

Time Period Library für .NET

Umfangreiche Zeitraumberechnungen und individuelle Kalenderperioden

Einführung

Für die Realisierung einer Software hatte ich verschiedene Anforderungen zur Berechnung von Zeiträumen. Diese Zeitberechnungen waren ein zentraler Bestandteil der Lösung und stellten daher hohe Anforderungen an korrekte Berechnungsergebnisse.

Die geforderte Funktionalität deckte folgende Bereiche ab:

- Behandlung von individuellen Zeiträumen
- Arbeiten mit Kalenderperioden im Kalenderjahr
- Arbeiten mit Kalenderperioden abweichend vom Kalenderjahr (Finanz- oder Schulperioden)

Die Zeitberechnungen sollten sowohl in Serverkomponenten (Web-Services und Tasks) als auch im Rich Client (Silverlight) zur Verfügung stehen.

Nach der Situationsanalyse musste ich feststellen, dass weder Komponenten im .NET Framework (was man auch nicht erwarten konnte), noch sonstige Tools den Anforderungen genügten. Da ähnliche Anforderungen schon früher aufgetreten waren, hatte ich mich dazu entschlossen, eine generische Bibliothek zu entwickeln.

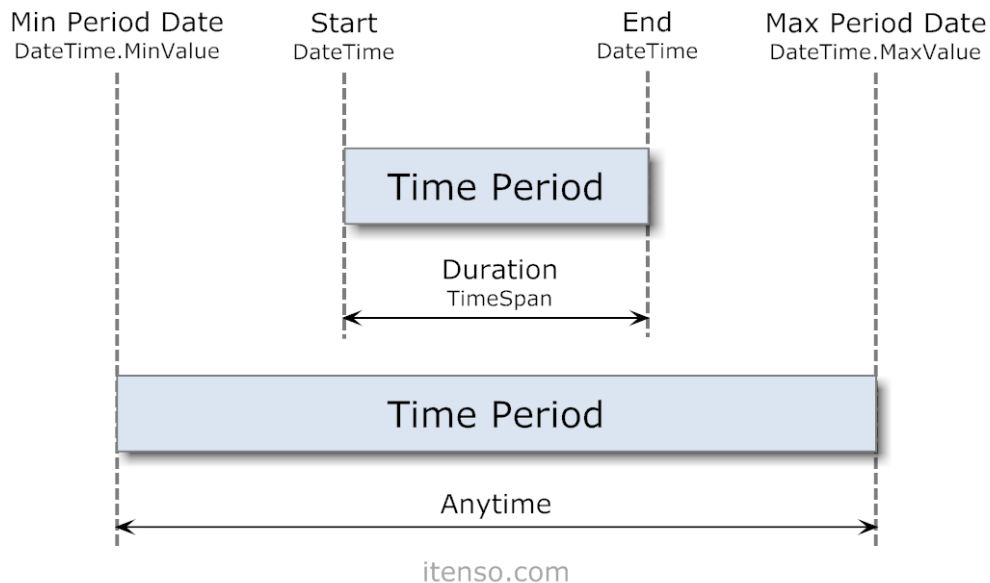
Nach einigen Iterationen resultierte die vorliegende Bibliothek **Time Period**, welche für folgende .NET Laufzeitumgebungen verfügbar ist:

- .NET Framework ab Version 2
- .NET Framework für Silverlight ab Version 4
- .NET Framework für Windows Phone ab Version 7

Zur Visualisierung einiger Bibliotheksfunktionen, habe ich unter <http://www.cpc.itenso.com/> die Silverlight Applikation **Calendar Period Collector** aufgeschaltet, welche die Suche nach Kalenderperioden demonstriert.

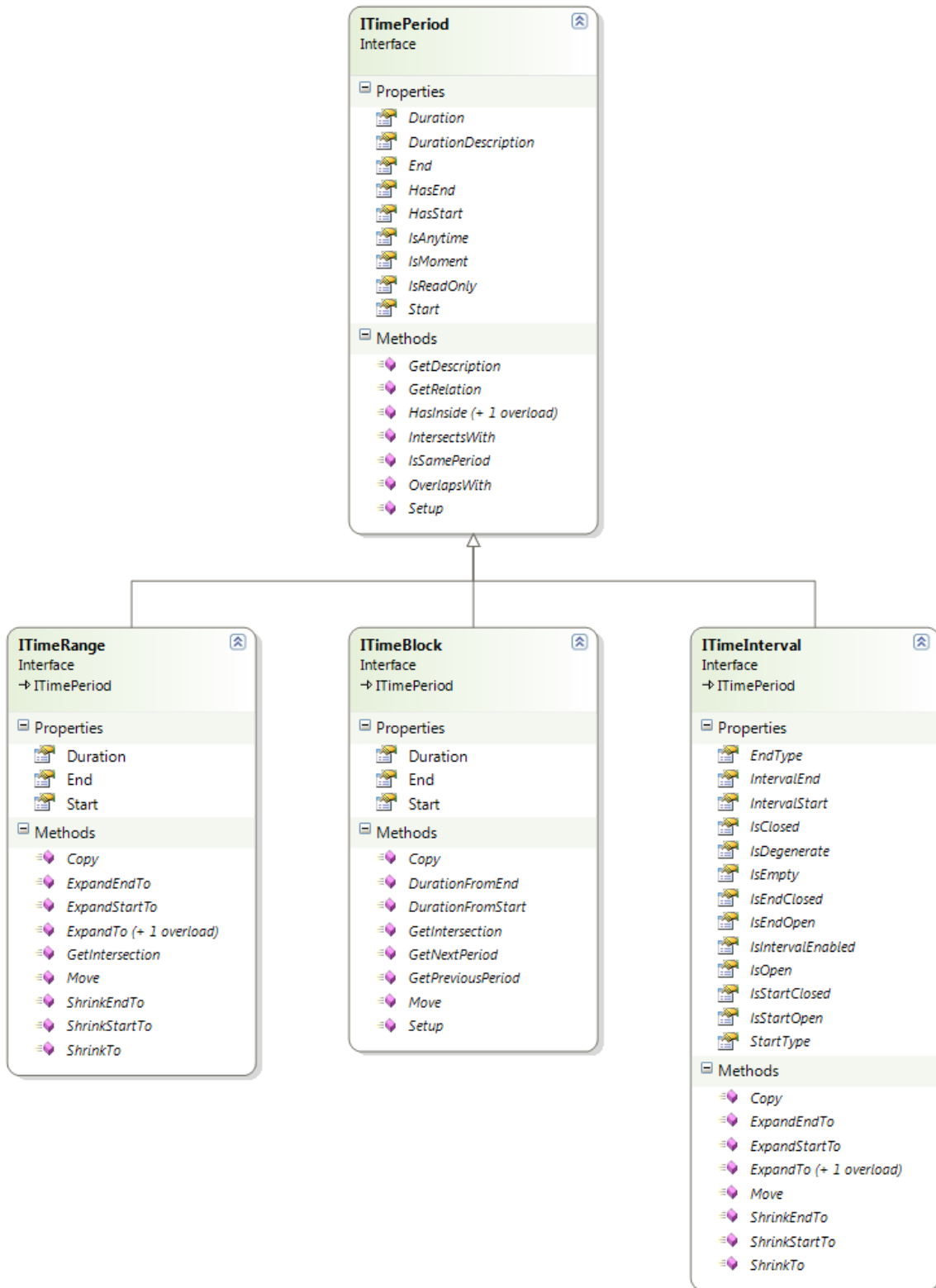
Time Periods

Mit den Klassen `DateTime` und `TimeSpan` bietet das .NET Framework Basisobjekte für Zeitberechnungen. Die Bibliothek **Time Period** erweitert das .NET Framework mit verschiedenen Werkzeugen zur Behandlung von Zeiträumen. Der Zeitraum beschreibt sich durch Start, Dauer und Ende:



Per Definition liegt der Start zeitlich immer vor dem Ende. Der Start gilt als undefiniert, wenn er den Minimalwert (`DateTime.MinValue`) besitzt. Analog dazu ist das Ende undefiniert, wenn er den Maximalwert (`DateTime.MaxValue`) besitzt.

Die Implementierung der Zeiträume basiert auf dem Interface `ITimePeriod` und beinhaltet die Spezialisierungen `ITimeRange`, `ITimeBlock` und `ITimeInterval`:



Das Interface `ITimePeriod` bietet Informationen und Operationen zu einem Zeitraum an, ohne festzulegen auf welche Weise die Eckdaten des Zeitraums berechnet werden:

- `Start`, `End` und `Duration` des Zeitraums
- `HasStart` ist True, wenn der Start definiert ist
- `HasEnd` ist True, wenn das Ende definiert ist
- `IsAnytime` ist True, wenn weder `Start` noch `End` definiert sind

- **IsMoment** ist True, wenn **Start** und **End** identisch sind
- **IsReadOnly** ist True für nicht veränderbare Zeiträume (Verwendung siehe weiter unten)

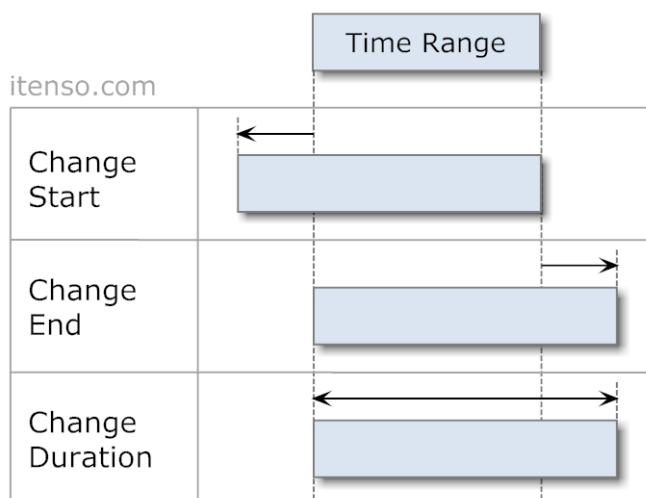
Das Zusammenspiel zweier Zeiträume wird durch die Aufzählung **PeriodRelation** beschrieben:

Period Relations				itenso.com			
				Is Same Period	Has Inside	Overlaps With	Intersects With
After							
Start Touching							✓
Start Inside						✓	✓
Inside Start Touching						✓	✓
Enclosing Start Touching					✓	✓	✓
Enclosing					✓	✓	✓
Enclosing End Touching					✓	✓	✓
Exact Match				✓	✓	✓	✓
Inside						✓	✓
Inside End Touching						✓	✓
End Inside						✓	✓
End Touching							✓
Before							

Funktionen wie **IsSamePeriod**, **HasInside**, **OverlapsWith** oder **IntersectsWith** stellen spezialisierte Varianten von einem Zeitraumverhältnis dar.

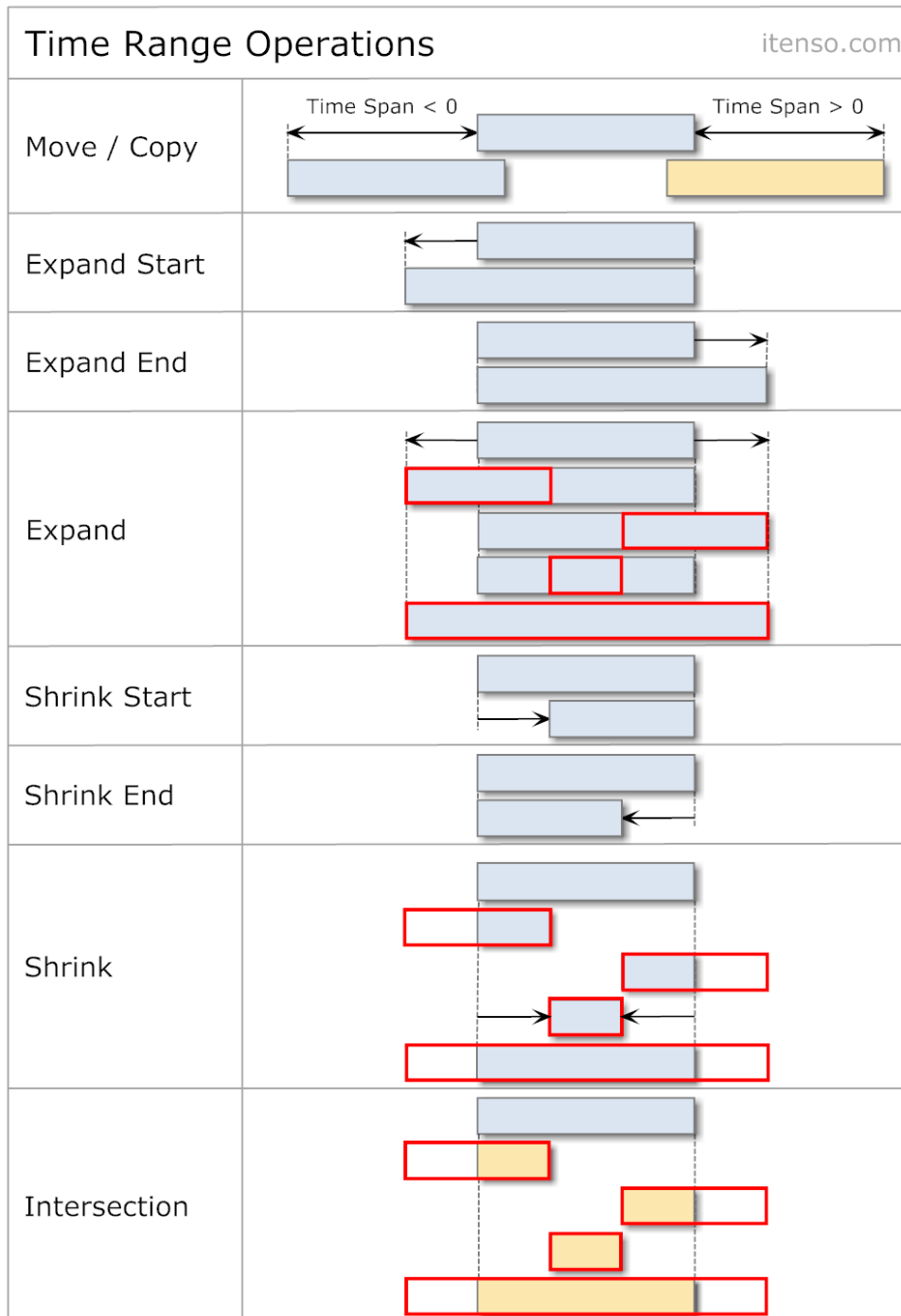
Time Range

TimeRange als Implementierung von **ITimeRange** bestimmt den Zeitraum durch **Start** und **Ende**; die Dauer wird berechnet:



Eine **TimeRange** kann mit **Start/End**, **Start/Duration** oder **Duration/End** erstellt werden. Falls erforderlich, werden **Start** und **Ende** zeitlich geordnet.

