

Name Stefan Kert

Points _____

Effort in hours 8h**1. Race Conditions****(3 + 1 + 3 Points)**

- a) What are *race conditions*? Implement a simple .NET application in C# that has a race condition. Document the race condition with appropriate test runs.
- b) What can be done to avoid race conditions? Improve your program from 1.a) so that the race condition is eliminated. Document your solution with some test runs again.
- c) Where is the race condition in the following code? How can the race condition be removed?

```
class RaceConditionExample {
    private const int N = 1000;
    private const int BUFFER_SIZE = 10;

    private double[] buffer;

    private AutoResetEvent signal;
    public void Run() {
        buffer = new double[BUFFER_SIZE];
        signal = new AutoResetEvent(false);

        // start threads
        var t1 = new Thread(Reader); var t2 = new Thread(Writer);
        t1.Start(); t2.Start();

        // wait
        t1.Join(); t2.Join();
    }

    void Reader() {
        var readerIndex = 0;
        for (int i = 0; i < N; i++) {
            signal.WaitOne();
            Console.WriteLine(buffer[readerIndex]);
            readerIndex = (readerIndex + 1) % BUFFER_SIZE;
        }
    }

    void Writer() {
        var writerIndex = 0;
        for (int i = 0; i < N; i++) {
            buffer[writerIndex] = (double)i;
            signal.Set();
            writerIndex = (writerIndex + 1) % BUFFER_SIZE;
        }
    }
}
```

2. Synchronization Primitives

(2 + 2 + 1 Points)

- a) The following code starts multiple threads to download multiple files in parallel. Change the code so that only maximally ten files are downloaded concurrently.

```
class LimitedConnectionsExample {
    public void DownloadFilesAsync(IEnumerable<string> urls) {
        foreach(var url in urls) {
            Thread t = new Thread(DownloadFile);
            t.Start(url);
        }
    }

    public void DownloadFile(object url) {
        // download and store file here
        // ...
    }
}
```

- b) Based on your version of the code in 2a) implement the synchronous method *DownloadFiles* that waits until all downloads are finished before returning.

- c) In the following code one thread waits for the result of another thread in a polling loop. Improve the code fragment to remove the polling.

```
class PollingExample {
    private const int MAX_RESULTS = 10;
    private volatile string[] results;
    private volatile int resultsFinished;
    private object resultsLocker = new object();

    public void Run() {
        results = new string[MAX_RESULTS];
        resultsFinished = 0;

        // start tasks
        for (int i = 0; i < MAX_RESULTS; i++) {
            var t = new Task((s) => {
                int _i = (int)s;
                string m = Magic(_i);
                results[_i] = m;
                lock(resultsLocker) {
                    resultsFinished++;
                }
            }, i);
            t.Start();
        }

        // wait for results
        while (resultsFinished < MAX_RESULTS) { Thread.Sleep(10); }

        // output results
        for (int i = 0; i < MAX_RESULTS; i++)
            Console.WriteLine(results[i]);
    }
}
```

3. Toilet Simulation

(4 + 4 + 4 Points)

Especially for simulation applications concurrent programming is very important, as real life is normally not sequential at all. So in order to simulate a realistic scenario as good as possible, parallel concepts are needed.

In this task you should implement a queue which handles jobs waiting to be processed (producer-consumer problem). In order to get the example a little bit more "naturalistic", imagine that the jobs are people waiting in front of a toilet (consumer).

