Systemnahes Programmieren

Teil 2: Parallele und verteilte Programmierung

Dieses Script war Teil der Vorlesungsunterlagen im Modul „Systemnahes Programmieren“ (PRGSY) der Hochschule Luzern und wurde durch Roger Diehl erstellt. Die handschriftlichen Notizen waren im Zusammenhang mit dem Unterricht entstanden. Keine Gewähr für Korrektheit.

Lucerne University of
Applied Sciences and Arts
HOCHSCHULE
LUZERN
Engineering & Architecture

TA.PRGSY
Systemnahes Programmieren

Teil 2
Parallele und verteilte Programmierung

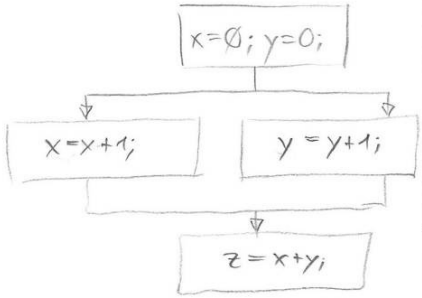
Roger Diehl, Luzern
© Copyright 2010

Inhalt

1. NEBENLÄUFIGKEIT 4
1.1. NEBENLÄUFIGE AUSFÜHRUNG 4
1.2. ATOMARE TEILE UND SYNCHRONISATION 5
1.3. EXPLIZITE SYNCHRONISATION 7
1.4. PROGRAMMEIGENSCHAFTEN 9
2. C# THREADS 11
2.1. DER THREAD LEBENSZYKLUS 11
2.2. ERZEUGEN VON THREADS 13
2.3. BEENDEN VON THREADS 15
2.4. EINFACHE SYNCHRONISATION 20
2.5. SYNCHRONISIERUNGSEREIGNISSE 24
3. ANWENDUNGEN 40
3.1. BOUNDED BUFFER 40
3.2. KOOPERATION ZWISCHEN THREADS 45
4. SOCKET KOMMUNIKATION 53
4.1. EIN- UND AUSGABE-STREAMS 53
4.2. SERIALISIERUNG 60
4.3. SOCKET-KOMMUNIKATION 66
4.4. CLIENT-SERVER 71
4.5. CLIENT-SERVER ANGEWANDT: HTTP-SERVER 76
5. BLUETOOTH FÜR DEN HTA ROBOTER 81
5.1. ÜBERSICHT 81
5.2. RESSOURCEN EINBINDEN 82
5.3. KLASSENSTRUKTUR 83
5.4. SERVER AKTIONEN 87
5.5. CLIENT AKTIONEN 88

Übungen

Übung 1: Multiplikation von Matrizen 10
Übung 2: Kopieren mit Hilfe eines Puffers 10
Übung 3: Programme mit Threads 19
Übung 4: Gemeinsamer Zähler 23
Übung 5: Pferderennen 25
Übung 6: Wait-Pool 29
Übung 7: Nested Monitor 30
Übung 8: Lost Signals 35
Übung 9: Prozessübergreifende Synchronisation 39
Übung 10: Parkhaus 44
Übung 11: Fibonacci mit Callback 48
Übung 12: Mandelbrot-Menge 52
Übung 13: Stream Verarbeitung 59
Übung 14: Nicht serialisierbare Attribute 64
Übung 15: Tiefe Objektkopien 65
Übung 16: Socket Implementationen 70
Übung 17: Einfacher File-Server 75
Übung 18: HTTP-Server 80
Übung 19: Bluetooth Implementationen 90



1. Nebenläufigkeit

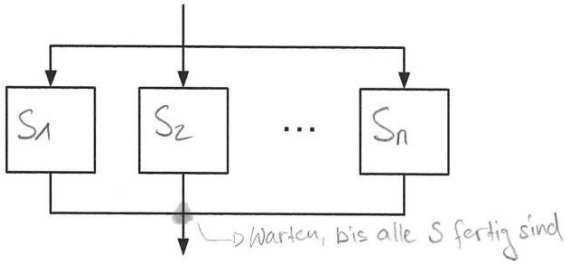
1.1. Nebenläufige Ausführung

Bei der parallelen oder nebenläufigen (concurrent) Ausführung von Programm-Statements S_1 bis S_n werden die Bestandteile der Statements überlappend und gleichzeitig ausgeführt. Statements S_n können natürlich auch Funktions- und Prozeduraufrufe oder Block-Statements sein.

Arbeitsblatt 1: Darstellung nebenläufige Ausführung

Parallele oder nebenläufige (concurrent) Ausführung von Programm-Statements notieren wir im folgenden durch die Schreibweise

Grafische Darstellung:



Notation:

con S_1; S_2; S_3; end;

bzw.

con i=1 to n do $S_{(i)}$; end; //sofern alle S gleich

Anmerkung: Das **con**-Statement terminiert, wenn alle Statements S_i bzw. $S_{(i)}$ terminiert sind.

Im folgenden Beispiel werden die Variablen x und y zuerst initialisiert und dann nebenläufig um 1 inkrementiert, bevor ihre Summe an z zugewiesen wird.

```
x = 0; y = 0;
con x = x + 1; y = y + 1; end;
z = x + y;
```

=> Pseudocode :)

Variablen, die ausserhalb von **con** deklariert werden, sind global. Variablen, die innerhalb von **con** deklariert werden, sind lokal zu dem jeweiligen sequentiellen Programmteil (auch wenn sie den gleichen Namen haben).

1.2. Atomare Teile und Synchronisation

Da innerhalb der Statements in **con** globale Variablen vorkommen können, kann es passieren, dass gleichzeitig auf ein und dieselbe Variable zugegriffen wird. An diesen Stellen ist einige Vorsicht geboten, da sonst die Inhalte dieser Variablen am Programmende falsche oder zufällige Werte enthalten können.

Arbeitsblatt 2: Zugriff auf einen gemeinsamen Speicher

Beispiel:

```
x = 0; y = 0;
con z = x + y; { x = 1; y = 2; } end;
```

Statements A	Statements B
`z = x + y`	`x = 1` `y = 2`

Begriffe: Kooperatives Multitasking = Prozess gibt Resource selber frei
Präemptives Multitasking = Scheduler entzieht die Resource nach bestimmter Zeit

gewünschter Ablauf (optimal)

möglicher Ablauf (schlecht)

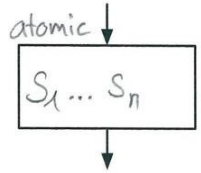
Aufgrund der gezeigten Problematik kommt man zu folgender Definition von elementaren oder atomaren Programmteilen.

Arbeitsblatt 3: Atomare Teile

Definition: Ein Programmteil heisst *Atomare Aktion* falls kein Zwischenschritt, kein Zwischenergebnis und kein Zwischenzustand in einem anderen parallel dazu abgearbeiteten Programmteil sichtbar ist.

Atomare Aktion notieren wir im folgenden durch die Schreibweise

Grafische Darstellung:

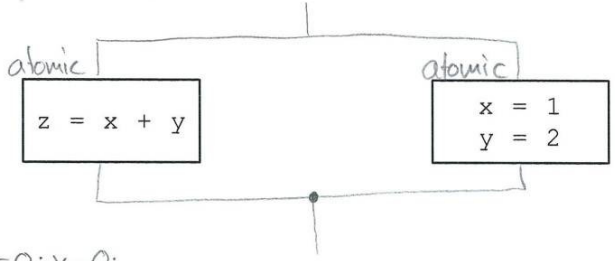


$S_1 - S_n$ werden atomar ausgeführt

Notation:

atomic S_1; S_2; ... S_n; end;

Wie muss das gezeigten Beispiel ergänzt werden, damit es atomar ausgeführt wird ? Wieviele mögliche Resultate liefert das Programm nun ?




```
x=0; y=0;
con
  atomic z= x+y; end;
  atomic x=1; x=2; end;
end;
```

} z entweder Ø oder 3

In der Theorie würde es reichen, wenn der linke ODER der rechte Prozess atomar abläuft.
In der Praxis müssen BEIDE Prozesse atomar ausgeführt werden!

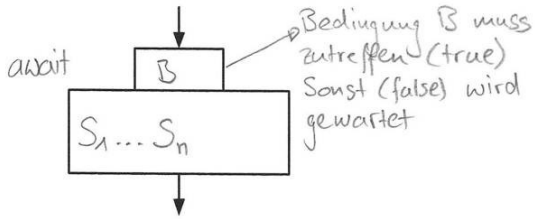
1.3. Explizite Synchronisation

Den kritischen Bereich des vorhergehenden Beispiels, die Summenbildung mit globalen Variablen, kann man mit **`atomic`** kontrollieren. Aber bevor überhaupt summiert werden kann, muss sicher sein, dass die Variablen initialisiert sind. Beim Summen-Beispiel macht man das, in dem **`con`** erst nach der Initialisierung aufgerufen wird. Bei komplexeren Datenstrukturen muss aber unter Umständen die Initialisierung mit in **`con`** einbezogen werden. In dieser Situation benötigen man eine Möglichkeit zu warten, bis dieser Schritt abgeschlossen ist.

Arbeitsblatt 4: Explizite Synchronisation

Bei nicht vermeidbaren Abhängigkeiten zwischen nebenläufigen Programmteilen, d.h. falls die explizite Synchronisation an Bedingungen gebunden ist, notieren wir im folgenden durch die Schreibweise

Grafische Darstellung:



Notation:

await B[then S_1;... S_n;] end;

Das **`await`**-Statement ist per Definition ein atomarer Programmteil, d.h. ausführlich geschrieben wäre es

atomic await B[then S_1;... S_n] end; end;

Anmerkung: Der Boolesche Ausdruck B bezeichnet die (Verzögerungs-)Bedingung, die erfüllt sein muss, bevor die Statements ausgeführt werden. Der **`then`**-Teil in eckigen Klammern ist optional. Wird er weggelassen, bewirkt das **`await`**-Statement nur eine Verzögerung bis die Bedingung wahr wird. Die Ausführung von **`await`** muss als atomar verlangt werden, damit in den Variablen, von denen B abhängt, keine Änderungen während der Ausführung der S_i sichtbar sind. Insbesondere ist B während der gesamten Ausführung der S_i wahr

Arbeitsblatt 4: Explizite Synchronisation

Beispiele:

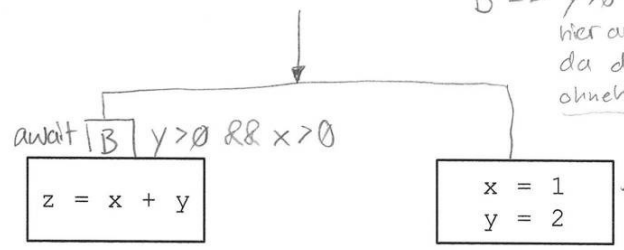
Hier wird gewartet

Hier wird nur unterschieden

`await s > 0 then s = s - 1; end;`	**`await`** wartet, bis die Variable s positiv geworden ist, dann wird s um 1 verringert.
`atomic if B then s = s + 1; end; end;`	Dieses Statement wartet nicht; falls die Bedingung B wahr ist, wird s um 1 erhöht, sonst passiert nichts.

Wie muss das bisher gezeigte Beispiel ergänzt werden, damit es nun absolut korrekt ausgeführt wird ?

B => y>0 würde hier auch reichen, da dieser Teil ohnehin atomar ist




```
x=0; y=0;
con
   await x>0 && y>0 then z=x+y; end;
   x=1;
   y=2;
end;
```

Hinweis: atomar muss nicht explizit erwähnt werden.

1.4. Programmeigenschaften

Wie die vorhergehenden Abschnitte gezeigt haben ist die Programmierung von parallelen oder nebenläufigen Algorithmen nicht einfach, da es bei nicht richtig synchronisierten Programmen vorkommen kann, dass Programmzustände in anderen Prozessen den Zustand in einem Prozess beeinflussen.

Arbeitsblatt 5: Korrektheit von Programmen

Ein Programm ist korrekt, wenn es sicher (keine fehlerhaften Zustände eintreten) und lebendig (irgendwann alle gewünschten Zustände eintreten) ist.

Korrektheit

- **Sicherheit**
  - Gegenseitiger Einfluss von Programmteilen
  - Deadlocks verhindern
  - Interferenz in kritischen Bereichen verhindern
  - Gemeinsamer Resourcenzugriff
- **Lebendigkeit**
  - Jeder Programmteil erhält faire Chance zur Ausführung
  - Lifelock / Starvation verhindern

Unconditionally Fair

```
cont = true;
con
  while cont do /* nothing */ end;
  cont = false;
end;
```

- Teile laufen für sich ohne Rücksicht auf andere

Weak Fair

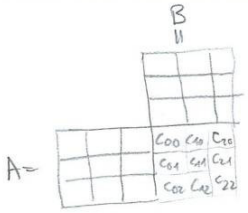
```
cont = true;
try = false;
con
  while cont do
    try = true;
    /* tick */
    try = false;
  end;
  await try then cont = false; end;
end;
```

Taskwechsel → /* tick */

→ keine Garantie auf Taskwechsel

Strong Fair

→ Taskwechsel garantiert
→ mit await erzwingen



$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Übung 1: Multiplikation von Matrizen

Das Problem der Multiplikation von Matrizen lässt sich leicht parallelisieren. Matrizen sind rechteckige, insbesondere quadratische, Zahlenanordnungen, die in vielen naturwissenschaftlichen und ökonomischen Berechnungen eine wichtige Rolle spielen.

Es seien A, B, C jeweils $n \times n$ Matrizen, (z.B. über den rationalen Zahlen, n eine natürliche Zahl). Gesucht ist das Produkt $C = A \cdot B$. Die Matrix $C = \{ c_{ij} \}$ berechnet sich für jedes i und j nach der Vorschrift

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \quad i, j = 1, \ldots, n$$

Erstellen Sie einen parallelen Algorithmus.

Komplexität $O(n^3)$
bei Parallelisierung noch $O(n)$

```
con ~~for~~ i=1 to n do
con ~~for~~ j=1 to n do
      s = ∅;
      for k=1 to n do
        s = s + a[i,k] * b[k,j];
      end;
      c[i,j] = s;
    end;
end;
```

$S_{(i,j)}$

Übung 2: Kopieren mit Hilfe eines Puffers

Gesucht ist ein Kopierprogramm zwischen Produzenten und Konsumenten. Kopieren Sie zwei Vektoren a und b der Länge n >= 0 mit zwei nebenläufigen Prozessen mit Hilfe eines gemeinsamen Puffers buf. Mit anderen Worten: implementieren Sie `buf = a[i], b[i] = buf`, für i von 1 bis n.

Zeigen Sie, dass Ihr Programm korrekt ist !

```
buf=0; p=∅; c=∅; n=100;
con
  producer;
  consumer;
end;
```

```
producer:
  while p<n do
    await p=c end;
    buf = a[p+1];
    p++;
  end;
```

⇒ Produzent wartet bis der Konsument das Produkt konsumiert hat

```
consumer:
  while c<n do
    await p>c end;
    b[c+1] = buf;
    c++;
  end;
```

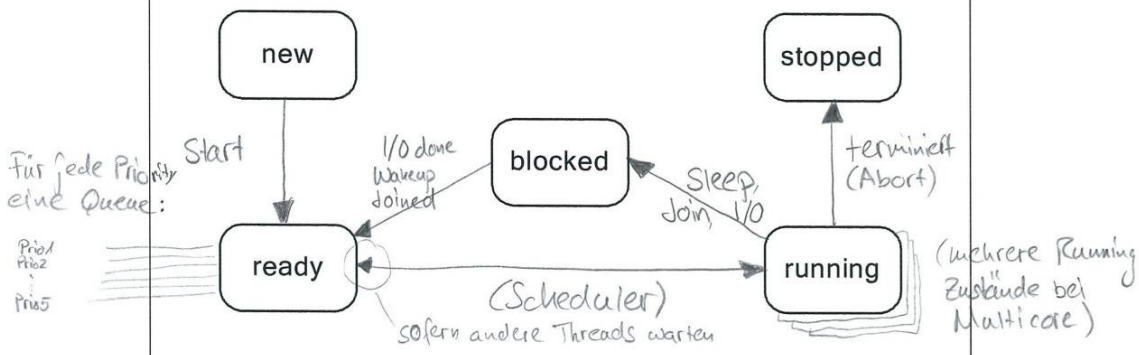
⇒ Konsument wartet auf neues Produkt

2. C# Threads

2.1. Der Thread Lebenszyklus

Arbeitsblatt 6: Thread Lebenszyklus C#

Da von der Common Language Runtime (CLR) mehrere Threads gestartet werden können, kann es sein, dass ein Thread mit einem oder mehreren anderen Threads interagiert. So kann z.B. die Ausgabe eines Threads als Eingabe für einen weiteren Thread verwendet werden. Müssen die Threads dadurch aufeinander warten, sind sie blockiert. Andererseits ist es auch möglich, dass ein laufender Thread, von der CLR unterbrochen wird. Das kann passieren, wenn der Scheduler der CLR auf einen anderen Thread umschalten will. Ein Thread kann nun während seiner Laufzeit prinzipiell folgende Lebenszyklen durchlaufen:



Legende

new	Das Thread-Objekt ist erzeugt, aber noch nicht gestartet.
ready	Der Thread ist gestartet, lokal Speicher (stack) ist zugeteilt, er wartet nur noch auf die Zuweisung des Prozessors.
running	Der Thread führt seine Anweisungen auf dem Prozessors aus.
blocked	Der Thread muss warten bis eine Bedingung erfüllt wird. Dies sind: Warten auf den Ablauf einer gewissen Zeit oder Ende eines anderen Threads. Ein Thread kann auch in den Zustand "blocked" gelangen, wenn eine Betriebsystemroutine aufgerufen wird. Das geschieht beispielsweise bei File-Operationen. Sollte von einem Stream gelesen werden und sind noch keine Daten vorhanden, so kann das Betriebsystem den Thread in den "blocked" Zustand führen.
stopped	Der Thread existiert nicht mehr. Das Thread-Objekt jedoch schon und kann, falls es referenziert ist, benutzt werden. Andernfalls wird es vom Garbage Collector entfernt.

Die C#-Implementierung von Threads ist im Namespace `System.Threading` enthalten. Die gesamte Implementierung umfasst etwa 28 Klassen und Strukturen, sowie Delegate- und Enummerationstypen.

Arbeitsblatt 7: Ein kleiner Thread Überblick

Wichtig: Es muss in allen Beispielen und Übungen der Namespace importiert werden:
`using System.Threading`

Für ~~die~~ das Erzeugen eines Threads muss die Thread-Klasse instanziert werden. Dem Konstruktor ist die Methode zu übergeben, die nebenläufig ausgeführt werden soll.

```
class ThreadTest1 {
   static void Main(string[] args) {
      Thread t = new Thread(WriteY);
      t.Start();
      while (true) Console.Write("x");
   }
   static void WriteY() {
      while (true) Console.Write("y");
   }
}
```

Jeder Thread erhält seinen eigenen Stack, sodass lokale Variablen darauf abgelegt werden können:

```
class ThreadTest2 {
   static void Main(string[] args) {
      new Thread(Go).Start();
      Go();
   }
   static void Go() {
      for (int i = 0; i < 5; i++) Console.Write('x');
   }
}
```

Threads können aber auch gemeinsam einen Stack miteinander teilen:

```
class ThreadTest3 {
   bool done;
   static void Main(string[] args) {
      ThreadTest3 tt = new ThreadTest3();
      new Thread(tt.Go).Start();
      tt.Go();
   }
   void Go() {
      if (!done) { done = true; Console.WriteLine("Done"); }
   }
}
```

wenn Kontextswitch genau hier, so wird "Done" zweimal ausgeführt

Frage: Welches Problem könnte in `ThreadTest3` auftreten ?