Zur Bearbeitung eines Zeitraumes stehen verschiedene Operationen zur Verfügung (Orange = neue Instanz):



Das folgende Beispiel zeigt die Verwendung von **TimeRange**:

```
// -----
public void TimeRangeSample()
{
    // --- time range 1 ---
    TimeRange timeRange1 = new TimeRange(
        new DateTime( 2011, 2, 22, 14, 0, 0 ),
        new DateTime( 2011, 2, 22, 18, 0, 0 ) );
    Console.WriteLine( "TimeRange1: " + timeRange1 );
    // > TimeRange1: 22.02.2011 14:00:00 - 18:00:00 | 04:00:00

    // --- time range 2 ---
    TimeRange timeRange2 = new TimeRange(
        new DateTime( 2011, 2, 22, 15, 0, 0 ),
        new TimeSpan( 2, 0, 0 ) );
    Console.WriteLine( "TimeRange2: " + timeRange2 );
    // > TimeRange2: 22.02.2011 15:00:00 - 17:00:00 | 02:00:00

    // --- time range 3 ---
    TimeRange timeRange3 = new TimeRange(
```

```

    new DateTime( 2011, 2, 22, 16, 0, 0 ),
    new DateTime( 2011, 2, 22, 21, 0, 0 ) );
Console.WriteLine( "TimeRange3: " + timeRange3 );
// > TimeRange3: 22.02.2011 16:00:00 - 21:00:00 | 05:00:00

// --- relation ---
Console.WriteLine( "TimeRange1.GetRelation( TimeRange2 ): " + timeRange1.GetRelation( timeRange2 ) );
// > TimeRange1.GetRelation( TimeRange2 ): Enclosing
Console.WriteLine( "TimeRange1.GetRelation( TimeRange3 ): " + timeRange1.GetRelation( timeRange3 ) );
// > TimeRange1.GetRelation( TimeRange3 ): EndInside
Console.WriteLine( "TimeRange3.GetRelation( TimeRange2 ): " + timeRange3.GetRelation( timeRange2 ) );
// > TimeRange3.GetRelation( TimeRange2 ): StartInside

// --- intersection ---
Console.WriteLine( "TimeRange1.GetIntersection( TimeRange2 ): " + timeRange1.GetIntersection( timeRange2 ) );
// > TimeRange1.GetIntersection( TimeRange2 ): 22.02.2011 15:00:00 - 17:00:00 | 02:00:00
Console.WriteLine( "TimeRange1.GetIntersection( TimeRange3 ): " + timeRange1.GetIntersection( timeRange3 ) );
// > TimeRange1.GetIntersection( TimeRange3 ): 22.02.2011 16:00:00 - 18:00:00 | 02:00:00
Console.WriteLine( "TimeRange3.GetIntersection( TimeRange2 ): " + timeRange3.GetIntersection( timeRange2 ) );
// > TimeRange3.GetIntersection( TimeRange2 ): 22.02.2011 16:00:00 - 17:00:00 | 01:00:00
} // TimeRangeSample

```

Das folgende Beispiel überprüft ob sich eine Reservation innerhalb der Arbeitszeit eines Tages befindet:

```

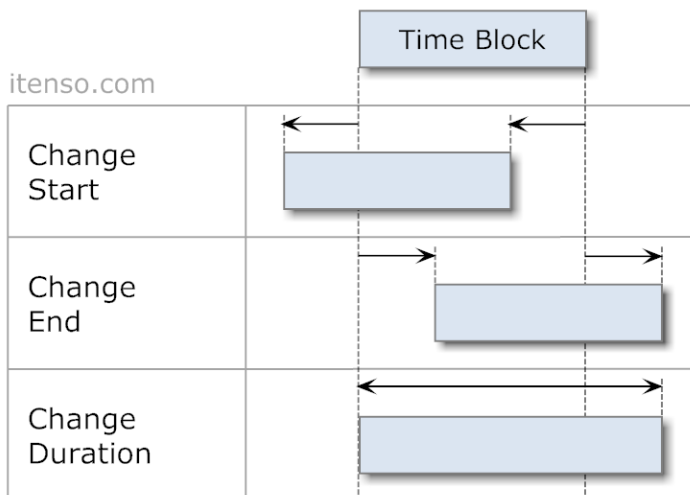
// -----
public bool IsValidReservation( DateTime start, DateTime end )
{
    if ( !TimeCompare.IsSameDay( start, end ) )
    {
        return false; // multiple day reservation
    }

    TimeRange workingHours = new TimeRange( TimeTrim.Hour( start, 8 ), TimeTrim.Hour( start, 18 ) );
    return workingHours.HasInside( new TimeRange( start, end ) );
} // IsValidReservation

```

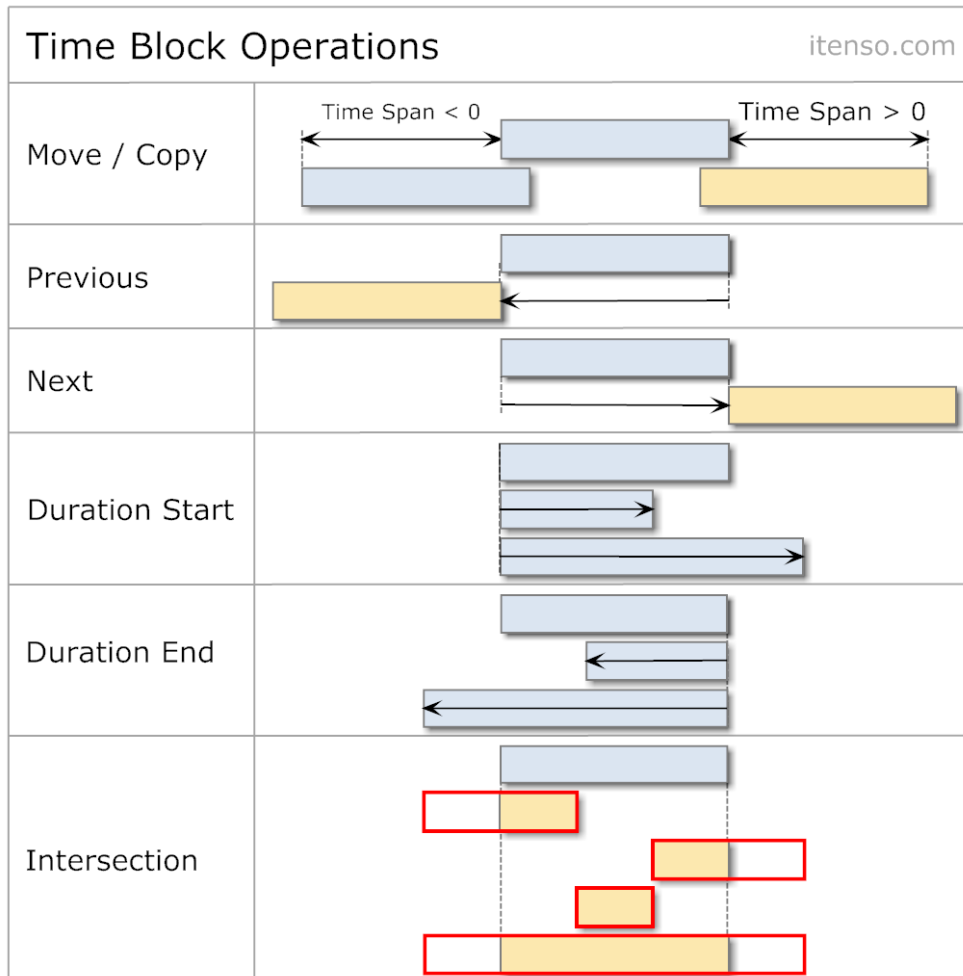
Time Block

TimeBlock implementiert das Interface **ITimeBlock** und bestimmt die Zeitperiode durch **Start** und **Dauer**; das **Ende** wird berechnet:



Wie **TimeRange** kann auch **TimeBlock** mit **Start/End**, **Start/Duration** oder **Duration/End** erstellt werden. Auch hier werden **Start** und **Ende** automatisch sortiert.

Zur Bearbeitung eines Zeitblocks stehen verschiedene Operationen zur Verfügung (Orange = neue Instanz):



Das folgende Beispiel zeigt die Verwendung von **TimeBlock**:

```
// -----
public void TimeBlockSample()
{
    // --- time block ---
    TimeBlock timeBlock = new TimeBlock(
        new DateTime( 2011, 2, 22, 11, 0, 0 ),
        new TimeSpan( 2, 0, 0 ) );
    Console.WriteLine( "TimeBlock: " + timeBlock );
    // > TimeBlock: 22.02.2011 11:00:00 - 13:00:00 | 02:00:00

    // --- modification ---
    timeBlock.Start = new DateTime( 2011, 2, 22, 15, 0, 0 );
    Console.WriteLine( "TimeBlock.Start: " + timeBlock );
    // > TimeBlock.Start: 22.02.2011 15:00:00 - 17:00:00 | 02:00:00
    timeBlock.Move( new TimeSpan( 1, 0, 0 ) );
    Console.WriteLine( "TimeBlock.Move(1 hour): " + timeBlock );
    // > TimeBlock.Move(1 hour): 22.02.2011 16:00:00 - 18:00:00 | 02:00:00




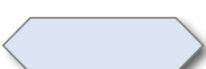
    // --- previous/next ---
    Console.WriteLine( "TimeBlock.GetPreviousPeriod(): " + timeBlock.GetPreviousPeriod() );
    // > TimeBlock.GetPreviousPeriod(): 22.02.2011 14:00:00 - 16:00:00 | 02:00:00
    Console.WriteLine( "TimeBlock.GetNextPeriod(): " + timeBlock.GetNextPeriod() );
    // > TimeBlock.GetNextPeriod(): 22.02.2011 18:00:00 - 20:00:00 | 02:00:00
    Console.WriteLine( "TimeBlock.GetNextPeriod(+1 hour): " + timeBlock.GetNextPeriod( new TimeSpan( 1, 0, 0 ) ) );
    // > TimeBlock.GetNextPeriod(+1 hour): 22.02.2011 19:00:00 - 21:00:00 | 02:00:00
    Console.WriteLine( "TimeBlock.GetNextPeriod(-1 hour): " + timeBlock.GetNextPeriod( new TimeSpan( -1, 0, 0 ) ) );
    // > TimeBlock.GetNextPeriod(-1 hour): 22.02.2011 17:00:00 - 19:00:00 | 02:00:00
} // TimeBlockSample
```

Time Interval

`ITimeInterval` bestimmt den Zeitraum wie `ITimeRange` durch `Start` und `Ende`. Zusätzlich kann mit der Aufzählung `IntervalEdge` die Interpretation von `Start` und `Ende` gesteuert werden:

- **Closed**: Der Eckpunkt wird bei Berechnungen einbezogen. Dies entspricht dem Verhalten von `ITimeRange`
- **Open**: Der Eckpunkt repräsentiert einen Grenzwert welcher in Berechnungen ausgeschlossen wird

Die möglichen Intervallvarianten sehen wie folgt aus:

Time Interval	itenso.com	Is Start Closed	Is End Closed	Is Closed	Is Start Open	Is End Open	Is Open
Start = Closed End = Closed		✓	✓	✓			
Start = Closed End = Open		✓				✓	
Start = Open End = Closed			✓		✓		
Start = Open End = Open					✓	✓	✓

