

Waterwolrd Analysis

Ing. Thomas Herzog

Version 1.0, March 31 2016

Folgendes Dokument beschäftigt sich mit der Analyse und Verbesserung des C# Programms *Waterworld*.

Die angewandten Verbesserungen wurden in der originalen *Visual Studio Solution* bzw. den enthaltenen Projekt in folgender Verzeichnisstruktur organisiert:

Wator/Wator/

- *Orginal*
- *Part1* (Point Instanzen vermeiden)
- *Part2* (SelectNeighbor entfernen)
- *Part3* (GenerateRandomMatrix, RandomizeMatrix)

Alle Implementierungen von *IWatorWorld* wurden in der Klasse *Version* hinzugefügt und können über die Einstellungen des Programms *WatorWorld* ausgewählt werden.

Im ersten Kapitel [Heap-Analyse](#) wird das Verhalten am *Heap* der Originalversion zu den Verbesserungen betrachtet.

Im Kapitel [Laufzeit-Analyse](#) wird das Laufzeitverhalten der Originalversion zu den Verbesserungen betrachtet.

Im Kapitel [Quelltextverbesserungen](#) werden die drei angewandten Verbesserungen beschrieben und begründet.

Heap-Analyse

Dieses Kapitel beschäftigt sich mit der *Heap*-Analyse, die aufzeigen wird, wie sich der Heap zur Programmlaufzeit der verschiedenen Programmversionen verhält und welche Objekte am *Heap* in welcher Verteilung und Häufigkeit vorzufinden sind. Diese Analyseergebnisse werden bei den Verbesserungen benötigt, da sie aufzeigen werden, wo in den verschiedenen Programmversionen Probleme am *Heap* vorzufinden sind.

Gleichzeitig werden die Analyseergebnisse auch aufzeigen, inwiefern sich das Verhalten am *Heap* durch die angewandten Verbesserungen verändert hat.

Ergebnisse: *Originalversion*

Folgender Teil beschäftigt sich mit den Ergebnissen der *Heap*-Analyse der Originalversion.

Functions Allocating Most Memory

Name	Bytes %
VSS.Wator.Original.OriginalWatorWorld.GetNeighbors(class System.Type, valuetype System.Drawing.Point)	92.55
VSS.Wator.Original.OriginalWatorWorld..ctor(class VSS.Wator.Settings)	2.73
VSS.Wator.Original.Fish.Spawn()	2.40
System.Windows.Forms.Form.ShowDialog(class System.Windows.Forms.IWin32Window)	0.88
VSS.Wator.Original.OriginalWatorWorld.GenerateRandomMatrix(int32, int32)	0.65

Types With Most Memory Allocated

Name	Bytes %
System.Drawing.Point[]	92.55
VSS.Wator.Original.Fish	3.03
VSS.Wator.Original.Animal[]	1.30
System.Byte[]	0.72
System.Int32[]	0.66

Types With Most Instances

Name	Instances %
System.Drawing.Point[]	94.79
VSS.Wator.Original.Fish	3.32
System.Version	0.41
VSS.Wator.Original.Shark	0.27
System.Drawing.KnownColor	0.21

Figure 1. Verhalten am Heap der "Originalversion"

Eklatant fällt hier auf, dass es sehr viele *Point*-Objekte am Heap gibt, die sehr kurzlebig sind und daher den *Garbage Collector* stark belasten. Daher sollte hier angesetzt werden, um zu versuchen diese *Point*-Objekte zu vermeiden.

Des Weiteren sieht man in Abbildung [Verhalten am Heap der "Originalversion"](#), dass die Methode `GetNeighbors(Type, Point)` die meisten *Point*-Objekte produziert, daher sollte man hier zuerst ansetzen. In der Methode `GetNeighbors(Type, Point)` werden alle Nachbarn gesucht und die Methode `SelectNeighbor(Type, Point)` entscheidet per Zufall welcher Nachbar herangezogen wird. Dadurch dass immer alle Nachbarn gesucht aber nur einer herangezogen wird, sind zu viele Zugriffe auf die Matrix erforderlich um die Nachbarn zu erhalten, was die Laufzeit negativ beeinflusst.

