Kapitel 10

Multithreading mit WPF

In diesem Kapitel:

Eine neue Ausnahme	442
Das Dispatcher-Objekt	444
Die Klasse DispatcherTimer	450
Die Klasse BackgroundWorker	451
WPF und die Task Parallel Library (TPL)	454
Zusammenfassung	463

Bernd Marquardt: Microsoft Windows Presentation Foundation - Crashkurs, 2. überarbeitete Auflage, Microsoft Press 2011 (ISBN 978-3-86645-553-5)

Die Programmierung von Parallelisierungen von Anwendungen mit mehreren Threads, sodass also mehrere Aufgaben nach Möglichkeit auf die verschiedenen Kerne des Prozessors verteilt gleichzeitig ausgeführt werden können, war noch nie einfach. Und hier kommt dann gleich auch die schlechte Nachricht: Sie wird auch mit Windows Presentation Foundation nicht einfacher.

Auch mit WPF darf es nur einen Thread geben, der auf die Elemente der Benutzerschnittstelle zugreift. Dies ist der Thread, in dem die WPF-Elemente erzeugt worden sind. Diese goldene Regel gilt eigentlich schon so lange, wie es Multithread-Programmierung mit Windows gibt. Sie wurde jedoch oft einfach ignoriert. Und meistens ging auch alles irgendwie gut.

Stellen Sie sich aber einfach einmal vor, ein Thread stellt den Inhalt eines `ListBox`-Steuerelements aus Ihrer Benutzerschnittstelle dar. Mit einem zweiten Thread löschen Sie nun gleichzeitig eine Eintragung aus der `ListBox`. Das hört sich nicht wirklich gut an. Denken Sie auch daran, dass im Moment die Prozessoren mit mehreren Kernen zum Standard werden. Die Gefahr, dass bestimmte Codeteile in einer Anwendung mit mehreren Threads auch tatsächlich parallel laufen, ist nun wesentlich größer, als bei einem älteren Einprozessor-System.

Eine neue Ausnahme

Damit Sie möglichst einfach und schnell herausfinden können, ob Sie aus dem »richtigen« Thread auf ein Steuerelement der Benutzerschnittstelle zugreifen und somit die eben erwähnte goldene Regel nicht verletzen, hat Microsoft in .NET Framework (ab Version 3.0) die neue Ausnahme *InvalidOperationException* eingebaut, welche helfen soll, diese Fehler in WPF-Anwendungen schnell zu finden. Diese Ausnahme wird nicht nur in WPF-Anwendungen, sondern auch in Windows-Forms-Anwendungen geworfen, wenn Code, der im falschen Thread-Kontext läuft, auf die Benutzerschnittstelle zugreift.

Bei vielen Anwendungen gibt es nur einen Thread, der die Benutzerschnittstelle verwaltet, und es gibt dann mehrere weitere Threads, die zum Beispiel Berechnungen im Hintergrund ausführen. Wenn nun die Ergebnisse der Berechnungen in einem `TextBox`-Steuerelement ausgegeben werden sollen, so muss der Code, der die `Text`-Eigenschaft der `TextBox` manipuliert, im genau gleichen Thread laufen, in dem die `TextBox` auch erzeugt wurde.

Natürlich kann es auch Anwendungen geben, in denen die Benutzerschnittstelle mit mehreren Threads gleichzeitig läuft. Stellen Sie sich eine Anwendung vor, die aus mehreren Fenstern besteht, die gleichzeitig geöffnet sind. Sie können zum Beispiel jedem Fenster einen eigenen Thread zur Verfügung stellen. In diesem Fall müssen Sie aber sicherstellen, dass beim Zugriff auf die Ressourcen eines Fensters immer in den richtigen Thread umgeschaltet wird.

Das erste Beispiel in diesem Kapitel soll zeigen, wie man es nicht machen darf. In der Anwendung, die aus Listing 10.1 und Listing 10.2 besteht, werden im XAML-Code zwei Schaltflächen erzeugt, die jeweils eine Ereignismethode aufrufen. Mit der oberen Schaltfläche wird im C#-Code (Methode: *OnTextOutputNormal*) die Methode *DoOutput* ganz normal im Thread der Benutzerschnittstelle aufgerufen. Dieser Aufruf sollte ordnungsgemäß funktionieren. In der Ereignismethode der zweiten Schaltfläche (Methode: *OnTextOutputThread*) wird jedoch zuerst ein neuer Thread erzeugt und danach wird die Methode *DoOutput* in diesem Thread aufgerufen. Dieser direkte Zugriff auf das `TextBox`-Steuerelement aus einem anderen Thread heraus sollte fehlschlagen.

Bernd Marquardt: Microsoft Windows Presentation Foundation - Crashkurs, 2. überarbeitete Auflage, Microsoft Press 2011 (ISBN 978-3-86645-553-5)

Wenn Sie die Test-Anwendung (Abbildung 10.1) im *Debug*-Modus starten (Menübefehl: *Debug* > *Start with Debugging*) und die zweite Schaltfläche anklicken, wird die *InvalidOperationException* ausgelöst und der Zugriff auf die `TextBox` in der Methode *DoOutput* findet nicht statt. Der Debugger hält die Anwendung genau in dieser Zeile des Codes an.

```
<Window x:Class="WpfThread_FALSCH.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="200" Width="200">
  <Grid>
    <StackPanel>
      <Button Margin="5" Height="30" Click="OnTextOutputNormal"
              Content="Text schreiben (Normal)" />
      <Button Margin="5" Height="30" Click="OnTextOutputThread"
              Content="Text schreiben (mit Thread)" />
      <Label Margin="5">Text-Ausgabe:</Label>
      <TextBox Margin="5" Name="textBoxOut" FontSize="14" />
    </StackPanel>
  </Grid>
</Window>
```

Listing 10.1 Die Benutzerschnittstelle für die Test-Anwendung

```
using System;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Threading;

namespace WpfThread_FALSCH
{
  public partial class MainWindow : Window
  {
    public MainWindow()
    {
      InitializeComponent();
    }

    private void OnTextOutputNormal(object sender, RoutedEventArgs e)
    {
      // TextBox-Methode aus richtigem Thread aufrufen
      DoOutput();
    }

    private void OnTextOutputThread(object sender, RoutedEventArgs e)
    {
      ThreadStart thStart = new ThreadStart(this.DoOutput);
      Thread th = new Thread(thStart);

      // TextBox-Methode aus anderem Thread aufrufen
      // Es wird eine Exception geworfen...
      th.Start();
      th.Join();
    }
```

```
    private void DoOutput()
    {
      // Diese Methode greift auf die Benutzerschnittstelle zu
      textBoxOut.Text = "Hallo, WPF!";
    }
  }
}
```

Listing 10.2 Der Code, der falsch auf die `TextBox` zugreift

Abbildung 10.1 Die Beispielanwendung hat den Text »normal« in der `TextBox` ausgegeben

Das Dispatcher-Objekt

Wie können Sie nun das Problem, welches im vorherigen Absatz aufgezeigt wurde, umgehen und warum gibt es dieses Problem in WPF überhaupt?