Im Normalfall haben Zeitraumpunkte den Wert `IntervalEdge.Closed` was dazu führt, dass bei angrenzenden Zeiträumen ein Schnittpunkt existiert. Sobald einer der angrenzenden Punkte den Wert `IntervalEdge.Open` besitzt, ist kein Schnittpunkt vorhanden:

```
// -----
public void TimeIntervalSample()
{
    // --- time interval 1 ---
    TimeInterval timeInterval1 = new TimeInterval(
        new DateTime( 2011, 5, 8 ),
        new DateTime( 2011, 5, 9 ) );
    Console.WriteLine( "TimeInterval1: " + timeInterval1 );
    // > TimeInterval: [08.05.2011 - 09.05.2011] | 1.00:00

    // --- time interval 2 ---
    TimeInterval timeInterval2 = new TimeInterval(
        timeInterval1.End,
        timeInterval1.End.AddDays( 1 ) );
    Console.WriteLine( "TimeInterval2: " + timeInterval2 );
    // > TimeInterval2: [09.05.2011 - 10.05.2011] | 1.00:00

    // --- relation ---
    Console.WriteLine( "Relation: " + timeInterval1.GetRelation( timeInterval2 ) );
    // > Relation: EndTouching
    Console.WriteLine( "Intersection: " + timeInterval1.GetIntersection( timeInterval2 ) );
    // > Intersection: [09.05.2011]

    timeInterval1.EndEdge = IntervalEdge.Open;
    Console.WriteLine( "TimeInterval: " + timeInterval1 );
    // > TimeInterval: [08.05.2011 - 09.05.2011] | 1.00:00

    timeInterval2.StartEdge = IntervalEdge.Open;
    Console.WriteLine( "TimeInterval2: " + timeInterval2 );
    // > TimeInterval2: (09.05.2011 - 10.05.2011] | 1.00:00

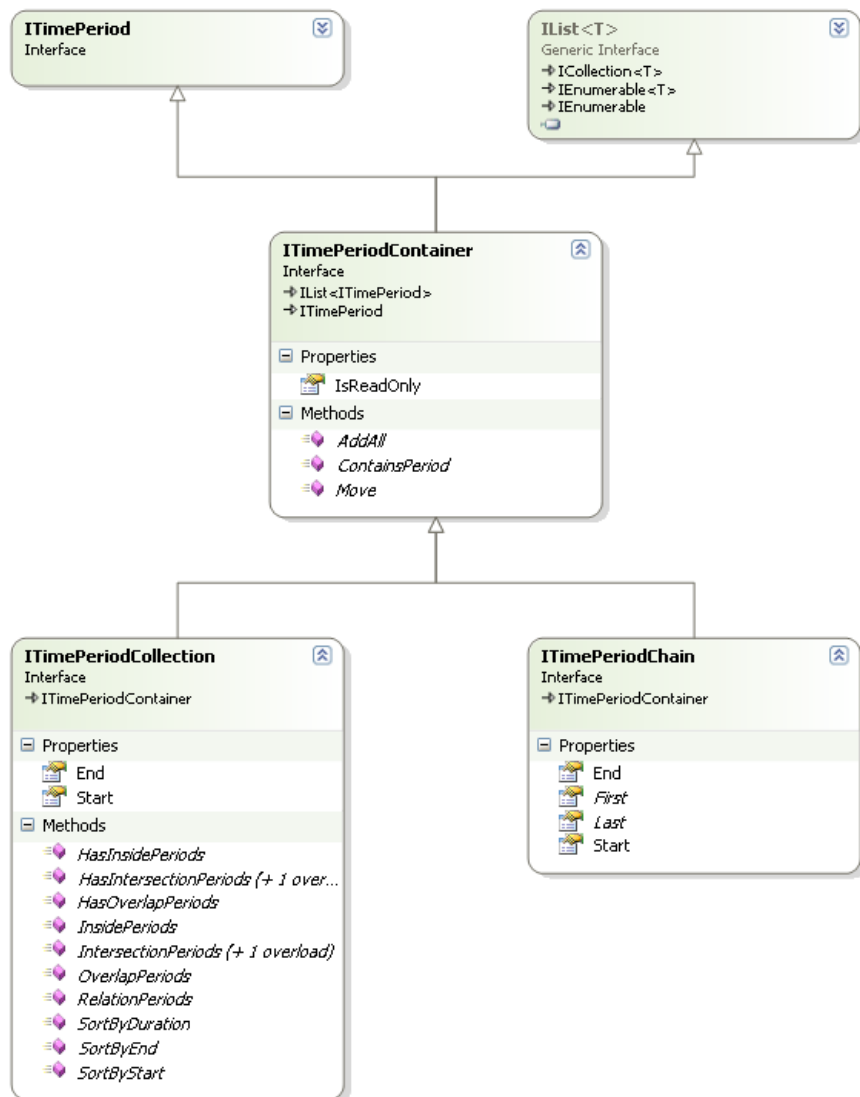
    // --- relation ---
    Console.WriteLine( "Relation: " + timeInterval1.GetRelation( timeInterval2 ) );
    // > Relation: Before
    Console.WriteLine( "Intersection: " + timeInterval1.GetIntersection( timeInterval2 ) );
    // > Intersection:
} // TimeIntervalSample
```

Für gewisse Szenarien, wie die Suche nach Zeitraumlücken, kann der Ausschluss der Periodenenden zu unerwünschten Ergebnissen führen. Für solche Situationen kann mit dem Property `IsIntervalEnabled`, der Ausschluss deaktiviert werden.

Unbeschränkte Zeitintervalle können mittels `TimeSpec.MinPeriodDate` für den `Start`, beziehungsweise `TimeSpec.MaxPeriodDate` für das `Ende` erzeugt werden.

Time Period Container

In der Praxis sind bei Zeitberechnungen oft mehrere Zeiträume involviert, welche sinnvollerweise in Container zusammengefasst werden. Die **Time Period** Bibliothek bietet folgende Container für Zeiträume:

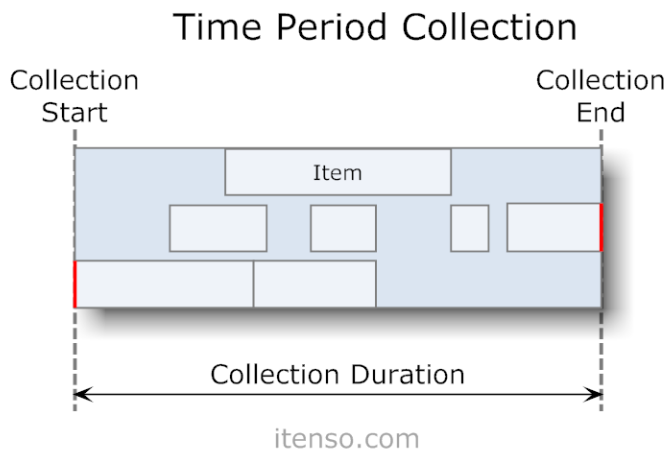


Alle Container basieren auf dem Interface **ITimePeriod**, so dass der Container selbst einen Zeitraum darstellt. Dadurch kann er in Berechnungen mit anderen Zeitperioden wie z.B. mit **ITimeRange** kombiniert werden.

Das Interface **ITimePeriodContainer** dient als Grundlage für Container und gewährleistet mittels **IList<ITimePeriod>** die Listenfunktionalität.

Time Period Collection

Die `ITimePeriodCollection` kann ein beliebiges Element vom Typ `ITimePeriod` beinhalten und interpretiert den frühesten Start seiner Elemente als Startzeitpunkt des Collection-Zeitraums. Entsprechend wird das letzte Ende des letzten Elementes als Endzeitpunkt verwendet:



Die Time Period Collection bietet folgende Operationen an:

Time Period Collection Operations itenso.com	
Inside Periods	
Intersection Periods by Moment	
Intersection Periods by Period	
Overlap Periods	

Das folgende Beispiel zeigt die Verwendung der Klasse `TimePeriodCollection`, welche das Interface `ITimePeriodCollection` implementiert:

```
// -----
public void TimePeriodCollectionSample()
{
    TimePeriodCollection timePeriods = new TimePeriodCollection();

    DateTime testDay = new DateTime( 2010, 7, 23 );

    // --- items ---
    timePeriods.Add( new TimeRange( TimeTrim.Hour( testDay, 8 ), TimeTrim.Hour( testDay, 11 ) ) );
    timePeriods.Add( new TimeBlock( TimeTrim.Hour( testDay, 10 ), Duration.Hours( 3 ) ) );
    timePeriods.Add( new TimeRange( TimeTrim.Hour( testDay, 16, 15 ), TimeTrim.Hour( testDay, 18, 45 ) ) );
    timePeriods.Add( new TimeRange( TimeTrim.Hour( testDay, 14 ), TimeTrim.Hour( testDay, 15, 30 ) ) );
    Console.WriteLine( "TimePeriodCollection: " + timePeriods );
    // > TimePeriodCollection: Count = 4; 23.07.2010 08:00:00 - 18:45:00 | 0.10:45
    Console.WriteLine( "TimePeriodCollection.Items" );
    foreach ( ITimePeriod timePeriod in timePeriods )

```

```

{
    Console.WriteLine( "Item: " + timePeriod );
}
// > Item: 23.07.2010 08:00:00 - 11:00:00 | 03:00:00
// > Item: 23.07.2010 10:00:00 - 13:00:00 | 03:00:00
// > Item: 23.07.2010 16:15:00 - 18:45:00 | 02:30:00
// > Item: 23.07.2010 14:00:00 - 15:30:00 | 01:30:00

// --- intersection by moment ---
DateTime intersectionMoment = new DateTime( 2010, 7, 23, 10, 30, 0 );
ITimePeriodCollection momentIntersections = timePeriods.IntersectionPeriods( intersectionMoment );
Console.WriteLine( "TimePeriodCollection.IntesectionPeriods of " + intersectionMoment );
// > TimePeriodCollection.IntesectionPeriods of 23.07.2010 10:30:00
foreach ( ITimePeriod momentIntersection in momentIntersections )
{
    Console.WriteLine( "Intersection: " + momentIntersection );
}
// > Intersection: 23.07.2010 08:00:00 - 11:00:00 | 03:00:00
// > Intersection: 23.07.2010 10:00:00 - 13:00:00 | 03:00:00

// --- intersection by period ---
TimeRange intersectionPeriod = new TimeRange( TimeTrim.Hour( testDay, 9 ), TimeTrim.Hour( testDay, 14, 30 ) );
ITimePeriodCollection periodIntersections = timePeriods.IntersectionPeriods( intersectionPeriod );
Console.WriteLine( "TimePeriodCollection.IntesectionPeriods of " + intersectionPeriod );
// > TimePeriodCollection.IntesectionPeriods of 23.07.2010 09:00:00 - 14:30:00 | 0.05:30
foreach ( ITimePeriod periodIntersection in periodIntersections )
{
    Console.WriteLine( "Intersection: " + periodIntersection );
}
// > Intersection: 23.07.2010 08:00:00 - 11:00:00 | 03:00:00
// > Intersection: 23.07.2010 10:00:00 - 13:00:00 | 03:00:00
// > Intersection: 23.07.2010 14:00:00 - 15:30:00 | 01:30:00
} // TimePeriodCollectionSample

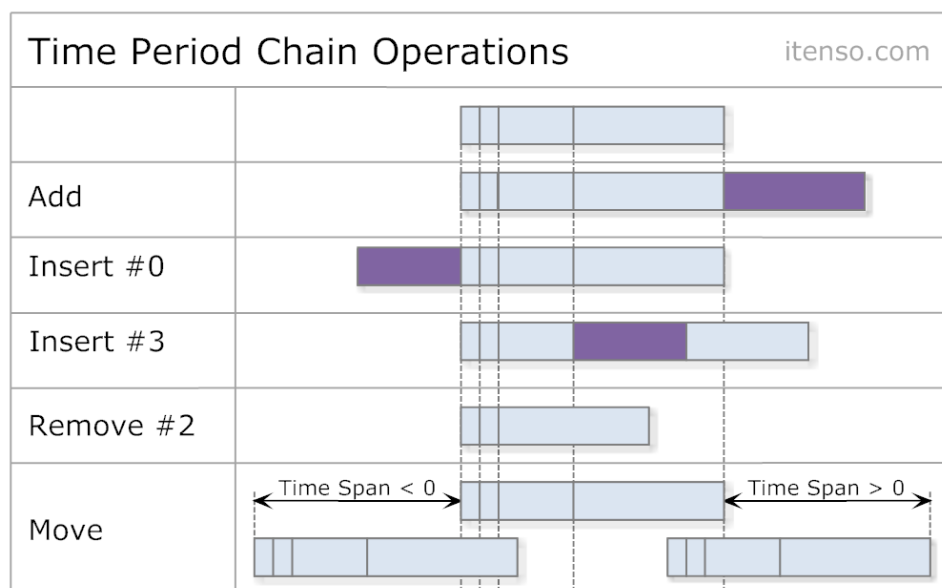
```

Time Period Chain

ITimePeriodChain verkettet mehrere Zeitperioden vom Typ **ITimePeriod** und gewährleistet, dass zwischen angrenzenden Zeitperioden keine Zeitlücke vorhanden ist.



