

Sinn und Zweck der Methode Dispose

Mülltrennung

Wenige .NET-Konzepte stiften so viel Verwirrung wie die IDisposable-Schnittstelle. In Foren und Newsgroups liest man abenteuerliche Formulierungen wie „Dispose markiert ein Objekt für die Zerstörung durch die Garbage Collection“. Dieser Artikel beleuchtet die Hintergründe und erläutert den richtigen Einsatz von Dispose.

Eigentlich ist doch alles ganz einfach: Der Programmierer legt neue Objekte unter Einsatz des *new*-Operators an, die Speicherverwaltung erledigt den Rest.

Objekte sind Instanzen von Klassen und werden auf dem Heap gespeichert. Um mit einem Objekt arbeiten zu können, benötigt man dessen Referenz. Besitzt man keine solche Referenz mehr, dann kann man auf das betreffende Objekt auch nicht mehr zugreifen. Für den Programmcode ist es somit wertlos.

Hier setzt die Speicherverwaltung ein. Objekte, die im Programm nicht mehr referenziert werden, können durch den Garbage Collector vom Heap entfernt werden. Selbst Zirkularreferenzen, wie sie bei VB-Classic und COM programmatisch aufgelöst werden müssen, stellen für moderne Speicherverwaltungen kein Problem dar. Der Programmierer muss sich somit um die Zerstörung der angelegten Objekte keine Gedanken mehr machen. Also ist doch alles bestens, oder?

Der Pferdefuß mit den Betriebssystemressourcen

Wann die Speicherverwaltung nicht mehr benötigte Objekte vom Heap entfernt, kann nicht vorausgesagt werden. Aus Performance-Gründen wird dies in der Regel erst der Fall sein, wenn der zur Verfügung stehende Speicher zu knapp zum Anlegen neuer Objekte wird. Würde ständig jedes nicht mehr referenzierte Objekt entsorgt, würde der Prozessor unnötig belastet.

Nun könnte es Programmierern wie Anwendern ja eigentlich egal sein, wann die Speicherbereinigung vorgenommen wird, wäre da nicht das leidige Problem mit diversen Ressourcen, die vom Betriebssystem verwaltet werden.

Für viele Windows-Funktionen müssen solche Ressourcen vom Betriebssystem angefordert werden. Beispielsweise benötigt man zum Zeichnen ein *Graphics*-Objekt, das letztlich einen Gerätekontext der GDI-Schnittstelle belegt. Für alle Zeichenobjekte wie *Pen*, *Brush*, *Font* und so weiter werden zusätzliche Ressourcen belegt. Auch Handles für Zugriffe auf Dateien oder serielle Schnittstellen belegen Ressourcen, die nicht in beliebiger Menge bereitstehen. Diese Ressourcen werden auch „nicht verwaltete Ressourcen“ genannt, da sie nicht der Speicherverwaltung unterliegen.

Verlässt man sich nun auf die selbstreinigende Kraft der Speicherverwaltung, kann es passieren, dass diese Ressourcen über einen langen Zeitraum völlig unnötig blockiert werden. Der Grund: Objekte, die derartige Ressourcen belegen, werden zwar vom Programm nicht mehr referenziert, wurden aber vom Garbage Collector noch nicht entfernt, weil die Speicherverwaltung noch keinen Bedarf gesehen hat.

Nun könnte man mithilfe verschiedener Methoden der Klasse GC direkt in die Speicherverwaltung eingreifen und diese zur Zerstörung der Objekte zwingen.

Hiervon ist aber dringend abzuraten. Sie greifen dem Busfahrer ja auch nicht ins Steuer, weil Sie eine Abkürzung sehen oder die andere Fahrspur gerade frei ist. Microsoft hat sicherlich viel in die Entwicklung der Speicherverwaltung investiert, ist sie doch eine der tragenden Säulen von .NET. Zusätzliche Eingriffe seitens des Programms werden sich in der Regel eher störend auswirken.

Um dem Dilemma zu entkommen, hat man einen anderen Weg beschritten. Klassen, deren Objekte Ressourcen belegen, implementieren die Schnittstelle *IDisposable*, in der lediglich die parameterlose Methode *Dispose* deklariert ist. Jede dieser Klassen muss somit *Dispose* implementieren. Und diese Implementierung ist es, die die belegten Ressourcen freigibt.

Ein Aufruf von *Dispose* für ein Objekt wie ein *Graphics*- oder *Pen*-Objekt greift also nicht in die Speicherverwaltung ein. Die Garbage Collection bleibt davon völlig unberührt. Die Implementierung von *Dispose* sorgt lediglich für die sofortige Freigabe der belegten Ressourcen.