Sehr viele WPF-Objekte sind *Thread-affin*, d.h., sie gehören zu einem bestimmten Thread und dürfen auch nur von genau diesem Thread aus manipuliert werden. Dieses Verarbeitungsmodell bezeichnen wir als *Single Threaded Apartment-Modell* (STA). Für WPF hat dieses Modell gegenüber dem Multi Threaded Apartment-Modell (MTA-Modell) zwei wichtige Vorteile. Einerseits ist das STA-Modell sehr einfach und übersichtlich. Sie können ganz normal programmieren, ohne einen Gedanken an die Synchronisierung verschiedener Tätigkeiten zu verschwenden, da es ja nur einen Thread gibt. Der zweite wichtige Vorteil für dieses Modell liegt in der Kompatibilität zu existierenden Benutzerschnittstellen-Bibliotheken, wie Win32, Microsoft Foundation Classes (MFC) oder Windows Forms.

WPF benutzt also auch das STA-Modell und aus diesem Grund benötigen wir einen Mechanismus, um bei Anwendungen mit mehreren Threads wieder in genau den Thread umzuschalten, in dem die WPF-Elemente, auf die zugegriffen werden soll, erzeugt worden sind.

Fast jede Klasse in WPF ist von der Klasse `DependencyObject` abgeleitet. Bei der Erzeugung eines WPF-Objekts sorgt diese Klasse nun dafür, dass im erzeugten Objekt eine Referenz auf genau das `Dispatcher`-Objekt gespeichert wird, in dessen Kontext das Objekt erstellt wurde. Jeder Thread hat ein solches `Dispatcher`-Objekt, welches die einzelnen Arbeitseinheiten des Threads steuert. Somit gibt es eine eindeutige Zuordnung von einem Steuerelement zu einem Thread.

Für jedes WPF-Element, welches von der `DependencyObject`-Klasse abgeleitet wurde, können Sie nun das dazugehörige `Dispatcher`-Objekt ermitteln, welches zwei sehr wichtige Methoden enthält: `CheckAccess` und `VerifyAccess`. Die beiden Methoden prüfen, ob von der aktuellen Aufrufstelle im Code auf das WPF-Element

zugegriffen werden darf, d.h., ob wir uns in dem Thread befinden, in welchem das WPF-Element erzeugt wurde. `CheckAccess` gibt das Ergebnis als Variable vom Typ `bool` zurück, während `VerifyAccess` eine Ausnahme auslöst, wenn wir uns nicht im richtigen Thread befinden. Weiterhin finden wir im `Dispatcher`-Objekt die beiden Methoden `Invoke` und `BeginInvoke`, die wir benutzen können, um ggf. in den korrekten Thread-Kontext umzuschalten. Das Szenario soll nun an einem Beispiel vertieft werden.

Die Benutzerschnittstelle der Anwendung (Listing 10.3) enthält nur zwei Schaltflächen, welche in einem `StackPanel`-Element untereinander angeordnet sind. Für beide Schaltflächen wird das `Click`-Ereignis jeweils auf eine andere Ereignismethode geleitet. Für die obere Schaltfläche soll die Benutzerschnittstelle nun ganz normal im Thread der Anwendung modifiziert werden. In der Ereignismethode der unteren Schaltfläche wird dagegen zuerst ein neuer Thread gestartet, aus dem dann der Text auf der Schaltfläche geändert werden soll. In diesen beiden Szenarien ist eine unterschiedliche Vorgehensweise erforderlich (Listing 10.4).

```
<Window x:Class="Thread1.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Benutzung eines Threads" Height="300" Width="300"
    >
    <Grid>
      <StackPanel>
        <Button Margin="10" Width="150" Height="30" Click="OnClickNormal">Test Normal</Button>
        <Button Margin="10" Width="150" Height="30" Click="OnClickThread">Test Thread</Button>
      </StackPanel>
    </Grid>
</Window>
```

Listing 10.3 Die Benutzerschnittstelle für den Multithreading-Test

```
using System;
using System.Threading;
using System.Windows;
using System.Windows.Threading;
using System.Windows.Controls;

namespace Thread1
{
  public partial class MainWindow :Window
  {
    // Funktionszeiger für den Invoke-Aufruf aus dem anderen Thread
    delegate void UpdateUIDelegate(Button btn, bool bInvoked);
    Button btn = null;

    public MainWindow()
    {
      InitializeComponent();
    }
    private void OnClickNormal(object sender, RoutedEventArgs e)
    {
      btn = sender as Button;
      DoIt();   // Normaler Aufruf
    }
```

```
    private void OnClickThread(object sender, RoutedEventArgs e)
    {
      btn = sender as Button;

      ThreadStart thStart = new ThreadStart(DoIt);   // Aufruf aus neuem Thread
      Thread th = new Thread(thStart);
      th.Start();
    }

    private void DoIt()
    {
      if (btn != null)
      {
        // Darf dieser Thread auf das Objekt btn zugreifen?
        if (btn.Dispatcher.CheckAccess())
        {
          // Ja! Normaler Aufruf!
          UpdateButtonUI(btn, false);
        }
        else
        {
          // Nein!
          // Die Methode muss über die Invoke-Methode des
          // Dispatcher-Objekts ausgeführt werden.
          UpdateUIDelegate delUpdate = new UpdateUIDelegate(UpdateButtonUI);
          btn.Dispatcher.Invoke(DispatcherPriority.Normal, delUpdate, btn, true);
        }
      }
    }

    private void UpdateButtonUI(Button btn, bool bInvoked)
    {
      // Hier greifen wir auf die Schaltfläche zu
      if (bInvoked)
      {
        btn.Content = "WPF - Mit Invoke!";
      }
      else
      {
        btn.Content = "WPF - Ohne Invoke!";
      }
    }
  }
}
```

Listing 10.4 Der Text auf der Schaltfläche wird aus einem Thread geändert

Schauen wir uns nun genau an, was in Listing 10.4 in den beiden Fällen passiert. Betrachten wir zunächst die einfache Möglichkeit ohne Threading. Sie klicken auf die Schaltfläche *Test Normal* und es wird die Ereignismethode `OnClickNormal` aufgerufen. Wir konvertieren den Parameter `sender` der Ereignismethode in ein `Button`-Element und rufen die Methode `DoIt` auf. Die Methode `DoIt` soll hier in diesem Beispiel keine Parameter enthalten, damit sie auch im zweiten Teil des Beispiels für das `ThreadStart`-Objekt benutzt werden kann. Darum haben wir hier in der Klasse `MainWindow` die Klassenvariable `btn` vom Typ `Button` angelegt. Dort steht nun das `sender`-Objekt zur Verfügung, welches immer die Schaltfläche angibt, auf die geklickt wurde. Allerdings können Sie bei einem `Invoke`-Aufruf als weiteren Parameter ein Array vom Typ `object` angeben, welches die Methodenparameter enthält.