→ Hauptsächlich 2 Threads: Main und t. (Plus GC Thread)
→ Beide dieselbe Priorität, daher ca. gleich viele x wie y

2.2. Erzeugen von Threads

Arbeitsblatt 8: Erzeugen und starten von Threads

Im vorhergehenden Arbeitsblatt wurden Threads in der verkürzten Schreibeweise erstellt und gestartet. Üblicherweise wird mit Hilfe der Methode `public delegate void ThreadStart();` angegeben, wo ein Thread zu starten hat.

```
class ThreadTest4 {
   static void Main(string[] args) {
      Thread t = new Thread (new ThreadStart(Go));
      t.Start();
      Go();
   }
   static void Go() {
      Console.WriteLine("hello!");
   }
}
```

Selbstverständlich lassen sich auch Argumente an eine als Thread auszuführende Methode übergeben. Dazu wird die Methode `public delegate void ParameterizedThreadStart(object obj);` benutzt:

```
class ThreadTest5 {
   static void Main(string[] args) {
      Thread t = new Thread(new ParameterizedThreadStart(Go));
      t.Start(true);
      Go(false);
   }
   static void Go(object upperCase) {
      bool upper = (bool)upperCase;
      Console.WriteLine(upper ? "HELLO!" : "hello!");
   }
}
```

object oder kein Parameter

Auch hier gibt es die Möglichkeit der verkürzten Schreibeweise:

```
class ThreadTest6 {
   static void Main(string[] args) {
      Thread t = new Thread(Go);
      t.Start(true);
      Go(false);
   }
   static void Go(object upperCase) {
      bool upper = (bool)upperCase;
      Console.WriteLine(upper ? "HELLO!" : "hello!");
   }
}
```


Arbeitsblatt 9: Erzeugen und starten von Threads

Ein Thread soll Zahlen von 1 bis n addieren und die Summe am Schluss ausgeben. Die Klasse `ThreadAddition` ist eine gewöhnliche C#-Klasse. Sie implementiert eine Methode `Add` mit der geforderten Funktionalität.

```
class ThreadAddition {
   int n;
   String id;

   public ThreadAddition(String id, int n) {
      this.id = id;
      this.n = n;
   }

   public void Add() {
      long sum = 0;
      for (int i = 0; i <= n; i++)
         sum += i;
      Console.WriteLine(id + ": SUM" + n + " -> "+sum);
   }
}
```

Um die Klasse `ThreadAddition` zu testen werden drei Thread-Objekte erzeugt und gestartet.

```
class ThreadTest7 {
   static void Main(string[] args) {
      ThreadAddition work1 = new ThreadAddition("A", 100000000);
      ThreadAddition work2 = new ThreadAddition("B", 100000);
      ThreadAddition work3 = new ThreadAddition("C", 100);

      Thread t1 = new Thread(new ThreadStart(work1.Add));
      Thread t2 = new Thread(new ThreadStart(work2.Add));
      Thread t3 = new Thread(new ThreadStart(work3.Add));

      t1.Start();
      t2.Start();
      t3.Start();
   }
}
```

Fragen:

- Was stellen Sie beim Ausführen der drei Threads fest ?
- Was stellen Sie bei der Ausführung fest, nachdem Sie die folgende Zeile vor dem Thread-Start eingefügt haben ?
  `t2.Priority = ThreadPriority.Lowest;`
- Wie erklären Sie sich das Programmverhalten ?

2.3. Beenden von Threads

Der Thread Lebenszyklus besagt, dass mit dem Ende der Thread-Methode auch der Thread beendet wird. Ist in der Methode eine Endlosschleife programmiert, würde dies dann ein theoretisch nie endender Thread bilden.

Arbeitsblatt 10: Beenden eines Threads

Allgemein ist ein Thread beendet, wenn eine der folgenden Bedingungen zutrifft:

- Die Thread-Methode wird ohne Fehler beendet.
- In der Thread-Methode tritt eine Ausnahme (`Exception`) auf, welche die Methode beendet.
- Der Thread wird von aussen abgebrochen.

Das Beispiel mit dem Thread, der Zahlen von 1 bis n addiert und am Schluss die Summe ausgibt, hat gezeigt wie eine `Add`-Methode ohne Fehler beendet werden kann.

Tritt in der Thread-Methode eine Ausnahme auf, so kann vermutlich nicht mehr ordentlich weitergerechnet werden und die Beendung des Threads ist eine zwingende Folge davon.

Die dritte Möglichkeit, den Thread von aussen abzubrechen, soll das folgende Beispiel zeigen. Dieses erste Beispiel ist der Möglichkeit eines Interrupts in Java nachempfunden. Dabei wird die Addition unterbrochen, wenn die Berechnung länger als 10msec dauert.

```
class ThreadStop1
{
   static void Main(string[] args) {
      Thread t = new Thread(delegate() {
         long sum = 0;
         for (int i = 0; i <= 100000000; i++)
            sum += i;
         Console.WriteLine("Summe= " + sum);
      });

      t.Start();
      Thread.Sleep(10);
      Console.WriteLine("interrupt...");
      t.Interrupt();
   }
}
```

Fragen:

- Was stellen Sie beim Ausführen des Threads fest ?
- Wie erklären Sie sich das Programmverhalten ?
- Wie kann der Thread mit Hilfe von `Interrupt` unterbrochen werden ?

Inline-Methode definieren:
`()=> {...}` oder `delegate() {...}`

Arbeitsblatt 10: Beenden eines Threads (Fortsetzung)

Anders als in Java bewirkt der Aufruf von `Interrupt` nur die Befreiung des Threads aus dem blocked Zustand. Wird `Interrupt` aufgerufen ohne, dass sich der Thread im blocked Zustand befindet, läuft der Thread weiter. Ein Aufruf von `Interrupt` sollte also in der folgenden Form ausgeführt werden.

```
class ThreadStop2
{
   static void Main(string[] args) {
      Thread t = new Thread(delegate() {
         try {
            long sum = 0;
            for (int i = 1; i <= 100; i++) {
               Thread.Sleep(1);
               sum += i;
            }
            Console.WriteLine("Summe = " + sum);
         }
         catch (ThreadInterruptedException) {
            Console.WriteLine("Thread interrupted.");
         }
         finally {
            Console.WriteLine("Thread ends.");
         }
      });

      t.Start();
      Thread.Sleep(500);
      if ((t.ThreadState & ThreadState.WaitSleepJoin) > 0) {
         t.Interrupt();
      }
   }
}
```

Achtung: Wenn die Thread-Methode nicht explizit für Interrupts entworfen wurde, sollte man den Thread `Interrput` nicht verwenden. Der Grund dafür liegt in der potentiellen Unsicherheit des Aufrufs, denn es ist nicht voraussagbar und auch nicht definiert, an welcher Stelle ein Thread unterbrochen wird, wenn ein Aufruf von `Interrput` erfolgt. Es kann nämlich insbesondere vorkommen, dass der Abbruch innerhalb eines kritischen Abschnitts erfolgt oder in einer anwendungsspezifischen Transaktion auftritt, die aus Konsistenzgründen nicht unterbrochen werden darf.

- Thread nur mit Interrupt() beenden, wenn Status WaitSleepJoin !
- Im Zweifelsfall kein Interrupt einsetzen.

Arbeitsblatt 11: Stoppen eines Threads

Das Terminieren eines Threads wird mit der Methode `Abort` durchgeführt. Der Aufruf löst die Ausnahme `ThreadAbortException` aus. Damit ist es möglich, die Methode ordnungsgemäss zu beenden.

```
class ThreadStop3
{
   static void Main(string[] args) {
         Thread t = new Thread(delegate() {
            try {
               Console.WriteLine("Thread starts.");
               while (true) ;
            }
            catch (ThreadAbortException) {
               Console.WriteLine("ThreadAbortException.");
               for (int i = 0; i < 100000000; i++) ;
            }
            finally {
               Console.WriteLine("Thread ends.");
            }
         });

         Console.WriteLine("1. " + t.ThreadState);   //Unstarted

         t.Start();
         Thread.Sleep(1000);
         Console.WriteLine("2. " + t.ThreadState);   // Running

         t.Abort();
         Console.WriteLine("3. " + t.ThreadState);   //AbortRequested

         t.Join();
         Console.WriteLine("4. " + t.ThreadState);   //Stopped
   }
}
```

Starten Sie das Programm `ThreadStop3`.

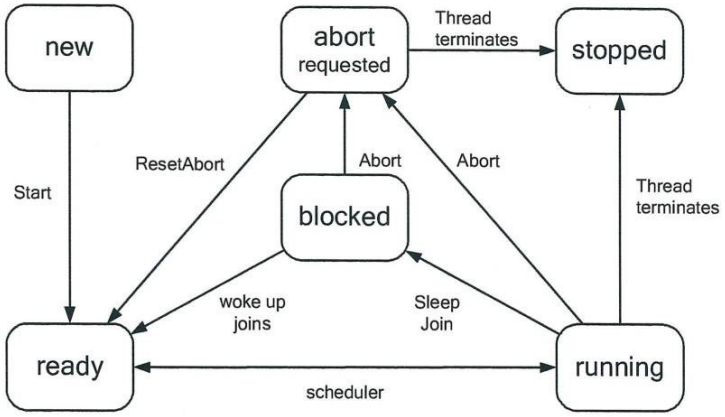
Fragen:

- Was stellen Sie beim Ausführen des Threads fest ?
- Was schliessen Sie aus dem Programmverhalten ?

Bei Abort wird die ThreadAbortException aufgerufen.

Arbeitsblatt 12: Thread Lebenszyklus mit AbortRequested

Der zusätzliche Thread-Zustand `AbortRequested` führt dazu, dass ein Thread nun während seiner Laufzeit prinzipiell folgende Lebenszyklen durchlaufen kann:



Somit ist es möglich ein gestoppten Thread wieder in den ausführungsbereiten Zustand zu versetzen:

```
class ThreadStop4 {
   static void Main(string[] args) {
      Thread t = new Thread(Work);
      t.Start();
      Thread.Sleep(1000); t.Abort();
      Thread.Sleep(1000); t.Abort();
      Thread.Sleep(1000); t.Abort();
   }
   static void Work() {
      while (true) {
         try { while (true); }
         catch (ThreadAbortException) { Thread.ResetAbort(); }
         Console.WriteLine("I will not die!");
      }
   }
}
```

=> Interrupt wirkt nur aus BLOCKED State
=> Abort wirkt aus Running wie auch Blocked State

Übung 3: Programme mit Threads

Nebenläufige Ausführung

Sie erinnern sich an das erste Beispiel aus dem Kapitel "Nebenläufigkeit".

```
x = 0; y = 0;
con x = x + 1; y = y + 1; end;
z = x + y;
```

Randbedingung: Die Variablen `x, y, z` sind global deklariert.

Beantworten Sie die Fragen und erstellen Sie das Programm:

- Wie setzen Sie in C# die Anweisungen innerhalb des `con`-Statement um ?
- Welches spezielle Verhalten im Zusammenhang mit dem `con`-Statement ist noch zu beachten und wie setzen Sie diese Verhalten um ?
- Wie verhält sich das Thread Programm auf den verschiedenen Plattformen ?

– 2 Threads erzeugen
– Starten und mit Join() das Ende von x++ und y++ abwarten.
– z = x+y ausgeben

2.4. Einfache Synchronisation

Die Thread-Synchronisierung umfasst Techniken, die sicherstellen sollen, dass mehrere Threads ihre Zugriffe auf gemeinsam benutzte Ressourcen koordinieren. Diese Koordination kann auf unterschiedliche Weise erreicht werden.

Arbeitsblatt 13: Einfache Blockierung

Idee: Die Threads stimmen sich zeitlich ab.

Im folgenden Beispiel teilen sich der Main-Thread und der Thread `t` die gemeinsam Ressource `long sum`. Thread `t` führt die Addition durch, der Main-Thread gibt das Resultat aus.

```
class SimpleBlocking
{
   static void Main() {
      long sum = 0;

      Thread t = new Thread(delegate() {
         for (int i = 0; i <= 1000; i++)
            sum += i;
      });

      t.Start();

      // Version 1                  => Zustand = Running
      while (sum < 500500) ;        => doof, da Endresultat bekannt sein muss
      // Version 2                  => Zustand => Main = Blocked, t = Running
      Thread.Sleep(1);              => doof, da Wartezeit unbekannt
      // Version 3                  => Zustand = Blocked
      t.Join();   => gute Lösung

      Console.WriteLine("Summe = " + sum);
   }
}
```

Starten Sie das Programm `SimpleBlocking`. mit jeweils einer Version der einfachen Blockierung.

Fragen:

- Funktionieren alle drei Versionen ? Bedingt. Join am besten.
- In welchen Zuständen befinden sich die Threads in den drei Versionen ?
- Bewerten Sie die drei Versionen. Welche ist die beste Variante ? Version 3.

Erstellen Sie einen neuen Thread `s`, der die Ausgabe der Summe übernimmt.

Frage: Was stellen Sie jetzt fest ?

Sleep(0) => Yield, wartet bis zum nächsten Interrupt

Arbeitsblatt 14: lock-Konstrukt

Idee: Die Threads reservieren einen Codebereich für sich.

Zur Synchronisation nebenläufiger Threads wurde das Konzept des Monitors implementiert. C. A. R. Hoare hat dieses Konzept im Aufsatz »Communicating Sequential Processes« von 1978 erstmals veröffentlicht. Ein Monitor ist die Kapselung eines kritischen Bereichs (also eines Programmteils, der nur von jeweils einem Thread zur Zeit durchlaufen werden darf) mit Hilfe einer automatisch verwalteten Sperre. Diese Sperre wird beim Betreten des Monitors gesetzt und beim Verlassen wieder zurückgenommen. Ist sie beim Eintritt in den Monitor bereits von einem anderen Thread gesetzt, muss der aktuelle Thread warten, bis der Konkurrent die Sperre freigegeben und den Monitor verlassen hat.

C# bietet hierfür die Anweisung `lock`, die sicherstellt, dass immer nur ein Thread zu einem gewissen Zeitpunkt einen Codebereich betreten kann. Das `lock`-Konstrukt von C# entspricht dem `synchronized` in Java. Allerdings können mit dem `lock`-Konstrukt nur Codeblöcke geschützt werden und nicht ganze Methoden. ⇒ gut so!

Im folgenden Beispiel teilen sich zwei Threads (`t1` und `t2`) die gemeinsam Ressource `long sum`. Die kritischen Codebreiche werden durch das gemeinsame Objekt `locker` geschützt.

```
class LockingConstruct1
{
   static object locker = new object();

   static void Main() {
      long sum = 0;

      Thread t1 = new Thread(delegate() {
         lock (locker) {
            for (int i = 0; i <= 1000; i++)
               sum += i;
         }
      });

      Thread t2 = new Thread(delegate() {
         lock (locker) {
            Console.WriteLine("Summe = " + sum);
         }
      });

      t1.Start();
      t2.Start();
   }
}
```

} Race Condition! t1 muss lock zuerst erhalten!

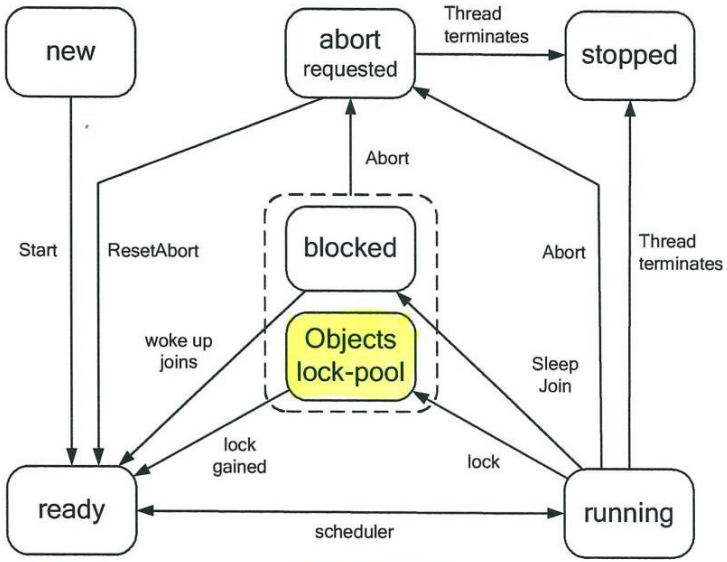
Fragen:

- Warum wird die Summe richtig berechnet ?
- Können Sie sich eine Situation vorstellen, wo die Summe nicht richtig berechnet wird ?

Monitor überwacht Zugriff auf Datenelemente

Arbeitsblatt 15: Thread Lebenszyklus mit lock-Konstrukt

Mit dieser Art der Synchronisation erhält der Thread Lebenszyklus einen zusätzlichen Zustand, den Objekt lock-pool. Jedes Objekt hat genau einen Objekt lock-pool.



Das `lock`-Konstrukt von C# ist eine syntaktische Kurzform und entspricht folgendem Code:

```
static object locker = new object();
...
Monitor.Enter(locker);
try {
   // kritischer Bereich
}
finally {
   Monitor.Exit(locker);
}
```

Die `Monitor` Klasse stammt aus dem Namespace `System.Threading`. Ein Beispiel zur `Monitor` Klasse finden Sie im Programm `LockingConstruct2`.

Übung 4: Gemeinsamer Zähler

Die Schreiboperation muss nicht offensichtlich sein, wie das folgende Programm zeigt. Hier wird auf einen für alle Threads zugreifbarer Zähler (Klasse `Counter`) deklariert. Ein Thread wartet in der `Go`-Methode erst 100msec und gibt dann seine ID und den Zählerstand mit der Methode `NextNumber` aus.

```
class Counter {
   int count = 0;
   public int NextNumber() {
      count++;
      return count;
   }
}

class CounterTest {
   private String id;
   private Counter counter;

   public CounterTest(String id, Counter counter) {
      this.id = id;
      this.counter = counter;
   }

   void Go() {
      while (true) {
         Thread.Sleep(100);
         System.Console.WriteLine(id + counter.NextNumber());
      }
   }
}
```

⇒ bei parallelem Zugriff muss dieser Abschnitt mit lock(...) geschützt werden

Startet man das Programm so wird in den meisten Fällen der `Counter` eine eindeutige Nummer liefern. Allerdings könnte eine Ausgabe auch so aussehen:

```
T2: 9
T3: 10
T2: 12 <---
T1: 12 <---
T3: 13
T1: 15 <---
T2: 15 <---
T3: 16
```

Beim Zählerstand 12, bzw. 15 ist offensichtlich bei der Ausgabe etwas passiert, was nicht gewollt war.

Fragen:

- Wie erklären Sie sich das Programmverhalten ?
- Wo genau im Code tritt das Problem auf ?
- Wie müsste eine korrekte Lösung aussehen ?

2.5. Synchronisierungsereignisse

Arbeitsblatt 16: WaitHandle

Idee: Die Threads warten an einem inaktiven Ereignisobjekt, bis dieses aktiv (frei) geschaltet wird.

Es gibt zwei Arten von Synchronisierungsereignissen: `AutoResetEvent` und `ManualResetEvent`. Der einzige Unterschied zwischen den beiden besteht darin, dass `AutoResetEvent` automatisch von signalisiert (aktiv) zu nicht signalisiert (inaktiv) geändert wird, wenn das Ereignis einen Thread aktiviert. Umgekehrt ist es mit `ManualResetEvent` möglich, eine beliebige Anzahl von Threads über den signalisierten Zustand zu aktivieren, und das Ereignis wird nur in den nicht signalisierten Zustand zurückgesetzt, wenn seine Reset-Methode aufgerufen wird.

Dieses C# Konstrukt entspricht nicht dem `wait` in Java.

Im folgenden Beispiel wird ein inaktives `EventWaitHandle` vom Typ `AutoResetEvent` erzeugt. Ein Thread ruft in der `Waiter`-Methode am `EventWaitHandle` die `WaitOne`-Methode auf. Damit wartet der Thread, bis das `EventWaitHandle` Objekt mit Hilfe der `Set`-Methode aktiviert, und damit der Thread aus dem Wartezustand befreit wird.

```
class BasicWaitHandle
{
   static EventWaitHandle wh = new AutoResetEvent(false);

   static void Main() {
      new Thread(Waiter).Start();
      Thread.Sleep(1000);
      wh.Set();
   }

   static void Waiter() {
      Console.WriteLine("Waiting...");
      wh.WaitOne();
      Console.WriteLine("Notified");
   }
}
```

⇒ inaktiv

gibt WaitHandle frei

Starten Sie das Programm `BasicWaitHandle`.

Fragen:

- In welchem Zustand (an welchem Ort des Thread-Lebenszyklus) befindet sich ein Thread, der an der Methode `WaitOne` wartet ? ⇒ Wait-Pool
- Was passiert, wenn Sie einen zweiten Thread starten ?
- Wie erklären Sie sich das Verhalten des zweiten Threads ?
- Gibt es eine Möglichkeit, dass der zweite Threads sich gleich wie der Erste verhält ?

- Ein zweiter Thread müsste bei AutoResetEvent mit einem zusätzlichen Set() losgelassen werden.
- Mit ManualResetEvent würde ein einziges Set() reichen, um alle Threads gleichzeitig loszulassen.

Übung 5: Pferderennen

Latch

Ein praktischer Synchronisationsmechanismus ist das Latch. Latches sperren so lange, bis sie einmal ausgelöst werden und danach sind sie frei passierbar. Sie sollen nun ein Latch in C# schreiben und testen. Dazu ist ein Interface `ISynch` gegeben, das implementiert werden soll.

```
interface ISynch
{
   void Acquire();
   void Release();
}
```

Für den Test veranstalten wir ein kleines Pferderennen. Dazu gibt es Pferde...

```
class RaceHorse
{
   private ISynch startSignal;
   private int nr;

   public RaceHorse(int nr, ISynch startSignal) {
      this.nr = nr;
      this.startSignal = startSignal;
   }

   public void Run() {
      startSignal.Acquire();
      Thread.Sleep(new Random().Next(3000));
      System.Console.WriteLine("Rennpferd "+nr+" ist im Ziel.");
   }
}
```

=> Gatter, welches Pferd blockiert am Start

...und eine Rennbahn.

```
class Turf
{
   static void Main() {
      ISynch starterBox = new Latch();
      for (int i = 1; i < 6; i++)
         new Thread(new RaceHorse(i, starterBox).Run).Start();
      starterBox.Release();
   }
}
```

=> Rennen starten

Damit alle Pferde fair gestartet werden, kommen diese in eine Starterbox. Sobald diese geöffnet wird, sollen die Pferde loslaufen. Implementieren Sie mit diesen Vorgaben die Klasse `Latch`.

