir das hier erlangte Wissen zur konstruktiven Problemlösung nutzen können.

```
1  public override void Sieht(Obst obst)
2  {
3      if (BrauchtNochTräger(obst))
4      {
5          if (Ziel == null)
6          {
7              GeheZuZiel(obst);
8          }
9      }
10 }

12 public override void ZielErreicht(Obst obst)
```

```
13 {
14    Nimm(obst);
15    GeheZuBau();
16 }
```

Jetzt können wir endlich den Lösungsvorschlag genauer betrachten. Sieht eine Ameise einen Apfel, dann schaut sie in `Sieht(Obst obst)` nach, was sie tun soll. Zunächst findet sie mit `BrauchtNochTräger(obst)` in Zeile 3 heraus, ob noch weitere Träger benötigt werden. Danach überprüft die Ameise mit `Ziel == null` in Zeile 5, ob sie nichts zu tun hat. Hat die Ameise keine Aufgabe, setzt sie in Zeile 7 mit `GeheZuZiel(obst)` den gesehenen Apfel als ihr Ziel. Alles in einem, haben wir in `C#` genau das übersetzt, was wir haben wollten.

Kommt die Ameise beim Apfel irgendwann an, tritt das Ereignis `ZielErreicht(Obst obst)` ein. Die Ameise nimmt mit `Nimm(obst)` den Apfel auf und setzt als ihr Ziel mit `GeheZuBau()` den eigenen Ameisenbau.

Wie funktioniert eine Zuckerstraße?

Dieses Problem besteht im Kern aus folgenden zwei Teilproblemen:

1. Der Transport von Zucker an sich.
2. Die Weitergabe von Weginformationen über den Zuckerhügel an alle anderen nahegelegenen Ameisen.

Zu 1.: Das erste Teilproblem lässt sich fast genau so lösen, wie das bei den Äpfeln:

```
1  public override void Sieht(Zucker zucker)
2  {
3     if (Ziel == null)
4     {
5        GeheZuZiel(zucker);
6     }
7  }

9  public override void ZielErreicht(Zucker zucker)
10 {
11    Nimm(zucker);
12    GeheZuBau();
13 }
```

Zur Lösung des ersten Teilproblems sind für uns die Ereignisse `Sieht(Zucker zucker)` und `ZielErreicht(Zucker zucker)` interessant.

Die Ameise schaut in `Sieht(Zucker zucker)` nach, was sie tun soll, wenn sie einen Zuckerhügel entdeckt. Zunächst überprüft die Ameise, ob sie zur Zeit „arbeitslos“ ist. Sollte dies der Fall sein, setzen wir den entdeckten Zuckerhügel als ihr nächstes Ziel mit `GeheZuZiel(zucker)` in Zeile 5.

Beim Zuckerhügel angekommen, erfährt sie ihre nächsten Befehle, indem sie in `ZielErreicht(Zucker zucker)` nachschaut. In dieser Situation soll sie mit `Nimm(zucker)` so viel Zucker wie möglich aufnehmen und mit `GeheZuBau()` wird ihr aufgetragen, zum eigenen Bau, zusammen mit dem aufgenommenen Zucker, zu laufen.

Zu 2.: Hier werden wir mit sogenannten *Markierungen* arbeiten, um Informationen an andere Ameisen weiterzureichen. Wir zeigen euch einen Lösungsvorschlag mit den Ereignissen `Tick()` und `RiechtFreund(Markierung markierung)`:

```
private const int SMALLESTDIST = 0;

public override void Tick()
{
    if(AktuelleLast > 0) //Trägt die Ameise Nahrung?
    {
        if(GetragenesObst == null) //Trägt die Ameise Zucker?
        {
            SprüheMarkierung(Richtung + 180, SMALLESTDIST);
        }
    }
}

public override void RiechtFreund(Markierung markierung)
{
    if(markierung.Information < 1000)
    {   //Ist Markierung Wegweiser zum Zuckerhügel?
        if(Ziel == null)
        {
            //Drehe Richtung Zuckerhügel
            DreheInRichtung(markierung.Information);
            GeheGeradeaus();
        }
    }
}
```

Hinweis: Bestehenden Code in `Tick()` könnt ihr natürlich stehen lassen. Schreibt den neuen Code in Zeile 5-11 unter eurem alten Code.