Verantwortung übernehmen

Das Konzept der *IDisposable*-Schnittstelle nimmt den Programmierer in die Pflicht. Er muss entscheiden, wann die Ressourcen wieder freigegeben werden müssen. In vielen Situationen ist diese Entscheidung recht einfach. So können beispielsweise die genannten Zeichenobjekte nach Abschluss der Zeichnung wieder freigegeben werden (Listing 1). Werden Objekte allerdings in verschiedenen Programmteilen benutzt, kann die Entscheidung schon einmal etwas kniffliger werden.

Was passiert nun, wenn für ein Objekt *Dispose* versehentlich mehrmals aufgerufen wird? Hier sieht die Dokumentation ausdrücklich vor, dass der mehrmalige Aufruf nicht zu einer Exception führen darf. Der erste Aufruf muss alle belegten

Auf einen Blick

Autor

Dr. Joachim Fuchs ist Software-Entwickler und Dozent, seit dem Jahr 2001 mit dem Schwerpunkt .NET. Sie erreichen ihn über www.fuechse-online.de/beruflich/index.html.



dotnetpro.code
A0505Dispose

Sprachen C#

Technik Garbage Collector, Dispose

Voraussetzungen VS.NET

Listing I

Ressourcen anfordern, verwenden und wieder freigeben.

```
// Ressourcen anfordern
Graphics g = this.CreateGraphics();
Pen stift = new Pen(Color.Blue);

// Ressourcen benutzen
g.DrawEllipse(stift, 0, 0, 100, 100);

// Ressourcen freigeben
stift.Dispose();
g.Dispose();
```

Ressourcen freigeben, alle weiteren müssen ignoriert werden. Allerdings darf danach das Objekt nicht mehr dergestalt benutzt werden, dass die inzwischen freigegebenen Ressourcen wieder benötigt werden. Sonst kommt es in vielen Fällen zu einem Laufzeitfehler.

Einige Klassen bieten die Möglichkeit, die belegten Ressourcen durch Aufruf an-

derer Methoden wieder freizugeben. Beispiele hierfür sind die Stream-Klassen oder die Datenbankverbindungen, die auch durch Aufruf der Methode *Close* geschlossen werden können. Meist ist diese Methode so implementiert, dass sie direkt *Dispose* aufruft. Ein zusätzlicher Aufruf von *Dispose* ist dann zwar unnötig, sollte aber auch nicht zu einem Fehler führen.

Wenn Sie Objekte verwenden, die *IDisposable* implementieren, dann sollten Sie grundsätzlich *Dispose* aufrufen, bevor Sie die Referenz aufgeben, es sei denn, in der Dokumentation der betreffenden Klasse wird ausdrücklich eine andere Vorgehensweise vorgeschrieben.

Bei verschachtelten Objekthierarchien, etwa bei Stream-Klassen, muss das oberste Objekt – zum Beispiel ein *StreamReader* – die *Dispose*-Methode der zugrunde liegenden Objekte – etwa eines *Streams* – aufrufen. Der Programmierer ruft nur für das von ihm angelegte *StreamReader*-Objekt die Methode *Dispose* auf, der Rest passiert hinter den Kulissen. Auch muss eine *Dispose*-Implementierung die *Dispose*-

Methode der Basisklasse aufrufen, sofern diese *IDisposable* implementiert.

Pillen gegen Vergesslichkeit

Allerdings gibt es keine einfache Möglichkeit, sicherzustellen oder zu überprüfen, dass alle Ressourcen ordnungsgemäß wieder freigegeben worden sind. Ein fehlendes *Dispose* bei Zeichenobjekten im *Paint*-Event wird zu einem Fehler führen, wenn das Programm lange Zeit läuft.

Glücklicherweise sorgt ein Mechanismus dafür, dass die Ressourcen spätestens dann freigegeben werden, wenn der Garbage Collector das betreffende Objekt entfernt. Die Dokumentation zu *IDisposable* schreibt nämlich vor, dass eine Klasse, die die Schnittstelle implementiert, auch einen Finalizer implementieren muss, der *Dispose* aufruft.

Sollten Sie einmal in die Verlegenheit kommen, eine Klasse zu entwerfen, die nicht verwaltete Ressourcen belegt, dann sollten Sie sich strikt an die Vorgaben der Dokumentation halten. |||||

Die aktuellen
redtec-
publishing-
Zeitschriften –
jetzt online
bestellen:

www.redtec.de

Oder direkt am
Kiosk kaufen.