Frage: Ist das Rennen wirklich gerecht ?

Arbeitsblatt 17: Wait und Pulse

Idee: Die Threads warten an einem Monitorobjekt, bis dieses einen Impuls erhält, um einen oder mehrere Threads frei zu schalten.

Wenn der Zugang zu einem kritischen Abschnitt von bestimmten Bedingungen oder Zuständen abhängt, so reicht das Konzept der einfachen Synchronisation mit `lock` allein nicht aus. Man muss diesen Zugang entsprechend auscodieren. Um einen Thread auf eine Bedingung oder Zustand warten zu lassen stellt C# die Monitor-Methoden `Wait` und `Pulse` zur Verfügung.

```
public static bool Wait(Object obj)
public static bool Wait(Object obj, int millisecondsTimeout)
public static bool Wait(Object obj, TimeSpan timeout)

public static void Pulse(Object obj)
public static void PulseAll(Object obj)
```

Die Methoden `Wait` und `Pulse` werden mit der Klasse `Monitor` aufgerufen.

```
Monitor.Wait(obj);
Monitor.Pulse(obj);
```

Wichtig: Sowohl `Wait` als auch `Pulse` dürfen nur aufgerufen werden, wenn das Objekt gesperrt ist, also nur innerhalb eines `lock`-Blocks für dieses Objekt.

Bei Aufruf von `Wait` wird der aufrufende Thread in einen Wartezustand versetzt, und gleichzeitig wird der Lock auf diesen Abschnitt freigegeben. Es führen genau drei Wege aus dem Warte-Zustand wieder heraus:

- Ein anderer Thread signalisiert den Zustandwechsel mittels `Pulse` bzw. `PulseAll`.
- Die angegebene Zeit (Timeout) ist abgelaufen.
- Ein anderer Thread ruft die Methode `Abort` des wartenden Threads auf.

Dann wartet der Thread eventuell noch einmal, bis er den Lock auf den Abschnitt wieder erhält, und wird dann wieder "ready", d.h. bereit zur erneuten Ausführung. `Pulse` weckt genau einen Thread im wait-Zustand auf. Falls mehrere Threads warten, ist nicht vorhersehbar oder bestimmbar, welcher Thread aufgeweckt wird. `PulseAll` weckt alle an diesem Objekt wartenden Threads auf. Der Rückgabewert der `wait` Methode ist in diesem Fall `true`.

Die Variante von `Wait` mit Timeout ist das 'timed wait', bei dem eine Zeit angegeben wird, wie lange maximal auf das Eintreten einer Bedingung gewartet werden soll. Falls die Zeit verstrichen ist, terminiert die Methode, als ob ein `Pulse` stattgefunden hätte. Das heisst, auch der Lock wird dann wieder von diesem Thread gehalten. Der Rückgabewert der `wait` Methode ist in diesem Fall `false`.

Achtung: Es sollte die Bedingung, die zum Wartezustand geführt hat, erneut getestet werden, um festzustellen, ob nur die Zeit abgelaufen ist oder die Bedingung tatsächlich erfüllt ist.

Ähnlich wie beim Blocked-Zustand, kann ein Thread im Warte-Zustand von einem anderen Thread unterbrochen werden, in dem er die `Abort`-Methode des wartenden Threads aufruft.

=> Ist "unconditionally fair"
=> Pulse startet Threads in undefinierter Reihenfolge

Arbeitsblatt 17: Wait und Pulse (Fortsetzung)

Dieses C# Konstrukt entspricht dem `wait` und `notify` in Java.

C#	Java

`Object x = new Object();`
`lock (x) {`
`  try {`
`  Monitor.Wait(x);`
`  } catch (ThreadInterruptedException) {}`
`  // weitere Aktionen...`
`}`
	`Object x = new Object();`
	`synchronized(x) {`
	`  try {`
	`  x.wait();`
	`  } catch (InterruptedException e) {}`
	`  // weitere Aktionen...`
	`}`

`// aufwecken...`
`lock (x) {`
`  Monitor.Pulse(x);`
`}`
	`// aufwecken...`
	`synchronized(x) {`
	`  x.notify();`
	`}`

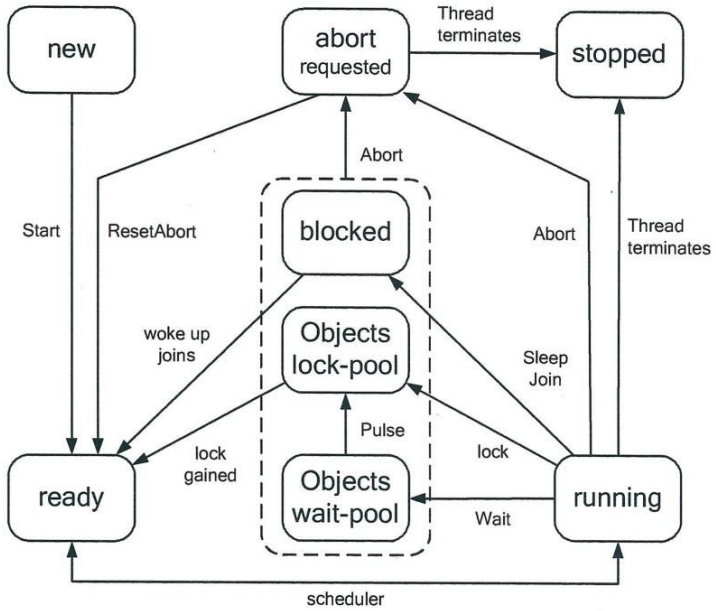
Im folgenden Beispiel wird vom Main-Thread ein Thread erzeugt und gestartet. Dieser ruft in der `Waiter`-Methode die `Wait`-Methode auf. Damit wartet der Thread, bis der Main-Thread die Pulse `Pulse` Methode auftruft und damit der Thread aus dem Wartezustand befreit wird. Falls jedoch das Timeout Argument kleiner ist als die Sleeptime des Main-Thread, wird der Thread ebenfalls aus dem Wartezustand befreit. Den Lock auf das `synch`-Objekt erhält er aber erst, wenn dieses durch den Main-Thread freigeben wurde.

```
class WaitAndPulse
{
   static Object synch = new Object();

   static void Main() {
      new Thread(Waiter).Start();
      Thread.Sleep(1000);
      lock (synch) {
         Monitor.Pulse(synch);
      }
   }

   static void Waiter() {
      lock (synch) {
         Console.WriteLine("Waiting...");
         if (Monitor.Wait(synch,1000)) // lock wird hier abgegeben!
            Console.WriteLine("Notified");
         else
            Console.WriteLine("Timeout");
      }
   }
}
```

Achtung: `Pulse` wird nicht gespeichert. Dies bedeutet, wenn `Pulse` vor dem `Wait` aufgerufen wird, wird der Thread ewig warten.

Arbeitsblatt 18: Thread Lebenszyklus mit Warteliste

Zusätzlich zu dem bereits erwähnten lock-Pool, der einem Objekt zugeordnet ist, besitzt ein Objekt auch noch eine Warteliste, einen wait-Pool. Dabei handelt es sich um eine (möglicherweise leere) Menge von Threads, die vom Scheduler unterbrochen wurden und auf ein Ereignis warten, um fortgesetzt werden zu können.



Wichtig: Der Objects lock-pool und der Objects wait-pool müssen zum gleichen Objekt gehören. Ist dies nicht der Fall führt dies während der Laufzeit zu einem Fehler:

`System.Threading.SynchronizationLockException`

Dieser Fehler besagt, dass der `Wait` aufrufende Thread nicht im Besitzt des Locks von dem Objekt ist, dessen wait-pool er benutzen möchte. Es muss sichergestellt sein, dass die beiden Object-pools zum selben Objekt gehören:

```
Object synch = new Object();
lock (synch) {
   Monitor.Wait(synch);
}
```

=> wait- und lock-Pool müssen von demselben Objekt sein, sonst funktioniert das Pulse nicht.

=> Nie ein Sleep in einen lock-Abschnitt implementieren

Übung 6: Wait-Pool

Man möchte den Wait-Pool eines Objektes testen mit folgende Klassen.

```
class MyThread
{
   private Object synch;

   public MyThread(Object synch) {
      this.synch = synch;
   }

   public void Run() {
      Console.WriteLine("warten...");
      lock (synch) {
         Monitor.Wait(this);
      }
      Console.WriteLine("...aufgewacht");
   }
}

class TestWaitPool
{
   static void Main() {
      Object synch = new Object();
      MyThread myThread = new MyThread(synch);
      new Thread(myThread.Run).Start();
      Thread.Sleep(1000);
      Monitor.Pulse(synch);
   }
}
```

Starten Sie das Programm `TestWaitPool`.

Fragen:

- Was passiert bei der Ausführung von `TestWaitPool` ?
- Wie erklären Sie sich das Verhalten der Klassen ?
- Welche Korrekturen sind nötig ?

Übung 7: Nested Monitor

Gegeben ist die Klasse `NotifyingQueue`. Eine Warteschlange, die zwei Methoden besitzt, um ein Objekt abzulegen, bzw. herzunehmen. Die Warteschlange ist als Ringbuffer implementiert. Ein Thread, der die Methode `dequeue` aufruft, muss warten falls die `NotifyingQueue` leer ist.

```
class NotifyingQueue
{
   private const int SIZE = 10;
   private Object[]  queue = new Object[SIZE];
   private int       head  = 0;
   private int       tail  = 0;

   public void enqueue( Object item ) {
      lock (this) {
         head++;
         head %= SIZE;
         queue[head] = item;
         Monitor.Pulse(this);
      }
   }

   public Object dequeue() {
      lock (this) {
         if (head == tail) {
            Monitor.Wait(this);
         }
         tail++;
         tail %= SIZE;
         return queue[tail];
      }
   }
}
```

WHILE immer dann, wenn aufgrund einer Bedingung ein Wait gemacht werden muss.

Fragen:

- Was ist Falsch an der Klasse `NotifyingQueue` ?

Übung 7: Nested Monitor (Fortsetzung)

Zusätzlich ist die Klasse `BlackHole` gegeben. In einem `BlackHole` kann etwas verschwinden (`put`) oder etwas herauskommen (`get`).

```
class BlackHole
{
   private NotifyingQueue queue = new NotifyingQueue();

   public void put(Object thing) {
      lock (this)    => wenn schon lock(queue)
         queue.enqueue(thing);
   }

   public Object get() {
      lock (this)    => lock(queue)
         return queue.dequeue();
   }
}
```

weglassen

Zugegeben, es ist nicht erwiesen, dass irgend etwas, das in einem schwarzen Loch verschwunden ist auch tatsächlich wieder herauskommt. Nun, Sie sollen das beweisen mit folgendem Programm:

```
class TestNestedMonitor
{
   public static BlackHole blackhole = new BlackHole();

   public static void Main() {
      Thread t1 = new Thread(delegate() {
         Console.WriteLine(blackhole.get().ToString());
      });

      Thread t2 = new Thread(delegate() {
         blackhole.put("Sonne, Licht, irgendetwas...");
      });

      Console.WriteLine("Wir starten die Untersuchung...");
      t1.Start();
      t2.Start();
      t1.Join();
      Console.WriteLine("Was ist passiert...");
   }
}
```

Fragen:

- Was stellen Sie beim Ausführen von `TestBlackHole` fest ?
- Wie erklären Sie sich das Programmverhalten ?
- Wie müsste eine korrekte Lösung aussehen ?

=> Thread t1 generiert über BlackHole ein lock und hält diesen, während Monitor.Wait wartet.
=> t2 vor t1 starten, dann fällt er nicht in Monitor.Wait
=> Nested Monitors VERMEIDEN!

Arbeitsblatt 19: Implementierung von await in C#

Die Herausforderung bei der Implementierung des `await`-Statements ist, dass die Bedingung ein beliebiger boolescher Ausdruck sein darf, man aber in C# nur eine "boolesche Variable" zur Verfügung hat (in Gestalt des Synchronisations- und Warte-Objekts). Die Grundidee bei der Lösung ist es, zunächst für jede semantisch verschiedene Bedingung eine eigene Bedingungsvariable (d.h. ein eigenes Objekt) einzuführen.

```
await B then S1; ... Sn; end;
```

Zur Implementierung des `await` Sprachkonstruktes muss die Arbeit auf einen Tester und einen Waiter aufgeteilt werden.

```
/* init */
   Object synch = new Object();

/* waiter */
   lock (synch) {
      Monitor.Wait(synch);
      S1; ...; Sn;
   }

/* tester */
   lock (synch) {
      // wenn Bedingung B wahr wird
      if (B) Monitor.Pulse(synch);
   }
```

An allen Programmstellen, an denen sich B ändern könnte muss der Tester-Teil eingefügt werden.

Anmerkung: Falls sich die Bedingungen semantisch überlappen, wird die Implementierung noch wesentlich schwieriger.

Arbeitsblatt 20: Eine AWait-Klasse

Es soll das Beispiel aus dem Abschnitt "Explizite Synchronisation" in C# umgesetzt werden.

```
x = 0; y = 0;
con
     await x > 0 and y > 0 then z = x + y; end;
     { x = 1; y = 2; }
end;
```

Da `await` in C# nicht richtig implementiert werden kann, soll die folgende Klasse Abhilfe schaffen:

```
delegate void Runnable();

class AWait
{
   Boolean cond = false;

   public void Await(Boolean b, Runnable s) {
      lock (this) {
         cond = b;
         while (!cond) {
            Monitor.Wait(this);
         }
         if (s != null)
            s();
      }
   }

   public void Test(Boolean b) {
      lock (this) {
         cond = b;
         if (cond) {
            Monitor.Pulse(this);
         }
      }
   }
}
```

Fragen:

- Warum wird die Bedingung (`cond`) in der `Await`-Methode mit einer `while`-Anweisung geprüft ? Bedingung könnte falsch sein
- Welche Randbedingungen gelten, wenn Sie die obige `AWait`-Klasse verwenden ? Jedes AWait ein Test, zuerst AWait, dann Pulse !
- Was passiert, wenn der Test nie ausgeführt wird ? Alle Wait bleiben hangen
- Was passiert, wenn der Test nie positiv (`cond` wahr) wird ?

Arbeitsblatt 20: Eine AWait-Klasse (Fortsetzung)

Weiter werden zwei Threads implementiert. Ein Thread übernimmt die Addition. Er wartet bis die Bedingung B zutrifft. Der andere Thread übernimmt die Zuweisung und testet die Bedingung B. Die Bedingung B trifft zu, wenn `x` und `y` grösser als Null sind. Am Schluss wird die Summe `z` ausgeben.

```
public class AwaitTest
{
   static int x = 0;
   static int y = 0;
   static int z = 0;
   static AWait b = new AWait();

   public static void sum() {
      z = x + y;
   }

   public static void Main()
   {
      for (int i = 1; i <= 10; i++) {
         x = y = z = 0;

         Thread s1 = new Thread(delegate() {
            // Warten, bis Bedingung B zutrifft...
            Boolean B = x > 0 && y > 0;
            b.Await(B, sum);
         });

         Thread s2 = new Thread(delegate() {
            Warte.Zeit(0.5);
            x = 1;
            Warte.Zeit(0.5);
            y = 2;
            // Prüfen, ob Bedingung B zutrifft...
            Boolean B = x > 0 && y > 0;
            b.Test(B);
         });

         s1.Start();
         s2.Start();
         s1.Join();
         s2.Join();
         Console.WriteLine(i + ". z = " + x + "+" + y + " = " + z);
      }
   }
}
```

Fragen:

- Testen Sie das Beispiel mit und ohne `AWait` Objekt. Wie reagiert der `AwaitTest` ?

Übung 8: Lost Signals

Objekte der Klasse `Worker` können bei einem Objekt der Klasse `Operator` eine bestimmte Anzahl (`MAXCOUNT`) Aufgaben abholen. Bevor die Aufgaben zur Abholung bereit stehen muss der `Operator` diese initialisieren. Ist die Initialisierung abgeschlossen, werden `MAXCOUNT` am `ISych` Signal wartenden `Worker` (hier vier) befreit. Hat ein `Worker` seine Aufgabe abgeschlossen, so informiert er mit Aufruf der Methode `Done` den `Operator`. Sobald alle ausstehenden Aufgaben gelöst sind, initialisiert der `Operator` eine neue Anzahl Aufgaben.

```
class Operator
{
   public const int MAXCOUNT = 4;
   private int counter = 0;
   private ISynch signal;

   public Operator(ISynch signal) {
      this.signal = signal;
   }

   private void Init() {
      Console.WriteLine("Jetzt werden Operationen vorbereitet...");
      counter = MAXCOUNT;
   }

   public int Operation() {
      return 1000;
   }

   public void Done() {
      counter++;
      Console.WriteLine(counter.ToString() + ". Operation ist beendet.");
   }

   public void Do() {
      while (true) {
         Init();
         for (; counter > 0; counter--) {
            signal.Release();
         }
         while (counter < MAXCOUNT) {
            Thread.Sleep(1);
         }
      }
   }
}
```


Übung 8: Lost Signals (Fortsetzung)

Ein `Worker` wartet am `ISych` Signal bis ein Release vom `Operator` ausgeführt wird. Dann wird eine Aufgabe geholt (hier Aufruf der Methode `Operation`) und ausgeführt. Ist die Aufgabe beendet, wird der `Operator` mit der Methode `Done` benachrichtigt.

```
class Worker
{
   private ISynch signal;
   private Operator op;

   public Worker(ISynch signal, Operator op) {
      this.signal = signal;
      this.op = op;
   }

   public void Do() {
      while (true) {
         signal.Acquire();
         Thread.Sleep(op.Operation());
         op.Done();
      }
   }
}

class AWaitTest
{
   static void Main() {
      ISynch signal = new AWait();
      Operator op = new Operator(signal);
      new Thread(op.Do).Start();
      for (int i = 0; i < 10; i++) {
         Worker worker = new Worker(signal, op);
         new Thread(worker.Do).Start();
      }
   }
}
```

Schreiben Sie eine Klasse `AWait`, die nun das Interface `ISynch` implementiert.

Anmerkung: Das oben beschriebene Verfahren ähnelt der `Scatter/gather`-Verarbeitung. Ein Verfahren zur Berechnung eines komplexen Problems in mehreren Schritten. In der `scatter`-Phase werden dabei Teilprobleme an einzelne Prozesse (workers) zugeteilt, die daraufhin unabhängig voneinander Zwischenlösungen berechnen. In der `gather`-Phase werden diese Zwischenlösungen eingesammelt und zu einer Gesamt-Zwischenlösung zusammengefasst, die dann in einem weiteren `scatter/gather`-Zyklus erneut in Teilprobleme zerlegt wird. Dies wird solange wiederholt, bis das Endergebnis vorliegt.

Arbeitsblatt 21: Semaphore

Ein allgemeineres Konzept für die Synchronisation sind die so genannten Semaphoren (Signalmasten, Leuchttürme). Die Signalmasten zeigen ähnlich wie bei der Eisenbahn an, ob in eine Strecke (hier jetzt ein kritischer Bereich) eingefahren werden darf, oder ob ein Zug (hier ein Thread oder Prozess) halten muss. Im Gegensatz zum Mutex können sie nicht nur die Zustände 'verschlossen' und 'frei' einnehmen, sondern beliebig viele (abzählbar viele). Semaphoren wurden zuerst entwickelt, um das Problem der "lost signals" zu lösen.

Das Semaphorkonzept wurde 1968 von E.W. Dijkstra eingeführt. Es gibt zwei Operationen 'V' (Abkürzung für holländisch Verhogen = Erhöhen) und 'P' (für Proberen = Probieren) auf das Semaphor `sema` angewendet:

`sema.P()`: entspricht dem Eintritt (Passieren) in einen synchronisierten Bereich, wobei mitgezählt wird, der wievielte Eintritt es ist.

await s>0 then s=s-1; end;

`sema.V()`: entspricht dem Verlassen (Freigeben) eines synchronisierten Bereiches, wobei ebenfalls mitgezählt wird, wie oft der Bereich verlassen wird.

atomic s=s+1; end;

Im folgenden Beispiel wird ein Semaphore erzeugt, dessen Zähler beim Start der Applikation auf 3 initialisiert wird. Das heisst, maximal drei Thread können den kritischen Abschnitt betreten.

initialCount=3;

```
class SemaphoreTest {
   static Semaphore s = new Semaphore(3, 3);  // ~~Available=3~~; Capacity=3

   static void Main() {
      for (int i = 0; i < 10; i++) {
         new Thread(Go).Start(i);
      }
   }
   static void Go(object number) {
      while (true) {
         Console.WriteLine(number.ToString() + ". Thread waits.");
         s.WaitOne();
         Console.WriteLine(
            number.ToString()+ ". Thread is in critical section");
         Thread.Sleep(1000);   // Only 3 threads can get here at once
         s.Release();
         Console.WriteLine(number.ToString() + ". Thread leaves.");
      }
   }
}
```

Starten Sie das Programm `SemaphoreTest`.

