Übungsblatt 5 (Lösung)

Aufgabe 13 Multi-Threading Einstieg

Die Aufgabe dient lediglich dazu, sich mit den prinzipiellen Aufbau eines Multi-Threadingfähigen Programms in der eigenen Programmierumgebung zu beschäftigen. Der Aufbau des Programms kann sehr einfach sein.

```
static void Main(string[] args)
{
    var thread = new Thread(Countdown);
    thread.Start();
    ConsoleInput();
}

static void ConsoleInput()
{
    while (true)
        Console.ReadLine();
}

static void Countdown()
{
    var time = 10;

    while (time > 0)
    {
        Console.WriteLine("Noch {0} Sekunden.", time--);
        Thread.Sleep(1000);
    }

    Environment.Exit(0);
}
```

Im .NET-Framework gibt es eine *Thread*-Klasse. Diese benötigt nur eine Startfunktion um diese in einem neuen Thread laufen zu lassen. Die Klasse hat auch einige statische Methoden wie *Sleep*. Diese Methoden reflektieren immer auf den Thread, von dem der Aufruf ausgeht, zurück.

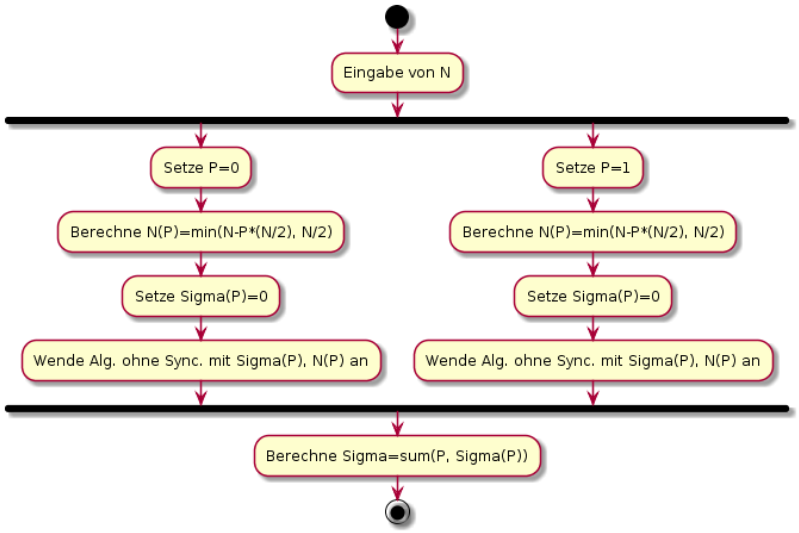
Aufgabe 14 Thread-Synchronisierung

Das dargelegt Problem ist sehr einfach. Im Prinzip ist es möglich, die gesamte Aufgabenstellung in eine einzige Klasse zu packen. Der dargestellte Code löst den einfachen, beschriebenen Fall. Eine Verbesserung sind daher sehr leicht möglich und wird in dem anschließend gezeigten Aktivitätsdiagramm gezeigt.

```
class Workgroup
{
    Int32 sigma;
    Int32 iterationsPerThreads;
```

```
    Boolean synchronized;
    Object sync;
    Dictionary<Boolean, Action> workers;

    public Workgroup()
    {
        sigma = 0;
        sync = new Object();
        workers = new Dictionary<Boolean, Action>();
        workers[true] = () => { lock (sync) sigma++; };
        workers[false] = () => { sigma++; };
    }

    public Int32 Sigma
    {
        get { return sigma; }
    }

    public Int32 IterationsPerThread
    {
        get { return iterationsPerThreads; }
        set { iterationsPerThreads = value; }
    }

    public Boolean IsSynchronized
    {
        get { return synchronized; }
        set { synchronized = value; }
    }

    public void DoWork()
    {
        var worker = workers[synchronized];

        for (var i = 0; i < iterationsPerThreads; i++)
            worker();
    }
}
```

Die Synchronisierung ist hier über einen *lock*-Block implementiert. Diese Blockade benötigt zusätzlich Zeit. Ohne diesen notwendigen Schritt ist eine Race Condition vorhanden, welche letztlich zu falschen Ergebnissen führen wird.
Ein optimierter Algorithmus würde versuchen soviel Arbeit wie möglich lokal auszuführen, ohne dass eine Blockade notwendig wäre. Am Ende wäre nur noch eine Operation notwendig, welche alle Teilergebnisse zusammenfasst.



Nicht mehr kann man nahezu die gesamte Arbeit ohne die Information aus den anderen Threads ausführen. Nichtsdestotrotz besteht ein hochprozentiger Anteil bei der Entwicklung von Algorithmen für parallele Ausführung darin, die vorhandenen Abhängigkeiten zu minimieren und lokalisieren.