Okaaay, das sieht auf den ersten Blick kompliziert aus, aber wir versprechen euch, dass alles seinen Sinn hat :). Für eine erfolgreiche Kommunikation braucht es in der Regel einen *Sender* und einen *Empfänger*. Sonst könnte es eine sehr einseitige Unterhaltung werden. Wir programmieren zuerst den *Sender* und danach den *Empfänger*.

Den *Sender* können wir in `Tick()` implementieren. Wir erinnern uns aus dem Vortrag, dass die Ameise in `Tick()` zu Beginn jeder Runde nachschaut, was sie tun soll. Unsere Ameise soll nur dann ihre Freunde in Kenntnis setzen wo sich ein Zuckerhügel befindet, wenn sie Zucker trägt. Dafür bedarf es zweier Abfragen.

Als Erstes überprüft unsere Ameise in Zeile 5 mit `AktuelleLast > 0`, ob sie überhaupt Nahrung trägt. Sollte sie tatsächlich Nahrung tragen, müssen wir ausschließen, dass sie einen Apfel trägt. Dies können wir mit `GetragenesObst == null` sicherstellen.

Sollten beide Abfragen wahr sein, sprühen wir eine Markierung. In diesem Fall sprühen wir als Information `Richtung + 180`. Was zunächst recht kryptisch anmutet, lässt sich einfach erklären. Unsere Ameise hat mit der *Zustandseigenschaft* `Richtung` eine Art „biologischen Kompass“, der besagt, in welche Himmelsrichtung sie läuft. Genauer gesagt steht in `Richtung` zu jedem Zeitpunkt ein bestimmter Winkel α in Grad. Angenommen, unsere Ameise befindet sich mit ihrer Beute auf dem Weg zu ihrem Ameisenbau. Gibt α die Richtung an, in die unsere beladene

Ameise läuft, dann geben wir mit $\alpha + 180$ an, in welcher Richtung ein Zuckerhügel liegt, denn die Addition von 180 auf einen Winkel in Grad entspricht der wortwörtlichen 180° Kehrtwende (Da war was, mit Schule und Mathe...).

Die *Konstante* `SMALLESTDIST` gibt die Größe unserer Markierung an. Dabei gilt die Faustregel: *Je größer eure Markierung ist, desto schneller verflüchtigt sich diese.* Für unsere Zuckerstraße möchten wir eine möglichst langlebige Markierung wählen. Die Größe 0 scheint wenig intuitiv, aber hier nutzen wir die Spielmechaniken etwas aus. Diese Größe ist tatsächlich sichtbar für eure Ameisen und die Markierung bleibt lange bestehen.

Nun kommen wir zum Ereignis `RiechtFreund(Markierung markierung)`. Hier implementieren wir den *Empfänger*.

Ihr werdet in der Informatik häufiger sehen, dass man Informationen *kodiert*. Das heißt, man weist einer bedeutungslosen Folge von Symbolen eine *semantische* Bedeutung zu. Wir haben mit unserem Code willkürlich festgelegt, dass alle Markierungen mit einem Wert $\in \{-2^{31}, \ldots, 999\}$ für die Kennzeichnung einer Zuckerstraße stehen. Negative Zahlen sind *Kwadj* in diesem Kontext, aber so steht es „technisch“ gesehen erst mal da. In Zeile 16 prüfen wir mit der Abfrage `markierung.Information < 1000`, ob wir es tatsächlich mit einer Markierung für eine Zuckerstraße zu tun haben. Andere Markierungen mit einem höheren Wert werden vorerst ignoriert.