In der Methode DoIt ist nun der eigentlich wichtige Code des Beispiels implementiert. Bevor wir überhaupt irgendwelche Methoden aufrufen, prüfen wir, ob das Objekt btn eine null-Referenz enthält. In diesem Fall wird kein weiterer Code ausgeführt. Ansonsten prüfen wir danach mit der Methode CheckAccess, ob wir uns im korrekten Thread befinden, in dem die Schaltfläche manipuliert werden darf. Dazu holen wir uns über die Eigenschaft Dispatcher der Schaltfläche das entsprechende Dispatcher-Objekt, welches dann die benötigte Methode CheckAccess zur Verfügung stellt. Befinden wir uns nun im »richtigen« Thread, d.h. in dem Thread, in welchem die Schaltfläche erzeugt worden ist, dann können wir ohne weitere Vorkehrungen auf das Button-Element zugreifen. Es wird also die Methode UpdateButtonUI mit dem Button-Objekt und dem Parameter false aufgerufen. Dieser zweite Parameter entscheidet nur über den Text, der auf das Button-Objekt geschrieben werden soll (Listing 10.4).

HINWEIS Es gibt zwei Möglichkeiten, wie Sie prüfen können, ob der Zugriff auf ein WPF-Steuerelement von einem bestimmten Code-Teil aus möglich ist. Zunächst können Sie aus der Dispatcher-Klasse die Methode CheckAccess benutzen, die Ihnen *true* zurückgibt, wenn ein Zugriff erlaubt ist. Weiterhin gibt es in der gleichen Klasse die Methode VerifyAccess, die jedoch eine Ausnahme auslöst, wenn ein Zugriff auf das Steuerelement nicht möglich ist.

Nun klicken wir auf die zweite Schaltfläche *Test Thread* und die Ereignismethode OnClickThread wird aufgerufen. Auch hier wird zunächst der sender-Parameter in ein Button-Objekt konvertiert. Danach erzeugen wir ein neues Thread-Objekt und weisen diesem über das ThreadStart-Objekt die Methode DoIt zu. Dieser neue Thread wird mittels der Start-Methode angestoßen. Sobald der Thread vom Betriebssystem eine Zeitscheibe bekommt, läuft er los und führt den Code der Methode DoIt aus. Dieses Mal gibt die Methode CheckAccess jedoch den Wert false zurück, denn wir befinden uns nicht in dem Thread, in dem die Schaltfläche btn instanziiert wurde.

Nun ist eine andere Vorgehensweise erforderlich, denn wir dürfen die Methode UpdateButtonUI nicht mehr direkt aufrufen. Wir benutzen jetzt den »Umweg« über die Invoke-Methode des Dispatcher-Objekts der Schaltfläche. Dazu legen wir eine Instanz des Delegaten UpdateUIDelegate an, der oben in der Window1-Klasse deklariert wurde. Dieser Funktionszeiger zeigt auf die Methode UpdateButtonUI. Bei Aufruf der Invoke-Methode werden die gewünschte Ausführungspriorität, der Funktionszeiger und weitere benötigte Parameter angegeben. Mithilfe des Invoke-Aufrufs wird nun sichergestellt, dass der Code, auf den der Funktionszeiger gesetzt ist (UpdateButtonUI) in genau dem Thread ausgeführt wird, in welchem die Schaltfläche btn erzeugt wurde (Listing 10.4). Damit ist der Affinitätsregel von Windows genüge getan.

Allerdings funktioniert das natürlich nur dann, wenn der Thread, zu dem die Schaltfläche gehört, nicht blockiert ist. In einem solchen Fall blockiert auch der Invoke-Aufruf und dadurch steht natürlich auch unser zweiter Thread still. Der Code, der im Invoke-Aufruf (hier: UpdateButtonUI) ausgeführt werden soll, läuft also erst dann, wenn der entsprechende Thread nicht mehr durch andere Aufgaben blockiert wird. Dieses Verhalten kann so genannte *Deadlocks* hervorrufen und ist normalerweise nicht wünschenswert. Es kann durch eine kleine Modifikation des Codes in Listing 10.4 abgeändert werden. Wir benutzen nämlich statt des Invoke-Aufrufs die Methode BeginInvoke mit den gleichen Parametern:

```
btn.Dispatcher.BeginInvoke(DispatcherPriority.Normal, delUpdate, btn, true);
```

Die Methode BeginInvoke arbeitet asynchron, d. h., die Methode kehrt nach dem Aufruf sofort zum Aufrufer zurück. Der zweite Thread kann nun sofort weiterlaufen, er muss nicht mehr auf die Beendigung des Invoke-Aufrufs warten. Ist der Thread der Schaltfläche beim BeginInvoke-Aufruf blockiert, wird der Aufruf der Methode UpdateButtonUI in eine Warteschlange gestellt und erst dann ausgeführt, wenn der benötigte Thread frei ist.

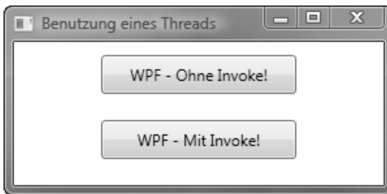

Abbildung 10.2 Nach dem Anklicken beider Schaltflächen

Sie haben im letzten Beispiel gesehen, dass alles, was in WPF mit Multithreading zu tun hat, meistens auch mit dem `Dispatcher`-Objekt abgewickelt wird. Sie können das `Dispatcher`-Objekt des aktuellen Threads durch Aufruf der statischen Methode `CurrentDispatcher` der `Dispatcher`-Klasse abfragen:

```
Dispatcher disp = Dispatcher.CurrentDispatcher;
```

Von diesem `Dispatcher`-Objekt wird jedem in diesem Thread neu erzeugten WPF-Element eine Referenz »mitgegeben«, die dann später dazu dient, den korrekten Thread-Kontext zu ermitteln, wenn das WPF-Element modifiziert werden soll.

Wie bereits erwähnt, arbeitet die Methode `Invoke` synchron. Das bedeutet, dass der Thread, aus dem die Methode aufgerufen wird, solange blockiert, bis der `Invoke`-Aufruf beendet wird. Die Methode `BeginInvoke` arbeitet dagegen asynchron und kehrt sofort zum Aufrufer zurück, sodass der aufrufende Thread sofort weiterlaufen kann. `BeginInvoke` gibt als Rückgabeparameter ein Objekt vom Typ `DispatcherOperation` zurück, über das Sie den aktuellen Status des Aufrufs abfragen können. Außerdem stellt die Klasse `DispatcherOperation` verschiedene Ereignisse zur Verfügung, die Sie über den Ablauf der asynchronen Operation informieren.