Fragen:

- Was haben die Argumente des Konstruktors der Klasse `Semaphore` für eine Bedeutung ?
- Was passiert, wenn Sie die Initialisierung des Semaphors ändern ?
- Wie sieht die Lösung für die Übung "Lost Signals" mit der Klasse `Semaphore` aus ?

Arbeitsblatt 22: Mutex

Mutex (Abk. für engl. Mutual Exclusion, "wechselseitiger Ausschluss") bezeichnet Verfahren, mit denen verhindert wird, dass nebenläufige Prozesse bzw. Threads gleichzeitig auf Daten zugreifen und so unter Umständen inkonsistente Zustände herbeiführen.

`bin.P()`: await b==true then b=false; end;

`bin.V()`: ~~atomic b=true; end;~~ → beliebig oft aufrufbar → lost Signals
await b==false then b=true; end;

Wichtig: C# bietet die `Mutex` Klasse an. Sie ist gegenüber dem lock-Konstrukt eine redundante Implementation. Aber das Mutex bietet die Möglichkeit das Sperren von Codebereichen Thread und Prozess übergreifend einzusetzen zu können.

Im folgenden Beispiel wird ein eindeutiges Mutex erzeugt, das beim Start der Applikation während 5 Sekunden geprüft wird, ob es frei geschaltet ist. Ist dies nicht der Fall wird die Applikation wieder beendet.

```
class MutexTest {
   // Use a name unique to the application (eg include your company URL)
   static Mutex mutex = new Mutex(false, "OneAtATimeDemo");

   static void Main() {
      // Wait 5 seconds if contended - in case another instance
      // of the program is in the process of shutting down.
      if (!mutex.WaitOne(TimeSpan.FromSeconds(5), false)) {
         Console.WriteLine(
            "Another instance of the app is running. Bye!");
         return;
      }
      try {
         Console.WriteLine("Running - press Enter to exit");
         Console.ReadLine();
      }
      finally {
         mutex.ReleaseMutex();
      }
   }
}
```

Starten Sie das Programm `MutexTest`.

Fragen:

- Was passiert in diesem Programm ?
- Wo kommen in diesem Programm Threads und wo Prozesse vor ?
- Erklären Sie anhand dieses Programms was "Prozess übergreifend" bedeutet.
- In welchem Zustand befindet sich ein Thread, der an der Methode `WaitOne` wartet ?
- Wie sieht der Code aus, um das lock-Konstrukt mit der Klasse `Mutex` zu ersetzen ?

⇒ Nur 1 Mutex auf System.
⇒ Wenn Anwendung 1 stirbt werden alle weiteren Anwendungen mit mutex.WaitOne mit Exception informiert
⇒ Anwendungsübergreifend

Übung 9: Prozessübergreifende Synchronisation

Gegeben sind die folgenden drei Programme. Ein Zahlengenerator (`RandomCounter`), ein Addierer (`AddNumbers`), der die Zahlenreihe zusammenzählt und ein MinMax Programm (`MinMaxNumber`), das den minimalen und maximalen Wert der Zahlenreihe ermittelt. Die Zahlenreihe steht im File "daten.txt".

```
class RandomCounter
{
    static void Main() {
        Random rnd = new Random();
        using (StreamWriter sw = new StreamWriter("daten.txt")) {
            int n = rnd.Next(5000);
            for (int i = 0; i < n; i++) {
                sw.WriteLine(rnd.Next(1000));
            }
        }
        Console.WriteLine("done.");
    }
}

class AddNumbers
{
    static void Main() {
        int sum = 0;
        using (StreamReader sr = new StreamReader("daten.txt")) {
            string line;
            while ((line = sr.ReadLine()) != null) {
                sum += Int32.Parse(line);
            }
        }
        Console.WriteLine("Sum := "+sum);
    }
}

class MinMaxNumber
{
    static void Main() {
        int min = 0;
        int max = 0;
        using (StreamReader sr = new StreamReader("daten.txt")) {
            string line;
            while ((line = sr.ReadLine()) != null) {
                int num = Int32.Parse(line);
                min = min > num ? num : min;
                max = max < num ? num : max;
            }
        }
        Console.WriteLine("Min := " + min);
        Console.WriteLine("Max := " + max);
    }
}
```

Alle drei Programme sollen so schnell wie möglich und parallel ausgeführt werden. Natürlich müssen die entsprechenden Daten für die Programme vorhanden sein. Aus diesem Grund wird eine prozessübergreifende Synchronisation nötig sein. Zwei Konstrukte für prozessübergreifende Synchronisation kennen Sie, das Semaphore und das Mutex.

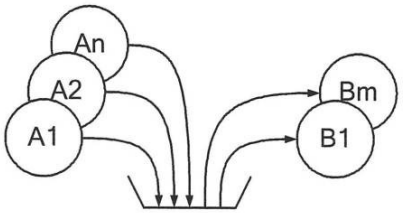
3. Anwendungen

3.1. Bounded Buffer

Arbeitsblatt 23: Anwendung von Semaphoren im Bounded-Buffer

Eine sehr häufige Aufgabenstellung in der nebenläufigen Programmierung ist, dass ***n*** Produzenten Daten erzeugen, die an ***m*** Konsumenten weitergegeben werden sollen.

Zur Weitergabe der Daten dient ein Puffer, der in der Lage sein soll, ***k*** Daten zu speichern. Dabei soll m >= 1, n >= 1 und k >= 1 sein. Falls der Puffer voll ist, sollen die Produzenten warten, bis wieder Platz ist, und falls der Puffer leer ist, sollen die Konsumenten warten, bis wieder Daten vorhanden sind.



Eine Synchronisation muss sicherstellen, dass der Produzent wartet, falls der Puffer voll ist, und der Konsument wartet, falls der Puffer leer ist. Damit ergibt sich folgender Ansatz:

```
global
   x = 0; // Anz. belegte Plätze
   k = bufferSize;

Producer(i):
   while (true) {
      /* Daten erzeugen */
      await x < k then
         /* Daten in Puffer schreiben */
         x = x + 1;
      end;
   }

Consumer(i):
   while (true) {
      await x >= 1 then
         /* Daten aus Puffer lesen */
         x = x - 1;
      end;
      /* Daten verbrauchen */
   }
```

Frage: Was stimmt an diesem Ansatz des Bounded-Buffers nicht ?

→ await ist implizit atomar, also nur 1 Prozess kann eintreten
→ Producer teilen sich x

Arbeitsblatt 24: Effizientere Synchronisation des Bounded-Buffers

Da `wait until` atomar ausgeführt werden muss, können die Produzenten und Konsumenten allerdings nur abwechselnd auf die Variable *x* und damit auf den Puffer zugreifen. Eine Abhilfe schafft die Verwendung von zwei Zählern, z.B. `empty` und `full`.

```
empty = bufferSize;      // freie Plätze
full = 0;                // belegte Plätze
con
   Producer(1); ...; Producer(n);
   Consumer(1); ...; Consumer(m);
end;
```

Dies bedeutet, dass die Produzenten und Konsumenten wie folgt definiert werden.

```
Producer(i):
   while (true) {
      /* Daten erzeugen */
      await empty>0 then empty--; end;
      // etwas produzieren
      atomic full++; end;   //wartet sofern full in Consumer benutzt wird!

   }

Consumer(i):
   while (true) {
      await full>0: then full--; end;
      //etwas konsumieren
      atomic empty++; end;

      /* Daten verbrauchen */
   }
```

Zusätzlich muss noch sichergestellt werden, dass nicht mehrere Produzenten bzw. mehrere Konsumenten gleichzeitig auf den Puffer zugreifen. Das heisst, `/* Daten in Puffer schreiben */` muss auf Produzentenseite atomar sein, und `/* Daten aus Puffer lesen */` muss auf Konsumentenseite atomar sein.

Gleiches Prinzip wie Semaphore

Arbeitsblatt 25: Implementation des Bounded-Buffers

Die zentrale Funktionalität eines Bounded-Buffer ist, das Schreiben und Lesen eines Puffers. Um diese Funktionalität abstrakt einzusetzen, wird das Interface `Queue` definiert.

```
interface IQueue {
   void Enqueue(Object x);
   Object Dequeue();
}
```

Der Bounded-Buffer muss sicherstellen, dass falls der Puffer voll ist, ein Produzent nicht mehr in den Puffer schreiben kann, und falls der Puffer leer ist, ein Konsument nicht aus dem Puffer lesen kann.

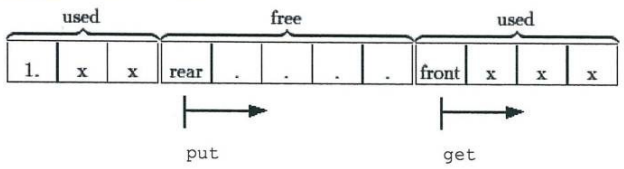
Status	Bedingung	Enqueue	Dequeue
voll	Anzahl == Kapazität	nein	ja
teilweise	0 < Anzahl < Kapazität	ja	ja
leer	Anzahl == 0	ja	nein

Der Bounded-Buffer benutzt die Klasse `RingBufferArray` (siehe Implementation Ring-Buffer-Array) und initialisiert das Attribut `buf` mit der gewünschten Grösse `size`. Für die Synchronisation des Buffers werden die Semaphore `empty` und `full` verwendet. Sie sind Zähler, die den Status "voll", bzw. "leer" überprüfen und `put/take` Zugriffe von Threads zu- oder die Threads warten lassen.

```
class BoundedBufferWithSemaphor : IQueue
{
   protected Semaphore empty;  --> 2 wait pools
   protected Semaphore full;   -->
   protected RingBufferArray buf;

   public BoundedBufferWithSemaphor(int size) {
      buf = new RingBufferArray(size);
      empty = new Semaphore(size,size);
      full = new Semaphore(0,size);
   }

   public void Enqueue(Object x) {
      empty.WaitOne();
      buf.Put(x);
      full.Release();
   }

   public Object Dequeue(){
      full.WaitOne();
      Object x = buf.Get();
      empty.Release();
      return x;
   }
}
```

Arbeitsblatt 26: Implementation des Ring-Buffer-Arrays

Am einfachsten lässt sich der Bounded-Buffer als Ringpuffer implementieren, da in einem solchen Puffer die Daten kontinuierlich ein- und ausgelesen werden können. Dazu wird die Klasse `RingBufferArray` implementiert. Die Attribute `front` und `rear` zeigen jeweils auf den nächsten vollen Platz zum Lesen des Puffers bzw. auf den nächsten freien Platz zum Schreiben.




```
class RingBufferArray
{
   protected Object[] array;
   protected int front = 0;
   protected int rear = 0;
   private Object putLock = new Object();
   private Object getLock = new Object();

   public RingBufferArray(int n) {
      array = new Object[n];
   }

   public void Put(Object x) {
      lock (putLock) {
         array[rear] = x;
         rear = (rear + 1) % array.GetLength(0);
      }
   }

   public Object Get() {
      lock (getLock) {
         Object x = array[front];
         array[front] = null;
         front = (front + 1) % array.GetLength(0);
         return x;
      }
   }
}
```

→ Array sorgt dafür, dass sich Rear + Front nicht überlappen

Übung 10: Parkhaus

Die Klasse `Parkhaus` soll die Zufahrten und Ausfahrten eines Parkhauses kontrollieren. Das heisst, wenn keine Parkplätze mehr vorhanden sind, werden die Autos an der Zufahrt gehindert. Die Autos könnten in der folgenden Art simuliert sein.

```
class Auto {
   private Parkhaus parkhaus;
   //...
   public void Drive() {
      while(true) {
         // ...fährt ein
         parkhaus.Passieren();
         // ...ist parkiert
         // ...fährt aus
         parkhaus.Verlassen();
      }
   }
}
```

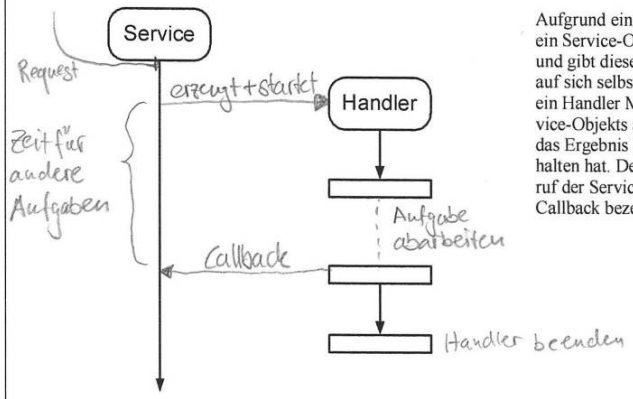
Zeigen Sie zwei völlig unterschiedliche Möglichkeiten, wie die Klasse `Parkhaus` die Zu- und Ausfahrten korrekt überwacht. Die vorgegebenen Methoden-Definitionen dürfen nicht verändert werden.

Variante 1 Mutex	Variante 2 Semaphore
`class Parkhaus {`	`class Parkhaus {`
`public Parkhaus(int plaetze){`	`public Parkhaus(int plaetze){`
`}`	`}`
`public void Passieren(){`	`public void Passieren(){`
`}`	`}`
`public void Verlassen(){`	`public void Verlassen(){`
`}`	`}`
`}`	`}`

3.2. Kooperation zwischen Threads

Arbeitsblatt 27: Asynchrone Aufrufe mit Callback

Ein Entwurfsmuster, das man häufig in verteilten Anwendungen vorfindet, ist der asynchrone Aufruf. Bei einem normalen (synchronen) Aufruf muss der Aufrufende warten, bis der Aufgerufene die ihm zugeteilte Aufgabe erledigt hat, und ist so blockiert. In C# (auch Java, o.a. Programmiersprachen) sind Methodenaufrufe stets synchron. Asynchrone Methodenaufrufe müssen durch Nebenläufigkeit hergestellt werden.



Aufgrund einer Anfrage erzeugt ein Service-Objekt einen Handler und gibt diesem eine Referenz auf sich selbst mit. Damit kann ein Handler Methoden des Service-Objekts aufrufen, sobald er das Ergebnis berechnet oder erhalten hat. Der asynchrone Aufruf der Servicemethode wird als Callback bezeichnet.

Das Entwurfsmuster für asynchronen Aufruf mit Callback gibt ein Interface mit der Methode `handle` vor.

```
interface ICallbackHandler {
   void Handle(Object res);
}

class Service : ICallbackHandler {
   public void Request() {
      // Die Aufgabe und eine Referenz auf sich selbst an den Handler
      // übergeben und starten...
      Handler handler = new Handler(this);
      new Thread(handler.Do).Start();
      // ...weitere Aufgaben bearbeiten
   }

   public void Handle(Object res) {
      // ...Resultat steht zur Verfügung, oder als Parameter erhalten.
   }
}
```

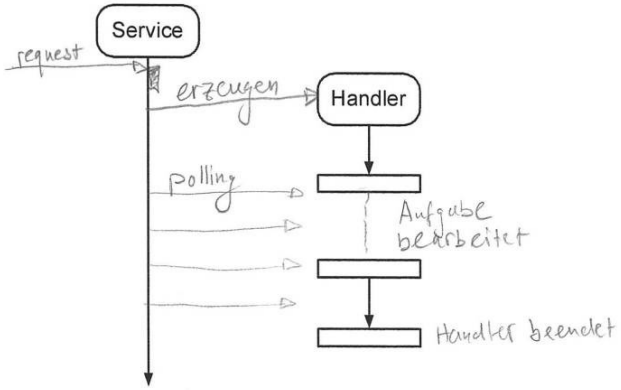
→ keine Fortschrittsanzeige
→ keine Prüfung auf Thread-Abstürze

Arbeitsblatt 27: Asynchrone Aufrufe mit Callback (Fortsetzung)

```
class Handler
{
   ICallbackHandler service;

   public Handler (ICallbackHandler service) {
      this.service = service;
   }

   public void Do() {
      // Hier ist die Aufgabe zu lösen...
      // ...Aufgabe beendet,
      // nun den Service benachrichtigen...
      service.Handle(result);
   }
}
```

Arbeitsblatt 28: Asynchrone Aufrufe mit Polling

Ein weiteres Entwurfsmuster, das man häufig in verteilten Anwendungen vorfindet, ist der asynchrone Aufruf mit Polling. Dabei kann das Service-Objekt den Handler laufend fragen, ob das Ergebnis schon eingetroffen ist. Wenn dies nicht der Fall ist, kann der Service sich um andere Aufgaben kümmern. Ist das Ergebnis aber eingetroffen, kann der Service es auslesen und verwerten. Dieses Verfahren nennt man Polling.



→ CPU intensiv!

Arbeitsblatt 28: Asynchrone Aufrufe mit Polling (Fortsetzung)

Entwurfsmuster für asynchronen Aufruf mit Polling. Dazu wird der Thread jeweils mit der Methode `isAlive` überprüft, ob er noch läuft.

```
class Service
{
   protected Handler handler = new Handler();

   public void Request() {
      // Die Aufgabe dem Handler übergeben und starten...
      Thread worker = new Thread(handler.Do);
      worker.Start();
      // ...auf das Resultat warten...
      while(worker.IsAlive) {
         // Hier können andere Aufgaben bearbeitet werden...
      }
      // ...Resultat steht zur Verfügung.
      handler.GetResult();
   }
}

class Handler
{
   // Resultat der Aufgabe
   private Object result;

   public void Do() {
      // Die Aufgabe lösen...
      // ...Aufgabe beendet.
   }

   public Object GetResult() {
      return result;
   }
}
```

Anmerkung: Man sollte sich bewusst sein, dass diese Arten von Polling unter Umständen unnötige Kommunikation (und damit Rechenkapazität) zu den Handler-Threads führen. Ein Grund warum man trotzdem diese Arten von Polling einsetzt, ist der Arbeitsfortschritt oder Zwischenresultate der Handler-Threads abzufragen.

Übung 11: Fibonacci mit Callback

Die Fibonacci Zahlenreihe definiert ein festes wiederkehrendes Muster, das in allen Bereichen des Lebens zu finden ist. Die Klasse `FibonacciHandler`, errechnet eine Fibonacci Zahl. Die Berechnung nach dem Bildungsgesetz `Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)` für `n > 2`, kann bei grossen Zahlen durchaus längere Zeit dauern.

```
class FibonacciHandler {

   public static BigInteger getFibonacci(int value) {
      BigInteger n = new BigInteger(value);
      BigInteger zero = new BigInteger(0);
      BigInteger one = new BigInteger(1);
      if (n == zero) return zero;
      if (n == one) return one;
      BigInteger i = one;
      BigInteger a = zero;
      BigInteger b = one;
      while (i < n) {
         BigInteger temp = b;
         b = a+b;
         a = temp;
         i++;
      }
      return b;
   }
}

class FiboService {

   public void Request() {
      Console.Write("Eingabe ? ");
      String line = Console.ReadLine();
      int n = Convert.ToInt32(line);
      BigInteger fibo = FibonacciHandler.getFibonacci(n);
      Console.WriteLine("Fibo {0} = {1}", n, fibo);
      // Hier könnten andere Aufgaben bearbeitet werden...
   }

   public static void Main() {
      FiboService service = new FiboService();
      while (true) {
         service.Request();
      }
   }
}
```

Die Klasse `FiboService` benutzt die Methode `getFibonacci` der Klasse `Fibonacci-Handler` synchron um eine Fibonacci Zahl zu errechnen. Schreiben Sie den Code für einen asynchronen Aufruf um.

Arbeitsblatt 29: Executor

In den vorangegangenen Entwurfsmustern wurde mit jedem Aufruf der Methode `Request` der Service-Klasse ein Handler-Thread erzeugt, um diesem die Service-Anfrage weiterzugeben. Diese Strategie macht Sinn und ist effizient, wenn Handler auf I/O-Aktionen oder sonstig beschränkte Ressourcen warten muss und wenn die Applikation auf einem Multiprozessorsystem läuft. Wenn allerdings zu viele Anfragen an den Service gestellt werden, kann dies schnell zur Überlastung der Ressourcen oder der CPU führen. Damit dieser Umstand entschärft werden kann, muss die direkte Beziehung zwischen dem Service-Objekt und dem Handler-Start im Programmcode gelockert werden. Am Besten mit einem Interface.