Darauf folgt die wohlbekannte Abfrage in Zeile 18 nach dem Ziel mit `Ziel == null`. Hat unsere Ameise keine Aufgabe, dreht sie sich mit `DreheInRichtung(markierung.Information)` zum Zuckerhügel und geht mit dem Befehl `GeheGeradeaus()` dort hin.

Ameisen, welche vom Zuckerhügel kommen und am eigenen Ameisenbau Zucker ablegen, riechen ihre eigene Markierung mit der Richtungsinformation und gehen wieder zum Zuckerhügel zurück. Dadurch entsteht die, für Ameisen charakteristische, Zuckerstraße.

Damit ist die Erklärung zur Zuckerstraße abgeschlossen. Ihr könnt gerne eure Zuckerstraße weiter verbessern. Das Grundprinzip solltet ihr nun verstanden haben.

Wir haben gehört, dass man die Eigenschaften von Ameisen ändern kann. Wie machen wir das?

Sucht in eurem Code nach dem folgenden Codefragment:

```
1  [Kaste(
2          Name = "Standart", // Name der Berufsgruppe
3          AngriffModifikator = 0, // Angriffsstärke einer Ameise
4          DrehgeschwindigkeitModifikator = 0, // Drehgeschwindigkeit einer Ameise
5          EnergieModifikator = 0, // Lebensenergie einer Ameise
6          GeschwindigkeitModifikator = 0, // Laufgeschwindigkeit einer Ameise
7          LastModifikator = 0, // Tragkraft einer Ameise
8          ReichweiteModifikator = 0, // Ausdauer einer Ameise
9          SichtweiteModifikator = 0  // Sichtweite einer Ameise
10 )]
```

Die Grundeigenschaften in der Kaste könnt ihr modifizieren. Zum Beispiel können wir die Modifikatoren und den Namen der Berufsgruppe wie folgt abändern:

```
[Kaste(
        Name = "Turbosammler", // Name der Berufsgruppe
        AngriffModifikator = -1, // Angriffsstärke einer Ameise
        DrehgeschwindigkeitModifikator = 0, // Drehgeschwindigkeit einer Ameise
        EnergieModifikator = -1, // Lebensenergie einer Ameise
        GeschwindigkeitModifikator = 2, // Laufgeschwindigkeit einer Ameise
        LastModifikator = 2, // Tragkraft einer Ameise
        ReichweiteModifikator = -1, // Ausdauer einer Ameise
        SichtweiteModifikator = -1  // Sichtweite einer Ameise
)]
```

Am Ende muss die *Summe* aller Modifikatoren 0 ergeben! Weiter unten findet ihr eine Auflistung der möglichen Modifikatoren und deren Auswirkungen.

Modifikator	-1	0	1	2
Geschwindigkeit	3 Schritte/Runde	4 Schritte/Runde	5 Schritte/Runde	6 Schritte/Runde
Drehgeschwindigkeit	6 Grad/Runde	8 Grad/Runde	12 Grad/Runde	16 Grad/Runde
Last	4 Einheiten Zucker	5 Einheiten Zucker	7 Einheiten Zucker	10 Einheiten Zucker
Sichtweite	45 Schritte	60 Schritte	75 Schritte	90 Schritte
Reichweite	$0.75 \cdot$ Standard	$1 \cdot$ Standard	$1.5 \cdot$ Standard	$2 \cdot$ Standard
Energie	50 LP	100 LP	175 LP	250 LP
Angriff	kein Angriff	10 LP/Runde	20 LP/Runde	30 LP/Runde

Tabelle 1: Eigenschaften einer Ameise und die Auswirkungen der Modifikatoren.

Wir haben uns einen Masterplan mit 13 verschiedenen Klassen von Ameisen überlegt. Wie übersetzen wir nun diese Klassen in Code?