Ergebnisse: *Point* Instanzen vermeiden

Folgende Teil zeigt die Analyseergebnisse der *Heap*-Analyse der Verbesserung *Point* Instanzen vermeiden.

Functions Allocating Most Memory

Name	Bytes %
VSS.Wator.Part1.Part1WatorWorld.SelectNeighbor(class System.Type, valueType System.Drawing.Point&)	84.06
VSS.Wator.Part1.Fish.Spawn()	6.80
VSS.Wator.Part1.Part1WatorWorld..ctor(class VSS.Wator.Settings)	4.90
System.Windows.Forms.Form.ShowDialog(class System.Windows.Forms.IWin32Window)	1.29
VSS.Wator.Part1.Part1WatorWorld.GenerateRandomMatrix(int32, int32)	1.17

Types With Most Memory Allocated

Name	Bytes %
System.Drawing.Point[]	84.06
VSS.Wator.Part1.Fish	7.92
VSS.Wator.Part1.Animal[]	2.34
System.Byte[]	1.44
System.Int32[]	1.18

Types With Most Instances

Name	Instances %
System.Drawing.Point[]	92.13
VSS.Wator.Part1.Fish	5.79
System.Version	0.42
VSS.Wator.Part1.Shark	0.30
System.Drawing.KnownColor	0.21

Figure 2. Verhalten am Heap der Verbesserung "Point Instanzen vermeiden"

Diese Verbesserung hat erreicht das anstatt ~ **94 %** nur mehr ~ **84 %** der Objekte am *Heap Point*-Instanzen sind. Dadurch ist der *Garbage Collector* weniger stark ausgelastet, was einen positiven Effekt auf das Laufzeitverhalten hat.

Trotzdem sei angemerkt, dass sich herausgestellt hat, dass der Ansatz des *Random*-Zugriff auf die *Direction* beim Ermitteln der Nachbarn die Laufzeit um ~ **500ms** verschlechtert, daher wurde der Randomzugriff entfernt. Dadurch wird aber immer in die gleiche Richtung gegangen bzw nach Nachbar gesucht, was zur Folge hat, dass dies auf der grafischen Asugabe sichtbar wird.

Ergebnisse: *SelectNeighbor* entfernen

Folgender Teil zeigt die Analyseergebnisse der *Heap*-Analyse der Verbesserung *SelectNeighbor* entfernen.

Functions Allocating Most Memory

Name	Bytes %
VSS.Wator.Part2.Fish.Spawn()	69.91
VSS.Wator.Part2.Shark.Spawn()	13.15
VSS.Wator.Part2.Part2WatorWorld..ctor(class VSS.Wator.Settings)	8.78
System.Windows.Forms.Form.ShowDialog(class System.Windows.Forms.IWin32Window)	3.92
VSS.Wator.Part2.Part2WatorWorld.GenerateRandomMatrix(int32,int32)	2.09

Types With Most Memory Allocated

Name	Bytes %
VSS.Wator.Part2.Fish	71.93
VSS.Wator.Part2.Shark	13.64
VSS.Wator.Part2.Animal[]	4.18
System.Byte[]	2.30
System.Int32[]	2.11

Types With Most Instances

Name	Instances %
VSS.Wator.Part2.Fish	78.50
VSS.Wator.Part2.Shark	14.89
System.Version	1.83
System.Drawing.KnownColor	0.92
System.WeakReference	0.29

Figure 3. Verhalten am *Heap* der Verbesserung "*SelectNeighbor* entfernen"

Durch das Entfernen der Methode *SelectNeighbor*(*Type*, *Point*) sind die *Point*-Objekte nicht mehr die dominanten Objekte am *Heap*, sondern die *Fish*-Objekte, die aber nicht zu vermeiden sind, da es nur soviele Fische am *Heap* gibt, als wie in der Matrix vorhanden. Und die *Fish*-Objekte sind langlebige Objekte im Unterschied zu den *Point*-Objekten. Diese Verbesserung hat also den *Heap* entlastet.