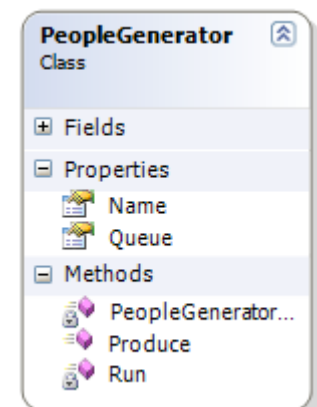
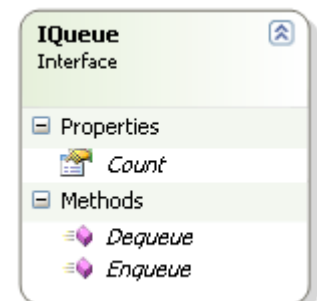
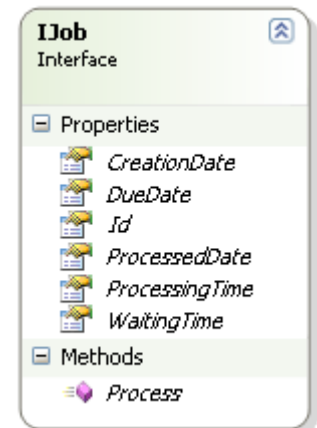
On Moodle you find a simple framework which already provides some parts of the simulation:

The interface *IJob* defines the data relevant for every job (id, creation date, due date, processing time, waiting time, time when the job was finally processed). It also has a method *Process* which is called by the consumer to process the job.

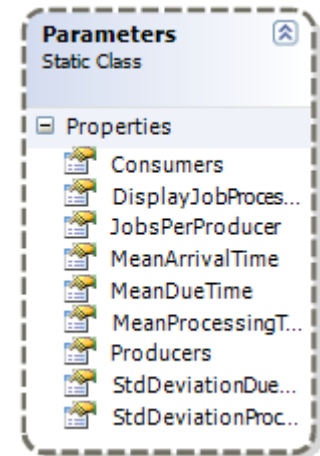
The class *Person* implements *IJob*. In the constructor of *Person* the time period available for processing is chosen randomly (normally distributed). Based on that time period the due date (*DueDate*) is set. Additionally the processing time (*ProcessingTime*) is also randomly set (normally distributed).

The interface *IQueue* defines the relevant methods for a queue which are used by the producer to enqueue jobs (*Enqueue*) and by the consumer to dequeue jobs (*Dequeue*).

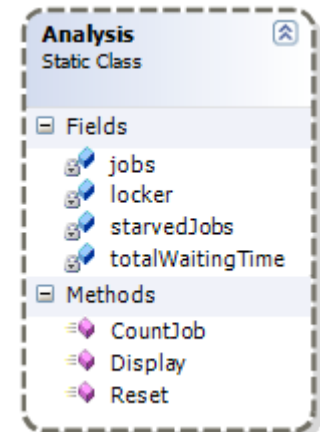
The producer *PeopleGenerator* uses a separate thread to create new jobs (instances of *Person*) and to enqueue them in the queue. The time between the creation of two *Person* objects is exponentially distributed (Poisson process).



The class *Parameters* contains all relevant parameters configuring the simulation. Especially, there is the number of producers and consumers, the number of jobs to generate per producer and the mean value and standard deviation of the arrival time, the due time and the processing time.



Analysis is used to analyze the job management in a queue. After a job is processed the job is counted by calling *CountJob*. The results of the analysis can be displayed with *Display* giving the total number of jobs, the number of "starved" jobs, the starvation ratio and the total and average waiting time.



The classes *NormalRandom* and *ExponentialRandom* are helper classes to create normally and exponentially distributed random variables.

ToiletSimulation contains the main method which is creating all required objects (producers, consumers, queue), starting the simulation and displaying the results.

- Implement a simple consumer *Toilet* which is dequeuing and processing jobs from the queue in an own thread. Especially think about when the consumer should terminate. How can the synchronization be done?
- Implement a first-in-first-out queue *FIFOQueue* and test it with the following parameter settings:

Producers	2
JobsPerProducer	200
Consumers	2
MeanArrivalTime	100
MeanDueTime	500
StdDeviationDueTime	150
MeanProcessingTime	100
StdDeviationProcessingTime	25

Execute some independent test runs and besides the individual results also document the mean value and the standard deviation.

- As you can see from 2.b), the performance of *FIFOQueue* is not that good. "Starvation" occurs quite regularly, in other words many jobs are not processed in time. And what that means according to our simulation scenario ... well you might know ;-).