Zuerst ein kleiner Tipp von uns: Überlegt euch gut, was ihr da tut. Da gibt es einen ganz gemeinen Gegner, der hinter jeder Ecke lauert: *Komplexität*. Je mehr Klassen ihr erstellen und mit Leben füllen möchtet, desto wahrscheinlicher ist es, dass euch alles um die Ohren fliegt.

Nun folgt, wie ihr eure Klassen in Code übersetzen könnt:

```
[
    Kaste(
        Name = "Turbosammler",
        AngriffModifikator                  = -1,
        DrehgeschwindigkeitModifikator      = 0,
        EnergieModifikator                  = -1,
        GeschwindigkeitModifikator          = 2,
        LastModifikator                     = 2,
        ReichweiteModifikator               = -1,
        SichtweiteModifikator               = -1
    ),

    Kaste(
        Name = "Wanzenkiller",
        AngriffModifikator                   = 2,
        DrehgeschwindigkeitModifikator       = 0,
        EnergieModifikator                   = 1,
        GeschwindigkeitModifikator           = 0,
        LastModifikator                      = -1,
        ReichweiteModifikator                = -1,
```

```
21          SichtweiteModifikator                    = -1
22      )
23  ]
```

Ihr müsst im Endeffekt einen weiteren „Kastenblock“ in den eckigen Klammern einfügen. Nach diesem Muster könnt ihr eure weiteren elf Klassen von Ameisen einfügen.

In `BestimmeKaste(Dictionary<string, int> anzahl)` könnt ihr nun bestimmen, welche Klasse von Ameisen wann generiert wird. Im merkwürdigen Konstrukt

`Dictionary<string, int> anzahl`

wird gespeichert, wie viele eurer lebenden Ameisen welcher Klasse angehören. Wollt ihr wissen, wie viele der Ameisen auf dem Spielfeld einer bestimmten Klasse angehören, dann geht das so:

```
1 anzahl["<Name einer Klasse>"]
```

Ihr könntet eure `BestimmeKaste(Dictionary<string, int> anzahl)` z.B. folgendermaßen gestalten:

```
1 public override string BestimmeKaste(Dictionary<string, int> anzahl)
2 {
3     if(anzahl["Turbosammler"] < 50)
4     {
5        return "Turbosammler";
6     } else {
7        return "Wanzenkiller";
8     }
9 }
```

Mit dem *Schlüsselwort* `return` und dem anschließenden Klassennamen sagt ihr, welcher Klasse die nächste generierte Ameise angehören wird.

Nun wollen wir für jede unserer 13 Klassen das Verhalten einzeln spezifizieren!

Gut, dann lest euch die Antwort zum übernächsten Punkt durch. Dort findet ihr ein Beispiel, wie ihr das machen könnt.

Unsere armen Ameisen sterben andauernd :(Woran liegt das?

Unser herzliches Beileid zu eurem Verlust :(. Für das vorzeitige Ableben eurer Ameisen kommen zwei mögliche Ursachen in Betracht:

- Die Lebenspunkte von euren Ameisen sind auf 0 gefallen. Das kann passieren, wenn eure Ameise von Wanzen oder feindlichen Ameisen angegriffen wird.

- Eure Ameisen verhungern. Ameisen haben eine Grundeigenschaft namens `Reichweite`.

  `Reichweite` gibt an, wie viele Schritte eure Ameise überleben kann, ohne zum Bau zu gehen. Die Anzahl der Schritte, welche eure Ameise seit dem letzten Besuch im Ameisenbau zurückgelegt hat, wird in der Zustandseigenschaft `ZurückgelegteSchritte` gespeichert.

  Wenn `ZurückgelegteSchritte` > `Reichweite` gilt, ist eure Ameise leider verhungert. Dies verhindert ihr, indem ihr eure Ameise *rechtzeitig* zum Ameisenbau schickt. Dadurch wird `ZurückgelegteSchritte` auf 0 zurückgesetzt.