Aufgabe 15 Ein einfacher Thread-Pool

Zunächst die Definition der beiden notwendigen Interfaces.

```
interface IWorker
{
    void Run();
}
interface IWorkUnit
{
    void Process();
}
```

Die einzige Konkretisierung wird in diesem Fall *WorkerThread* genannt. Der Name setzt sich aus der Verwendung eines Workers in einem Thread zusammen.

```
class WorkerThread : IWorker
{
    IWorkUnit unit;

    public WorkerThread(IWorkUnit unit)
    {
        this.unit = unit;
    }

    public void Run()
    {
        unit.Process();
    }

    public override string ToString()
    {
```

```
        return String.Format("Worker for unit {0}", unit.ToString());
    }
}
```

Letztlich nimmt dieser einfache *WorkerThread* nur eine *IWorkerUnit* auf, welche prozessiert wird, sobald die *Run* Methode aufgerufen wird.
In dieser Aufgabe gibt es nur eine Implementierung des *IWorkUnit* Interfaces. Der *CountdownWorker* soll dabei eine gewisse Zeit schlafen. Die Methode wurde so implementiert, dass es theoretisch möglich ist, die aktuell verstrichene Zeit in Sekunden abzufragen.

```
class CountdownWorker : IWorkUnit
{
    readonly Int32 target;
    Int32 elapsed;

    public CountdownWorker(Int32 time)
    {
        target = time;
        elapsed = 0;
    }

    public void Process()
    {
        while (elapsed != target)
        {
            Thread.Sleep(1000);
            elapsed++;
        }
    }

    public override String ToString()
    {
        return String.Format("Countdown to {0}s", target.ToString());
    }
}
```

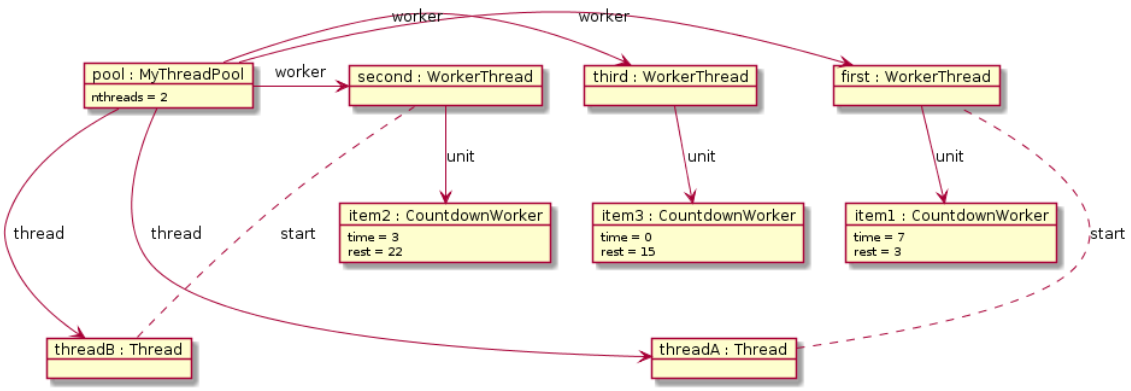
Alles was noch fehlt ist eine Implementierung des Thread-Pools. Der wichtigste Teil dieser Klasse besteht darin, notwendige Synchronisierungsblöcke einzubauen, so dass eine Race-Condition ausgeschlossen ist.

```
class MyThreadPool : IDisposable
{
    Thread[] workers;
    Queue<IWorker> queue;
    Boolean initialized;
    Object sync;

    public MyThreadPool(Int32 size)
    {
        sync = new Object();
        queue = new Queue<IWorker>();
        workers = new Thread[size];
        Logger("Created pool with {0} threads", size);
    }
```

```
public Boolean Running
{
    get { return initialized; }
}

public void Initialize()
{
    if (initialized)
        return;

    for (var i = 0; i < workers.Length; i++)
    {
        workers[i] = new Thread(PoolKernel);
        workers[i].Start(i);
    }

    initialized = true;
    Logger("Pool initialized");
}

public void Shutdown()
{
    if (!initialized)
        return;

    for (var i = 0; i < workers.Length; i++)
        workers[i].Abort();

    initialized = false;
    Logger("Pool shutdown");
}

void PoolKernel(Object id)
{
    while (true)
    {
        IWorker worker = null;

        lock (sync)
        {
            if (queue.Count != 0)
                worker = queue.Dequeue();
        }

        if (worker != null)
        {
            Logger("Running {0} from thread {1}", worker, id);
            worker.Run();
            Logger("Finished {0} from thread {1}", worker, id);
        }
        else
            Thread.Sleep(10);
    }
}
```

```
    public void Queue(IWorker worker)
    {
        Logger("Queuing worker {0}", worker);
        lock (sync) queue.Enqueue(worker);
    }

    void Logger(String message, params Object[] args)
    {
        Console.WriteLine("{0} {1}.", DateTime.Now.ToString("hh:mm:ss"), String.
            Format(message, args));
    }

    void IDisposable.Dispose()
    {
        Shutdown();
    }
}
```

In einem Szenario wie von der Aufgabe beschrieben, besteht daher ein 1:1 Verhältnis zwischen *IWorker* und *IWorkUnit* Instanzen. Warum gibt es diese Aufteilung überhaupt? Sie stellt eine wesentlich losere Kopplung dar, und gibt dem Entwickler mehr Flexibilität. So wäre es auch möglich eine Klasse zu entwickeln, welche *IWorker* implementiert und mehrere *IWorkUnit* Instanzen aufnehmen kann. Die Instanzen würden dann z.B. nacheinander abgearbeitet werden.
Das UML Objektdiagramm zum beschriebenen Zeitpunkt:



Wichtig ist, dass die 1:1 Relation erfüllt ist, aber nur 2 *IWorker* mit Threads verbunden sind. Obwohl jedem Objekt ein Name gegeben worden ist, sind die meisten Objekte in diesem Fall anonym (gespeichert in einer Datenstruktur).