Develop a better queue (*ToiletQueue*) which has a better performance according to the total number of starved jobs. Which strategy could be used to choose the next job from the queue that should be processed?

Repeat the test runs you have done in 2.b) for the improved queue and compare.

Note: Upload your report which contains all documentation and all changed or new source code of your program to Moodle.

Don't forget to give meaningful solution descriptions, so that one can easily get the main idea of your approach.

If necessary, you are allowed to extend or change the given classes. If you do so, please motivate and document such changes clearly in the solution description.

VPS5 - UE2

Stefan Kert

14. April 2016

1 Race Conditions

1.1 Was sind *Race Conditions*?

In Programmen kann es zu sogenannten Race conditions kommen, wenn Ergebnisse einer Operation von der zeitlichen Abfolge parallel ablaufender Threads abhängig sind und es zu einem unverhersebaren Ergebnis kommen kann. Ein Beispiel für eine solche Race Condition befindet sich in folgendem Listing:

```
public class RaceConditionExample
{
    private static readonly object LockObject = new object();
    static int result = 0;

    public static void IncreaseResult()
    {
        result++;
    }

    public static void Run(int numberOfIncrements, int threadCount)
    {
        var tasks = new Task[threadCount];
        var raceConditionCount = 0;
        result = 0;
        for (int i = 0; i < numberOfIncrements; i++)
        {
            for (int j = 0; j < threadCount; j++)
            {
                tasks[j] = new Task(() => IncreaseResult());
                tasks[j].Start();
            }

            Task.WaitAll(tasks);

            if (result != i * threadCount)
                raceConditionCount++;
        }

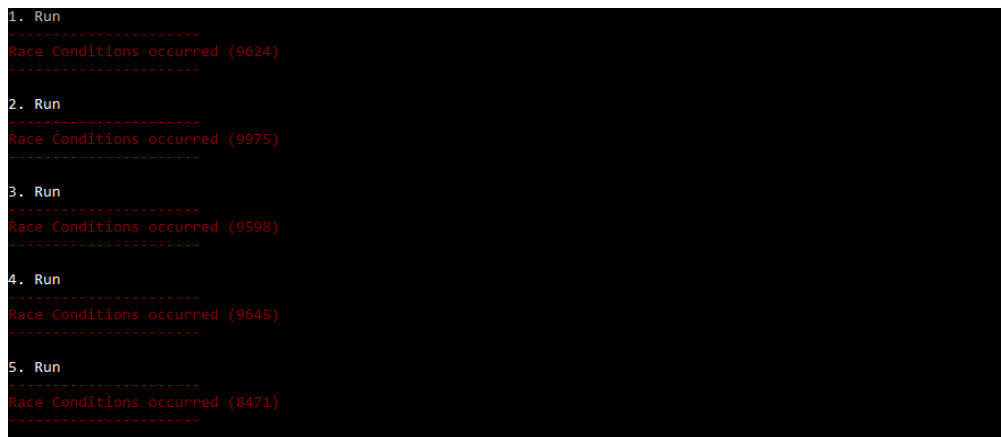
        .....
    }
}
```

```

    }
}

```

In der Methode *IncreaseResult()* wird die statische Variable *result* erhöht. Diese Methode werden schließlich mehrere Threads erzeugt, welche diese Methode gleichzeitig aufrufen. Hier kann es zu Race Conditions kommen, wodurch die Variable *result* falsch erhöht werden kann. Bei dem gegebenen Beispiel kommt es bei 5 Threads und 10000 Schleifendurchläufen zu folgendem Ergebnis:



```

1. Run
-----
Race Conditions occurred (9624)
-----

2. Run
-----
Race Conditions occurred (9975)
-----

3. Run
-----
Race Conditions occurred (9598)
-----

4. Run
-----
Race Conditions occurred (9645)
-----

5. Run
-----
Race Conditions occurred (8471)
-----