Ist eine Ameise 1/3 ihrer `Reichweite` gelaufen, schaut sie in `WirdMüde()` nach, was sie tun soll. Dort könnt ihr eurer Ameise mit `GeheZuBau()` befehlen zum Bau zurückzukehren, damit sie sich erholen kann.

Wir schlagen euch jedoch eine etwas intelligentere Implementierung vor:

```
1  public override void Tick()
2  {
3      if(Ziel == null)
4      {
5          if(Reichweite / (ZurückgelegteStrecke + 1) < 2)
6          {
7              GeheZuBau();
8          }
9      }
10 }
```

> *Hinweis:* Ist eure `Tick()`-Methode nicht leer, dann schreibt den neuen Code in Zeile 3-9 unter eurem alten Code.

Wir finden es etwas früh, die Ameise zum Bau zu schicken, wenn sie erst 1/3 ihrer `Reichweite` aufgebraucht hat. Wir wollen die Ameise erst zu ihrem Ameisenbau schicken, wenn sie seit ihrem letzten Besuch im Ameisenbau ungefähr die Hälfte ihrer `Reichweite` gelaufen ist. Dafür ist `WirdMüde()` aber komplett nutzlos, denn `WirdMüde()` wird nicht kontinuierlich aufgerufen, sobald die Ameise 1/3 ihrer Reichweite aufgebraucht hat, sondern nur in genau diesem Moment. Würde man unsere Lösung in `WirdMüde()` einfügen, dann würde die Anweisung `GeheZuBau()` nie ausgeführt werden (Warum?).

Betrachten wir jetzt den Code weiter oben. Zuerst erfolgt die obligatorische Überprüfung in Zeile 3, ob die Ameise ein Ziel hat. Schließlich sollte die Ameise zuerst ihre (hoffentlich sinnvolle) Aufgabe erledigt haben, bevor sie sich in ihrem Ameisenbau ausruht.

Der Ausdruck in der zweiten Abfrage in Zeile 5 ist erst wahr, wenn die Ameise ungefähr die Hälfte ihrer Reichweite gelaufen ist. Der wenig nützlich erscheinende Summand 1 ist eher technischer Natur und vermeidet eine *Division durch 0*, falls bei unserer Ameise die *Zustandseigenschaft* `ZurückgelegteStrecke` durch einen Besuch im Ameisenbau auf 0 gesetzt wurde.

Böse Wanzen und feindliche Ameisen töten unsere unschuldigen Sammler! Wir wollen uns rächen!

Klar könnt ihr das tun! Wenn das Lebensmotto eurer Ameisen „Auge um Auge“ lauten soll, ist der Befehl `GreifeAn()` genau richtig für euch! Wir zeigen euch, wie man einer Ameise befehlen kann eine Wanze anzugreifen. Gegnerische Ameisen könnt ihr auf analoge Art und Weise ebenfalls überfallen.

```
private const int MEDIUMDIST = 100;
private const int THRESHHOLD = 5;
private const int BUG = 1001;

public override void SiehtFeind(Wanze wanze)
{
    if(Ziel == null)
    {
        if(Kaste == "Wanzenkiller")
        {
            SprüheMarkierung(BUG, MEDIUMDIST);
            if(AnzahlAmeisenDerSelbenKasteInSichtweite > THRESHHOLD)
            {
                GreifeAn(wanze);
            }
        } else if(Kaste == "Turbosammler") {
            GeheWegVon(wanze);
        }
    }
}
```

Wir geben zu, dass wir euch hier ein etwas komplizierteres Beispiel untergejubelt haben, als notwendig. In diesem Fall liegt eine Steuerung in Abhängigkeit von der Zugehörigkeit zu einer Kaste vor. Auf diese Weise könnt ihr ein ziemlich komplexes Ameisenvolk programmieren!

Wie so oft, folgt in Zeile 7 die Abfrage nach dem Ziel der Ameise. Hat die Ameise kein Ziel, wird sie irgendwie auf die gesehene Wanze reagieren.