Da `ITimePeriodChain` die Position der Elemente verändert, können keine Read-Only Zeitperioden eingefügt werden. Dies führt zu einer `NotSupportedException`. `ITimePeriodChain` bietet folgende Funktionen:



Das folgende Beispiel zeigt die Verwendung der Klasse `TimePeriodChain`, welche das Interface `ITimePeriodChain` implementiert:

```
// -----
public void TimePeriodChainSample()
{
    TimePeriodChain timePeriods = new TimePeriodChain();

    DateTime now = ClockProxy.Clock.Now;
    DateTime testDay = new DateTime( 2010, 7, 23 );

    // --- add ---
    timePeriods.Add( new TimeBlock( TimeTrim.Hour( testDay, 8 ), Duration.Hours( 2 ) ) );
    timePeriods.Add( new TimeBlock( now, Duration.Hours( 1, 30 ) ) );
    timePeriods.Add( new TimeBlock( now, Duration.Hour ) );
    Console.WriteLine( "TimePeriodChain.Add(): " + timePeriods );
    // > TimePeriodChain.Add(): Count = 3; 23.07.2010 08:00:00 - 12:30:00 | 0.04:30
    foreach ( ITimePeriod timePeriod in timePeriods )
    {
        Console.WriteLine( "Item: " + timePeriod );
    }
    // > Item: 23.07.2010 08:00:00 - 10:00:00 | 02:00:00
    // > Item: 23.07.2010 10:00:00 - 11:30:00 | 01:30:00
    // > Item: 23.07.2010 11:30:00 - 12:30:00 | 01:00:00

    // --- insert ---
    timePeriods.Insert( 2, new TimeBlock( now, Duration.Minutes( 45 ) ) );
    Console.WriteLine( "TimePeriodChain.Insert(): " + timePeriods );
    // > TimePeriodChain.Insert(): Count = 4; 23.07.2010 08:00:00 - 13:15:00 | 0.05:15
    foreach ( ITimePeriod timePeriod in timePeriods )
    {
        Console.WriteLine( "Item: " + timePeriod );
    }
    // > Item: 23.07.2010 08:00:00 - 10:00:00 | 02:00:00
    // > Item: 23.07.2010 10:00:00 - 11:30:00 | 01:30:00
    // > Item: 23.07.2010 11:30:00 - 12:15:00 | 00:45:00
    // > Item: 23.07.2010 12:15:00 - 13:15:00 | 01:00:00
} // TimePeriodChainSample
```

Calendar Time Periods

Bei Berechnungen mit Kalenderperioden kommt der Faktor zum Tragen, dass das Ende einer Zeitperiode nicht gleich ist, wie der Anfang der folgenden Periode. Das folgende Beispiel zeigt die Werte für die Stunden zwischen 13 und 15 Uhr:

- 13:00:00.0000000 – 13:59:59.9999999
- 14:00:00.0000000 – 14:59:59.9999999

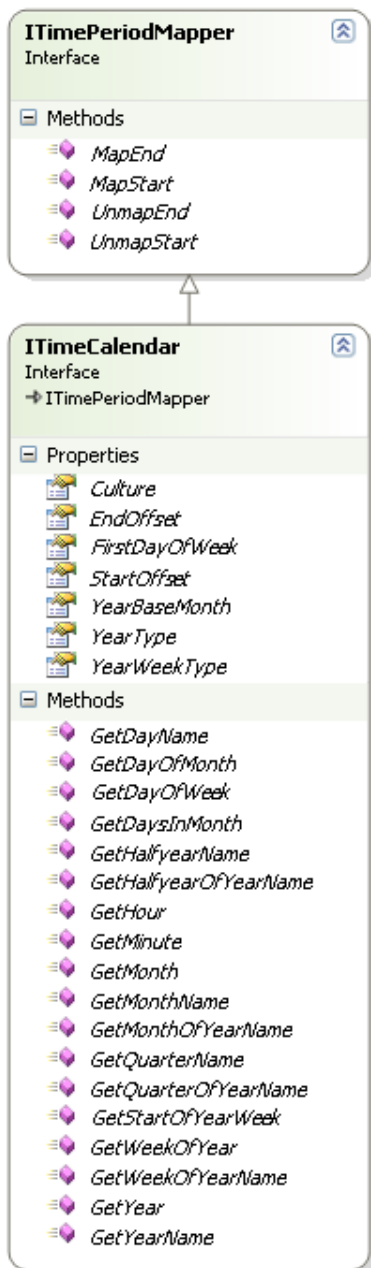
Das Ende ist um einen Moment vorher als der nächste Start, was im Minimum 1 Tick = 100 Nanosekunden ist. Dies ist bei Zeitraumberechnung mit Kalenderelementen ein wichtiger Faktor und muss bei Zeitraumberechnungen berücksichtigt werden.

In der **Time Period** Bibliothek existiert das Interface **ITimePeriodMapper**, welches die Zeitpunkte eines Zeitraums in beide Richtungen konvertieren kann. Im oben geschilderten Szenario bedeutet dies:

```
// -----  
public void TimePeriodMapperSample()  
{  
    TimeCalendar timeCalendar = new TimeCalendar();  
    CultureInfo ci = CultureInfo.InvariantCulture;  
  
    DateTime start = new DateTime( 2011, 3, 1, 13, 0, 0 );  
    DateTime end = new DateTime( 2011, 3, 1, 14, 0, 0 );  
  
    Console.WriteLine( "Original start: {0}", start.ToString( "HH:mm:ss.fffffff", ci ) );  
    // > Original start: 13:00:00.0000000  
    Console.WriteLine( "Original end: {0}", end.ToString( "HH:mm:ss.fffffff", ci ) );  
    // > Original end: 14:00:00.0000000  
  
    Console.WriteLine( "Mapping offset start: {0}", timeCalendar.StartOffset );  
    // > Mapping offset start: 00:00:00  
    Console.WriteLine( "Mapping offset end: {0}", timeCalendar.EndOffset );  
    // > Mapping offset end: -00:00:00.0000001  
  
    Console.WriteLine( "Mapped start: {0}", timeCalendar.MapStart( start ).ToString( "HH:mm:ss.fffffff", ci ) );  
    // > Mapped start: 13:00:00.0000000  
    Console.WriteLine( "Mapped end: {0}", timeCalendar.MapEnd( end ).ToString( "HH:mm:ss.fffffff", ci ) );  
    // > Mapped end: 13:59:59.9999999  
}  
// TimePeriodMapperSample
```

Time Calendar

Die Interpretation der Zeiträume von Kalenderelementen ist im Interface `ITimeCalendar` zusammengefasst:



ITimeCalendar deckt folgende Bereiche ab:

- Zuordnung zur `CultureInfo` (default = `CultureInfo` des aktuellen Threads)
- Mapping der Periodengrenzen (`ITimePeriodMapper`)
- Basismonat des Jahres (default = Januar)
- Interpretationsweise der Kalenderwochen
- Bezeichnung der Perioden wie z.B. der Name des Jahres (Fiskaljahr, Schuljahr, ...)
- Diverse kalenderbezogene Berechnungen

Als Ableitung von `ITimePeriodMapper` werden die Zeiträume mit den Properties `StartOffset` (Default = 0) und `EndOffset` (Default = -1 Tick) gemappt.

Das folgende Beispiel zeigt die Spezialisierung des Zeitkalenders für ein Fiskaljahr:

```
// -----
public class FiscalTimeCalendar : TimeCalendar
{
    // -----
    public FiscalTimeCalendar()
    : base(
        new TimeCalendarConfig
        {
            YearBaseMonth = YearMonth.October, // October year base month
            YearWeekType = YearWeekType.Iso8601, // ISO 8601 week numbering
            YearType = YearType.FiscalYear // treat years as fiscal years
        } )
    {
    } // FiscalTimeCalendar
} // class FiscalTimeCalendar
```

Der Zeitkalender kann nun wie folgt verwendet werden:

```
// -----
public void FiscalYearSample()
{
    FiscalTimeCalendar calendar = new FiscalTimeCalendar(); // use fiscal periods

    DateTime moment1 = new DateTime( 2006, 9, 30 );
    Console.WriteLine( "Fiscal Year of {0}: {1}", moment1.ToShortDateString(), new Year( moment1, calendar ).YearName );
    // > Fiscal Year of 30.09.2006: FY2005
    Console.WriteLine( "Fiscal Quarter of {0}: {1}", moment1.ToShortDateString(), new Quarter( moment1,
calendar ).QuarterOfYearName );
    // > Fiscal Quarter of 30.09.2006: FQ4 2005

    DateTime moment2 = new DateTime( 2006, 10, 1 );
    Console.WriteLine( "Fiscal Year of {0}: {1}", moment2.ToShortDateString(), new Year( moment2, calendar ).YearName );
    // > Fiscal Year of 01.10.2006: FY2006
    Console.WriteLine( "Fiscal Quarter of {0}: {1}", moment2.ToShortDateString(), new Quarter( moment2,
calendar ).QuarterOfYearName );
    // > Fiscal Quarter of 30.09.2006: FQ1 2006
} // FiscalYearSample
```

Die Beschreibung der Klassen **Year** und **Quarter** folgt weiter unten.

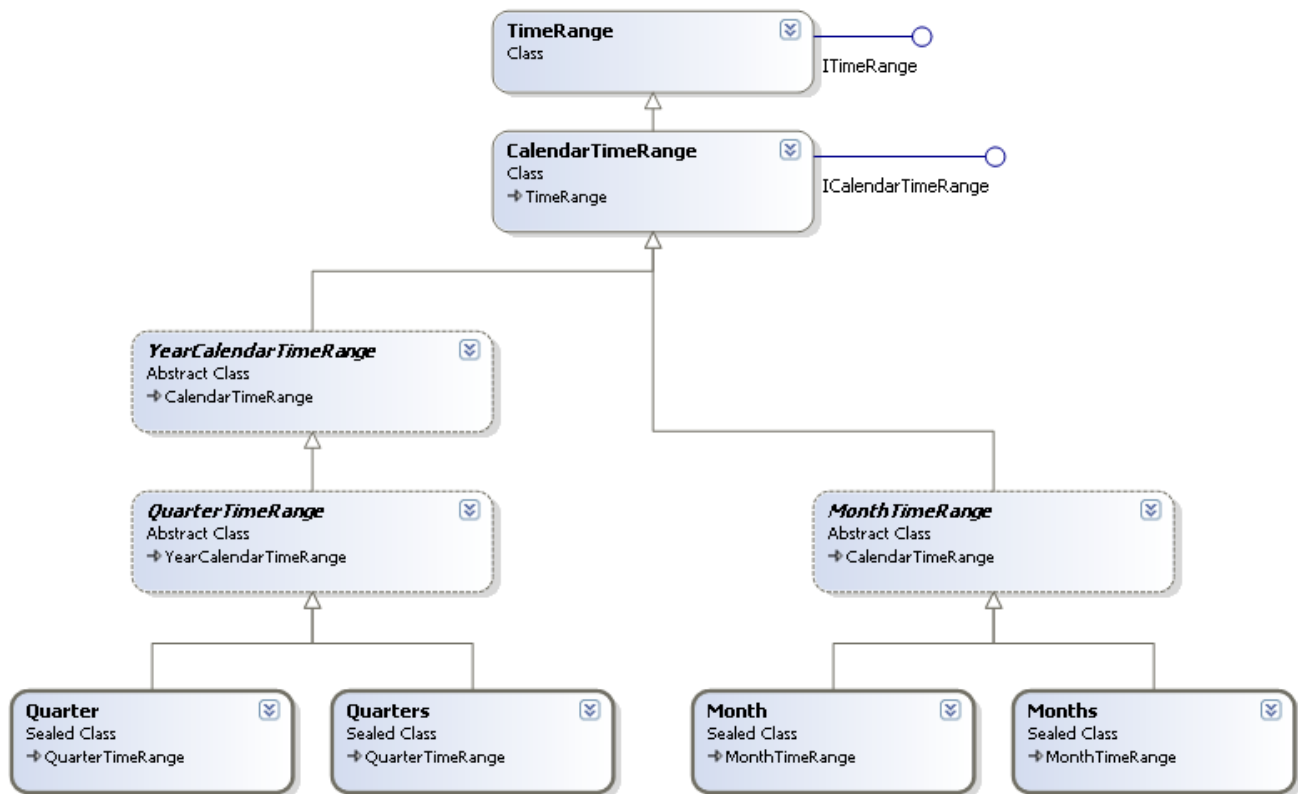
Kalendarelemente

Für die gängigsten Kalenderelemente stehen spezialisierte Klassen zur Verfügung:

Zeitraum	Einzelne Periode	Mehrere Perioden	Bezug Jahr Basismonat
Jahr	Year	Years	Ja
Halbjahr	Halfyear	Halfyears	Ja
Quartal	Quarter	Quarters	Ja
Monat	Month	Months	Nein
Jahreswoche	Week	Weeks	Nein
Tag	Day	Days	Nein
Stunde	Hour	Hours	Nein
Minute	Minute	Minutes	Nein

Bei Elementen über mehrere Perioden kann bei der Instanzierung die Anzahl der Perioden bestimmt werden.

Das folgende Diagramm zeigt exemplarisch die Kalenderelemente für die Elemente Quartal und Monat:



Alle Kalenderelemente besitzen die Basisklasse **CalendarTimeRange** welche wiederum von **TimeRange** abgeleitet ist. **CalendarTimeRange** beinhaltet den Zeitkalender **ITimeCalendar** und gewährleistet, dass die Werte des Zeitraums nachträglich nicht verändert werden können (**IsReadOnly=True**).

Weil das Kalenderelement durch die Basisklasse **TimePeriod** das Interface **ITimePeriod** implementiert, kann es für Berechnungen mit anderen Zeitperioden kombiniert werden.