Laufzeit-Analyse

Dieses Kapitel beschäftigt sich mit der *Runtime*-Analyse der Originalversion zu den angewandten Verbesserungen. Dieses Kapitel wird aufzeigen wie sich das Laufzeitverhalten der verschiedene Programmversionen verhält und verändert.

Folgendes Bild zeigt die festgesetzte Konfiguration für alle gemachten Testdurchläufe.

▼ Fish Settings	
FishBreedTime	10
InitialFishEnergy	10
InitialFishPopulation	20000
▼ General Settings	
DisplayInterval	1
DisplayWorld	False
Height	500
Iterations	500
Runs	2
Version	OriginalWaterWorld
Width	500
Workers	1
▼ Shark Settings	
InitialSharkEnergy	25
InitialSharkPopulation	5000
SharkBreedEnergy	50

Figure 4. Konfiguration der Testdurchläufe

Mit dieser Konfiguration wurden **10** Durchläufe vorgenommen, deren Ergebnisse folgend als Screenshots bei den verschiedenen Programmversionen angefügt sind.

Ergebnisse: der Originalversion

Folgende Teil zeigt die Analyseergebnisse der *Runtime*-Analyse der *Originalversion*.

Runs:	10
Iterations:	100
Runtime in Milliseconds:	53626.3151
Avg. Milliseconds / Run:	5362.63151
Std. Deviation:	63.4535428582706

Runtimes in Milliseconds:	
Run 01:	5413.8321
Run 02:	5216.142
Run 03:	5315.6675
Run 04:	5309.4686
Run 05:	5398.8009
Run 06:	5407.1325
Run 07:	5407.93
Run 08:	5339.8962
Run 09:	5385.8673
Run 10:	5431.578

Figure 5. Laufzeitverhalten der "Originalversion"

Ergebnisse: *Point Instanzen vermeiden*

Folgender Teil zeigt die Analyseergebnisse der *Runtime*-Analyse der Verbesserung *Point Instanzen vermeiden*.

Runs:	10
Iterations:	100
Runtime in Milliseconds:	46086.6748
Avg. Milliseconds / Run:	4608.66748
Std. Deviation:	42.2188100154815

Runtimes in Milliseconds:	
Run 01:	4648.7922
Run 02:	4563.9884
Run 03:	4611.6269
Run 04:	4623.2412
Run 05:	4660.7016
Run 06:	4566.9648
Run 07:	4557.5137
Run 08:	4672.7191
Run 09:	4623.5439
Run 10:	4557.583

Figure 6. Laufzeitverhalten der "*Point Instanzen vermeiden*"

Mit dieser ersten Verbesserung wurde die Laufzeit der Anwendung um ~ 0.7 **sec** verbessert.

Ergebnisse: *SelectNeighbor entfernen*

Folgende Teil zeigt die Analyseergebnisse der *Runtime*-Analyse dieser angewandten Verbesserung.

Laufzeitverhalten: SelectNeighbor entfernen

Folgender Teil zeigt die Analyseergebnisse der *Runtime*-Analyse der Verbesserung *SelectNeighbor entfernen*.

Runs:	10
Iterations:	100
Runtime in Milliseconds:	40555.7923
Avg. Milliseconds / Run:	4055.57923
Std. Deviation:	46.0901552732217

Runtimes in Milliseconds:	
Run 01:	4079.6591
Run 02:	4047.0637
Run 03:	4104.3861
Run 04:	4148.8433
Run 05:	4001.0242
Run 06:	4054.1996
Run 07:	3992.8962
Run 08:	4079.8272
Run 09:	4026.5757
Run 10:	4021.3172

Mit dieser zweiten Verbesserung wurde die Laufzeit der Anwendung um ~ 1.4 **sec** verbessert.