Der Executor stellt ein Interface dar, das eine Threadmethode via `ThreadStart` Delegate entgegennimmt. Die Schnittstelle ist ähnlich der eines Threads.

```
interface IExecutor {
   void Execute(ThreadStart threadStart);
}
```

Mit dieser Entkopplung sieht die Service-Klasse wie folgt aus:

```
public class Service
{
   protected final IExecutor executor;

   public Service(IExecutor executor) {
      this.executor = executor;
   }

   public void Request() {
      // Die Aufgabe dem Handler übergeben und starten...
      executor.Execute(new ThreadStart(new Handler().Do));
      // weitere Definitionen...
   }
}
```

Dem Konstruktor der Service-Klasse muss ein konkreter Executor übergeben werden. Die Klasse `PlainThreadExecutor` implementiert die `Execute` Methode, indem ein Thread erzeugt, das mit `Runnable` implementierte Objekt übergeben und gestartet wird.

```
public class PlainThreadExecutor : IExecutor
{
   void Execute(ThreadStart threadStart) {
      new Thread(threadStart).Start();
   }
}
```

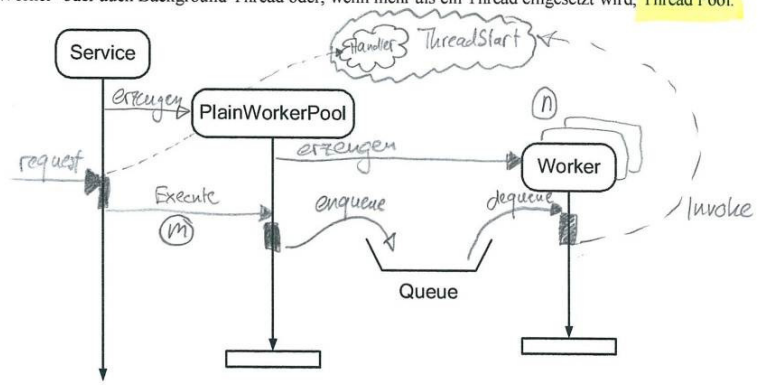
Damit wird der Service wie folgt gestartet:

```
   new Service(new PlainThreadExecutor()).Request();
```

=> Anzahl Worker = Anzahl Cores

Arbeitsblatt 30: Worker Threads

Mit Hilfe des Executor lassen sich nun Applikationen entwickeln, die eine limitiert nebenläufige Ausführungen von Handlern ermöglichen. In diesem Design werden Objekte mit `ThreadStart` Delegate Methoden erst in eine Queue gegeben, um diese dann durch einen weiteren Thread ausführen zu lassen. Diesen Thread nennt man Worker- oder auch Background-Thread oder, wenn mehr als ein Thread eingesetzt wird, Thread Pool.



Die Klasse `PlainWorkerPool` erlaubt es eine festgelegte Anzahl Threads zu starten, um Objekt aus der Queue zu holen und auszuführen. Damit ist es möglich Applikationen zu skalieren, d.h. mit einer begrenzte Anzahl Threads laufen zu lassen und damit Ressource zu schonen.

```
class PlainWorkerPool : IExecutor
{
   protected IQueue workQueue;

   public void Execute(ThreadStart threadStart) {
         workQueue.Enqueue(threadStart);
   }

   public PlainWorkerPool(IQueue workQueue, int nWorkers) {
      this.workQueue = workQueue;
      for (int i = 0; i < nWorkers; ++i)
         activate();
   }
   //...
```

=> m-Aufträge initiierbar, während nur n Threads gleichzeitig abgearbeitet

Arbeitsblatt 30: Worker Threads (Fortsetzung)

```
class PlainWorkerPool : IExecutor
{
   //...

   protected void activate() {
      Thread runLoop = new Thread(delegate() {
         while(true) {
            ThreadStart threadStart = (ThreadStart)workQueue.Dequeue();
            threadStart.Invoke();
         }
      });
      runLoop.Start();
   }
}
```

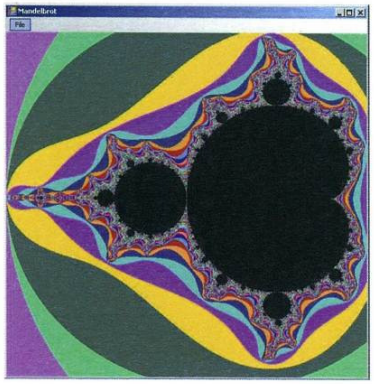
So kann beispielsweise der `PlainWorkerPool` beim Service Objekt Anwendung finden, so dass nun eine Skalierung möglich ist.

```
class TestWorkerPoolExec
{
   public static void Main() {
      PlainWorkerPool executor =
         new PlainWorkerPool(                  // Executor
         new BoundedBufferWithSemaphor(100),   // Queue
         10);                                  // Anzahl Workers

      Service service = new Service(executor);
      for (int i = 0; i < 10; i++) {
         service.Request();
      }
   }
}
```

Frage: Wenn Sie die `TestWorkerPoolExec` Anwendung starten, werden Sie feststellen, dass die `request` Methode zehn mal korrekt abgearbeitet wird, dann aber die Demo Anwendung nicht beendet. Warum ist dies so ? Wie könnte die Anwendung korrekt beendet werden ?

Übung 12: Mandelbrot-Menge

Benoit B. Mandelbrot vom Thomas J. Watson Forschungszentrum der IBM hat 1980 eine Arbeit mit dem Titel "Fractal Aspects of the Iteration of z -> dz(1-z) for complex d and z". Aus seiner Arbeit mit geometrischen Formen hat er ein Gebiet entwickelt, das er fraktale Geometrie nannte: die mathematische Untersuchung von Gebilden mit gebrochener Dimension. Im deutschen Volksmund werden diese Gebilde auch oft als "Apfelmännchen" bezeichnet. Die Mandelbrot-Menge besteht gerade aus der Menge der Zahlen c, deren Betrag unter beliebig langer Iteration gemäss der Vorschrift z = z*z + c endlich bleibt. Der Realteil ihrer Elemente liegt im Bereich zwischen –2 und 1, der Imaginärteil zwischen –1.5 und 1.5.



Mit Hilfe eines recht einfachen Algorithmus lässt sich eine Art mathematisches Mikroskop zur Untersuchung des Randes der Mandelbrot-Menge erstellen

```
z = 0; // komplexe Zahl
n = 0;
while (n < 100 && Betrag(z) < 2) {
    z = z*z + c;
    n++;
}
FarbeDesPunktesInC(n);
```

wobei z und c komplexe Zahlen sind. Die Farbe des Punktes c sollte dabei in Abhängigkeit von n sein. Zum Beispiel könnte die Farbe heller sein, je schneller z entkommt (wenn z überhaupt entkommt). Schwarz würde dann bedeuten, dass in der Iteration n = 100 erreicht hat, ohne dass der Betrag(z) grösser als 2 geworden wäre.

Erstellen Sie eine Windows-Applikation, das die Mandelbrot-Menge abbilden kann. Dabei soll das Programm die Iterationen nicht selbst durchführen, sondern an mehrere Handler weitergeben. Aber Achtung, zu viele Handler überlasten ein System schnell einmal.

4. Socket Kommunikation

In diesem Teil werden Implementierungen und Anwendungen von Kommunikation zwischen Applikationen mit Hilfe von Sockets mit TCP/IP besprochen. Das Transmission Control Protocol (TCP) und das Internet Protocol (IP) sind nur zwei der Protokolle. Es sind aber die am häufigsten verwendeten Protokolle und so bekam die Familie ihren Namen. Das User Datagram Protocol (UDP) ist ebenfalls Teil von TCP/IP. Welche Protokolle vom .NET Framework unterstützt werden, kann man in der .NET Reference nachlesen und zwar in der Dokumentation der Klasse `System.Net.Sockets.Socket`.

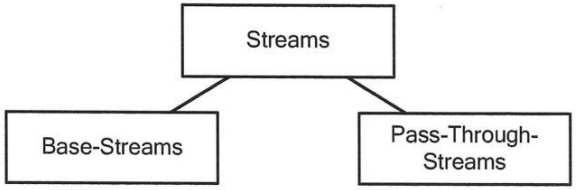
Da die C#-Netzwerk-Programmierung eigenständige und nicht zwingend grafikorientierte Anwendungen hervorbringt, bietet C# eine umfangreiche Bibliothek zur sequentiellen Ein- und Ausgabe an. Die dafür verwendeten Klassen realisieren das aus anderen Sprachen bekannte Konzept der *Streams*.

4.1. Ein- und Ausgabe-Streams

Streams dienen generell dazu, drei elementare Operationen ausführen zu können:

- Dateninformationen müssen in einen Stream geschrieben werden. Nach welchem Muster das geschieht, wird durch den Typ des Streams vorgegeben.
- Aus dem Datenstrom muss gelesen werden, ansonsten könnte man die Daten nicht weiterverarbeiten. Das Ziel kann unterschiedlich sein: Die Bytes können Variablen oder Arrays zugewiesen werden, sie könnten aber auch in einer Datenbank landen und zur Ausgabe an einem Peripheriegerät wie dem Drucker oder dem Monitor dienen.
- Nicht immer ist es erforderlich, den Datenstrom vom ersten bis zum letzten Byte auszuwerten. Manchmal reicht es aus, erst ab einer bestimmten Position zu lesen. Man spricht dann vom wahlfreien Zugriff.

Arbeitsblatt 31: Typen von Streams



Base-Streams, die direkt aus einem Strom Daten lesen oder in diesen hineinschreiben. Diese Vorgänge können z.B. in Dateien, im Hauptspeicher oder in einer Netzwerkverbindung enden.

Pass-Through-Streams ergänzen einen Base-Stream um spezielle Funktionalitäten. So können manche Streams verschlüsselt oder im Hauptspeicher gepuffert werden. Pass-Through-Streams lassen sich hintereinander in Reihe schalten, um so die Fähigkeiten eines Base-Streams zu erweitern. Auf diese Weise lassen sich sogar individuelle Streams konstruieren.

Die Stream-Klassen implementieren sequentielle Ein-/Ausgabe auf verschiedenen Datenquellen und Datensenken. Die I/O-Art kann sein

- zeichen- bzw. textorientiert (`StreamReader`, `StreamWriter`, `StringReader`, `StringWriter`): mit Wandlung zwischen interner Binärdarstellung und externer Textdarstellung. Grundlage ist die byteorientierte Ein- und Ausgabe mit den Klassen `TextReader` und `TextWriter`.
- binär (`BinaryReader`, `BinaryWriter`, Unterklassen von `Stream`): ohne Wandlung der Binärdarstellung.

Arbeitsblatt 32: Klassenstruktur I/O-Streams

```
System.Object
   System.IO.BinaryWriter
   System.IO.BinaryReader
   System.MarshalByRefObject
      System.IO.Stream
         Microsoft.JScript.COMCharStream
         Microsoft.WindowsMobile.DirectX.GraphicsStream
         System.Data.OracleClient.OracleBFile
         System.Data.OracleClient.OracleLob
         System.IO.BufferedStream
         System.IO.Compression.DeflateStream
         System.IO.Compression.GZipStream
         System.IO.FileStream
         System.IO.MemoryStream
         System.IO.UnmanagedMemoryStream
         System.Net.Security.AuthenticatedStream
         System.Net.Sockets.NetworkStream
         System.Security.Cryptography.CryptoStream
      System.IO.TextWriter
         System.CodeDom.Compiler.IndentedTextWriter
         System.IO.StreamWriter
         System.IO.StringWriter
         System.Web.HttpWriter
         System.Web.UI.HtmlTextWriter
      System.IO.TextReader
         System.IO.StreamReader
         System.IO.StringReader
```

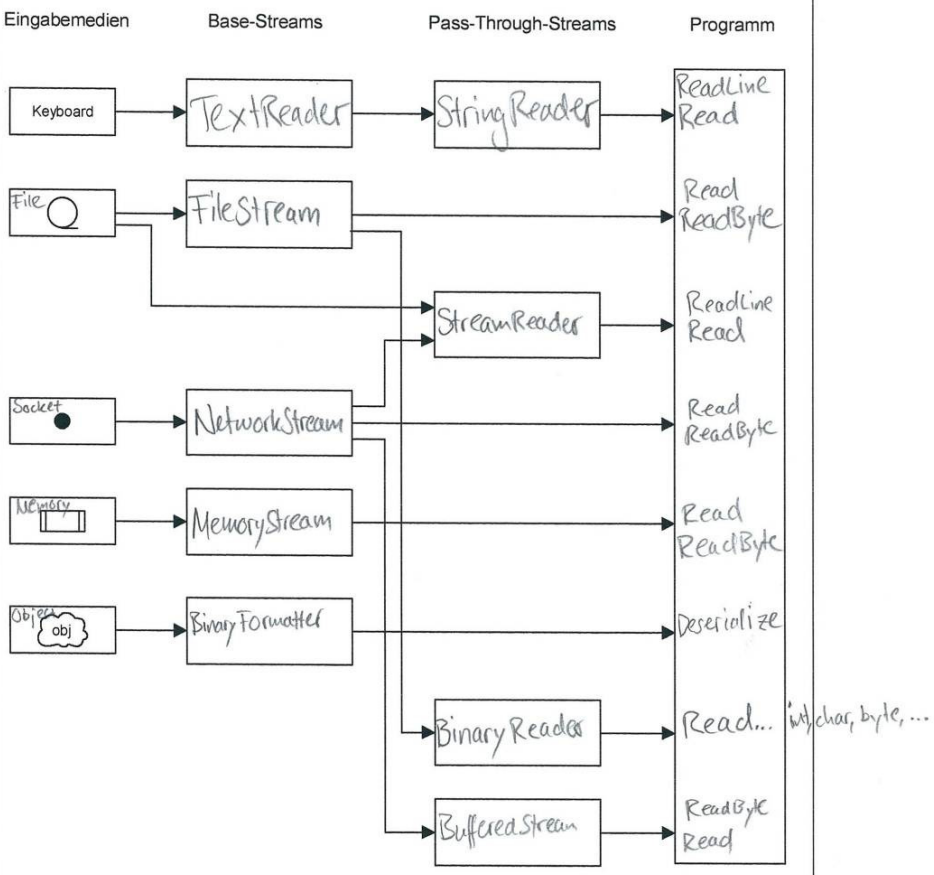
Die folgenden Klassen sind nicht Stream-Klassen, sondern unterstützen den `FileStream`.

```
   System.IO.DriveInfo
   System.IO.Directory
   System.MarshalByRefObject
      System.IO.FileSystemInfo
         System.IO.DirectoryInfo
         System.IO.FileInfo
```

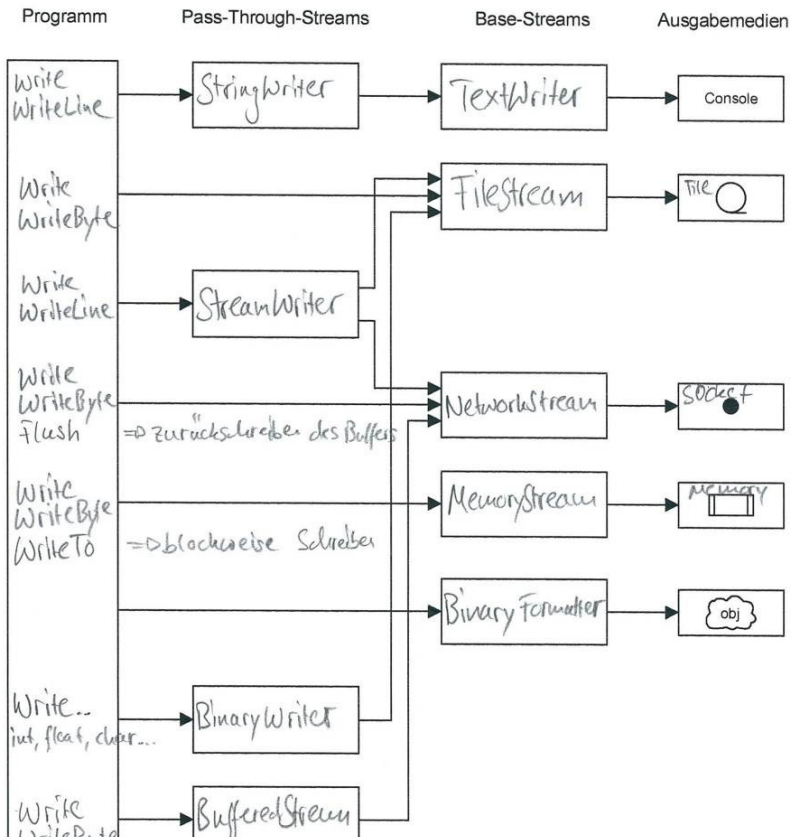
Alle Klassen zur Datenein- und -ausgabe befinden sich im Namespace `System.IO`. Um sie zu verwenden, die entsprechende Import-Anweisung an den Anfang eines Programms gestellt werden:

```
using System.IO;
```

Arbeitsblatt 33: Auswahl von Input Streams



Arbeitsblatt 34: Auswahl von Output Streams



Arbeitsblatt 35: Stream Beispiele

Die folgenden Beispiele zeigen einige Möglichkeiten der Verwendung von `Stream` Klassen.

Lesen von einer Tastatur und Schreiben in einen Bildschirm

```
class KeyboardRead {
   public static void Main() {
      string line;
      Console.Write("Bitte Eingabe: ");
      while ((line = Console.ReadLine()) != null) {
         Console.WriteLine("Eingabe war  : " + line);
         Console.Write("Bitte Eingabe: ");
      }
   }
}
```

Lesen aus einem Netzwerk-TCP-Socket:

```
class TcpRead {
   public static void Main() {
      TcpClient client = new TcpClient("192.53.103.103", 13);
      StreamReader inStream = new StreamReader(client.GetStream());
      Console.WriteLine(inStream.ReadLine());
      client.Close();
   }
}
```

Schreiben in eine Datei mit FileStream:

```
class FileWritel {
   public static void Main() {
      try {
         FileStream fs = new FileStream("daten.txt", FileMode.Create);
         StreamWriter sw = new StreamWriter(fs);
         string[] text = { "Titel", "Köln", "4711" };
         for (int i = 0; i < text.Length; i++) {
            sw.WriteLine(text[i]);
         }
         sw.Close();
         Console.WriteLine("fertig.");
      }
      catch (Exception e) {
         Console.WriteLine(e.ToString());
      }
   }
}
```


Arbeitsblatt 35: Stream Beispiele (Fortsetzung)

Schreiben in eine Datei mit implizitem FileStream:

```
class FileWrite2 {
   public static void Main() {
      try {
         using (StreamWriter sw = new StreamWriter("daten.txt")) {
            string[] text = { "Titel", "Köln", "4711" };
            for (int i = 0; i < text.Length; i++)
               sw.WriteLine(text[i]);
         }
         Console.WriteLine("fertig.");
      }
      catch (Exception e) { Console.WriteLine(e); }
   }
}
```

Lesen aus einer Datei:

```
class FileRead2 {
   public static void Main() {
      try {
         using (StreamReader sr = new StreamReader("daten.txt")) {
            string line;
            while ((line = sr.ReadLine()) != null) {
               Console.WriteLine(line);
            }
         }
      }
      catch (Exception e) { Console.WriteLine(e); }
   }
}
```

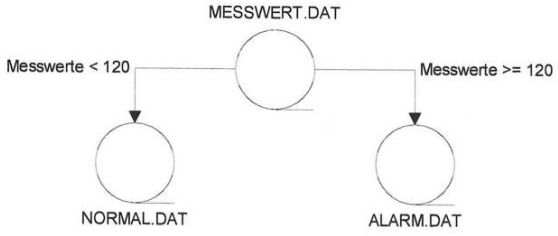
Lesen aus einer Datei mit einem Pass-Through-Stream:

```
class Crypto {
   public static void Main() {
      Stream stm =
         new FileStream("daten.txt", FileMode.Open, FileAccess.Read);
      ICryptoTransform ict = new ToBase64Transform();
      CryptoStream cs =
         new CryptoStream(stm, ict, CryptoStreamMode.Read);
      TextReader tr = new StreamReader(cs);
      string s = tr.ReadToEnd();
      Console.WriteLine(s);
   }
}
```

Übung 13: Stream Verarbeitung

Messstation

Eine Messstation für Luftschadstoffe liefert nach jeweils einem bestimmten Zeitabschnitt eine Datei von Messwerten. Dieses File ist vom Datentyp INTEGER und hat den Namen MESSWERT.DAT Sein Inhalt ist nicht sortiert. Ihre Aufgabe besteht nun darin die Integerwerte aus dieser Datei auszulesen und in zwei weitere Dateien NORMAL.DAT und ALARM.DAT abzuspeichern. Messwerte, die gleich oder grösser als 120 sind kommen in die Datei ALARM.DAT, alle anderen in die Datei NORMAL.DAT



4.2. Serialisierung

Arbeitsblatt 36: Prinzip

Die Serialisierung ist eine Technik, mit der Objekte in eine Folge von Bytes transformiert werden, die z.B. in Dateien gespeichert oder über ein Netzwerk übertragen werden können. In einem serialisierten Objekt wird die Klasseninformation und der aktuelle Zustand des Objektes zum Zeitpunkt der Serialisierung gespeichert. Unter Deserialisierung versteht man die Rekonstruktion eines serialisierten Objektes aus seiner serialisierten Form. Dazu stehen aus den Klassen `BinaryFormatter`, `SoapFormatter` und `XmlSerializer` die Methoden `Serialize` bzw. `Deserialize` zur Verfügung. Die Funktionsweisen dieser Methoden sind recht komplex. Es werden nämlich folgende Daten geschrieben bzw. gelesen.

- Der vollständig qualifizierte Klassename des zu serialisiernden Objekts.
- Die Signatur der Klasse.
- Alle nicht-statischen, nicht-transienten Attribute des Objekts inklusive aller aus den Oberklassen geerbten Attribute sowie die Attribute aller assoziierten Objekte.