Das folgende Beispiel zeigt verschiedene Kalenderelemente:

```

// -----
public void CalendarYearTimePeriodsSample()
{
    DateTime moment = new DateTime( 2011, 8, 15 );
    Console.WriteLine( "Calendar Periods of {0}:", moment.ToShortDateString() );
    // > Calendar Periods of 15.08.2011:
    Console.WriteLine( "Year      : {0}", new Year( moment ) );
    Console.WriteLine( "Halfyear  : {0}", new Halfyear( moment ) );
    Console.WriteLine( "Quarter   : {0}", new Quarter( moment ) );
    Console.WriteLine( "Month     : {0}", new Month( moment ) );
    Console.WriteLine( "Week      : {0}", new Week( moment ) );
    Console.WriteLine( "Day       : {0}", new Day( moment ) );
    Console.WriteLine( "Hour      : {0}", new Hour( moment ) );
    // > Year      : 2011; 01.01.2011 - 31.12.2011 | 364.23:59
    // > Halfyear  : HY2 2011; 01.07.2011 - 31.12.2011 | 183.23:59
    // > Quarter   : Q3 2011; 01.07.2011 - 30.09.2011 | 91.23:59
    // > Month     : August 2011; 01.08.2011 - 31.08.2011 | 30.23:59
    // > Week      : w/c 33 2011; 15.08.2011 - 21.08.2011 | 6.23:59
    // > Day       : Montag; 15.08.2011 - 15.08.2011 | 0.23:59
    // > Hour      : 15.08.2011; 00:00 - 00:59 | 0.00:59
} // CalendarYearTimePeriodsSample

```

Einzelne Kalenderelemente bieten Methoden an, um die Zeiträume der Subelement zu erhalten. Das folgende Beispiel zeigt die Quartale eines Kalenderjahres an:

```

// -----
public void YearQuartersSample()
{
    Year year = new Year( 2012 );
    ITimePeriodCollection quarters = year.GetQuarters();
    Console.WriteLine( "Quarters of Year: {0}", year );
    // > Quarters of Year: 2012; 01.01.2012 - 31.12.2012 | 365.23:59
    foreach ( Quarter quarter in quarters )
    {
        Console.WriteLine( "Quarter: {0}", quarter );
    }
}

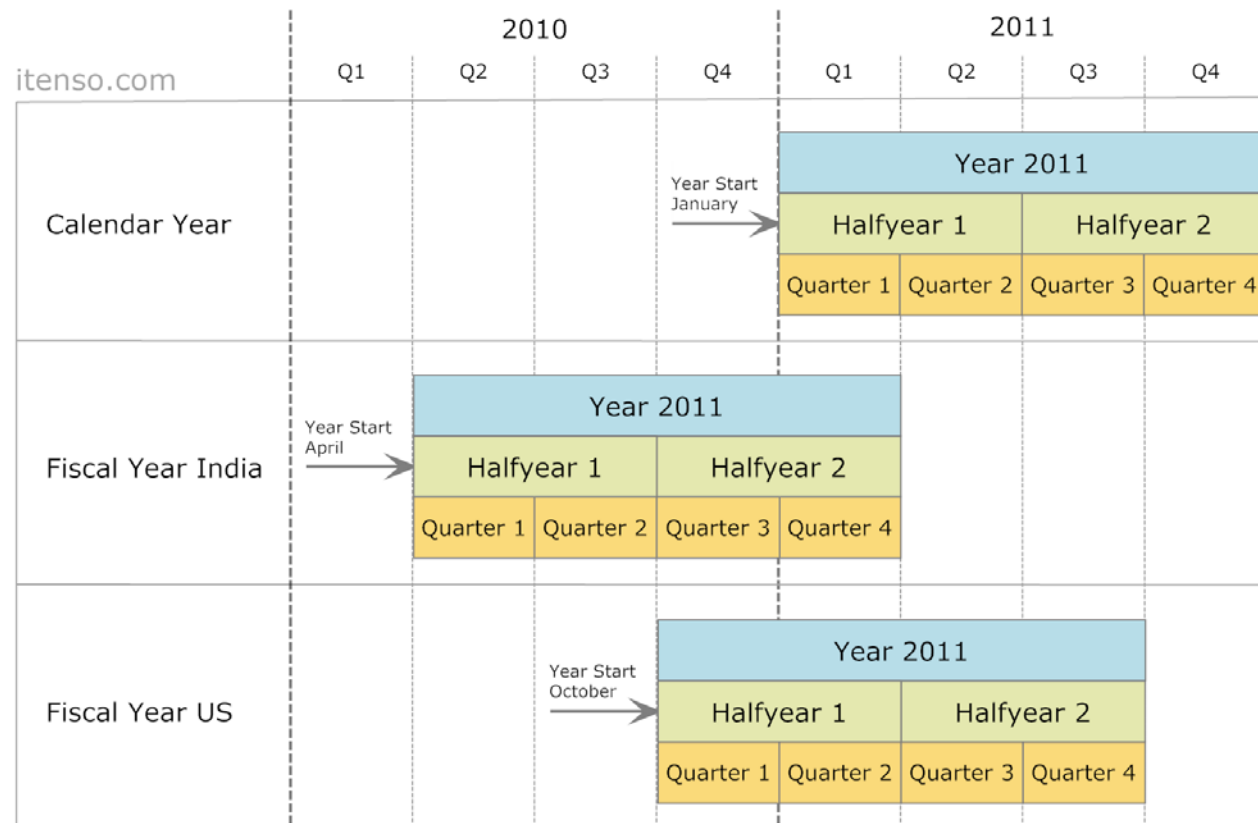
```



```
// > Quarter: Q1 2012; 01.01.2012 - 31.03.2012 | 90.23:59
// > Quarter: Q2 2012; 01.04.2012 - 30.06.2012 | 90.23:59
// > Quarter: Q3 2012; 01.07.2012 - 30.09.2012 | 91.23:59
// > Quarter: Q4 2012; 01.10.2012 - 31.12.2012 | 91.23:59
} // YearQuartersSample
```

Jahr und Jahresperioden

Eine Besonderheit der Kalenderelemente ist der Support von Kalenderperioden, welche vom Kalenderjahr abweichen:



Der Jahresanfang wird durch das Property `ITimeCalendar.YearBaseMonth` gesteuert und wird von den Kalenderelementen Jahr, Halbjahr und Quartal berücksichtigt. Als Jahresstart kann ein beliebiger Monat eingesetzt werden. Das Kalenderjahr stellt somit den Spezialfall `YearBaseMonth = YearMonth.January` dar.

Für die Interpretation der Jahresübergänge stehen folgende Properties zur Verfügung:

- `MultipleCalendarYears` ergibt True, wenn sich die Periode über die Jahresgrenze erstreckt
- `IsCalendarYear/Halfyear/Quarter` ergibt True, wenn die Periode derjenigen des Kalenderjahres entspricht

Fiskaljahre welche im Juli oder später starten, besitzen standardmässig die Jahreszahl des Folgejahrs. Die Kalendereigenschaft `FiscalYearBaseMonth` bietet die Möglichkeit, den Monat zu bestimmen, ab welchen das Fiskaljahr dem nächsten Jahr zugeordnet wird.

Das folgende Beispiel zeigt die Kalenderelemente eines Fiskaljahres auf:

```
// -----
public void FiscalYearTimePeriodsSample()
{
    DateTime moment = new DateTime( 2011, 8, 15 );
    FiscalTimeCalendar fiscalCalendar = new FiscalTimeCalendar();
    Console.WriteLine( "Fiscal Year Periods of {0}:", moment.ToShortDateString() );
    // > Fiscal Year Periods of 15.08.2011:
    Console.WriteLine( "Year      : {0}", new Year( moment, fiscalCalendar ) );
    Console.WriteLine( "Halfyear  : {0}", new Halfyear( moment, fiscalCalendar ) );
    Console.WriteLine( "Quarter   : {0}", new Quarter( moment, fiscalCalendar ) );
    // > Year      : FY2010; 01.10.2010 - 30.09.2011 | 364.23:59
    // > Halfyear  : FHY2 2010; 01.04.2011 - 30.09.2011 | 182.23:59
    // > Quarter   : FQ4 2010; 01.07.2011 - 30.09.2011 | 91.23:59
}
```

```
} // FiscalYearTimePeriodsSample
```

Die Verschiebung des Jahresbeginnes wirkt sich entsprechend auf alle Elemente und Operationen aus:

```
// -----
public void YearStartSample()
{
    TimeCalendar calendar = new TimeCalendar(
        new TimeCalendarConfig { YearBaseMonth = YearMonth.February } );

    Years years = new Years( 2012, 2, calendar ); // 2012-2013
    Console.WriteLine( "Quarters of Years (February): {0}", years );
    // > Quarters of Years (February): 2012 - 2014; 01.02.2012 - 31.01.2014 | 730.23:59

    foreach ( Year year in years.GetYears() )
    {
        foreach ( Quarter quarter in year.GetQuarters() )
        {
            Console.WriteLine( "Quarter: {0}", quarter );
        }
    }
    // > Quarter: Q1 2012; 01.02.2012 - 30.04.2012 | 89.23:59
    // > Quarter: Q2 2012; 01.05.2012 - 31.07.2012 | 91.23:59
    // > Quarter: Q3 2012; 01.08.2012 - 31.10.2012 | 91.23:59
    // > Quarter: Q4 2012; 01.11.2012 - 31.01.2013 | 91.23:59
    // > Quarter: Q1 2013; 01.02.2013 - 30.04.2013 | 88.23:59
    // > Quarter: Q2 2013; 01.05.2013 - 31.07.2013 | 91.23:59
    // > Quarter: Q3 2013; 01.08.2013 - 31.10.2013 | 91.23:59
    // > Quarter: Q4 2013; 01.11.2013 - 31.01.2014 | 91.23:59
} // YearStartSample
```

Es folgen nun Beispiele von Hilfsfunktionen, welche in der Praxis oft nützlich sein können:

```
// -----
public bool IntersectsYear( DateTime start, DateTime end, int year )
{
    return new Year( year ).Intersects( new TimeRange( start, end ) );
} // IntersectsYear

// -----
public void GetDaysOfPastQuarter( DateTime moment, out DateTime firstDay, out DateTime lastDay )
{
    TimeCalendar calendar = new TimeCalendar(
        new TimeCalendarConfig { YearBaseMonth = YearMonth.October } );
    Quarter quarter = new Quarter( moment, calendar );
    Quarter pastQuarter = quarter.GetPreviousQuarter();

    firstDay = pastQuarter.FirstDayStart;
    lastDay = pastQuarter.LastDayStart;
} // GetDaysOfPastQuarter

// -----
public DateTime GetFirstDayOfWeek( DateTime moment )
{
    return new Week( moment ).FirstDayStart;
} // GetFirstDayOfWeek

// -----
public bool IsInCurrentWeek( DateTime test )
{
    return new Week().HasInside( test );
} // IsInCurrentWeek
```

Wochen

Wochen werden in der Regel innerhalb einer Jahres von 1 bis 52/53 durchnummeriert. Das .NET Framework bietet mit `Calendar.GetWeekOfYear` eine Funktion zur Ermittlung der Kalenderwoche. Diese weicht jedoch von der [ISO 8601](http://www.iso.org/iso/8601) Definition ab, was zu Fehlinterpretationen führen kann.

Die **Time Period** Bibliothek beinhaltet die Aufzählung `YearWeekType`, welche die Berechnung von ISO 8601 Kalenderwochen steuert. `YearWeekType` wird durch `ITimeCalendar` unterstützt und regelt die Berechnungsvarianten:

```
// -----
// see also http://blogs.msdn.com/b/shawnste/archive/2006/01/24/517178.aspx
public void CalendarWeekSample()
{
    DateTime testDate = new DateTime( 2007, 12, 31 );

    // .NET calendar week
    TimeCalendar calendar = new TimeCalendar();
    Console.WriteLine( "Calendar Week of {0}: {1}", testDate.ToShortDateString(), new Week( testDate,
        calendar ).WeekOfYear );
    // > Calendar Week of 31.12.2007: 53

    // ISO 8601 calendar week
    TimeCalendar calendarIso8601 = new TimeCalendar(
```

```

    new TimeCalendarConfig { YearWeekType = YearWeekType.Iso8601 } );
    Console.WriteLine( "ISO 8601 Week of {0}: {1}", testDate.ToShortDateString(), new Week( testDate,
calendarIso8601 ).WeekOfYear );
    // > ISO 8601 Week of 31.12.2007: 1
} // CalendarWeekSample

```

Buchhaltung-Kalender

Zur Vereinfachung der Planung gliedern Industrien im Rechnungswesen das Jahr oft in Quartale ein, welche in Monate mit 4 oder 5 Wochen unterteilt sind ([4-4-5 Kalender](#)). Ein solches Jahr wird wahlweise auf

- den letzte Wochentag eines Monats (`FiscalYearAlignment.LastDay`)
- einem Wochentag nahe dem Monatsende (`FiscalYearAlignment.NearestDay`)

ausgerichtet.

Die Anordnung der Wochen erfolgt nach folgenden Gruppierungskriterien:

- 4-4-5 Wochen (`FiscalQuarterGrouping.FourFourFiveWeeks`)
- 4-5-4 Wochen (`FiscalQuarterGrouping.FourFiveFourWeeks`)
- 5-4-4 Wochen (`FiscalQuarterGrouping.FiveFourFourWeeks`)

Die Steuerung erfolgt im Kalender `ITimeCalendar` und gilt nur für Fiskaljahre (`YearType.FiscalYear`). Die Kalendereigenschaft `FiscalFirstDayOfYear` bestimmt den Wochentag, an welchem ein Jahr beginnt.