Listing 10.5 zeigt eine vereinfachte Benutzerschnittstelle, die es ermöglicht, einen Thread zu starten, und von dort aus über `BeginInvoke` des `Dispatcher`-Objekts wieder die Methode `UpdateButtonUI` aufzurufen, welche die Schaltfläche modifiziert.

```
<Window x:Class="Thread2.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Benutzung von BeginInvoke" Height="100" Width="300"
    >
    <Grid>
      <Button Margin="10" Width="150" Height="30" Click="OnClickThread">Test Thread</Button>
    </Grid>
</Window>
```

Listing 10.5 Eine einfache Benutzerschnittstelle zum Starten eines Threads

```
using System;
using System.Threading;
using System.Windows;
using System.Windows.Threading;
using System.Windows.Controls;

namespace Thread2
{
  public partial class MainWindow : Window
```

```
  {
    delegate void UpdateUIDelegate(Button thButton);
    Button btn = null;

    public Window1()
    {
      InitializeComponent();
    }

    private void OnClickThread(object sender, RoutedEventArgs e)
    {
      btn = sender as Button;

      ThreadStart thStart = new ThreadStart(DoIt);
      Thread th = new Thread(thStart);
      th.Start();
    }

    private void DoIt()
    {
      if (btn != null)
      {
        // Die Methode muss über die Invoke-Methode des
        // Dispatcher-Objekts ausgeführt werden.
        UpdateUIDelegate delUpdate = new UpdateUIDelegate(UpdateButtonUI);
        DispatcherOperation dispOp = btn.Dispatcher.BeginInvoke(DispatcherPriority.Normal, delUpdate, btn);
        dispOp.Completed += new EventHandler(dispOp_Completed);

        if (dispOp.Status == DispatcherOperationStatus.Completed)
        {
          MessageBox.Show("Das ging aber schnell!");
        }
      }
    }

    void dispOp_Completed(object sender, EventArgs e)
    {
      MessageBox.Show("Es ist vollbracht!");
    }

    private void UpdateButtonUI(Button btn)
    {
      btn.Content = "WPF - Mit Invoke!";
      btn.FontSize = 18;
    }
  }
}
```

Listing 10.6 Die Benutzung des `DispatcherOperation`-Objekts für die Statusabfrage

In Listing 10.6 finden wir eine fast identische Vorgehensweise wie im vorherigen Beispiel. Es wird allerdings die Methode `BeginInvoke` des `Dispatcher`-Objekts verwendet und der Rückgabewert einem Objekt vom Typ `DispatcherOperation` zugewiesen. Über dieses Objekt können Sie nun auch eine Ereignismethode für das Ereignis `Completed` deklarieren, die aufgerufen wird, wenn die Ausführung der Methode aus dem `BeginInvoke`-Aufruf

beendet wurde (hier die Methode: dispOp_Completed). Außerdem ist es möglich, den momentanen Status der Operation mit der Eigenschaft Status des DispatcherOperation-Objekts abzufragen. In Listing 10.6 wird nach dem BeginInvoke-Aufruf sofort geprüft, ob der Aufruf bereits beendet wurde. Diese Abfrage wird wohl im Normalfall nicht zutreffen. Es ist sogar sehr wahrscheinlich, dass mit der asynchronen Ausführung der Methode UpdateButtonUI noch gar nicht begonnen wurde. Ein Test der Status-Eigenschaft des Objekts dispOp auf den Wert DispatcherOperationStatus.Pending würde hier wohl eher erfolgreich sein.

Die Klasse DispatcherTimer

Wenn Sie in Ihrer Anwendung mit Timer-Objekten arbeiten müssen, können Sie sowohl den Timer aus dem .NET-Namensraum System.Timers, wie auch das Objekt DispatcherTimer aus dem WPF-Namensraum System.Windows.Threading benutzen. Allerdings ist Vorsicht geboten, denn zwischen beiden Timer-Objekten gibt es einen entscheidenden Unterschied. Wenn Sie den Timer aus .NET benutzen (System.Timers.Timer), wird die Ereignismethode beim Auslösen des Timers in einem Thread des .NET-Threadpools aufgerufen. Das bedeutet, dass Sie die Methoden Invoke oder BeginInvoke benutzen müssen, wenn Sie auf die WPF-Steuerelemente oder –Fenster zugreifen wollen.

Es ist einfacher, direkt mit dem WPF-Timer zu arbeiten. Diese Klasse heißt DispatcherTimer und wie der Name schon sagt, wird die Ereignismethode des Timers über das Dispatcher-Objekt ausgelöst. Der Thread, in dem die Ereignismethode läuft, ist immer der Thread, in welchem auch das Timer-Objekt erzeugt wurde. Wenn Sie also auf die Benutzerschnittstelle im Tick-Ereignis des DispatcherTimer-Objekts zugreifen wollen, müssen Sie das Timer-Objekt ganz einfach im gleichen Thread erzeugen, in dem Sie auch die Elemente der Benutzerschnittstelle generiert haben. Listing 10.7 zeigt ein Beispiel für die Anwendung des DispatcherTimer-Elements.

```
using System;
using System.Windows;
using System.Windows.Threading;
using System.Windows.Media;

namespace Thread3
{
  public partial class MainWindow : Window
  {
    delegate void UpdateUI();

    public MainWindow()
    {
      InitializeComponent();

      DispatcherTimer timer = new DispatcherTimer();

      timer.Tick += new EventHandler(timer_Tick);
      timer.Interval = TimeSpan.FromSeconds(1);
      timer.Start();
    }
```

```
    private void timer_Tick(object sender, EventArgs e)
    {
      if (this.Background == Brushes.Black)
      {
        this.Background = Brushes.Crimson;
      }
      else
      {
        this.Background = Brushes.Black;
      }
    }
  }
}
```

Listing 10.7 Die Anwendung des `DispatcherTimer`-Objekts

Im Konstruktor der Klasse `Window1` erstellen wir das `DispatcherTimer`-Objekt, welches damit im Thread-Kontext der Benutzerschnittstelle enthalten ist. Nach der Definition der Ereignismethode `timer_Tick` und dem Setzen der Eigenschaft `Interval` können wir den Timer starten. Wie Sie in Listing 10.7 sehen können, wird in der `Tick`-Ereignismethode die Benutzerschnittstelle der Anwendung direkt manipuliert. Ein `Invoke`- oder `BeginInvoke`-Aufruf ist nicht notwendig. Die XAML-Datei dieses Beispiels enthält nur den Code, wie er standardmäßig von Visual Studio erzeugt wird (Listing 10.8).

```
<Window x:Class="Thread.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Der DispatcherTimer" Height="300" Width="300"
    >
    <Grid>

    </Grid>
</Window>
```

Listing 10.8 Die Benutzerschnittstelle für die Threading-Anwendung

Nach dem Start der Anwendung wird die Hintergrundfarbe des Fensters im Sekundenabstand umgeschaltet.