Insbesondere der letzte Punkt verdient dabei besondere Beachtung. Ist ein Attribut vom Typ `static`, wird es nicht serialisiert, denn es gehört nicht zum Objekt, sondern zur Klasse des Objekts. Weiterhin werden alle Attribute ignoriert, die mit der Eigenschaft `[NonSerialized]` deklariert wurden. Auf diese Weise kann die Klasse Attribute definieren, die aufgrund ihrer Natur nicht serialisiert werden sollen oder dürfen. Wichtig ist weiterhin, dass ein Objekt nur dann serialisiert werden kann, wenn es mit der Eigenschaft `[Serializable]` deklariert wurde. Dies gilt nicht wenn Objekte mit Hilfe der Klasse `XmlSerializer` serialisiert werden.

Das folgende Beispiel soll zeigen, wie ein Objekt zum Schreiben und Lesen in bzw. aus einer Datei serialisiert wird. Dazu wird eine einfache Klasse konstruiert, die einen Integer-Wert und einen String kapselt

```
[Serializable]
class MyClass
{
   private int value = 0;
   private String id = null;

   public MyClass(String id, int value) {
      this.id = id;
      this.value = value;
   }

   public String getId() {
      return id;
   }

   public int getValue() {
      return value;
   }
}
```

Wichtig: Die Deklaration von `MyClass` muss mit der Eigenschaft [Serializable] deklariert werden. Die Methode `Serialize` testet das zu serialisierende Objekt auf diese Eigenschaft. Ist das nicht der Fall, wird eine Ausnahme des Typs `SerializationException` ausgelöst

Arbeitsblatt 36: Prinzip (Fortsetzung)

Mit Hilfe eines Objekts der Klasse `BinaryFormatter` kann ein `MyClass`-Objekt serialisiert und wieder hergestellt werden. Die Methoden `Serialize` bzw. `Deserialize` nehmen als Argumente jeweils ein Objekt der Klasse `Stream` entgegen.

```
class TestSerializeBinary
{
   public static void SerializeObject(String filename, Object obj) {
      FileStream fs = new FileStream(filename, FileMode.Create);
      BinaryFormatter binFormatter = new BinaryFormatter();
      binFormatter.Serialize(fs, obj);
      fs.Close();
   }

   public static Object DeserializeObject(String filename) {
      BinaryFormatter binFormatter = new BinaryFormatter();
      FileStream fs = new FileStream(filename, FileMode.Open);
      return binFormatter.Deserialize(fs);
   }

   public static void Main() {
      String filename = "daten.dat";
      SerializeObject(filename, new MyClass("Köln",4711));
      MyClass t = (MyClass)DeserializeObject(filename);
      Console.WriteLine(t.getId()+"("+t.getValue()+")");
   }
}
```

Hinweis: Es gibt keine verbindlichen Konventionen für die Benennung von Dateien mit serialisierten Objekten. Die im Beispiel verwendete Erweiterung `.dat` ist allerdings recht häufig zu finden. Wenn eine Anwendung viele unterschiedliche Dateien mit serialisierten Objekten hält, kann es auch sinnvoll sein, die Namen nach dem Typ der serialisierten Objekte zu vergeben.

Auch mit Hilfe eines Objekts der Klasse `SoapFormatter` kann ein `MyClass`-Objekt serialisiert und wieder hergestellt werden. Die Methoden werden identisch wie bei der Klasse `BinaryFormatter` angewendet. Im oben abgebildeten Code werden die Objekte `BinaryFormatter binFormatter` mit den folgenden `SoapFormatter soapFormatter` Objekte ersetzt.

```
SoapFormatter soapFormatter = new SoapFormatter();
soapFormatter.Serialize(fs, obj);
soapFormatter.Deserialize(fs);
```

Wichtig: Damit die Klasse `SoapFormatter` benutzt werden kann muss das C# Projekt auf die Bibliothek `System.Runtime.Serialization.Formatters.Soap.dll` referenzieren.

Arbeitsblatt 37: Serialisierung mit XML

Die Serialisierung mit der Klasse `XmlSerializer` unterscheidet sich erheblich von den Klassen `BinaryFormatter` und `SoapFormatter`:

- Die im XML-Format zu serialisierende Klasse muss `public` definiert sein.
- Es werden nur `public` deklarierte Attribute serialisiert.
- Die zu serialisierende Klasse muss einen öffentlichen, parameterlosen Konstruktor haben. Dieser wird von `XmlSerializer` aufgerufen.
- Die Steuerung der XML-Serialisierung erfolgt mit Attributen, die unter `System.Xml.Serialization` zu finden sind. Damit ist es beispielsweise möglich, bestimmte Felder vom Serialisierungsprozess auszuschliessen.
- Im Gegensatz zu `BinaryFormatter` und `SoapFormatter` ist die Eigenschaft `[Serializable]` nicht zwingend vorgeschrieben.

Obwohl unter Nutzung aller sich bietenden Möglichkeiten die XML-Serialisierung aufwändiger zu programmieren ist, hat sie einen entscheidenden Vorteil: XML ist ein offener Standard und deshalb plattformunabhängig. Die auf diese Weise serialisierten Daten lassen sich folglich von beliebigen Anwendungen verarbeiten.

Das folgende Beispiel soll zeigen, wie ein Objekt zum Schreiben und Lesen in bzw. aus einer XML-Datei serialisiert wird. Dazu wird eine einfache Klasse konstruiert, die einen Integer-Wert und einen String kapselt

```
public class MyClass
{
   public int value = 0;
   public String id = null;
   private String geheim = null;

   public MyClass() {
      geheim = "?";
   }

   public MyClass(String id, int value) {
      this.id = id;
      this.value = value;
      this.geheim = "Passwort";
   }

   public String getId() {
      return id;
   }

   public int getValue() {
      return value;
   }

   public String getGeheim() {
      return geheim;
   }
}
```

Arbeitsblatt 37: Serialisierung mit XML (Fortsetzung)

Mit Hilfe eines Objekts der Klasse `XmlSerializer` kann ein `MyClass`-Objekt serialisiert und wieder hergestellt werden. Die Methoden `Serialize` bzw. `Deserialize` nehmen als Argumente jeweils ein Objekt der Klasse `Stream` entgegen.

```
class TestSerializeXML
{
   public static void SerializeObject(String filename, MyClass obj)
   {
      FileStream fs = new FileStream(filename, FileMode.Create);
      XmlSerializer xml = new XmlSerializer(typeof(MyClass));
      xml.Serialize(fs, obj);
      fs.Close();
   }

   public static Object DeserializeObject(String filename)
   {
      XmlSerializer xml = new XmlSerializer(typeof(MyClass));
      FileStream fs = new FileStream(filename, FileMode.Open);
      return xml.Deserialize(fs);
   }

   public static void Main() {
      String filename = "daten.xml";
      SerializeObject(filename, new MyClass("Köln",4711));
      MyClass t = (MyClass)DeserializeObject(filename);
      Console.WriteLine(
         t.getId()+"("+t.getValue()+","+t.getGeheim()+")");
   }
}
```

Starten Sie das Programm `TestSerializeXML`.

Fragen:

- Was wird auf der Konsole ausgegeben ?
- Wie erklären Sie sich das Programmverhalten ?

Das serialisierte Objekt sieht dann so aus:

```
<?xml version="1.0"?>
<MyClass
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <value>4711</value>
   <id>Köln</id>
</MyClass>
```


Übung 14: Nicht serialisierbare Attribute

Es gibt eine Reihe von Objekttypen, die sich nicht serialisieren lassen. Das folgenden Programm versucht ein Objekt der Klasse `NotSerializable` zu serialisieren.

```
public class NotSerializable
{
   private Thread t;
   private String s;

   private void Print() {
      Console.WriteLine(s);
   }

   public NotSerializable() {
      t = new Thread(this.Print);
      s ="Fremde sind Freunde, die man nur noch nicht kennengelernt hat.";
      t.Start();
   }

   public static void Main() {
      try {
         MemoryStream ms = new MemoryStream(500);
         BinaryFormatter binFormatter = new BinaryFormatter();
         binFormatter.Serialize(ms, new NotSerializable());
         Console.WriteLine(Encoding.UTF8.GetString(ms.ToArray()));
      }
      catch (Exception e) {
         Console.WriteLine(e.ToString());
      }
   }
}
```

Fragen: Was stellen Sie fest, wenn Sie das Programm ausführen ?

_____

Wie erklären Sie sich das Programmverhalten ?

_____

Wie muss die Korrektur lauten, damit ein Objekt der obigen Klasse serialisiert werden kann ?

_____

Was ist der Grund, warum sich bestimmte Klassen nicht serialisieren lassen ?

_____

_____

Übung 15: Tiefe Objektkopien

Klassen können die `clone`-Methode von `Object` überschreiben und so eine Kopie des Zustands des Objekts liefern. Die Standardimplementierung ist jedoch so angelegt, dass diese Kopie flach ist. Das bedeutet, Referenzen auf Objekte, die von dem zu klonenden Objekt ausgehen, werden beibehalten und diese Objekte nicht extra kopiert.

Anhand eines kleinen Programms, können Sie sich überzeugen, dass die tiefe Kopie tatsächlich etwas anderes als ein `clone` ist...

```
class MakeDeepCopies
{
   public static Object DeepCopy(Object obj) {
      return null;
   }

   public static void WriteArray(string rem, ArrayList list) {
      Console.Write(rem);
      if (list != null) {
         foreach (Hashtable tbl in list)
            foreach (DictionaryEntry de in tbl)
               Console.WriteLine("{0}:{1}", de.Key, de.Value);
      }
      Console.WriteLine();
   }

   public static void Main() {
      Hashtable table = new Hashtable();
      table.Add(
         "Cul de Paris",
         "hinten unter dem Kleid getragenes Gestell oder Polster");

      ArrayList list1 = new ArrayList();
      list1.Add(table);
      WriteArray("list1: ",list1);
      ArrayList list2 = (ArrayList)list1.Clone();
      ArrayList list3 = (ArrayList)DeepCopy(list1);
      table.Clear();
      WriteArray("list1: ", list1);
      WriteArray("list2: ", list2);
      WriteArray("list3: ", list3);
   }
}
```

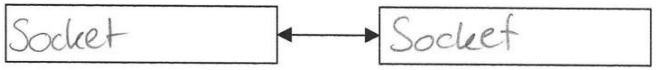
...das heisst natürlich nur, wenn Sie die statische Methode `deepCopy` in der Klasse `MakeDeepCopies` implementieren.

4.3. Socket-Kommunikation

Die Socket-Kommunikation kann allgemein als Interprozesskommunikation bezeichnet werden. Zwei oder mehr Prozesse (z.B. Applikationen) kommunizieren miteinander. Die Prozesse können auf demselben oder verschiedenen Computern oder auch anderen Maschinen laufen. Damit zwei Prozesse sich verstehen, müssen sie dieselbe Sprache sprechen, also gleiche Kommunikationsprotokolle verwenden. Die heute am häufigsten verwendeten Protokolle auf Netzwerkebene stammen aus der Protokollfamilie TCP/IP.

Arbeitsblatt 38: Socket Prinzip

Ende-zu-Ende Kommunikation



Sockets stellen eine Verbindung zwischen zwei Rechnern dar. Sie entsprechen etwa den Filehandles, die für den Zugriff auf Dateien angelegt und verwaltet werden müssen. Der Benutzer braucht sich nicht um technische Details der Datenübertragung zu kümmern. Er kann mit anderen Benutzern kommunizieren, die Details des Protokolls sind für ihn nicht sichtbar. Er kann einfach lesen und schreiben.

Was kann ein Socket:

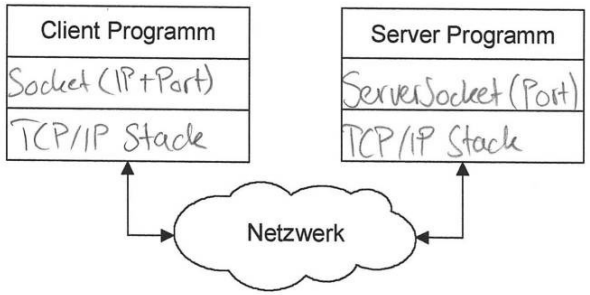
1. Verbindung zu einem entfernten Prozess aufbauen
2. Daten senden
3. Daten empfangen
4. Verbindung beenden
5. An einen Port binden (IP<->Port)
6. An bestimmtem Port auf auf Verbindung hören
7. Verbindungswunsch an Port akzeptieren

Der `System.Net.Sockets`-Namespace stellt eine verwaltete Implementierung der Windows Sockets-Schnittstelle (Winsock) bereit. Es bietet eine umfangreiche Socketfunktionalität, wie TCP-Sockets, Datagram-Sockets und Multicast-Sockets. Für vernetzte Anwendungen (z.B. für eigene Protokolle oder allgemeiner vernetzte Anwendungen), bietet C# die Klasse `Socket` als Abstraktion von standardmässigen Socket-Programmiertechniken. Die Sockets werden direkt mit `NetworkStreams` verknüpft. Damit können dann alle auf Streams aufbauenden Klassen und Methoden genutzt werden.

=> ähnlich wie Filehandling (urspr. aus UNIX)

Arbeitsblatt 39: Einsatz der Sockets

Sockets für TCP-Client und TCP-Server



Das folgende Beispiel benutzt den TcpClient-Konstruktor und überprüft, ob bestimmte Ports noch frei sind.

```
class PortScan
{
   public static void Main(string[] args) {
      TcpClient theClient;
      string host = "localhost";
      if (args.Length > 0) {
         host = args[0];
      }
      Console.WriteLine("Port Scanner");
      for (int port = 1; port < 65535; port++) {
         try {
            theClient = new TcpClient(host, port);
            Console.WriteLine(
               "Port {0} des Hosts {1} kennt TCP.", port, host);
         }
         catch (Exception e) {
            Console.WriteLine(
               "Port {0} des Hosts {1}: {2}", port, host,e.Message);
         }
      }
   }
}
```

Fragen:

- Was stellen Sie fest beim Ausführen des Programms ? – muss parallelisiert werden
- Wie erklären Sie sich das Programmverhalten ?
- Wie können Sie das Programmverhalten verhalten verbessern ? – mit Threads + Executer (Worker Queue)

– hier können sehr viele Threads gleichzeitig laufen gelassen werden (die Threads warten sowieso nur...)

Arbeitsblatt 40: Socket Eigenschaften

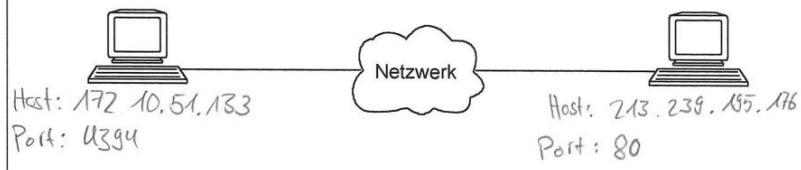
Das folgende Beispiel zeigt einige Eigenschaften, die Informationen über die Socket-Verbindung liefern.

```
class SocketInfo
{
   public static void Main(string[] args) {
      Console.WriteLine("Socket Information");
      for (int i = 0; i < args.Length; i++) {
         try {
            TcpClient theClient = new TcpClient(args[i], 80);
            Socket theSocket = theClient.Client;
            Console.WriteLine(
               "Der lokale Host (" +
               theSocket.LocalEndPoint +
               ") ist verbunden mit");
            Console.WriteLine(
               "dem entfernten Host " + args[i] + " (" +
               theSocket.RemoteEndPoint+ ")");
            Console.WriteLine(
               "über einen " + theSocket.ProtocolType + " Socket.");
            theClient.Close();
         }
         catch (Exception e) {
            Console.Error.WriteLine(e);
         }
      }
   }
}
```

Was stellen Sie fest, wenn Sie dieses Programm testen ?

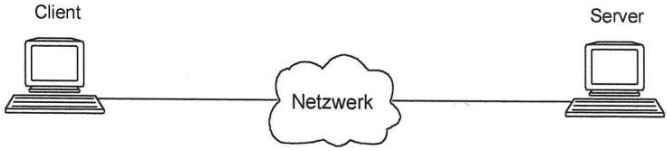
Der SendePort ist dynamisch und auf jeden Fall grösser als 1023 (well-known ports)

Was folgern Sie aus der obigen Feststellung ?



Arbeitsblatt 41: Server Socket

Ein `TcpListener` Objekt wird für einen bestimmten Port erzeugt und gestartet. Die Portnummer bestimmt die Art der Serveranwendung (www.iana.org). Anschliessend wird die Methode `AcceptTcpClient` aufgerufen, um auf einen eingehenden Verbindungswunsch zu warten. Die Methode `AcceptTcpClient` ist blockierend, d.h. es wird solange gewartet bis sich ein Client bei der Serveranwendung anmeldet. Ist der Verbindungsaufbau erfolgreich, liefert diese Methode ein Socket-Objekt, das wie bei einer Client-Anwendung zur Kommunikation mit der Gegenseite verwendet werden kann. Anschliessend steht der `TcpListener` für einen weiteren Verbindungsaufbau zur Verfügung.



Das folgende Server-Programm nimmt auf Port 13 einen Verbindungswunsch entgegen und sendet dem Client einen String mit Datum und Zeit.

```
class SimpleDayTimeServer
{
   public static void Main() {
         TcpListener listen = new TcpListener(13);
         listen.Start(); =D Starten des Abhörens
         while (true) {
            Console.WriteLine("Warte auf Verbindung auf Port "+
               listen.LocalEndpoint + "...");
            TcpClient client = listen.AcceptTcpClient();
            Console.WriteLine("Verbindung zu " +
               client.Client.RemoteEndPoint);
            StreamWriter sw = new StreamWriter(client.GetStream());
            sw.Write(DateTime.Now.ToString());
            sw.Flush();
            client.Close();
         }
      }
   }
```

Wie müssen Sie das `telnet`-Programm starten, um den `SimpleDayTime` Servers zu testen ?

telnet localhost 13

Was ist die grösste Einschränkung dieses `SimpleDayTime` Servers ?

Server kann Anfragen nicht parallel abarbeiten.

Übung 16: Socket Implementationen

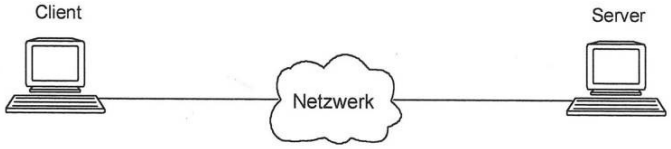
Erweiterung des `SimpleDayTime` Servers

Beheben Sie die Einschränkung des `SimpleDayTime` Servers. Testen Sie den neuen `DayTime` Server mit Hilfe eines selbst erstellten `DayTimeClients` lokal auf Ihrem Laptop. Der `DayTimeClient` kann eine einfache Konsolenapplikation sein oder auch eine kleine Windows-Anwendung.

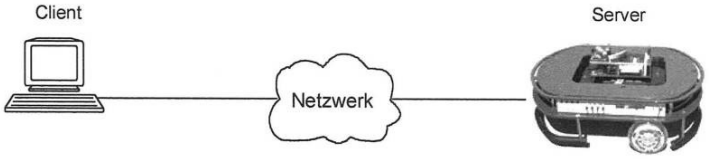
Test des `DayTime` Servers

Testen Sie den `DayTimeServer` mit dem `DayTimeClient` unter folgenden Bedingungen.

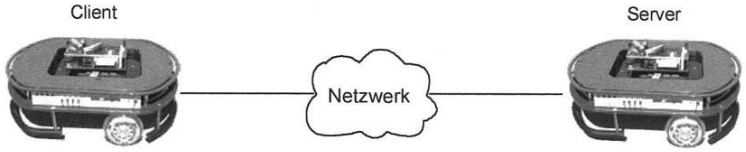
Verteilt auf verschiedenen Laptops



Verteilt auf Laptops und Roboter



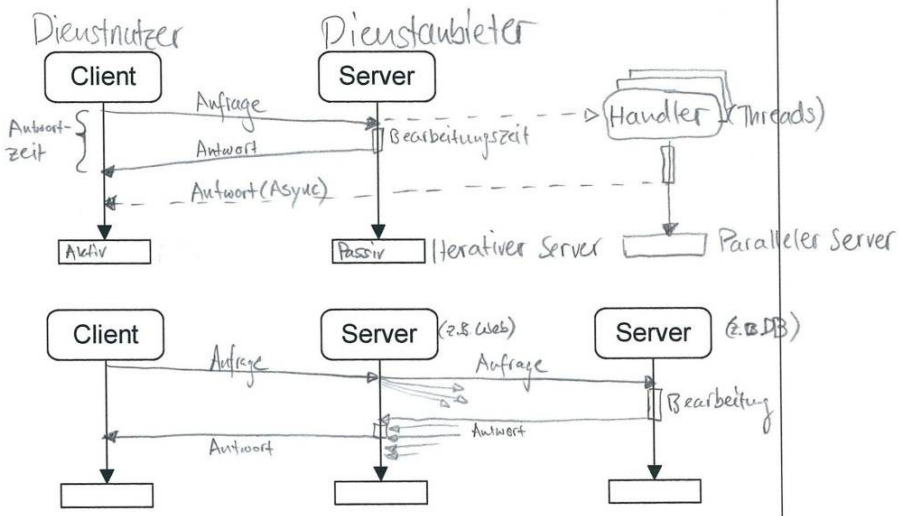
Verteilt auf verschiedenen Robotern



4.4. Client-Server

Das Client-Server-Verfahren ist das im Internet meist verwendete Konzept. Dabei bietet ein Server (Dienstleister) auf bekannten Kanälen Dienste an. Zum Beispiel FTP-Dienste oder auch News- oder Web-Dienste. Die Clients stellen eine Verbindung zu diesen Kanälen her und nehmen entsprechend dem Protokoll des Kanals den angebotenen Dienst in Anspruch.