```

Abbildung 1: Ergebnis Race Conditions

1.2 Was kann getan werden um *Race Conditions* zu vermeiden?

Eine Möglichkeit Race Conditions zu vermeiden sind Locks. Im folgenden Beispiel wurden diese dazu verwendet um die Operation zu synchronisieren. Im folgenden Listing befindet sich eine Variante, durch welche die Race Conditions nicht mehr auftreten.

```

public class RaceConditionExample
{
    private static readonly object LockObject = new object();
    static int result = 0;

    public static void IncreaseResultWithLock()
    {
        lock (LockObject)
        {
            result++;
        }
    }

    public static void Run(int numberOfIncrements, int threadCount,
        Action method)
    {
        var tasks = new Task[threadCount];
        var raceConditionCount = 0;

```

```

        result = 0;
        for (int i = 0; i < numberOfIncrements; i++)
        {
            for (int j = 0; j < threadCount; j++)
            {
                tasks[j] = new Task(method);
                tasks[j].Start();
            }

            Task.WaitAll(tasks);

            if (result != i * threadCount)
                raceConditionCount++;
        }
        .....
    }
}

```

Durch das Verwenden von *lock* in der Methode *IncreaseResultWithLock* wird das inkrementieren synchronisiert wodurch das erwartete Ergebnis erreicht wird:

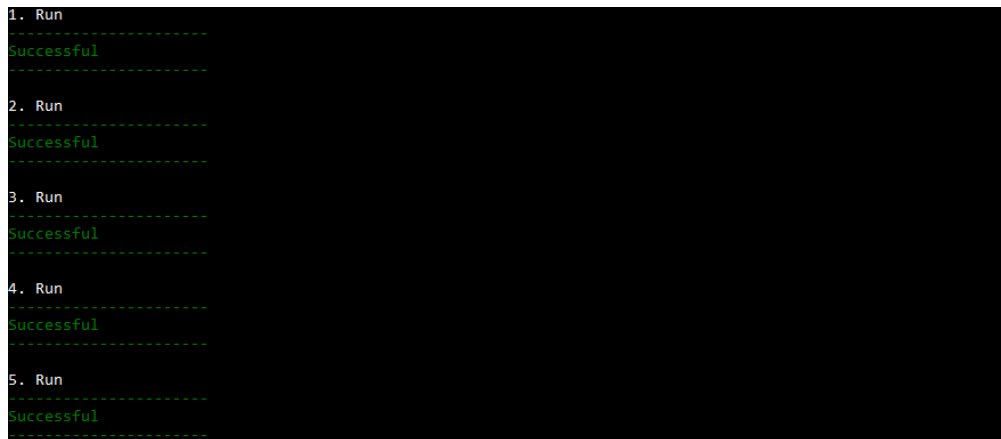


Abbildung 2: Ergebnis der behobenen Race Conditions

1.3 Wo befindet sich die Race Condition im folgenden Code und wie kann diese behoben werden?

Im folgenden Listing ist eine Klasse dargestellt die eine Race Condition enthält. Diese Race Condition zeigt sich in dem Ausmaß, dass der Writer Elemente überschreibt, welche sich im Buffer befinden, die vom Reader noch nicht gelesen wurden.

```

class RaceConditionExample
{
    private const int N = 1000;
    private const int BUFFER_SIZE = 10;
    private double[] buffer;

```



```

private AutoResetEvent signal;

public void Run()
{
    buffer = new double[BUFFER_SIZE];
    signal = new AutoResetEvent(false);
    // start threads
    var t1 = new Thread(Reader);
    var t2 = new Thread(Writer);
    t1.Start();
    t2.Start();
    // wait
    t1.Join();
    t2.Join();
}

void Reader()
{
    var readerIndex = 0;
    for (int i = 0; i < N; i++)
    {
        signal.WaitOne();
        Console.WriteLine(buffer[readerIndex]);
        readerIndex = (readerIndex + 1) % BUFFER_SIZE;
    }
}

void Writer()
{
    var writerIndex = 0;
    for (int i = 0; i < N; i++)
    {
        buffer[writerIndex] = (double)i;
        signal.Set();
        writerIndex = (writerIndex + 1) % BUFFER_SIZE;
    }
}
}