Die Klasse BackgroundWorker

Mit .NET Framework 2.0 wurde die Klasse `BackgroundWorker` vorgestellt. Diese Hilfsklasse verbirgt vor dem Entwickler die Notwendigkeit, beim Zugriff auf Elemente der Benutzerschnittstelle zunächst den korrekten Thread mit `Invoke` oder `BeginInvoke` aufzurufen. Für ein `BackgroundWorker`-Objekt können Sie verschiedene Ereignismethoden definieren, die aufgerufen werden, wenn z.B. die Fortschrittsanzeige aktualisiert werden muss oder der Worker-Thread beendet werden soll. Innerhalb dieser Ereignismethoden ist die Benutzung von `Invoke` oder `BeginInvoke` aus dem `Dispatcher`-Objekt nicht erforderlich, da der `BackgroundWorker` selbst in den richtigen Thread umschaltet. Dieses `BackgroundWorker`-Objekt können Sie auch in Ihren WPF-Anwendungen benutzen, um die Programmierung mit asynchronen Methoden zu vereinfachen.

```
<Window x:Class="Background.MainWindow"
    xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
    xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
    Title="BackgroundWorker" Height="80" Width="400"
    >
    <StackPanel Orientation="Horizontal">
      <Label FontSize="18" Margin="5">Fortschritt:</Label>
      <ProgressBar Name="status" Height="20" Width="165" Minimum="0" Maximum="100" Margin="5" />
      <Button FontSize="18" Margin="5" Click="OnBeenden">Abbrechen</Button>
    </StackPanel>
</Window>
```

Listing 10.9 Benutzerschnittstelle mit einer Fortschrittsanzeige

```
using System;
using System.Windows;
using System.ComponentModel;

namespace Background
{
  public partial class MainWindow : Window
  {
    private double dSum = 0.0;
    BackgroundWorker back = null;

    public MainWindow()
    {
      InitializeComponent();

      back = new BackgroundWorker();

      // Ereignismethoden definieren
      back.DoWork += new DoWorkEventHandler(back_DoWork);                             // Rechnen
      back.RunWorkerCompleted += new RunWorkerCompletedEventHandler(back_Completed);  // Fertig
      back.ProgressChanged += new ProgressChangedEventHandler(back_ProgressChanged);  // Fortschritt

      // Unterstützung der Fortschrittsanzeige und des Abbruchs
      back.WorkerReportsProgress = true;
      back.WorkerSupportsCancellation = true;

      // Start der asynchronen Operation
      back.RunWorkerAsync();
    }

    private void back_DoWork(object sender, DoWorkEventArgs e)
    {
      // Hier wird gerechnet...
      for (int j = 0; j < 100; j++)
      {
        // Es wurde abgebrochen, jetzt Berechnungen beenden
        if (back.CancellationPending)
        {
          e.Cancel = true;
          return;
        }
```

```
        // Hier findet die Berechnung statt
        for (int i = 0; i < 10000000; i++)
        {
          dSum += Math.Sqrt(j * 10000000 + i);
        }

        // Ausgabe des Fortschritts
        back.ReportProgress(j + 1);
      }
    }

    private void back_Completed(object sender, RunWorkerCompletedEventArgs e)
    {
      // Arbeit beendet
      if (e.Cancelled)
      {
        // Es wurde abgebrochen
        status.Value = 0;
      }
      else
      {
        // Es wurde alles beendet
        MessageBox.Show("Alles erledigt! Ergebnis: " + dSum.ToString());
      }
    }

    private void back_ProgressChanged(object sender, ProgressChangedEventArgs e)
    {
      // Fortschrittsanzeige
      status.Value = e.ProgressPercentage;
    }

    private void OnBeenden(object sender, RoutedEventArgs e)
    {
      // Abbrechen gedrückt
      back.CancelAsync();
    }
  }
}
```

Listing 10.10 Das `BackgroundWorker`-Objekt im Einsatz

Listing 10.9 und Listing 10.10 zeigen die Anwendung der `BackgroundWorker`-Klasse. Im Konstruktor der `MainWindow`-Klasse wird das Element instanziiert, die Ereignismethoden werden definiert und die gewünschten Eigenschaften gesetzt. Die Verarbeitung der asynchronen Methode wird mit `RunWorkerAsync` aus der `BackgroundWorker`-Klasse gestartet. Dabei wird die Methode `back_DoWork` aufgerufen, die für das Ereignis `DoWork` angegeben wurde. Während der Berechnung im separaten Thread wird in regelmäßigen Abständen geprüft, ob der Berechnungsvorgang abgebrochen werden soll. Hierzu wird die Eigenschaft `CancellationPending` abgefragt. Außerdem wird nach Ausführung einer bestimmten Rechenarbeit die Fortschrittsanzeige aktualisiert. Diese beiden Aktionen sollten nicht zu oft ausgeführt werden, um die Rechengeschwindigkeit nicht zu stark zu reduzieren. Die Methode `back_ProgressChanged` wird immer dann aufgerufen, wenn über `ReportProgress` die Fortschrittsanzeige aktualisiert wird. Da `ReportProgress` aus unserem Rechen-Thread aufgerufen wird, wird

Bernd Marquardt: Microsoft Windows Presentation Foundation - Crashkurs, 2. überarbeitete Auflage, Microsoft Press 2011 (ISBN 978-3-86645-553-5)

im `BackgroundWorker`-Objekt zunächst in den Thread der Benutzerschnittstelle umgeschaltet und dann der Code in der Ereignismethode ausführt. Danach geht es im Rechen-Thread weiter. Sie benötigen hier also keine `Invoke`-Aufrufe mehr, um das `ProgressBar`-Element zu steuern.

Wenn Sie die Schaltfläche *Abbrechen* anklicken, wird über die Methode `CancelAsync` des `BackgroundWorker`-Elements die Eigenschaft `CancellationPending` auf `true` gesetzt, die während der Berechnungen regelmäßig überprüft wird und diese gegebenenfalls unterbricht (Listing 10.10). Das Ergebnis dieser kleinen Demoanwendung sehen Sie in Abbildung 10.3.

In der `BackgroundWorker`-Klasse wird ein »kooperatives« Beenden eines Threads genutzt. Threads, die nicht mehr benötigt werden, sollten nicht »zwanghaft« beendet werden, sondern es sollte dem Thread die Möglichkeit gegeben werden, seine Arbeit in einem sinnvollen Zustand zu beenden. Der Thread sollte, bevor er sich selbst beendet, zum Beispiel alle Dateien, die er geöffnet hat, schließen oder Ressourcen, die er benutzt, wieder freigeben.

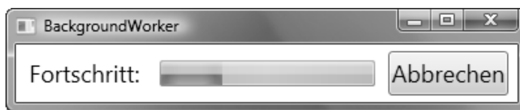

Abbildung 10.3 Während der Berechnungen mit dem `BackgroundWorker`-Objekt

WPF und die Task Parallel Library (TPL)

In .NET Framework 4.0 (Visual Studio 2010) wurde eine neue Bibliothek vorgestellt, die es ermöglicht, parallel ausführbaren Code einfacher zu erstellen, ohne direkt mit mehreren Threads zu programmieren. Diese Bibliothek heißt *Task Parallel Library* (TPL) und stellt dem Entwickler sehr viele Klassen zur Verfügung, welche die Erstellung von parallel auszuführendem Code wesentlich erleichtern.

Die bekannte `Thread`-Klasse in .NET Framework kann zwar weiterhin benutzt werden, aber sie konfrontiert uns mit einigen Schwierigkeiten und Problemen, auf die wir bei der Programmierung mit normalen Threads unbedingt achten müssen. Die neue TPL aus .NET Framework 4 kann mit der WPF-Klassenbibliothek zusammen benutzt werden. Beachten Sie dabei jedoch, dass das Framework 4.0 und die CLR 4.0 Voraussetzung für die Benutzung der TPL sind. Ältere Projekte müssen also entsprechend migriert werden, bevor die neuen Bibliotheken eingesetzt werden können.