Arbeitsblatt 42: Client-Server Verfahren



Dieses Verfahren wird typischerweise in folgenden Fällen eingesetzt.

- Die Anzahl der Clients ist nicht oder nur schwer zu bestimmen.
- Die Aufgaben sind eventuell nicht unabhängig voneinander, z.B. werden einige Details nur nachgefragt, wenn deren Information für den Client wichtig ist.
- Der Umfang der nachgefragten Leistungen variiert oder ist nicht genau bestimmbar, z.B. eine Web-Seite oder ein ganzer Teilbaum von Web-Seiten wird nachgefragt.
- Ein Client nimmt zum Teil die Leistungen mehrerer Server in Anspruch, bis die Aufgabe gelöst ist, z.B. Suchmaschine, Web-Server, FTP-Server.

Arbeitsblatt 43: AbstractServer

Das Server-Handler Entwurfsmuster soll für die Socket-Kommunikation verfeinert werden. Der Server erzeugt ein `TcpListener` Objekt und erhält von einem Client über das `TcpListener` Objekt einen Auftrag. Für die Erledigung des Auftrags wird ein `Handler`-Objekt erzeugt, dem der Client-Socket übergeben und in einem eigenen Thread gestartet wird. Ein Code-Muster dieses Servers sieht wie folgt aus:

```
public abstract class AbstractServer
{
   private Boolean running = true;
   private int port;
   private string host;
   public const int DEFAULTPORT = 4711;

   protected void Run() {
      IExecutor executor = CreateExecutor();
      Socket listen = CreateServerSocket();
      while (running) {
         AbstractHandler handler = CreateHandler(listen.Accept());
         executor.Execute(handler.Run);
      }
   }

   protected IExecutor CreateExecutor() {
      return new PlainThreadExecutor();
   }

   protected Socket CreateServerSocket() {
      IPAddress ipAddress = Dns.GetHostEntry(host).AddressList[0];
      TcpListener listener = new TcpListener(ipAddress, port);
      listener.Start();
      return listener.Server;
   }

   abstract protected AbstractHandler CreateHandler(Socket client);
   //...weitere Deklarationen
}
```

Ein konkreter Server kann, bzw. muss die folgenden drei Methoden implementieren:

`CreateExecutor` – Methode zur Erzeugung eines Executors. Mit dem Executor hat man die Möglichkeit den Server skalierbar zu machen. Je nach Bedürfnis kann der Executor mit verschiednen Buffern und unterschiedlicher Anzahl von ausführenden Threads initialisiert werden. Mit der Standardimplementierung wird ein einfacher Thread-Executor ohne Buffer erzeugt.

`CreateServerSocket` – Methode zur Erzeugung eines Serversockets. Diese Methode eröffnet die Möglichkeit einen Server Socket auf eine spezifische unterschiedliche Art zu erzeugen. Mit der Standardimplementierung wird ein `TcpListener`-Objekt am definierten Port erzeugt.

`CreateHandler` – Methode zur Erzeugung eines Handlers. Der Handler bestimmt die Funktion des Servers und muss daher mit durch eine konkrete Handler-Klasse implementiert werden.

Arbeitsblatt 44: AbstractHandler

Ein Handler in einer Client-Server-Anwendung hat in den meisten Fällen drei Aufgaben zu erledigen.

- Er verbindet den Client-Socket mit dem Input- und Outputstream.
- Er liest die Anfrage aus dem Inputstream und wertet diese aus.
- Er generiert eine entsprechende Antwort und sendet diese über den Outputstream dem Client.

Ein Muster dieses Handlers sieht daher wie folgt aus:

```
public abstract class AbstractHandler
{
   private Socket client;

   public AbstractHandler(Socket client) {
      this.client = client;
   }

   public void Run() {
      try {
         if (ReadRequest()) {
            CreateResponse();
         }
         client.Close();
      }
      catch (Exception e) {
         Console.Error.WriteLine(e);
      }
   }

   abstract protected Boolean ReadRequest();
   abstract protected void CreateResponse();
}
```

1 Anfrage, 1 Antwort, dann schliessen

Ein konkreter Handler muss einen Konstruktor und die folgenden zwei Methoden implementieren:

Konstruktor – Er verbindet den Client-Socket mit einem konkreten Input- und Outputstream. Das heisst, der konkreter Handler muss konkrete Input- und Outputstreams (z.B. `StreamReader`, `StreamWriter` oder `BinaryWriter`, `BinaryReader`, etc.) als Attribute definieren, sodass die Methoden `ReadRequest` und `CreateResponse` darauf zugreifen können.

`ReadRequest` – Methode zum Lesen der Anfrage des Klienten. Sie liest aus dem konkreten Inputstream Daten oder Anweisungen aus, welche für die Antwort benötigt werden.

`CreateResponse` – Methode zur Erzeugung einer entsprechenden Antwort. Über den konkreten Outputstream werden die Daten (oder Objekte) der Antwort an den Klienten gesendet.

Anmerkung: Eine Variante dieses `AbstractHandler` könnte in einer Iteration die Methoden `ReadRequest` und `CreateResponse` abarbeiten. Dies hängt davon ab, für welche Aufgabe ein konkreter Handlers eingesetzt wird. Während bei HTTP Anwendungen nach jeder Anfrage und Antwort die Socket-Verbindung wieder geschlossen wird, sind bei FTP Anwendungen viele Anfragen und Antworten mit der gleichen Socket-Verbindung möglich.

Arbeitsblatt 45: Hello-Server

Zur Verwendung des Server-Handler Entwurfsmuster soll ein einfaches Beispiel eines Hello-Servers dienen. Dieser Server macht nichts anderes, als auf eine Anfrage den String "Hello world" zu senden. Dazu muss die Klasse `HelloServer` von `AbstractServer` abgeleitet sein und die Methode `CreateHandler` implementieren. In dieser Methode wird ein `HelloHandler` Objekt erzeugt. Nun muss nur noch ein Thread der Klasse `HelloServer` erzeugt und gestartet werden.

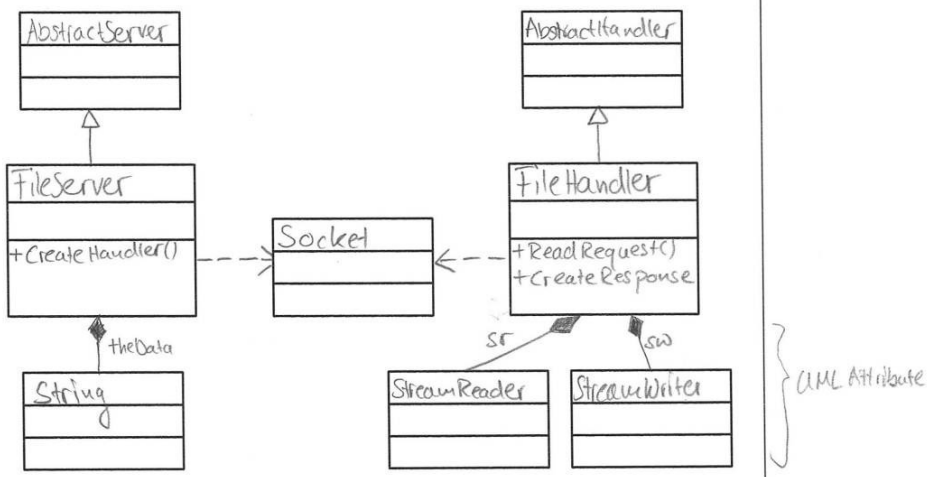
```
class HelloServer : AbstractServer
{
   override protected AbstractHandler CreateHandler(Socket client) {
      return new HelloHandler(client);
   }

   public static void Main() {
      new HelloServer().Start();
      Console.WriteLine("HelloServer auf {0} gestartet.",
         AbstractServer.AbstractServer.DEFAULTPORT);
   }
}
```

Die Klasse `HelloHandler` muss von `AbstractHandler` abgeleitet sein. Sie beinhaltet die beiden privaten Stream-Attribute `StreamReader sr` und `StreamWriter sw`. Zudem müssen ein Konstruktor mit einem `Socket`-Parameter und die abstrakten Methoden `ReadRequest` und `CreateResponse` mit der gewünschten Funktionalität implementiert werden.

```
class HelloHandler : AbstractHandler
{
   private StreamReader sr;
   private StreamWriter sw;

   public HelloHandler(Socket client) : base (client) {
      NetworkStream nws = new NetworkStream(client,true);
      sr = new StreamReader(nws);
      sw = new StreamWriter(nws);
   }

   override protected Boolean ReadRequest() {
      String request = sr.ReadLine();
      return true;
   }

   override protected void CreateResponse() {
      sw.WriteLine("Hello world.");
      sw.Flush();
   }
}
```

Übung 17: Einfacher File-Server

Als Vorstufe zum Web-Server sollen Sie einen sehr einfachen File-Server entwickeln. Ruft ein Client den Server an, so sendet dieser eine zu voraus bestimmte Datei mit dem Hypertext Transfer Protocol (siehe Anhang). Erstellen Sie dazu zuerst eine Klassenstruktur mit den für die Datenübertragung notwendigen Klassen. Tragen Sie in die Klassen alle notwendigen Methoden ein.



Implementieren und Testen Sie den File-Server und beantworten Sie folgende Fragen.

a) Welchen Port müssen Sie zum Start des File-Server angeben ?

Port > 1024

b) Wie muss ein Browser den File-Server adressieren ?

http://localhost:4711

c) Wie muss das Telnet den File-Server adressieren ?

telnet > o localhost 4711 > GET

d) Wie implementieren Sie Ihren File-Server, so dass er skalierbar ist ?

theData vorladen; in FileServer

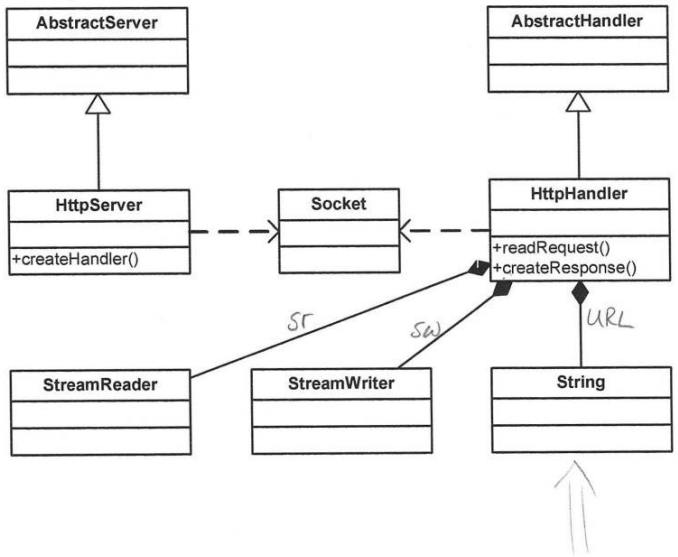
4.5. Client-Server angewandt: HTTP-Server

Die Kommunikation zwischen einem Browser und einem Web-Server entspricht etwa folgendem Schema (siehe auch den Anhang: Das Hypertext Transfer Protocol (HTTP)):

- Der Web-Browser baut eine Verbindung zum Server auf.
- Er schickt eine Seitenanforderung (und zusätzliche Informationen) in Form eines Requests gemäss HTTP-Spezifikation.
- Der Server analysiert den Request und schickt die gewünschte Datei (bzw. eine Fehlermeldung) an den Browser.
- Der Server beendet die Verbindung.

Hat der Browser auf diese Weise eine HTML-Seite erhalten, interpretiert er den HTML-Code und zeigt die Seite formatiert auf dem Bildschirm an. Enthält die Datei IMG-, APPLET- oder ähnliche Elemente, werden diese in derselben Weise vom Server angefordert und in die Seite eingebaut. Die wichtigste Aufgabe des Servers besteht also darin, eine Datei an den Client zu übertragen.

Arbeitsblatt 46: Klassenstruktur des HTTP-Servers



hier dynamisch, da der User die URL vorgibt

Arbeitsblatt 47: Implementation des HTTP-Servers

Der HTTP-Server besteht aus den beiden Klassen `HttpServer` und `HttpHandler`, die nach dem bisher vorgestellten Muster von den Klassen `AbstractServer` und `AbstractHandler` abgeleitet sind. In `HttpServer` wird ein Handler erzeugt und die weitere Bearbeitung des Requests an ein Objekt der Klasse `HttpHandler` delegiert. Mit der Erzeugung des Handlers werden die Ein- und Ausgabestreams zur Kommunikation mit dem Client-Socket initialisiert, sowie die beiden Methoden `readRequest` und `create-Response` implementiert. Zur Kontrolle gibt der Handler alle eingelesenen Request-Header-Zeilen und die entsprechenden Response Zeilen auf die Konsole aus. Zudem werden mit Hilfe des Integer-Attributs `call` der Klasse alle `HttpServer` eingehenden Anfragen gezählt, indem bei der Erzeugung des Handlers `call` inkrementiert und übergeben wird. Mit Überschreiben der Methode `createExecutor` ist der HTTP-Server skalierbar. Er kann bis zu 1000 Requests in kurzer Folge annehmen und arbeitet diese mit maximal 20 Handler-Threads ab. Das Verhältnis von Requests zu Handler-Threads ist willkürlich festgelegt und kann oder sollte je nach Rechner angepasst werden.

```
/**
 * Ein ganz einfacher Web-Server auf TCP und einem beliebigen Port.
 * Der Server ist in der Lage, Seitenanforderungen zu dem Luafwerk
 * und Verzeichnis zu bearbeiten, von dem er gestartet wurde.
 * Wurde der Server z.B. im Laufwerk c: gestartet, so würde eine
 * Seitenanforderung "http://localhost/test/index.html" die Datei
 * c:\test\index.html laden.
*/
class HttpServer : AbstractServer
{
   private int call = 0;

   override protected AbstractHandler CreateHandler(Socket client) {
      return new HttpHandler(client, ++call);
   }

   override protected IExecutor CreateExecutor() {
      return new PlainWorkerPool(new BoundedBufferWithSemaphor(1000),20);
   }

   public static void Main(String[] args) {
      int port = 80;
      if (args.Length > 0) {
         port = Int32.Parse(args[0]);
      }
      new HttpServer().Start(port);
      Console.WriteLine("HTTP server running on port: " + port);
   }
}
```



```
class HttpHandler : AbstractHandler
{
   // Multipurpose Internet Mail Extensions (MIME)
   // Internet Erweiterungen für die Einbindung von Daten.
   private string[,] mimetypes = {
     {"html","text/html"},
     {"htm", "text/html"},
     {"txt", "text/plain"},
     {"gif", "image/gif"},
     {"jpg", "image/jpeg"},
     {"jpeg","image/jpeg"}
   };
   private int id;
   private NetworkStream nws;
   private StreamReader sr;
   private StreamWriter sw;
   private string url;

   public HttpHandler(Socket client, int id) : base(client) {
      this.id = id;
      nws = new NetworkStream(client, true);
      sr = new StreamReader(nws);
      sw = new StreamWriter(nws);
   }

   override protected bool ReadRequest() {
      Console.WriteLine(id+". Incoming request...");
      string headerline;
      ArrayList request = new ArrayList();
      // Request-Header-Zeilen lesen bis zur Leerzeile
      while ((headerline = sr.ReadLine()) != null && headerline != "") {
         request.Add(headerline);
         Console.WriteLine("< " + headerline);
      }
      // 1. Request-Zeile auf HTTP Methode GET untersuchen
      string[] tokens = ((string)request[0]).Split(new char[] { ' ' });
      if (tokens.Length >= 2 && tokens[0] == "GET") {
         // URL ermittlen
         if (tokens[1].StartsWith("/"))
            url = tokens[1];
         // Start URL setzen
         if (url.EndsWith("/"))
            url += "index.html";
         return true;
      }
      else {
         WriteError(400, "Bad Request");
         return false;
      }
   }
```

```
override protected void CreateResponse() {
   try {
      string filename = url.Substring(1);
      FileStream fs =
         new FileStream(filename, FileMode.Open, FileAccess.Read);
      long len = fs.Length;
      byte[] bytes = new byte[len];
      fs.Read(bytes, 0, (int)len);
      // Alles OK
      WriteResult("HTTP/1.0 200 OK");
      // Servererkennung senden
      WriteResult("Server: ExperimentalWebServer 1.0");
      // Content-Length ermitteln und senden
      WriteResult("Content-Length: "+bytes.Length.ToString());
      // Content-Type ermitteln und senden
      // Beginnen mit unbekanntem Dateityp
      String mimestring = "application/octet-stream";
      for (int i = 0; i < mimetypes.Length/2; ++i) {
         if (url.EndsWith(mimetypes[i,0])) {
            mimestring = mimetypes[i,1];
            break;
         }
      }
      WriteResult("Content-Type: " + mimestring);
      // Leerzeile senden
      WriteResult("");
      // Daten senden
      nws.Write(bytes, 0, (int)len);
      nws.Flush();
      fs.Close();
   }
   catch (FileNotFoundException) {
      WriteError(404, "File Not Found");
   }
   catch (DirectoryNotFoundException) {
      WriteError(404, "File Not Found");
   }
   catch (Exception e) {
      WriteError(410, "Unknown Exception");
      Console.Error.WriteLine(e);
   }
}

public void WriteResult(string message) {
   Console.WriteLine("> " + message);
   sw.Write(message+"\r\n");
   sw.Flush();
}
```



```
public void WriteError(int status, string message)
{
   string output = "<h1>HTTP/1.0 " + status + " " + message + "</h1>";
   Console.WriteLine("> " + status.ToString() + " " + message);
   sw.Write(output);
   sw.Flush();
}
}
```

Übung 18: HTTP-Server

Untersuchen Sie den abgebildeten HTTP-Server und beantworten Sie die Fragen.

a) Welchen Port müssen Sie zum Start des Web-Server angeben ?

http://localhost:8080 (index.html)

b) Welche HTTP-Kommandos werden akzeptiert ?

GET

c) Welches ist die erste Seite die ein Browser vom HTTP-Server erhält ?

index.html

d) Welche Art von Seiten unterstützt der HTTP-Server ?

nur Client-seitige Programmierung => siehe Mime Types

e) Welche Fehler erkennt und generiert der HTTP-Server ?

404 (File not found), 410 (Unknown), 400 (Bad request)

f) Wie sicher ist der HTTP-Server ?

Sehr unsicher: • Files von fremden Systemen können gelesen werden
• GET Ø und leere Eingabe bringen Absturz

g) Was müsste getan werden, damit der HTTP-Server sicherer wird ?

• Angefragtes File muss den Basispfad des Webservers enthalten.
•

5. Bluetooth für den HSLU Roboter

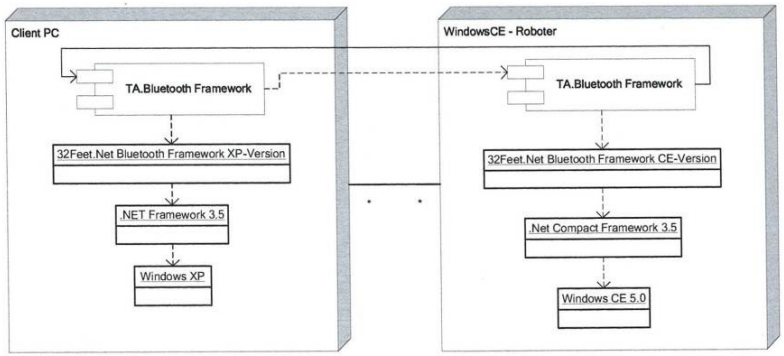
5.1. Übersicht

Bluetooth ist ein Funkstandard, der es ermöglicht, elektronische Geräte drahtlos miteinander zu verbinden. Hierzu zählen insbesondere Handys, Headsets, Drucker, moderne Autoradios uvm. Bluetooth-Adapter gibt es inzwischen für nahezu jedes Einsatzgebiet. Eine gute Einführung in Bluetooth finden man im Internet unter: http://de.wikipedia.org/wiki/Bluetooth.

Dieses Kapitel behandelt das Thema „Bluetooth für den HSLU Roboter". Dabei geht es um die Erweiterung des von der Hochschule Luzern – Technik & Architektur entwickelten Roboters um die Fähigkeit mittels Bluetooth zu kommunizieren.