```

Gelöst werden kann diese Race Condition durch das Verwenden von *AutoResetEvents*. Im Writer wird auf das readerSignal gewartet, welches im Reader gesetzt wird. Danach wird das Element zum Buffer hinzugefügt und das Signal, auf welches im Reader gewartet wird, wieder freigegeben. Dies führt dazu, dass der reader das Element ausliest und wieder zum Anfang springt und die Synchronisierung von vorne beginnt.

```

class FixedRaceConditionExample
{
    private const int N = 1000;
    private const int BUFFER_SIZE = 10;
    private double[] buffer;
    private AutoResetEvent writerSignal;
    private AutoResetEvent readerSignal;

    public void Run()
    {
        buffer = new double[BUFFER_SIZE];
    }
}

```

```

        writerSignal = new AutoResetEvent(false);
        readerSignal = new AutoResetEvent(true);
        // start threads
        var t1 = new Thread(Reader);
        var t2 = new Thread(Writer);
        t1.Start();
        t2.Start();
        // wait
        t1.Join();
        t2.Join();
    }
    void Reader()
    {
        var readerIndex = 0;
        for (int i = 0; i < N; i++)
        {
            readerSignal.Set();
            writerSignal.WaitOne();
            Console.WriteLine(buffer[readerIndex]);
            readerIndex = (readerIndex + 1) % BUFFER_SIZE;
        }
    }
    void Writer()
    {
        var writerIndex = 0;
        for (int i = 0; i < N; i++)
        {
            readerSignal.WaitOne();
            buffer[writerIndex] = (double)i;
            writerIndex = (writerIndex + 1) % BUFFER_SIZE;
            writerSignal.Set();
        }
    }
}

```

2 Synchronization Primitives

2.1 Wie kann die Anzahl an erzeugten Threads begrenzt werden?

Mit der Klasse *SemaphoreSlim* kann die Anzahl an verwendeten Threads begrenzt werden. Folgendes Listing demonstriert, wie mit Hilfe dieser Klasse die Anzahl der Threads begrenzt werden kann. Dazu wird der Konstruktor mit dem gewünschten Limit (in dem Beispiel 10) instanziiert und in der in einem Thread ausgeführten Methode wird ein Signal aufgerufen, welches anzeigt, dass ein neuer Thread gestartet wird. Für jeden Aufruf von *syncSemaphore.Wait* wird in der Klasse *SemaphoreSlim* intern ein Counter erhöht und bei jedem Aufruf von *syncSemaphore.Release* wird dieser wieder verringert. Mit dieser Funktionalität kann die Anzahl an Aufrufen einer Methode reguliert werden und somit auch die Anzahl an Threads die verwendet werden.

```

public void DownloadFilesAsync(IEnumerable<string> urls)
{

```

```

        _syncSemaphore = new SemaphoreSlim(10, 10);
        _threads = new List<Thread>();
        foreach (var url in urls)
        {
            Thread t = new Thread(DownloadFile);
            _threads.Add(t);
            t.Start(url);
        }
    }

    public void DownloadFile(object url) {
        _syncSemaphore.Wait();
        Console.WriteLine($"Downloading {url}");
        Thread.Sleep(1000);
        Console.WriteLine($"finished {url}");
        _syncSemaphore.Release();
    }
}

```

2.2 Synchrone Implementierung der Methode DownloadFilesAsync

Im folgenden Listing wird die Methode *DownloadFiles* gezeigt, welche wartet bis alle Threads beendet sind. Dies geschieht mittels der Methode *Join*.

```

public void DownloadFiles(IEnumerable<string> urls)
{
    _threads = new List<Thread>();
    foreach (var url in urls)
    {
        Thread t = new Thread(DownloadFile);
        _threads.Add(t);
        t.Start(url);
    }
    foreach (var thread in _threads)
    {
        thread.Join();
    }
}

public void DownloadFile(object url) {
    _syncSemaphore.Wait();
    Console.WriteLine($"Downloading {url}");
    Thread.Sleep(1000);
    Console.WriteLine($"finished {url}");
    _syncSemaphore.Release();
}