Ein Problem bei der Erstellung guter Benutzerschnittstellen ist die Tatsache, dass längere Rechenoperationen in einer Ereignismethode, zum Beispiel der *Click*-Ereignismethode eines Menüpunktes, die Benutzerschnittstelle blockieren. Das bedeutet, dass Sie keine anderen Befehle (Menüpunkte, Schaltflächen, usw.) mehr anklicken können, bis die Ereignismethode endlich ihre Arbeit beendet hat und verlassen wird. Während dieser Zeit wird für den Anwender in der Benutzerschnittstelle eine Sanduhr als Cursor dargestellt.

HINWEIS In modernen Windows-Versionen sieht man keine Sanduhr mehr, sondern einen kleinen rotierenden Ring. Die Benutzerschnittstelle verhält sich jedoch genauso wie beschrieben.

Nun kann man durch eine einfache Parallelisierung des Codes zumindest die Zeit, welche in einer Ereignismethode verbracht wird, verkürzen. Gänzlich verschwinden lassen kann man die lästige Sanduhr damit aber nicht. Hierzu werden spezielle Verfahren benötigt, die es ermöglichen, bestimmten Code separat in einem *Task* auszuführen.

In den folgenden drei Beispielen möchte ich Ihnen im Ansatz zeigen, wie die Task Parallel Library (TPL) funktioniert und wie Sie damit in Ihren Anwendungen parallelen Code ausführen können. Hierbei wollen wir schrittweise vorgehen und eine einfache Anwendung immer weiter verbessern.

Wir beginnen mit einer ganz normalen Anwendung, in welcher der gesamte Code sequenziell ausgeführt wird. In diesem Fall wird für eine längere Zeit die Sanduhr zu sehen sein und der Anwender kann nicht weiter arbeiten. In der Beispielanwendung sollen die Quadratwurzeln von 0 bis 99.999.999 berechnet und addiert werden. Dabei kommt eine relativ große Zahl heraus und die Berechnung dauert – je nach Rechner – ungefähr eine Sekunde.

```
<Window x:Class="Wpf_TPL1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="200" Width="200">
  <Grid>
    <StackPanel>
      <Button Height="30" Margin="5" Content="Wurzelsumme berechnen"
              Click="OnButtonCalculate" Name="btnStart" />
      <Label Margin="5" Content="Ergebnis:" />
      <TextBox Margin="5" Name="textBoxResult" FontSize="14" Background="Pink" />
      <StackPanel Orientation="Horizontal" Margin="5">
        <TextBlock FontSize="14" Text="Zeit: " />
        <TextBlock FontSize="14" Width="100" Name="textBlockZeit"
                   Background="LightGreen"/>
        <TextBlock FontSize="14" Text=" mSek" />
      </StackPanel>
    </StackPanel>
  </Grid>
</Window>
```

Listing 10.11 Die Benutzerschnittstelle für die Anwendung mit parallelem Code – Variante 1

Die Benutzerschnittstelle der Testanwendung zeigt Listing 10.11. Über die Schaltfläche *Wurzelsumme berechnen* können Sie die lange Berechnung starten. Darunter wird in einer `TextBox` das Ergebnis ausgegeben, sobald die gesamte Schleife abgearbeitet ist. Die Zeit, die dazu benötigt wird, wird im unteren `TextBlock`-Steuerelement in Millisekunden ausgegeben. Abbildung 10.4 zeigt das Ergebnis der sequenziellen Addition der Wurzeln. Der Code in der Ereignismethode der Schaltfläche (Listing 10.12) ist selbst erklärend.

```
using System;
using System.Text;
using System.Windows;
using System.Windows.Input;
using System.Windows.Controls;
using System.Diagnostics;
```

```
namespace Wpf_TPL1
{
  public partial class MainWindow : Window
  {
    Stopwatch sw = new Stopwatch();

    public MainWindow()
    {
      InitializeComponent();
    }

    private void OnButtonCalculate(object sender, RoutedEventArgs e)
    {
      // "Sanduhr" als Cursor
      this.Cursor = Cursors.Wait;

      // Zeitmessung starten
      sw.Reset();
      sw.Start();

      // Berechnung der Wurzelsumme
      double dSumme = 0.0;

      for (int i = 0; i < 100000000; i++)
      {
        dSumme += Math.Sqrt(i);
      }

      // Zeitmessung beenden
      sw.Stop();

      // Ausgabe Ergebnis und Zeit
      textBoxResult.Text = dSumme.ToString();
      textBlockZeit.Text = sw.ElapsedMilliseconds.ToString();

      // Cursor wieder auf den normalen Pfeil setzen
      this.Cursor = Cursors.Arrow;
    }
  }
}
```

Listing 10.12 Sequenzielle Addition der Quadratwurzeln

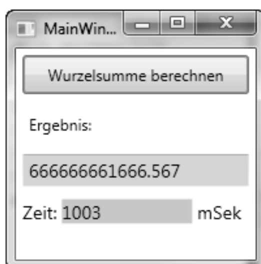

Abbildung 10.4 Nach der sequenziellen Berechnung der Wurzeln

Nun wollen wir den Code in der Ereignismethode parallelisieren. Das bedeutet, dass mehrere Prozessoren bzw. Prozessorkerne für die Berechnung verwendet werden. Die Benutzerschnittstelle wird nach wie vor noch blockieren, aber nicht mehr so lange wie im ersten Beispiel, da die Berechnungszeit für die Quadratwurzeln im parallelen Code natürlich kürzer ist. Der neue C#-Code wird in Listing 10.13 gezeigt.

```
using System;
using System.Text;
using System.Windows;
using System.Windows.Input;
using System.Windows.Controls;
using System.Diagnostics;
using System.Threading.Tasks;

namespace Wpf_TPL2
{
  public partial class MainWindow : Window
  {
    Stopwatch sw = new Stopwatch();
    object summeLock = new object();

    public MainWindow()
    {
      InitializeComponent();
    }

    private void OnButtonCalculate(object sender, RoutedEventArgs e)
    {
      // "Sanduhr" als Cursor
      this.Cursor = Cursors.Wait;

      // Zeitmessung starten
      sw.Reset();
      sw.Start();

      // Berechnung der Wurzelsumme
      double dSumme = 0.0;

      // Parallele Berechnung der Quadratwurzeln
      Parallel.For(0, 100000000,
        () => 0.0,              // Initialisierung LocalState
        (i, pls, tls) =>        // Berechnung einer Teilsumme
        {
          tls += Math.Sqrt(i);
          return tls;
        },
        (teilSumme) =>          // Summierung der Teilsummen
        {
          lock (summeLock)
          {
            dSumme += teilSumme;
          }
        });
```

```
      // Zeitmessung beenden
      sw.Stop();

      // Ausgabe Ergebnis und Zeit
      textBoxResult.Text = dSumme.ToString();
      textBlockZeit.Text = sw.ElapsedMilliseconds.ToString();

      // Cursor wieder auf den normalen Pfeil setzen
      this.Cursor = Cursors.Arrow;
    }
  }
}
```

Listing 10.13 Parallele Berechnung der Quadratwurzeln mit *Parallel.For*

In Listing 10.13 wird die statische Methode `For` aus der Klasse `Parallel` benutzt, um innerhalb dieser Methode die Quadratwurzeln parallel zu berechnen. Hier wird eine so genannte *Lambda-Funktion* benutzt, um den Schleifenkörper darzustellen. Die einzelnen Schleifendurchläufe müssen vollständig unabhängig voneinander sein und können in beliebiger Reihenfolge ausgeführt werden.