Das folgende Beispiel zeigt ein Fiskaljahr welches am letzten Samstag im Monat August endet:

```

// -----
public void FiscalYearLastDay()
{
    ITimeCalendar calendar = new TimeCalendar( new TimeCalendarConfig
    {
        YearType = YearType.FiscalYear,
        YearBaseMonth = YearMonth.September,
        FiscalFirstDayOfYear = DayOfWeek.Sunday,
        FiscalYearAlignment = FiscalYearAlignment.LastDay,
        FiscalQuarterGrouping = FiscalQuarterGrouping.FourFourFiveWeeks
    } );

    Years years = new Years( 2005, 14, calendar );
    foreach ( Year year in years.GetYears() )
    {
        Console.WriteLine( "Fiscal year {0}: {1} - {2}", year.YearValue,
            year.Start.ToString( "yyyy-MM-dd" ), year.End.ToString( "yyyy-MM-dd" ) );
    }
} // FiscalYearLastDay

```

Das folgende Fiskaljahr endet jeweils am Samstag, welcher näher beim Ende des Monats August liegt:

```

public void FiscalYearNearestDay()
{
    ITimeCalendar calendar = new TimeCalendar( new TimeCalendarConfig
    {
        YearType = YearType.FiscalYear,
        YearBaseMonth = YearMonth.September,
        FiscalFirstDayOfYear = DayOfWeek.Sunday,
        FiscalYearAlignment = FiscalYearAlignment.NearestDay,
        FiscalQuarterGrouping = FiscalQuarterGrouping.FourFourFiveWeeks
    } );

    Years years = new Years( 2005, 14, calendar );
    foreach ( Year year in years.GetYears() )
    {
        Console.WriteLine( "Fiscal year {0}: {1} - {2}", year.YearValue,
            year.Start.ToString( "yyyy-MM-dd" ), year.End.ToString( "yyyy-MM-dd" ) );
    }
} // FiscalYearNearestDay

```

Rundfunk-Kalender

The [Broadcast Calendar](#) is supported by the classes `BroadcastYear`, `BroadcastMonth` and `BroadcastWeek`:

```

// -----
public void BroadcastCalendar()

```

```

{
    BroadcastYear year = new BroadcastYear( 2013 );
    Console.WriteLine( "Broadcast year: " + year );
    // > Broadcast year: 2013; 31.12.2012 - 29.12.2013 | 363.23:59

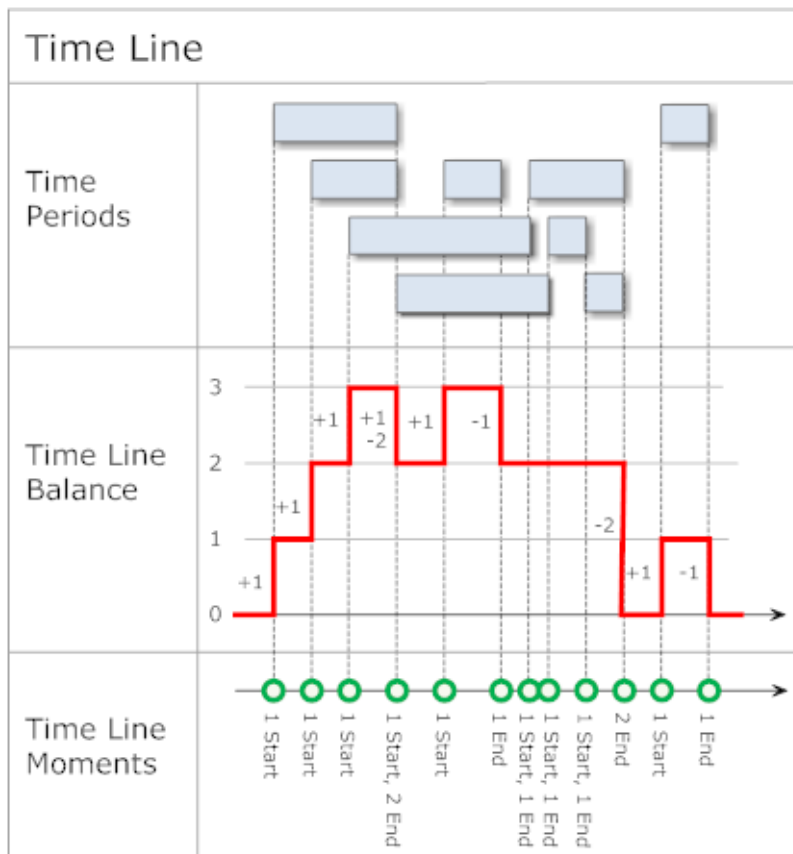
    foreach ( BroadcastMonth month in year.GetMonths() )
    {
        Console.WriteLine( "    Broadcast month: " + month );
        foreach ( BroadcastWeek week in month.GetWeeks() )
        {
            Console.WriteLine( "        Broadcast week: " + week );
        }
    }
} // BroadcastCalendar

```

Time Period Berechnungstools

Time Line

Die Klasse **TimeLine** dient als Herzstück zur Berechnung von Zeitlücken und Überlappungen. Sie analysiert die Zeiträume eine Collection, durch das zeitliche anordnen aller auftretenden Momente. Jeder Moment der Zeitachse wird durch **ITimeLineMoment** beschrieben und beinhaltet die Informationen, welche Zeiträume zu einem bestimmten Moment anfangen und aufhören. Diese Betrachtungsweise erlaubt bei der Abarbeitung der Zeitlinien, durch addieren und subtrahieren die laufende Balance zu berechnen.



Die Momente der Zeitachse werden in der Collection **ITimeLineMomentCollection** gespeichert, welche den effizienten Zugriff für Iterationen und Zeitpunktzugriffe erlaubt.

Differenz zweier Zeitpunkte

Die **TimeSpan** Struktur des .NET Framework bietet für den Zeitraum nur die Werte in Tagen, Stunden, Minuten, Sekunden und Millisekunden an. Aus Sicht der Benutzerfreundlichkeit macht es oft Sinn, auch Monate und Jahre eines Zeitraumes darzustellen:

- Letzter Besuch vor 1 Jahr, 4 Monaten und 12 Tagen

- o Aktuelles Alter: 28 Jahre

Die **Time Period** Bibliothek beinhaltet die Klasse **DateDiff**, welche von zwei Datumswerten die Zeitdifferenz sowie die abgelaufene Zeitdauer anbietet. Dabei werden Kalenderperioden berücksichtigt um zum Beispiel verschiedene Monatsdauern korrekt zu behandeln:

```
// -----
public void DateDiffSample()
{
    DateTime date1 = new DateTime( 2009, 11, 8, 7, 13, 59 );
    Console.WriteLine( "Date1: {0}", date1 );
    // > Date1: 08.11.2009 07:13:59
    DateTime date2 = new DateTime( 2011, 3, 20, 19, 55, 28 );
    Console.WriteLine( "Date2: {0}", date2 );
    // > Date2: 20.03.2011 19:55:28

    DateDiff dateDiff = new DateDiff( date1, date2 );

    // differences
    Console.WriteLine( "DateDiff.Years: {0}", dateDiff.Years );
    // > DateDiff.Years: 1
    Console.WriteLine( "DateDiff.Quarters: {0}", dateDiff.Quarters );
    // > DateDiff.Quarters: 5
    Console.WriteLine( "DateDiff.Months: {0}", dateDiff.Months );
    // > DateDiff.Months: 16
    Console.WriteLine( "DateDiff.Weeks: {0}", dateDiff.Weeks );
    // > DateDiff.Weeks: 70
    Console.WriteLine( "DateDiff.Days: {0}", dateDiff.Days );
    // > DateDiff.Days: 497
    Console.WriteLine( "DateDiff.Weekdays: {0}", dateDiff.Weekdays );
    // > DateDiff.Weekdays: 71
    Console.WriteLine( "DateDiff.Hours: {0}", dateDiff.Hours );
    // > DateDiff.Hours: 11940
    Console.WriteLine( "DateDiff.Minutes: {0}", dateDiff.Minutes );
    // > DateDiff.Minutes: 716441
    Console.WriteLine( "DateDiff.Seconds: {0}", dateDiff.Seconds );
    // > DateDiff.Seconds: 42986489

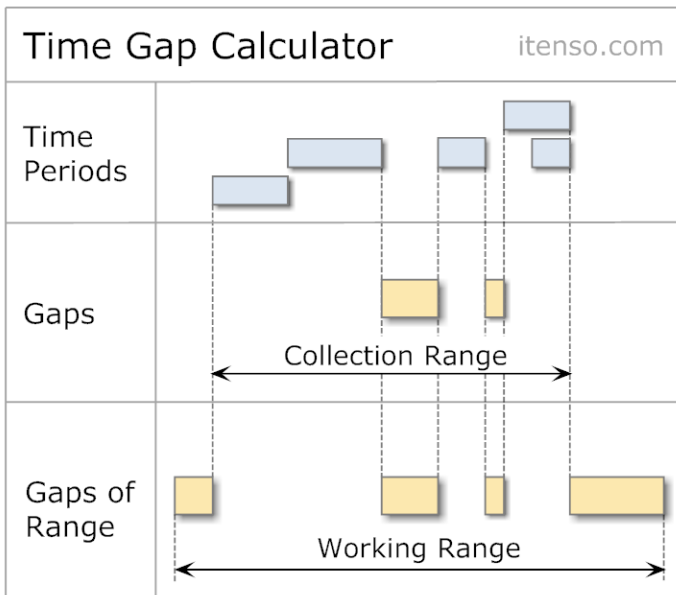
    // elapsed
    Console.WriteLine( "DateDiff.ElapsedYears: {0}", dateDiff.ElapsedYears );
    // > DateDiff.ElapsedYears: 1
    Console.WriteLine( "DateDiff.ElapsedMonths: {0}", dateDiff.ElapsedMonths );
    // > DateDiff.ElapsedMonths: 4
    Console.WriteLine( "DateDiff.ElapsedDays: {0}", dateDiff.ElapsedDays );
    // > DateDiff.ElapsedDays: 12
    Console.WriteLine( "DateDiff.ElapsedHours: {0}", dateDiff.ElapsedHours );
    // > DateDiff.ElapsedHours: 12
    Console.WriteLine( "DateDiff.ElapsedMinutes: {0}", dateDiff.ElapsedMinutes );
    // > DateDiff.ElapsedMinutes: 41
    Console.WriteLine( "DateDiff.ElapsedSeconds: {0}", dateDiff.ElapsedSeconds );
    // > DateDiff.ElapsedSeconds: 29

    // description
    Console.WriteLine( "DateDiff.GetDescription(1): {0}", dateDiff.GetDescription( 1 ) );
    // > DateDiff.GetDescription(1): 1 Year
    Console.WriteLine( "DateDiff.GetDescription(2): {0}", dateDiff.GetDescription( 2 ) );
    // > DateDiff.GetDescription(2): 1 Year 4 Months
    Console.WriteLine( "DateDiff.GetDescription(3): {0}", dateDiff.GetDescription( 3 ) );
    // > DateDiff.GetDescription(3): 1 Year 4 Months 12 Days
    Console.WriteLine( "DateDiff.GetDescription(4): {0}", dateDiff.GetDescription( 4 ) );
    // > DateDiff.GetDescription(4): 1 Year 4 Months 12 Days 12 Hours
    Console.WriteLine( "DateDiff.GetDescription(5): {0}", dateDiff.GetDescription( 5 ) );
    // > DateDiff.GetDescription(5): 1 Year 4 Months 12 Days 12 Hours 41 Mins
    Console.WriteLine( "DateDiff.GetDescription(6): {0}", dateDiff.GetDescription( 6 ) );
    // > DateDiff.GetDescription(6): 1 Year 4 Months 12 Days 12 Hours 41 Mins 29 Secs
} // DateDiffSample
```

Mit der Methode **DateDiff.GetDescription** kann die Zeitdauer mit einer variablen Darstellungstiefe dargestellt werden.

Suche von Zeitraumlücken

Der **TimeGapCalculator** berechnet die Lücken von Zeiträumen einer Collection:



Zur Interpretation der Zeitpunkte kann ein **ITimePeriodMapper** verwendet werden.