Gehört die Ameise der Klasse Wanzenkiller an, wird die Abfrage in Zeile 9 wahr. Die Ameise sprüht in diesem Falle eine Markierung. Wir erinnern uns, dass wir in `RiechtFreund(Markierung markierung)` bestimmen können, wie unsere Ameisen auf diese Markierung mit der konkreten Kodierung `BUG` (1001) reagieren sollen.

Sind genügend andere Wanzenkiller in der Nähe unserer Ameise, wird die Abfrage in Zeile 12 wahr und die Ameise wagt einen Angriff auf die Wanze mit `GreifeAn(wanze)`.

Sollte unsere Ameise stattdessen ein fragiler Turbosammler sein, wird statt der Abfrage in Zeile 9 die Abfrage in Zeile 16 wahr (*Verzweigung!*). In diesem Fall wird sich der Turbosammler mit `GeheWegVon(wanze)` von der Wanze entfernen.

Die Wanzen sind aber ziemlich hartnäckig... Wieso haben wir so große Schwierigkeiten sie zu töten?

Wir verraten euch mal ein paar Eigenschaften einer Wanze:

- *Energie:* 1000LP
- *Angriff:* 50LP/Runde
- *Drehgeschwindigkeit:* 3 Schritte/Runde
- *Geschwindigkeit:* 3 Schritte/Runde
- *Besonderheit:* Regeneriert langsam verlorene LP!

Wie man sieht, ist eine Wanze sehr stark und kann viele Treffer einstecken. Mit den entsprechenden Modifikatoren, können eure Ameisen höchstens 250LP Energie haben und 30 Angriff/Runde ausrichten. Dazu kommt, dass eine Wanze ihre verlorene LP wieder regeneriert!

Unsere Ameisen verhalten sich nicht so, wie wir es möchten und wir wissen nicht, wo der Wurm steckt :(

Die schlechte Nachricht: Das Finden von Defekten in einem Programm ist mühsam und eine Wissenschaft für sich.

Die gute Nachricht: Eure Ameisen können sich nicht nur halbwegs intelligent verhalten, sondern sogar *denken*! Mit dem Befehl `denke(string nachricht)` können eure Ameisen eine Textzeile in die Simulation projizieren. Um diese Textzeilen in Form von Denkblasen zu sehen, drückt in der Simulation die *D-Taste*.

Ist das alles? Ist unser Ameisenvolk nun perfekt?

Wir müssen euch leider, und auch zum Glück, in diesem Punkt enttäuschen. Ihr solltet ein ganz *passables* Ameisenvolk haben, wenn ihr dieses „Stück Papier“ durchgegangen seid. Ihr könnt nun euer Ameisenvolk beliebig komplex gestalten. Hier eine Liste von weiteren Verbesserungsvorschlägen:

- Ihr könnt versuchen das Verhalten eurer Ameisen noch intelligenter zu gestalten (offensichtlich).
- Ihr könnt den niedlichen, aber ineffizienten, „Zick-Zack“ Lauf in `GeheZuZiel(Spielobjekt ziel)` durch eigenen Code optimieren.
- Ihr könnt ein zentrales Ameisengehirn implementieren!
- Ihr könnt die Generierung von Ameisen dynamisch durch äußere Einflüsse steuern. Z.B.: Wenig getötete Ameisen $\Rightarrow$ Mehr Sammler produzieren
- Ihr könnt dafür sorgen, dass immer der, zum eigenen Ameisenbau nächst gelegene Zuckerhügel abgearbeitet wird.
- ...

Lasst eurer Kreativität freien Lauf! Für weitere Inspirationen könnt ihr euch auch die schon bestehenden Ameisenvölker in AntMe! anschauen. Vielleicht beobachtet ihr Verhalten, welches euch gefällt und ihr imitieren wollt.

Viel Spaß!