Bei Berechnungen, die alle Ergebnisse in einer Summenvariablen (hier: *dSumme*) aufaddieren sollen, gibt es jedoch ein Problem. Wenn nämlich die Berechnung der Wurzeln parallel in mehreren Threads ausgeführt wird, kann es zu Situationen kommen, in denen mehrere Threads gleichzeitig schreibend auf die Variable *dSumme* zugreifen und ein neues Ergebnis hinzuaddieren wollen. Mehrfach schreibende Zugriffe auf Variablen sind jedoch nicht erlaubt. Solche Mehrfachzugriffe bezeichnet man auch als *Data Races*.

Aus diesem Grund wird in der Beispielanwendung der Thread-lokale Status (hier: *tls*) benutzt, um die jeweiligen Zwischenergebnisse zu berechnen. Diese Zwischenergebnisse werden dann in eine andere Lambda-Funktion weitergegeben, in der sie zur Gesamtsumme aufaddiert werden. In dieser Lambda-Funktion muss allerdings ein `lock`-Befehl verwendet werden, damit kein *Data Race* entstehen kann. Die Summierungs-Funktion wird sehr selten aufgerufen, sodass mit Geschwindigkeitseinbußen nicht zu rechnen ist.

Ich habe den Code in diesem Fall auf einem Rechner mit sechs Prozessorkernen laufen lassen. In dieser Variante beträgt die Berechnungszeit der Wurzeln nur noch etwa 0,2 Sekunden, sodass die Blockierungszeit der Benutzerschnittstelle eigentlich kaum noch auffällt. Allerdings findet diese Blockierung statt, da der Aufruf von `Parallel.For(...)` ein synchroner Aufruf ist, der erst dann beendet ist, wenn alle Threads, welche die Berechnungen durchführen, beendet wurden.

HINWEIS Beachten Sie bitte, dass eine Geschwindigkeitssteigerung der Berechnung der Wurzelzahlen mit den parallel arbeitenden Beispielprogrammen nur dann zu erkennen ist, wenn Sie einen Rechner mit einem Mehrkernprozessor verwenden.

Darum wollen wir das Beispiel nun dahingehend erweitern, dass die Benutzerschnittstelle gar nicht mehr blockiert. In diesem Fall müssen wir die Berechnung der Quadratwurzeln komplett mit einem `Task`-Objekt entkoppeln. In Listing 10.14 können Sie die Vorgehensweise bei der Benutzung eines Tasks sehen.

```
using System;
using System.Text;
using System.Windows;
using System.Windows.Input;
using System.Windows.Controls;
using System.Diagnostics;
```

```
using System.Threading.Tasks;
using System.Windows.Threading;

namespace Wpf_TPL3
{
  public partial class MainWindow : Window
  {
    delegate void UpdateUIDelegate(double dResult, long iTime, bool bInvoke);
    Stopwatch sw = new Stopwatch();

    public MainWindow()
    {
      InitializeComponent();
    }

    private void OnButtonCalculate(object sender, RoutedEventArgs e)
    {
      // Start-Button ausschalten
      btnStart.IsEnabled = false;

      // Zeitmessung starten
      sw.Reset();
      sw.Start();

      // Nebenläufige Berechnung der Quadratwurzeln
      Task.Factory.StartNew(() =>
      {
        double dSumme = 0.0;

        for (int i = 0; i < 100000000; i++)
        {
          dSumme += Math.Sqrt(i);
        }

        return dSumme;

      }).ContinueWith(dTask =>
      {
        // Aufruf, wenn Berechnung beendet ist
        sw.Stop();

        // Sind wir im "richtigen" Thread?
        if (textBoxResult.Dispatcher.CheckAccess())
        {
          // Ja, direkte Ausgabe der Ergebnisse
          UpdateUI(dTask.Result, sw.ElapsedMilliseconds, false);
        }
        else
        {
          // Nein, Invoke benutzen
          UpdateUIDelegate delUpdate = new UpdateUIDelegate(UpdateUI);
          textBoxResult.Dispatcher.Invoke(DispatcherPriority.Normal, delUpdate,
            dTask.Result, sw.ElapsedMilliseconds, true);
        }
      });
    }
```

```
    private void UpdateUI(double dResult, long iTime, bool bInvoke)
    {
      textBoxResult.Text = dResult.ToString() + " [" + bInvoke.ToString() + "]";
      textBlockZeit.Text = iTime.ToString();

      // Start-Button wieder einschalten
      btnStart.IsEnabled = true;
    }
  }
}
```

Listing 10.14 Nebenläufige Berechnung mit einem Task aus der TPL

In Listing 10.14 mussten einige Änderungen durchgeführt werden. Wir benutzen nun ein `Task`-Objekt, um die Berechnung »nebenläufig« auszuführen. Das bedeutet jedoch, dass die Berechnung wieder etwa eine Sekunde dauert, aber die Benutzerschnittstelle nun nicht mehr blockiert wird. Wenn das `Task`-Objekt mit der `StartNew`-Methode erzeugt und gestartet wird, müssen wir bedenken, dass es sich hier um einen asynchronen Aufruf handelt. Der Code, der zu der `Task`-Methode gehört, ist vor dem Aufruf von `ContinueWith` beendet. Darum wird die Ereignismethode nicht blockiert und der Anwender könnte sofort weiter arbeiten. Wenn nun die erste Lambda-Funktion mit der Berechnung der Wurzelsumme fertig ist, wird die zweite Lambda-Funktion über `ContinueWith` aufgerufen. Dort wird geprüft, ob wir im richtigen Thread sind und danach wird die Methode `UpdateUI` entweder direkt oder über `Dispatcher.Invoke` aufgerufen. Wenn `Invoke` benutzt werden muss, wird dies in der Ergebnis-`TextBox` durch dem Hinweis [True] ausgegeben. Dies würde bedeuten, dass der Code der `ContinueWith`-Lambda-Funktion nicht im Thread der Benutzerschnittstelle ausgeführt wird.

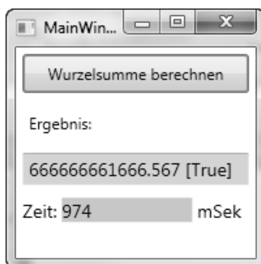

Abbildung 10.5 Nach der Ausführung der Berechnung im `Task`-Objekt

In dieser Variante der Beispielanwendung wurde noch eine weitere Änderung implementiert: Während die Berechnung läuft, wird nämlich die Schaltfläche zum Starten der Aktion ausgeschaltet. Dadurch ist es nicht möglich, gleichzeitig mehrere Wurzelberechnungen durchzuführen. Allerdings ist der entstandene Code nicht gut zu warten oder zu ändern, da der entsprechende Code an vielen Stellen verteilt ist.