```

2.3 Optimieren des Codes, der mittels Polling auf das fertigstellen der Threads wartet

Der folgende Codeabschnitt enthält eine Schleife, in der überprüft wird, ob alle Threads beendet wurden. Dass der Prozessor nicht zu stark belastet wird, wird in der Schleife immer ein paar Millisekunden gewartet und danach erneut überprüft.

```

while (resultsFinished < MAX_RESULTS) {
    Thread.Sleep(10);
}

```

Die in den neueren Version des .NET Frameworks vorhandene TPL bietet eine einfache Möglichkeit, wie ein solches Polling verhindert werden kann:

```
Task.WaitAll(tasks);
```

Mit der Methode *WaitAll* wird gewartet bis alle *Tasks* die übergeben wurden beendet sind. Wodurch nicht mehr gepollt werden muss.

3 Toilet Simulation

3.1 Implementieren eines einfachen Consumers Toilet

In der Übung wurde bereits der rudimentäre Consumer Toilet implementiert. Der Consumer ist fertig sobald alle Elemente abgearbeitet sind. Zu diesem Zweck wurde das Flag *IsCompleted* zum Interface *IQueue* hinzugefügt. Über die Methode *TryDequeue* kann ein Wert ausgelesen werden. Falls kein Wert enthalte ist blockiert diese Methode.

3.2 FifoQueue Implementierung

Bei der Implementierung der FifoQueue wird mit Hilfe einer Semaphore die Synchronisierung realisiert. Wenn ein Element hinzugefügt wird, wird die Semaphore erhöht, sobald ein Element wieder ausgelesen wird wird sie wieder verringert. Weiters wurden alle Zugriffe auf die Queue mittels einem lock synchronisiert.

```

using System;
using System.Linq;
using System.Threading;

namespace VSS.ToiletSimulation
{
    public abstract class Queue : IQueue
    {
        private int countOfCompletedProducers;
        protected IList<IJob> _queue;

        public int Count
        {
            get
            {
                lock (_queue)
                {
                    return _queue.Count;
                }
            }
        }

        protected Queue()
        {

```

```

        _queue = new List<IJob>();
    }

    public abstract void Enqueue(IJob job);

    public abstract bool TryDequeue(out IJob job);

    public virtual void CompleteAdding()
    {
        lock (_queue)
        {
            countOfCompletedProducers++;
            if (countOfCompletedProducers == Parameters.Producers)
                IsCompleted = true;
        }
    }

    public bool IsCompleted
    {
        get; private set;
    }
}

public class FIFOQueue : Queue
{
    private readonly SemaphoreSlim _syncSem;

    public FIFOQueue()
    {
        _syncSem = new SemaphoreSlim(0);
    }

    public override void Enqueue(IJob job)
    {
        if (IsCompleted)
            throw new InvalidOperationException("The queue already
                is completed.");

        lock (_queue)
        {
            _queue.Add(job);
        }
        _syncSem.Release();
    }

    public override bool TryDequeue(out IJob job)
    {
        job = null;
        if (IsCompleted)
            return false;
    }
}

```

```

        _syncSem.Wait();

        lock (_queue)
        {
            if (Count == 0)
                return false;

            job = DequeueNextJob();
            return true;
        }
    }

    protected virtual IJob DequeueNextJob()
    {
        lock (_queue)
        {
            var job = _queue.First();
            _queue.Remove(job);
            return job;
        }
    }
}

```

Die FIFOQueue wurde mit folgenden Parametern aufgerufen:

- Producers: 2
- JobsPerProducer: 200
- Consumers: 2
- MeanArrivalTime: 100
- MeanDueTime: 500
- StdDeviationDueTime: 150
- MeanProcessingTime: 100
- StdDeviationProcessingTime: 25

Damit ergibt sich folgendes Ergebnis:

3.3 ToiletQueue Implementierung

Bei der FifoQueue ist das größte Problem, dass die Elemente in einer ungünstigen Reihenfolge ausgelesen werden. Durch eine Sortierung und eine priorisierung der einzelnen Jobs kann ein besseres Ergebnis erzielt werden. Die Queue wird dazu nach dem *DueDate* sortiert um diejenigen Jobs zu erhalten, welche als nächstes fällig werden. Im Falle, dass ein Job nicht rechtzeitig abgearbeitet werden kann, sollte diese weniger hoch priorisiert

Run	Starved Jobs	Starvation Ratio	Total Wating Time	Mean Waiting Time
1	181	0,4525	0:2:46.99	0:0:0.41
2	201	0,5025	0:2:36.71	0:0:0.40
3	190	0,475	0:2:54.12	0:0:0.42
4	212	0,53	0:2:31.42	0:0:0.51
5	174	0,435	0:2:16.87	0:0:0.48
Schnitt	191,6	0,479	0:2:36.62	0:0:0.44
Std. Abw.	13,63231455	0,034080786	13,01	4,3

Abbildung 3: Ergebnis für FifoQueue

werden, da die Zeit für diese bereits abgelaufen ist. Mittels der Linq Methode *FirstOrDefault* kann das erste Element in der sortierten Queue ausgelesen werden, welches nach der aktuellen Zeit fällig ist. Wenn die Queue leer ist, werden die restlichen, bereits abgelaufenen Jobs abgearbeitet.

Implementiert wurde diese Logik in der überschriebenen Methode *DequeueNextJob* welche bereits in der FifoQueue zum Auslesen des nächsten Jobs verwendet wurde.

```
using System;
using System.Linq;
using System.Threading;

namespace VSS.ToiletSimulation
{
    public class ToiletQueue : FIFOQueue
    {
        protected override IJob DequeueNextJob()
        {
            lock (_queue)
            {
                IJob result = _queue.OrderBy(x =>
                    x.DueDate).FirstOrDefault(x => x.DueDate >
                    DateTime.Now) ?? _queue.First();
                _queue.Remove(result);
                return result;
            }
        }
    }
}
```

Durch diese Änderung stellt sich eine starke Performanceverbesserung, sowie eine starke Verbesserung des Verhältnisses von erfolgreichen und fehlgeschlagenen Jobs dar. Die Testläufe wurden mit folgenden Parametern durchgeführt:

- Producers: 2
- JobsPerProducer: 200

- Consumers: 2
- MeanArrivalTime: 100
- MeanDueTime: 500
- StdDeviationDueTime: 150
- MeanProcessingTime: 100
- StdDeviationProcessingTime: 25

Die folgende Tabelle listet die Ergebnisse der einzelnen Durchläufe auf und zeigt wie stark sich die Performanz und das Verhältnis verbessert hat.

Run	Starved Jobs	Starvation Ratio	Total Wating Time	Mean Waiting Time
1	16	0,04	0:0:56.81	0:0:0.14
2	47	0,1175	00:01:28.14	0:0:0.21
3	64	0,16	0:1:29.06	0:0:0.23
4	25	0,0625	0:0:59.81	0:0:0.17
5	30	0,075	0:1:10.41	0:0:0.19
Schnitt	36,4	0,091	00:01:12.2	0:0:0.18
Std. Abw.	17,09502852	0,042737571	13,73	3,1241

Abbildung 4: Ergebnis für ToiletQueue

3.4 Vergleich FIFOQueue und ToiletQueue

Ein Vergleich zeigt, dass die Zahl der durchschnittlich fehlgeschlagenen Jobs von 191 auf 36 reduziert wurde, was einer mehr als 5 fachen Verbesserung entspricht. Die durchschnittlich benötigte Gesamtzeit wurde um mehr als eine Minute verringert. Die Personen können durch die verbesserte Ordnung der Elemente schneller verarbeitet werden, was in der verbesserten Laufzeit resultiert. Im Großen und Ganzen lässt sich sagen, dass durch eine richtige Sortierung von Elementen eine starke Verbesserung der Performanz erreicht werden kann.