Im folgenden Beispiel soll innerhalb eines Zeitraums die grösstmögliche Lücke zwischen vorhandenen Buchungen gefunden werden. Dabei sollen die Wochenenden als Sperrzeiten betrachtet werden:

```
// -----
public void TimeGapCalculatorSample()
{
    // simulation of some reservations
    TimePeriodCollection reservations = new TimePeriodCollection();
    reservations.Add( new Days( 2011, 3, 7, 2 ) );
    reservations.Add( new Days( 2011, 3, 16, 2 ) );

    // the overall search range
    CalendarTimeRange searchLimits = new CalendarTimeRange( new DateTime( 2011, 3, 4 ), new DateTime( 2011, 3, 21 ) );

    // search the largest free time block
    ICalendarTimeRange largestFreeTimeBlock = FindLargestFreeTimeBlock( reservations, searchLimits );
    Console.WriteLine( "Largest free time: " + largestFreeTimeBlock );
    // > Largest free time: 09.03.2011 00:00:00 - 11.03.2011 23:59:59 | 2.23:59
} // TimeGapCalculatorSample

// -----
public ICalendarTimeRange FindLargestFreeTimeBlock( IEnumerable<ITimePeriod> reservations,
    ITimePeriod searchLimits = null, bool excludeWeekends = true )
{
    TimePeriodCollection bookedPeriods = new TimePeriodCollection( reservations );

    if ( searchLimits == null )
    {
        searchLimits = bookedPeriods; // use boundary of reservations
    }

    if ( excludeWeekends )
    {
        Week currentWeek = new Week( searchLimits.Start );
        Week lastWeek = new Week( searchLimits.End );
        do
        {
            ITimePeriodCollection days = currentWeek.GetDays();
            foreach ( Day day in days )
            {
                if ( !searchLimits.HasInside( day ) )
                {
                    continue; // outside of the search scope
                }
                if ( day.DayOfWeek == DayOfWeek.Saturday || day.DayOfWeek == DayOfWeek.Sunday )
                {
                    bookedPeriods.Add( day ); // // exclude weekend day
                }
            }
            currentWeek = currentWeek.GetNextWeek();
        } while ( currentWeek.Start < lastWeek.Start );
    }

    // calculate the gaps using the time calendar as period mapper
    TimeGapCalculator<TimeRange> gapCalculator = new TimeGapCalculator<TimeRange>( new TimeCalendar() );
```

```

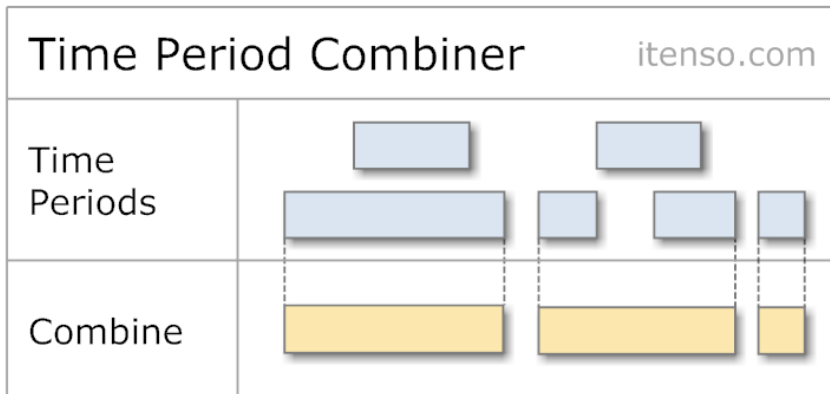
ITimePeriodCollection freeTimes = gapCalculator.GetGaps( bookedPeriods, searchLimits );
if ( freeTimes.Count == 0 )
{
    return null;
}

freeTimes.SortByDuration(); // move the largest gap to the start
return new CalendarTimeRange( freeTimes[ 0 ] );
} // FindLargestFreeTimeBlock

```

Zusammenführen von Zeiträumen

Für bestimmte Aufgaben ist es sinnvoll, überlappende und angrenzende Zeiträume konsolidiert zu betrachten. Die Klasse `TimePeriodCombiner` bietet die Möglichkeit, Zeiträume zusammenzufassen:



Das folgende Beispiel zeigt die Verwendung von `TimePeriodCombiner`:

```

// -----
public void TimePeriodCombinerSample()
{
    TimePeriodCollection periods = new TimePeriodCollection();

    periods.Add( new TimeRange( new DateTime( 2011, 3, 01 ), new DateTime( 2011, 3, 10 ) ) );
    periods.Add( new TimeRange( new DateTime( 2011, 3, 04 ), new DateTime( 2011, 3, 08 ) ) );

    periods.Add( new TimeRange( new DateTime( 2011, 3, 15 ), new DateTime( 2011, 3, 18 ) ) );
    periods.Add( new TimeRange( new DateTime( 2011, 3, 18 ), new DateTime( 2011, 3, 22 ) ) );
    periods.Add( new TimeRange( new DateTime( 2011, 3, 20 ), new DateTime( 2011, 3, 24 ) ) );

    periods.Add( new TimeRange( new DateTime( 2011, 3, 26 ), new DateTime( 2011, 3, 30 ) ) );

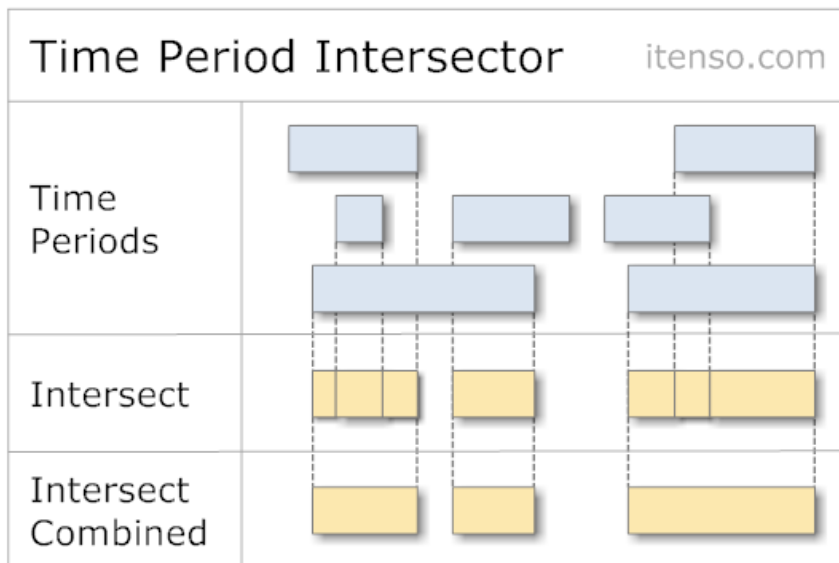
    TimePeriodCombiner<TimeRange> periodCombiner = new TimePeriodCombiner<TimeRange>();
    ITimePeriodCollection combinedPeriods = periodCombiner.CombinePeriods( periods );

    foreach ( ITimePeriod combinedPeriod in combinedPeriods )
    {
        Console.WriteLine( "Combined Period: " + combinedPeriod );
    }
    // > Combined Period: 01.03.2011 - 10.03.2011 | 9.00:00
    // > Combined Period: 15.03.2011 - 24.03.2011 | 9.00:00
    // > Combined Period: 26.03.2011 - 30.03.2011 | 4.00:00
} // TimePeriodCombinerSample

```

Schnittmengen von Zeiträumen

Die Klasse `TimePeriodIntersector` dient zur Ermittlung aller Schnittmengen von überlappenden Zeiträumen:



Im Normalfall werden die resultierenden Schnittmengen in einen Zeitraum zusammengefasst. Um die Schnittmengen beizubehalten, kann der Parameter `combinePeriods` der Methode `IntersectPeriods` auf `false` gesetzt werden.

Das folgende Beispiel zeigt die Verwendung von `TimePeriodIntersector`:

```
// -----
public void TimePeriodIntersectorSample()
{
    TimePeriodCollection periods = new TimePeriodCollection();

    periods.Add( new TimeRange( new DateTime( 2011, 3, 01 ), new DateTime( 2011, 3, 10 ) ) );
    periods.Add( new TimeRange( new DateTime( 2011, 3, 05 ), new DateTime( 2011, 3, 15 ) ) );
    periods.Add( new TimeRange( new DateTime( 2011, 3, 12 ), new DateTime( 2011, 3, 18 ) ) );

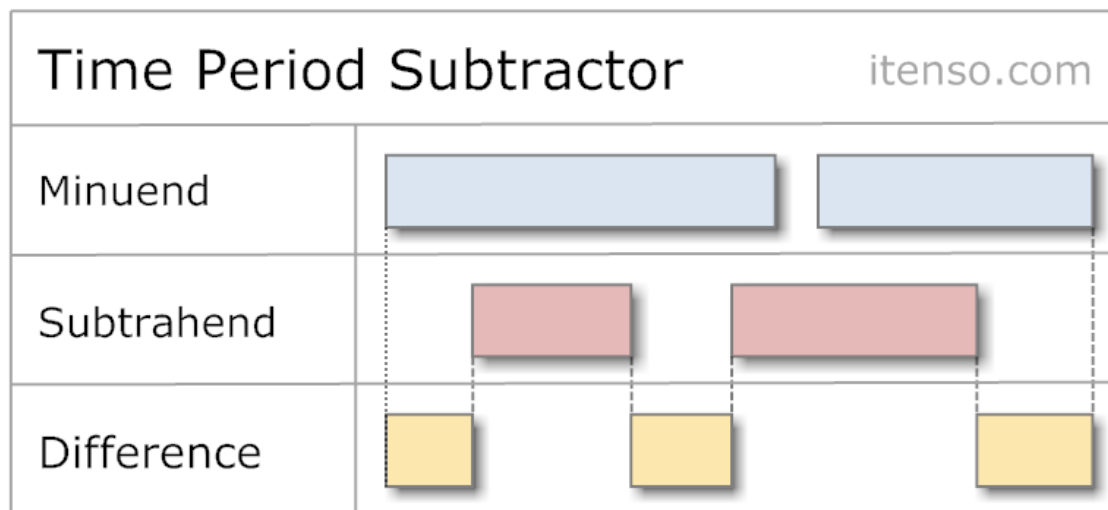
    periods.Add( new TimeRange( new DateTime( 2011, 3, 20 ), new DateTime( 2011, 3, 24 ) ) );
    periods.Add( new TimeRange( new DateTime( 2011, 3, 22 ), new DateTime( 2011, 3, 28 ) ) );
    periods.Add( new TimeRange( new DateTime( 2011, 3, 24 ), new DateTime( 2011, 3, 26 ) ) );

    TimePeriodIntersector<TimeRange> periodIntersector = new TimePeriodIntersector<TimeRange>();
    ITimePeriodCollection intersectedPeriods = periodIntersector.IntersectPeriods( periods );

    foreach ( ITimePeriod intersectedPeriod in intersectedPeriods )
    {
        Console.WriteLine( "Intersected Period: " + intersectedPeriod );
    }
    // > Intersected Period: 05.03.2011 - 10.03.2011 | 5.00:00
    // > Intersected Period: 12.03.2011 - 15.03.2011 | 3.00:00
    // > Intersected Period: 22.03.2011 - 26.03.2011 | 4.00:00
} // TimePeriodIntersectorSample
```


Subtraktion von Zeiträumen

Mit der Klasse `TimePeriodSubtractor` ist es möglich, mehrere Zeiträume (Subtrahend) von anderen Zeiträumen (Minuend) zu subtrahieren:



Das Ergebnis beinhaltet die Differenzen beider Zeitraumcontainer:

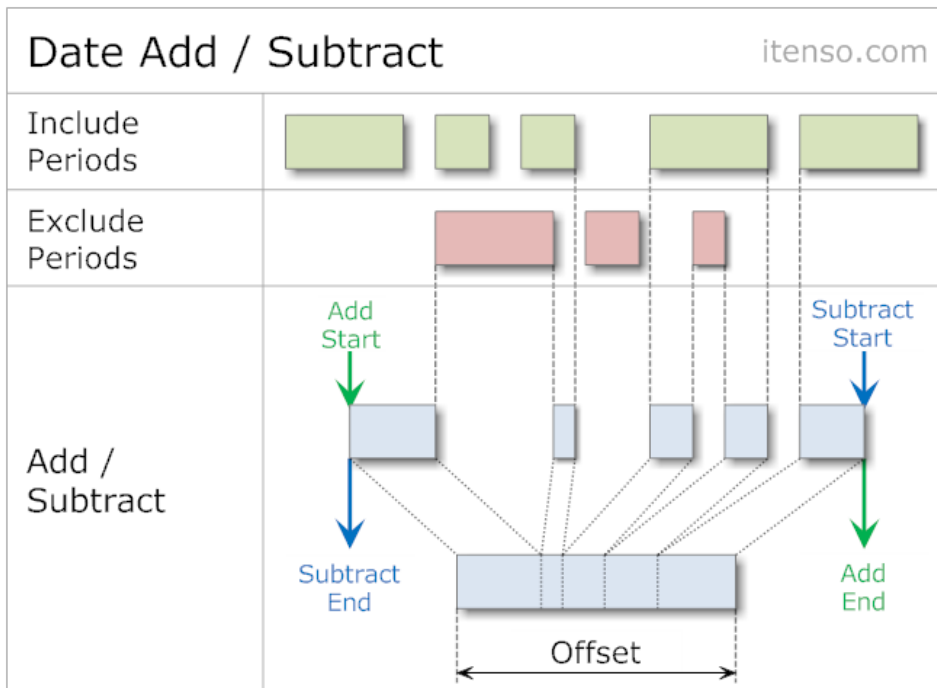
```
// -----  
public void TimePeriodSubtractorSample()  
{  
    DateTime moment = new DateTime( 2012, 1, 29 );  
    TimePeriodCollection sourcePeriods = new TimePeriodCollection  
    {  
        new TimeRange( moment.AddHours( 2 ), moment.AddDays( 1 ) )  
    };  
  
    TimePeriodCollection subtractingPeriods = new TimePeriodCollection  
    {  
        new TimeRange( moment.AddHours( 6 ), moment.AddHours( 10 ) ),  
        new TimeRange( moment.AddHours( 12 ), moment.AddHours( 16 ) )  
    };  
  
    TimePeriodSubtractor<TimeRange> subtractor = new TimePeriodSubtractor<TimeRange>();  
    ITimePeriodCollection subtractedPeriods = subtractor.SubtractPeriods( sourcePeriods, subtractingPeriods );  
    foreach ( TimeRange subtractedPeriod in subtractedPeriods )  
    {  
        Console.WriteLine( "Subtracted Period: {0}", subtractedPeriod );  
    }  
    // > Subtracted Period : 29.01.2012 02:00:00 - 06:00:00 | 0.04:00  
    // > Subtracted Period : 29.01.2012 10:00:00 - 12:00:00 | 0.02:00  
    // > Subtracted Period : 29.01.2012 16:00:00 - 30.01.2012 00:00:00 | 0.08:00  
} // TimePeriodSubtractorSample
```

Datum Addieren und Subtrahieren

Oft soll zu einem bestimmten Datum ein Zeitraum hinzugefügt werden, und daraus der Zielzeitpunkt berechnet werden. Was einfach klingt, wird durch verschiedene Faktoren erschwert:

- Es dürfen nur die Geschäftszeiten berücksichtigt werden
- Wochenende, Feiertage, Service- und Wartungsperioden sollen ausgeschlossen werden

Sobald solche Anforderungen existieren, ist eine einfache Datumsarithmetik nicht mehr möglich. In solchen Fällen kann die Klasse `DateAdd` eingesetzt werden:



Anders als es der Name der Klasse vermuten lässt, kann Addiert und Subtrahiert werden. Die Besonderheit von `DateAdd` besteht darin, dass einzuschliessende Perioden `DateAdd.IncludePeriods` sowie auszuschliessende Perioden `DateAdd.ExcludePeriods` bestimmt werden können. Man ist auch offen, nur die einzuschliessenden bzw. nur die auszuschliessenden Perioden festzulegen. Sind beide undefiniert, verhält sich das Tool wie `DateTime.Add` bzw. `DateTime.Subtract`.