Die letzte gezeigte Variante der Anwendung mit dem `Task`-Objekt soll darum nun vereinfacht werden. Es soll der TaskScheduler der TPL verwendet werden, um den Code in der `ContinueWith`-Lambda-Funktion automatisch im richtigen Thread der Benutzerschnittstelle auszuführen (Listing 10.15).

```
using System;
using System.Text;
using System.Windows;
using System.Windows.Input;
using System.Windows.Controls;
```

```
using System.Diagnostics;
using System.Threading.Tasks;

namespace Wpf_TPL4
{
  public partial class MainWindow : Window
  {
    Stopwatch sw = new Stopwatch();

    public MainWindow()
    {
      InitializeComponent();
    }

    private void OnButtonCalculate(object sender, RoutedEventArgs e)
    {
      // Thread-Kontext für Ausgaben merken
      var uiKontext = TaskScheduler.FromCurrentSynchronizationContext();

      // Start-Button ausschalten
      btnStart.IsEnabled = false;

      // Zeitmessung starten
      sw.Reset();
      sw.Start();

      // Nebenläufige Berechnung der Quadratwurzeln
      Task.Factory.StartNew(() =>
      {
        double dSumme = 0.0;

        for (int i = 0; i < 100000000; i++)
        {
          dSumme += Math.Sqrt(i);
        }

        return dSumme;

      }).ContinueWith(dTask =>
      {
        // Aufruf, wenn Berechnung beendet ist
        sw.Stop();

        // Ausgabe ist jetzt möglich
        textBoxResult.Text = dTask.Result.ToString();
        textBlockZeit.Text = sw.ElapsedMilliseconds.ToString();

        // Start-Button wieder einschalten
        btnStart.IsEnabled = true;

      }, uiKontext);
    }
  }
}
```

Listing 10.15 Direkte Ausgabe der Ergebnisse in `ContinueWith`

In Listing 10.15 merken wir uns am Anfang der Ereignismethode den momentanen Thread-Kontext (hier: `uiKontext`) durch den Aufruf der Methode `FromCurrentSychronisationContext` aus der Klasse `TaskScheduler`. An dieser Stelle wird genau der Thread-Kontext der Benutzerschnittstelle zurückgegeben. Nun wird wieder die Berechnung der Quadratwurzeln in der ersten Lambda-Funktion des `Task`-Objekts gestartet. Die Ereignismethode wird wiederum nicht blockiert. Wenn die Berechnung beendet wurde, wird die zweite Lambda-Funktion (`ContinueWith`) aufgerufen. Allerdings wird nun als Parameter die Variable `uiKontext` mit dem Thread-Kontext der Benutzerschnittstelle übergeben. Der TaskScheduler von WPF lässt den Code in der `ContinueWith`-Methode nun im Thread der Benutzerschnittstelle ablaufen und wir können direkt auf die Steuerelemente zugreifen und diese manipulieren.

Die Berechnungszeit beträgt jedoch immer noch etwa eine Sekunde. Nun kann als letzte Änderung in diesem Beispiel in der Lambda-Funktion des `Task`-Objekts statt der normalen `for`-Schleife die Methode `Parallel.For` benutzt werden, um die Geschwindigkeit zu steigern. In diesem Fall kann für die Schleife wieder der Code aus dem zweiten Beispiel benutzt werden (Listing 10.16).

```
private void OnButtonCalculate(object sender, RoutedEventArgs e)
{
  // Thread-Kontext für Ausgaben merken
  var uiKontext = TaskScheduler.FromCurrentSynchronizationContext();

  // Start-Button ausschalten
  btnStart.IsEnabled = false;

  // Zeitmessung starten
  sw.Reset();
  sw.Start();

  // Nebenläufige Berechnung der Quadratwurzeln
  Task.Factory.StartNew(() =>
  {
    double dSumme = 0.0;

    // Parallele Berechnung der Quadratwurzeln
    Parallel.For(0, 100000000,
      () => 0.0,              // Initialisierung LocalState
      (i, pls, tls) =>        // Berechnung einer Teilsumme
      {
        tls += Math.Sqrt(i);
        return tls;
      },
      (teilSumme) =>          // Summierung der Teilsummen
      {
        lock (summeLock)
        {
          dSumme += teilSumme;
        }
      });

    return dSumme;

  }).ContinueWith(dTask =>
  {
    // Aufruf, wenn Berechnung beendet ist
    sw.Stop();
```

Bernd Marquardt: Microsoft Windows Presentation Foundation - Crashkurs, 2. überarbeitete Auflage, Microsoft Press 2011 (ISBN 978-3-86645-553-5)

```
        // Ausgabe ist jetzt möglich
        textBoxResult.Text = dTask.Result.ToString();
        textBlockZeit.Text = sw.ElapsedMilliseconds.ToString();

        // Start-Button wieder einschalten
        btnStart.IsEnabled = true;

      }, uiKontext);
    }
```

Listing 10.16 `Task`-Objekt mit parallel ausgeführter Schleife

Nun wird die Summierung der Quadratwurzeln sehr schnell durchgeführt und die Benutzerschnittstelle wird während der Berechnung nicht blockiert (Abbildung 10.6). Dies ist der Idealfall, der aber in vielen realen Anwendungssituationen nicht immer erreicht werden kann.

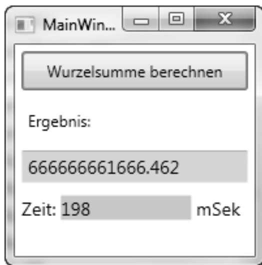

Abbildung 10.6 Das fertige parallel arbeitende Beispielprogramm

Zusammenfassung

Auch mit WPF müssen Sie darauf achten, dass der Code, welcher auf die Benutzerschnittstelle zugreift, im richtigen Thread ausgeführt wird. Mit dem `Dispatcher`-Objekt können Sie den korrekten Thread ermitteln bzw. über die `Invoke`- oder `BeginInvoke`-Methode in diesen Thread umschalten und den Code ausführen lassen.

Die Programmierung von Anwendungen mit mehreren Threads bleibt allerdings auch mit Windows Presentation Foundation kompliziert und muss sorgfältig geplant werden. Neue Bibliotheken, wie die Task Parallel Library, bringen deutliche Vereinfachungen mit sich. Sie müssen auch hier eventuell für die Synchronisierung der Threads sorgen und dabei auch, wie bisher, Deadlocks, Data Races und andere Unannehmlichkeiten vermeiden.

Trotzdem ist die Programmierung von Anwendungen mit mehreren Threads in der letzten Zeit sehr wichtig geworden, da immer mehr Computer mit neuen Mehrkern-Prozessoren ausgerüstet werden. Die Rechenleistung dieser Prozessoren können Sie nur dann vollständig für Ihre Anwendung ausnutzen, wenn Sie mehrere Threads im Code benutzen.

Bernd Marquardt: Microsoft Windows Presentation Foundation - Crashkurs, 2. überarbeitete Auflage, Microsoft Press 2011 (ISBN 978-3-86645-553-5)