Ergebnisse von: *GenerateRandomMatrix*, *RandomizeMatrix*

Folgende Teil zeigt die Analyseergebnisse der *Runtime*-Analyse dieser angewandten Verbesserung.

```
Runs: 10
Iterations: 100
Runtime in Milliseconds: 35784.5365
Avg. Milliseconds / Run: 3578.45365
Std. Deviation: 18.5103239067533
-----
Runtimes in Milliseconds:
Run 01: 3606.3749
Run 02: 3595.0335
Run 03: 3593.541
Run 04: 3587.0629
Run 05: 3558.9391
Run 06: 3581.4392
Run 07: 3557.9559
Run 08: 3580.595
Run 09: 3543.0791
Run 10: 3580.5159
```

Figure 7. Ergebnisse von: *GenerateMatrix*, *RandomizeMatrix* modifizieren

Mit dieser letzten Verbesserung wurde die Laufzeit der Anwendung um ~ **1.8 sec** verbessert.

Quelltextverbesserungen

Folgender Teil beschäftigt sich mit den Optimierungen, die angewendet wurden, um das Laufzeitverhalten zu verbessern. Es wurden drei Verbesserungen angewandt die nachfolgend aufgeführt sind:

- [Point Instanzen vermeiden](#)
- [SelectNeighbor entfernen](#)
- [GenerateRandomMatrix, RandomizeMatrix](#)

Point Instanzen vermeiden

Die Methode `GetNeighbors(Type, Point)` wurde dahingehend verändert, dass nicht mehr alle Nachbarn besucht und zurückgeliefert werden, sondern dass per Zufall, solange die möglichen Nachbarn besucht werden, bis das erwartete Resultat eintritt. Der erste gefundene Nachbar, der die Anforderungen erfüllt, wird zurückgeliefert.

Siehe hierzu die folgenden Methoden in *Part1WaterWorld*:

- `public Point GetNeighbors(Type type, Point position)`
- `public Point SelectNeighbor(Type type, Point position)`

In der Methode `SelectNeighbor(Type type, Point position)` wurde lediglich folgende Änderung vorgenommen

`Point[] neighbors = new Point[] { GetNeighbors(type, ref position) };`. Damit bleibt die restliche Implementierung von Änderungen unbetroffen.

Durch die Änderungen in `GetNeighbors(Type, Point)` wird das Verhalten teilweise verändert, da andere Wege als bei der Originalversion gegangen werden. Dies könnte als Änderung am Programm ausgelegt werden, wenn dieses Verhalten als essentiell eingestuft wird.

SelectNeighbor entfernen

Nachdem die Methode `GetNeighbors(Type, Point)` so verändert wurde, dass hbie bereits eine einzige Position eines Nachbarn zurückgeliefert wird, kann auf die Methode `SelectNeighbor` verzichtet werden, da die implementierte Logik keine Anwendung mehr findet. Im Zuge dessen wird die Methode `GetNeighbors(Type, Point)` unbenannt in `GetNeighbor(Type, Point)`, da diese Methode nur mehr ein Resultat und kein Array mehr zurückliefert. Dadurch wird ein *Methoden*-Aufruf vermieden und das Durchschleifen bzw. Kopieren (bei *Value Types*) der Aktualparameter.

GenerateRandomMatrix, RandomizeMatrix

Anstatt ein zweidimensionales Array zu verwenden, in dem die Indexes abgebildet werden, wird eine Liste von *Point*-Objekten beim erstmaligen Erstellen der Matrix erstellt und bei jedem Aufruf der Methode `ExecuteStep()` zufällig neu geordnet. Es werden zwar alle Positionen auf der Matrix über *Point*-Objekte abgebildet, aber diese *Point*-Objekte bleiben über die Laufzeit erhalten und werden daher vom *Garbage Collector* nicht beachtet, da immer eine Referenz auf diese Objekte

besteht. Das zufällige Besuchen der Positionen bleibt gewährleistet. Des Weiteren werden alle *Animal*-Objekte, die in der Methode `ExecuteStep()` als *Moved* markiert wurden, in einer Liste gespeichert und nachdem alle Positionen besucht wurden, *committed*. Dadurch wird ein weiteres Iterieren über alle Positionen vermieden.

Siehe dazu die folgenden Methoden in der Klasse *Part3WatorWorld*:

- `ExecuteStep()`
- `ShuffelPoints(Point[])` (Ersetzt `RandomizeMatrix`)