Die für die Kommunikation notwendige Software:

- Betriebssystem: Windows XP für Client, Windows CE 5.0 für HTA Roboter
- .Net Framework 2.0 für Windows XP .Net Compact Framework 2.0 WinCE
- 32Feet.net Framework in XP- und CE-Version Version 2.2
- TA Bluetooth Library



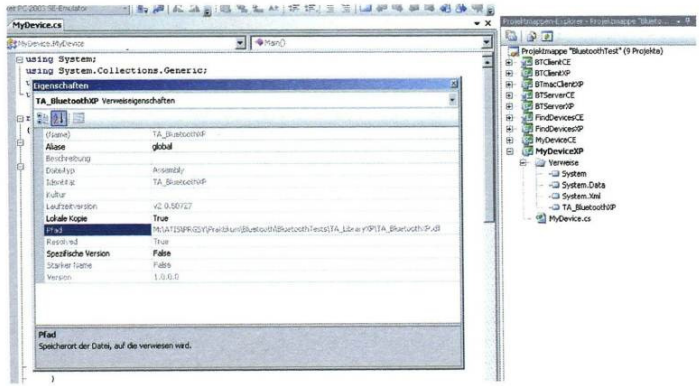
Client-PC
Über die TA Bluetooth-DLL wird auf die Funktionen des installierten Bluetooth-Geräts zugegriffen. Die Bibliothek verwendet dabei Methoden, die von 32feet.NET und dem .Net 3.5 Framework zur Verfügung gestellt werden.

HSLU Roboter
Über die TA Bluetooth-DLL wird auf die Funktionen des installierten Bluetooth-Geräts zugegriffen. Die Bibliothek verwendet dabei Methoden, die von 32feet.NET und dem .Net 3.5 Compact Framework zur Verfügung gestellt werden

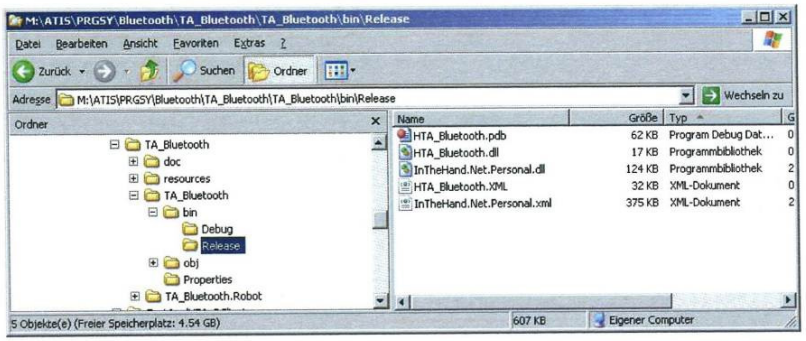
Deployment auf Produktiv-Umgebung
Durch das einbinden der Bibliothek in ein Projekt werden die nötigen Dateien automatisch ins Verzeichnis des generierten Programms kopiert, oder direkt darin integriert. Das daraus entstanden Bundle kann danach auf verschieden System verteilt und dort gestartet werden.

5.2. Ressourcen einbinden

Als erstes muss abhängig davon, ob das Programm am Ende auf einem Windows XP Client oder einem WinCE Roboter läuft die passende TA.Bluetooth.dll als externe Ressource eingebunden werden.

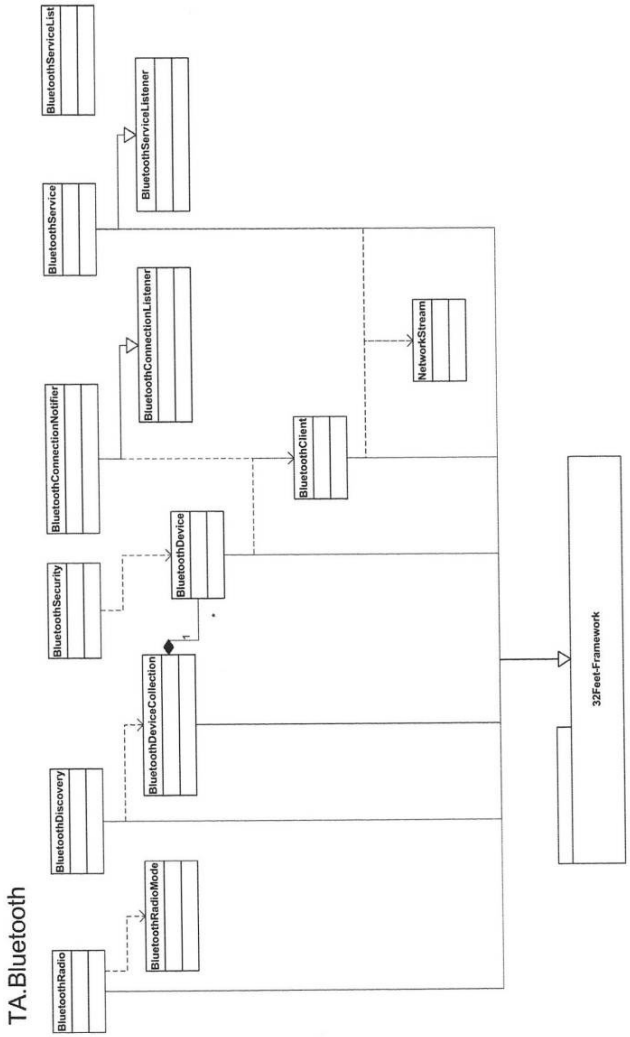


Dabei muss sich im selben Verzeichnis auch die InTheHand.Net.Personal.dll befinden. Diese ist die Basis der TA.Bluetooth.dll und wird automatisch mit dieser eingebunden.



Falls ein Programm auf beiden Plattformen laufen soll, können entweder innerhalb einer Solution zwei Projekte mit unterschiedlichen Ressourcen und Build-Targets erstellt werden, oder die beiden Bluetooth-Dlls nach dem kompilieren mit dem jeweils passenden DLL-Files überschrieben werden.

5.3. Klassenstruktur



BluetoothRadio
Stellt das lokale Bluetooth-Device dar.

IsSupported: Prüft, ob auf diesem System ein Bluetooth-Gerät vorhanden ist.

Name: Setzen des Gerätenamens, der für andere Bluetooth-Geräte angezeigt wird.

LocalAddress: Bluetooth-Adresse des Geräts

PrimaryRadio: Liefert das verwendete Bluetooth-Gerät zurück. Obwohl mehrere Bluetooth-Geräte gleichzeitig installiert werden können unterstützt der MS-Treiber nur ein aktives Gerät.

Mode: Ermöglicht das setzen der Bluetooth Sichtbarkeit. Wird momentan unter WinCE nicht unterstützt, da auf diesem die dazu benötigte bthutil.dll fehlt.

BluetoothRadioMode
Liste der möglichen Bluetooth Sichtbarkeitsmodi.

PowerOff: Bluetooth-Gerät ist deaktiviert.

Connectable: Verbindung zum Gerät kann hergestellt werden, Wird bei einer Suche nach erreichbaren Bluetooth-Geräten aber nicht angezeigt.

Discoverable: Gerät ist aktiviert und für andere sichtbar.

BluetoothDiscovery
Diese Klasse dient zum Auffinden von anderen Bluetooth-Geräten.

DiscoverDevices(): Liefert eine Liste aller erreichbaren Bluetooth-Geräte.

DiscoverDevices(serviceId): Liefert eine Liste aller erreichbaren Bluetooth-Geräte mit einem bestimmten Dienst.

DiscoverDevices(deviceName): Liefert eine Liste mit Bluetooth-Geräten, welche den übergebenen String als Name definiert haben.

DiscoverDevices(serviceList): Liefert eine Liste aller erreichbaren Bluetooth-Geräte mit mindestens einem Dienst aus der übergebenen Liste.

DiscoverDevices(count, authenticated**, remembered, unknown):** Sucht nach erreichbaren Bluetooth Geräten. Liefert maximal "count" Geräte zurück. Über die Boolean authenticated, remembered und unknown können die Geräte auf zusammengehängte (Paired devices), bereits bekannte und alle unbekannten eingeschränkt werden.

GetDiscoveredDevices(): Liefert die bei der letzten Suche gefundenen Geräte zurück ohne die Suche nochmals zu starten.

BluetoothServiceList
Liste zur Zuordnung von Guids zu bestimmten Diensten.

Robot01... Robot32: Generischer Dienst, der auf dem Roboter gestartet wird.

Um eigene Dienste zu Registrieren muss eine Klasse erstellt werden, welche die BluetoothServiceList Klasse erweitert. Darin wird als statische Variable eine neue Guid erstellt. Der Konstruktor dieser Klasse muss überschrieben werden. Bei der Instanzierung müssen alle in dieser Klasse vorhandenen statischen Guids zu der serverList-ArrayList hinzugefügt werden. Damit auch die Guids der übergeordneten Klasse in die Liste aufgenommen werden, sollte auch der übergeordnete Konstruktor aufgerufen werden, indem die Konstruktor-Deklaration mit dem base-Aufruf erweitert wird: public MyServiceList():**base()**{…}

GetServerList(): Liefert eine Liste von allen selbst definierten Guids.

BluetoothDeviceCollection
Liste von gefundenen Bluetooth-Geräten.

BluetoothDevice
Repräsentiert ein anderes Bluetooth-Geräte in Reichweite.

Address: 6 Byte Geräte-Adresse

Name: Bezeichnung des Bluetooth-Geräts

Authenticated: Zeigt, ob dieses Gerät durch Schlüsselaustausch mit dem Host verbunden wurde. Solche Geräte werden als PairedDevices bezeichnet.

OpenConnections: Liste aller geöffneten Verbindungen zu diesem Gerät.

Connect(serviceId): Versucht eine Verbindung zu diesem Service herzustellen.

Connect(macAddr, serviceId): Versucht eine Verbindung zu einem Gerät mit spezifischer MAC Adresse mit der Form "001122334455" und dem entsprechenden Service herzustellen.

HasService(serviceId): Prüft, ob auf dem RemoteDevice ein bestimmter Dienst gestartet wurde.

Close(): schliesst alle Verbindungen, die noch zu diesem Gerät aufgebaut sind.

GetClient(serviceId): Liefert die Verbindung zu dem gewünschten Dienst, falls sie bereit aufgebaut wurde.

BluetoothClient
Repräsentiert die Verbindung zu einem anderen Bluetooth-Gerät.

ServiceId: Dienst mit dem die Verbindung aufgebaut wurde.

Connected: Prüft, ob die Verbindung noch besteht.

GetStream(): Liefert einen NetworkStream zum Remote-Gerät.

GetSocket(): Liefert den Socket der Verbindung zum Remote-Gerät

Close(): Schliesst alle offenen Verbindungen.

BluetoothSecurity
Klasse, die Zugriff auf die Bluetooth-Sicherheitsfunktionen bietet.

PairRequest(device, pin): Versucht sich beim angegebenen Gerät mit dem übergebenen Pin zu authentifizieren.

RemoveDevice(device): Entfernt das Gerät von der Liste authentifizierter Geräte.

SetPin(address, pin): Definiert den für das Gerät mit dieser Adresse gültigen Pin.

BluetoothService
Bietet die Möglichkeit, auf diesem Gerät einen Dienst zu starten. Verbindungen über Bluetooth können nur zu einem Dienst aufgebaut werden.

CreateService(serviceId): Startet einen Dienst mit der gewünschten Guid.

StopService(): Stoppt den gestarteten Dienst.

WaitForConnection(): Wartet auf die nächste Verbindung zu diesem Gerät. Blockiert die Verarbeitung bis zum erstellen einer Verbindung.

CreateAsynchService(serviceId, serviceListener): Startet einen Dienst in einem anderen Thread. Beim Aufbau einer Verbindung wird diese and den serviceListener zur Verarbeitung übergeben.

AddConnectionListener(listener): Fügt dem aktuellen Service einen serviceListener hinzu. Beim Aufruf von waitForConnection wird die eingehende Verbindung zuerst an den Listener übergeben.

BluetoothServiceListener
Interface für ServiceListener. Als abstrakte Klasse ausgeführt.

HandleConnection(client): Nimmt einen BluetoothClient zur Verarbeitung entgegen. Wird als erstes aufgerufen.

HandleConnection(stream): Erhält einen NetworkStream zur Verarbeitung. Wird nur ausgeführt, wenn keine Methode mit einem BluetoothClient-Parameter definiert wurde.

ServiceTerminated(): Wird aufgerufen, wenn der Service gestoppt wird.

BluetoothConnectionNotifier
Klasse um auf über den NetworkStream übertragene Daten zu reagieren.

AddConnectionListener(listener): Fügt dem BluetoothClient einen ConnectionListener hinzu, welcher aufgerufen wird, sobald Daten über den NetworkStream übertragen werden.

StopListening(): Stoppt jede weitere Notifikation über anstehende Daten.

BluetoothConnectionListener
Interface für ConnectionListener. Als abstrakte Klasse ausgeführt.

HandleIncomingData(client): Nimmt einen BluetoothClient zur Verarbeitung entgegen. Wird als erstes aufgerufen:

HandleIncomingData(stream): Erhält einen NetworkStream zur Verarbeitung. Wird nur ausgeführt, wenn keine Methode mit einem BluetoothClient-Parameter definiert wurde.

Disconnected(client): Wird bei einem Verbindungsunterbruch aufgerufen.

Anmerkungen zu der Bibliothek

Derzeit gibt es noch drei offene oder fehlerhafte Punkte in der Bibliothek.

- **BluetoothRadio:** Das setzen des BluetoothRadioMode funktioniert nur unter WinCE. Da auf dem HSLU Roboter jedoch eine wichtige Bibliothek (bthutil.dll) fehlt, ist diese Funktion auch auf dem Roboter nicht ausführbar. Die restlichen Werte sind bisher nur Read-only.
- **BluetoothSecurity:** Die Authentifizierung eines Geräts nur durch Code ist nicht klar definiert. Einige Funktionen sind nur unter WinCE verfügbar. Mit der neuen Version des 32Feet-Frameworks sollten in diesem Bereich Verbesserungen möglich sein. Falls wirklich eine verschlüsselte Verbindung benötigt wird, können die Geräte vorgängig manuell authentisiert werden.
- **BluetoothClient:** Die Abfrage, ob ein Client noch verbunden ist, zeigt nur an, ob die Verbindung bei der letzten ausgeführten Datenübertragung noch verbunden war. Der aktuelle Status der Verbindung kann also nicht ermittelt werden. Dieses Verhalten wurde von Microsoft definiert. Eine alternative Möglichkeit zur Aktualisierung des Status funktioniert derzeit noch nicht zuverlässig. Aus diesem Grund kann es sein, dass der BluetoothConnectionNotifier eine unterbrochene Verbindung nicht bemerkt und deshalb endlos wartet. In diesem Fall muss der Notifier manuell gestoppt werden.

5.4. Server Aktionen

Service starten

Nach dem Import des Namespaces mit `using TA.Bluetooth;` kann mit den folgenden Code-Zeilen ein Bluetooth-Dienst gestartet werden, auf welchen man von anderen Bluetooth-Geräten aus verbinden kann. Die Dienste werden über eine Guid identifiziert. Ein Verbindungsaufbau zu einem Dienst erfolgt immer über seine serviceId.

```
// check if Bluetooth-Stick is available
if (BluetoothRadio.IsSupported)
{
    // Device visible for others
    // RadioMode setting not supported as long as the
    // WinCE image doesn't include the BthUtil.dll
    // BluetoothRadio.PrimaryRadio.Mode = BluetoothRadioMode.Discoverable;

    // set device Name
    BluetoothRadio.PrimaryRadio.Name = "HSLU Robot";
    // desired service
    Guid serviceId = BluetoothServiceList.Robot07;

    // start new service
    BluetoothService service = new BluetoothService();
    service.CreateService(serviceId);
}
```

Verbindung annehmen

Nachdem ein Bluetooth-Dienst gestartet wurde, können eingehende Verbindungen auf diesen Dienst verarbeitet werden. Bei einem Verbindungsaufbau zu einem Dienst wird ein BluetoothClient erzeugt. Über diesem Client kann ein NetworkStream geholt werden, mit welchem Daten über Bluetooth empfangen und versendet werden können.

```
BluetoothClient client;
System.Net.Sockets.NetworkStream ns;

while (true) {
    // accept bluetooth connection
    client = service.WaitForConnection();
    Console.WriteLine("Incoming connection");

    // transform into network stream
    ns = client.GetStream();

    // Output data to stream
    StreamWriter sw = new StreamWriter(ns);
    sw.WriteLine("Hello from Server!");

    // clear and close stream
    sw.Flush();
    client.Close();
}
```


5.5. Client Aktionen

Erreichbare Geräte Suchen

Damit man zu einem Bluetooth-Gerät verbinden kann müssen zuerst alle erreichbaren Geräte der Umgebung gesucht werden. Bereits beim Suchen nach erreichbaren Geräten mit „DiscoverDevices" können bestimmte Selektionskriterien wie Gerätename, gestartete Dienste und authentifizierte Geräte mitgegeben werden. Als Rückgabewert erhält man eine Liste aller gefundenen Bluetooth-Geräte.

```
// desired service
Guid service = BluetoothServiceList.Robot07;
// search for reachable devices
BluetoothDiscovery discovery = new BluetoothDiscovery();

// Search for maximum 12 devices
// Exclude paired devices
// Exclude remebered devices
// Include unknown devices
BluetoothDeviceCollection bdc =
    discovery.DiscoverDevices(12, false, false, true);

foreach (BluetoothDevice device in bdc) {
    if (device.HasService(service)) {
        Console.WriteLine(device.Name + ";" +device.DeviceAddress);
    }
}
```

Zu Gerät verbinden

Nachdem die erreichbaren Geräte ermittelt wurden, können sie verarbeitet werden. Jedes einzelne Gerät wird als BluetoothDevice repräsentiert. Diese Klasse bietet Informationen wie Name und Adresse des entfernten Bluetooth-Geräts. Über Hilfsfunktionen kann geprüft werden, ob auf diesem Gerät ein bestimmter Dienst gestartet wurde und bei Bedarf zu diesem Verbinden. Pro hergestellte Verbindung wird ein BluetoothClient erzeugt.

```
// test if desired service is started on this device
if (device.HasService(service)) {
    // connect to desired service
    BluetoothClient bc = device.Connect(service);

    // read transmitted data
    StreamReader sr = new StreamReader(bc.GetStream());

    // print
    Console.WriteLine("Message received : ");
    Console.WriteLine(sr.ReadLine());

    bc.Close();
}
else {
    Console.WriteLine("Required service does not exist on target");
}
```

Zu einem bekannten Gerät verbinden

Wenn das zu erreichende Gerät bekannt ist, kann man direkt damit verbinden. Das Gerät wird als BluetoothDevice repräsentiert. Diese Klasse bietet die Methode Connect mit den Argumenten MAC Adresse als String und den Service als Guid Pro hergestellte Verbindung wird ein BluetoothClient erzeugt.

Die Verbindungsaufnahme über die MAC Adresse ist um einige schneller, als die Suche über den Service.

```
// target
String mac = "001122334455";
// desired service
Guid service = BluetoothServiceList.Robot11;

// connect to target with desired service
Console.WriteLine("Connect to device "+mac+"...");
BluetoothClient bc = BluetoothDevice.Connect(mac, service);
if (bc != null)
{
   Console.WriteLine("Outgoing connection " +
      bc.GetSocket().LocalEndPoint + ".");

   // read transmitted data
   StreamReader sr = new StreamReader(bc.GetStream());

   // print
   Console.WriteLine("Message received : ");
   Console.WriteLine(sr.ReadLine());
   bc.Close();
}
else
{
   Console.WriteLine("Target does not exist");
}
```


Übung 19: Bluetooth Implementationen

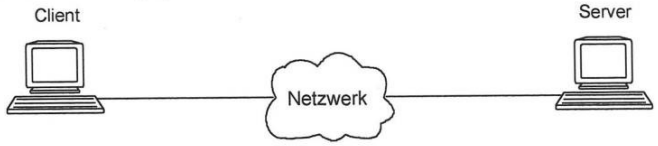
Erweiterung des `DayTime` Servers

Nehmen Sie den `DayTime` Server als Vorlage und erweitern diesen zu einem Bluetooth Server. Die Socket Funktionalität können Sie beibehalten oder auch weglassen. Testen Sie den Bluetooth `DayTime` Server können mit Hilfe eines selbst erstellten `DayTimeClients`. Der `DayTimeClient` kann eine einfache Konsolenapplikation sein oder auch eine kleine Windows-Anwendung.

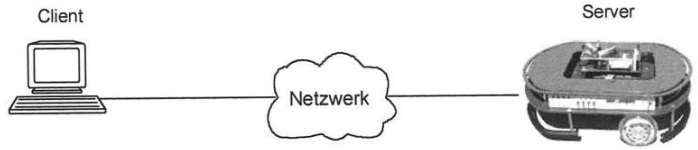
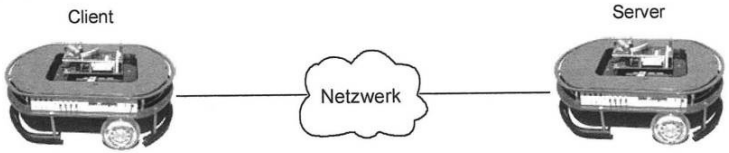
Test der Bluetooth `DayTime` Server und Client

Testen Sie den `DayTimeServer` mit dem `DayTimeClient` unter folgenden Bedingungen.

Verteilt auf verschiedenen Laptops



Verteilt auf Laptops und Roboter



Verteilt auf verschiedenen Robotern