Das folgende Beispiel zeigt die Verwendung von `DateAdd`:

```
// -----  
public void DateAddSample()  
{  
    DateAdd dateAdd = new DateAdd();  
  
    dateAdd.IncludePeriods.Add( new TimeRange( new DateTime( 2011, 3, 17 ), new DateTime( 2011, 4, 20 ) ) );  
  
    // setup some periods to exclude  
    dateAdd.ExcludePeriods.Add( new TimeRange(  
        new DateTime( 2011, 3, 22 ), new DateTime( 2011, 3, 25 ) ) );  
    dateAdd.ExcludePeriods.Add( new TimeRange(  
        new DateTime( 2011, 4, 1 ), new DateTime( 2011, 4, 7 ) ) );  
    dateAdd.ExcludePeriods.Add( new TimeRange(  
        new DateTime( 2011, 4, 15 ), new DateTime( 2011, 4, 16 ) ) );  
  
    // positive  
    DateTime dateDiffPositive = new DateTime( 2011, 3, 19 );  
    DateTime? positive1 = dateAdd.Add( dateDiffPositive, Duration.Hours( 1 ) );  
    Console.WriteLine( "DateAdd Positive1: {0}", positive1 );  
    // > DateAdd Positive1: 19.03.2011 01:00:00  
    DateTime? positive2 = dateAdd.Add( dateDiffPositive, Duration.Days( 4 ) );  
    Console.WriteLine( "DateAdd Positive2: {0}", positive2 );  
    // > DateAdd Positive2: 26.03.2011 00:00:00  
    DateTime? positive3 = dateAdd.Add( dateDiffPositive, Duration.Days( 17 ) );  
    Console.WriteLine( "DateAdd Positive3: {0}", positive3 );  
    // > DateAdd Positive3: 14.04.2011 00:00:00  
    DateTime? positive4 = dateAdd.Add( dateDiffPositive, Duration.Days( 20 ) );  
    Console.WriteLine( "DateAdd Positive4: {0}", positive4 );  
    // > DateAdd Positive4: 18.04.2011 00:00:00  
}
```

```
// negative
DateTime dateDiffNegative = new DateTime( 2011, 4, 18 );
DateTime? negative1 = dateAdd.Add( dateDiffNegative, Duration.Hours( -1 ) );
Console.WriteLine( "DateAdd Negative1: {0}", negative1 );
// > DateAdd Negative1: 17.04.2011 23:00:00
DateTime? negative2 = dateAdd.Add( dateDiffNegative, Duration.Days( -4 ) );
Console.WriteLine( "DateAdd Negative2: {0}", negative2 );
// > DateAdd Negative2: 13.04.2011 00:00:00
DateTime? negative3 = dateAdd.Add( dateDiffNegative, Duration.Days( -17 ) );
Console.WriteLine( "DateAdd Negative3: {0}", negative3 );
// > DateAdd Negative3: 22.03.2011 00:00:00
DateTime? negative4 = dateAdd.Add( dateDiffNegative, Duration.Days( -20 ) );
Console.WriteLine( "DateAdd Negative4: {0}", negative4 );
// > DateAdd Negative4: 19.03.2011 00:00:00
} // DateAddSample
```

Mit der Spezialisierung **CalendarDateAdd** können die Wochentage und Arbeitsstunden bestimmt werden, welche beim Addieren und Subtrahieren zu berücksichtigen sind. Das Property **IncludePeriods** kann nicht verwendet werden, da die verfügbaren Perioden durch die Woche bestimmt werden:

```
// -----
public void CalendarDateAddSample()
{
    CalendarDateAdd calendarDateAdd = new CalendarDateAdd();
    // weekdays
    calendarDateAdd.AddWorkingWeekDays();
    // holidays
    calendarDateAdd.ExcludePeriods.Add( new Day( 2011, 4, 5, calendarDateAdd.Calendar ) );
    // working hours
    calendarDateAdd.WorkingHours.Add( new HourRange( new Time( 08, 30 ), new Time( 12 ) ) );
    calendarDateAdd.WorkingHours.Add( new HourRange( new Time( 13, 30 ), new Time( 18 ) ) );

    DateTime start = new DateTime( 2011, 4, 1, 9, 0, 0 );
    TimeSpan offset = new TimeSpan( 22, 0, 0 ); // 22 hours

    DateTime? end = calendarDateAdd.Add( start, offset );

    Console.WriteLine( "start: {0}", start );
    // > start: 01.04.2011 09:00:00
    Console.WriteLine( "offset: {0}", offset );
    // > offset: 22:00:00
    Console.WriteLine( "end: {0}", end );
    // > end: 06.04.2011 16:30:00
} // CalendarDateAddSample
```

Suche nach Kalenderperioden

Mit dem **CalendarPeriodCollector** besteht die Möglichkeit, über einen Zeitraum verschiedene Kalenderperioden zu ermitteln. Dabei wird der Filter **ICalendarPeriodCollectorFilter** verwendet, welcher die Suche nach folgenden Kriterien einschränkt:

- Suche nach Jahren
- Suche nach Monaten
- Suche nach Tagen im Monat
- Suche nach Wochentagen

Ist kein Filter gesetzt wird, werden alle Zeiträume einer Periode berücksichtigt.

Beim Zusammenführen können folgende Zielbereiche bestimmt werden:

- Jahre: **CalendarPeriodCollector.CollectYears**
- Monate: **CalendarPeriodCollector.CollectMonths**
- Tage: **CalendarPeriodCollector.CollectDays**
- Stunden: **CalendarPeriodCollector.CollectHours**

Im normalen Modus werden alle Zeitbereiche des gefundenen Bereiches zusammengeführt. So werden zum Beispiel mit **CalendarPeriodCollector.CollectHours** alle Stunden eines Tages gefunden.

Um das Ergebnis weiter einzuschränken, lassen sich Zeitbereiche definieren:

- Welche Monate eines Jahres:
`ICalendarPeriodCollectorFilter.AddCollectingMonths`
- Welche Tage eines Monats: `ICalendarPeriodCollectorFilter.AddCollectingDays`
- Welche Stunden eines Tages:
`ICalendarPeriodCollectorFilter.AddCollectingHours`

Wird zum Beispiel ein Zeitbereich für Stunden von 08:00 bis 10:00 Uhr definiert, resultiert nur eine Zeitperiode und nicht für jede Stunde eine. Dies erweist sich beim Zusammenführen von umfangreichen Zeitperioden als eine hilfreiche, wenn nicht gar notwendige, Optimierung.

Im folgenden Beispiel werden über mehrere Jahre die Arbeitsstunden aller Freitage des Monats Januar ermittelt:

```
// -----
public void CalendarPeriodCollectorSample()
{
    CalendarPeriodCollectorFilter filter = new CalendarPeriodCollectorFilter();
    filter.Months.Add( YearMonth.January ); // only Januaries
    filter.WeekDays.Add( DayOfWeek.Friday ); // only Fridays
    filter.CollectingHours.Add( new HourRange( 8, 18 ) ); // working hours

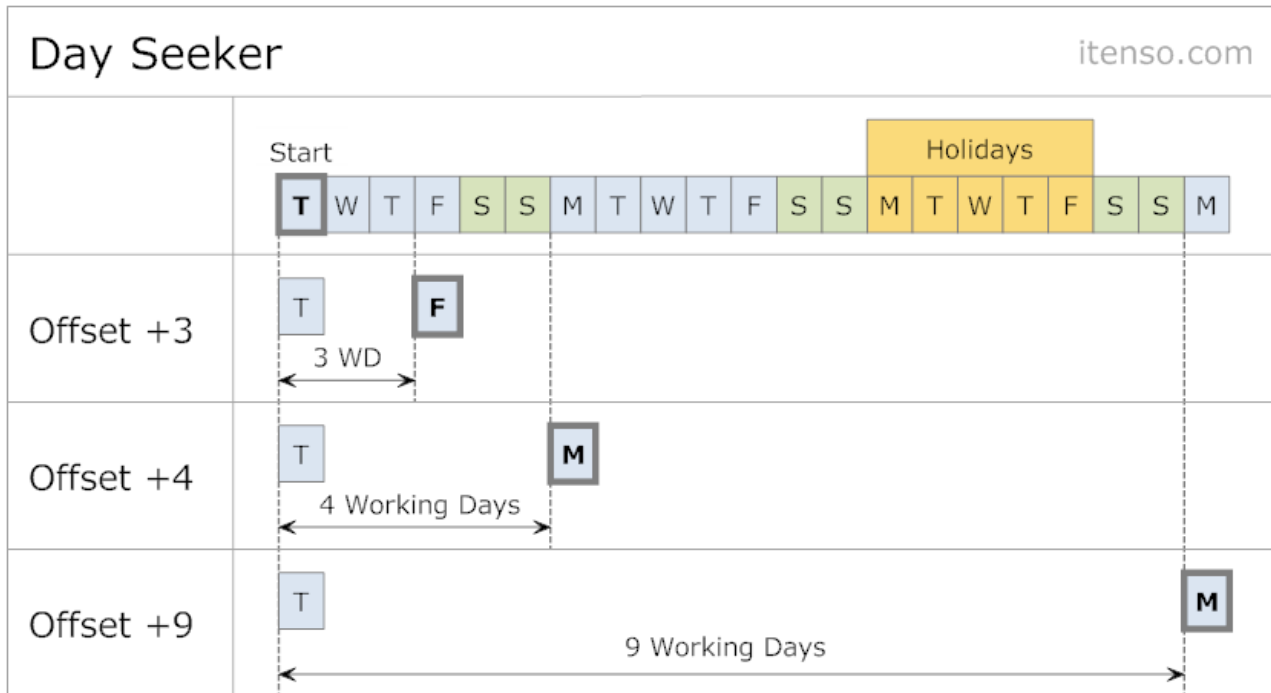
    CalendarTimeRange testPeriod = new CalendarTimeRange( new DateTime( 2010, 1, 1 ), new DateTime( 2011, 12, 31 ) );
    Console.WriteLine( "Calendar period collector of period: " + testPeriod );
    // > Calendar period collector of period: 01.01.2010 00:00:00 - 30.12.2011 23:59:59 | 728.23:59

    CalendarPeriodCollector collector = new CalendarPeriodCollector( filter, testPeriod );
    collector.CollectHours();
    foreach ( ITimePeriod period in collector.Periods )
    {
        Console.WriteLine( "Period: " + period );
    }
    // > Period: 01.01.2010; 08:00 - 17:59 | 0.09:59
    // > Period: 08.01.2010; 08:00 - 17:59 | 0.09:59
    // > Period: 15.01.2010; 08:00 - 17:59 | 0.09:59
    // > Period: 22.01.2010; 08:00 - 17:59 | 0.09:59
    // > Period: 29.01.2010; 08:00 - 17:59 | 0.09:59
    // > Period: 07.01.2011; 08:00 - 17:59 | 0.09:59
    // > Period: 14.01.2011; 08:00 - 17:59 | 0.09:59
    // > Period: 21.01.2011; 08:00 - 17:59 | 0.09:59
    // > Period: 28.01.2011; 08:00 - 17:59 | 0.09:59
} // CalendarPeriodCollectorSample
```

Suche nach Tagen

In der Praxis gibt es oft die Situation, dass anhand einer vorgegebenen Anzahl Arbeitstage, der nächste verfügbare Arbeitstag zu ermitteln ist. Bei der Zählung sollen Wochenende, Feiertage, Service- und Wartungsperioden sollen ausgeschlossen werden.

Zur Bewältigung dieser Aufgabe steht der **DaySeeker** zur Verfügung. Wie beim **CalendarPeriodCollector** kann auch diese Klasse mit vordefinierten Filtern gesteuert werden. Das folgende Beispiel zeigt die Suche nach Arbeitstagen, wobei alle Wochenende und Ferien übersprungen werden:



Die Implementierung dieses Beispiels sieht wie folgt aus:

```
// -----  
public void DaySeekerSample()  
{  
    Day start = new Day( new DateTime( 2011, 2, 15 ) );  
    Console.WriteLine( "DaySeeker Start: " + start );  
    // > DaySeeker Start: Dienstag; 15.02.2011 | 0.23:59  
  
    CalendarVisitorFilter filter = new CalendarVisitorFilter();  
    filter.AddWorkingWeekDays(); // only working days  
    filter.ExcludePeriods.Add( new Week( 2011, 9 ) ); // week #9  
    Console.WriteLine( "DaySeeker Holidays: " + filter.ExcludePeriods[ 0 ] );  
    // > DaySeeker Holidays: w/c 9 2011; 28.02.2011 - 06.03.2011 | 6.23:59  
  
    DaySeeker daySeeker = new DaySeeker( filter );  
    Day day1 = daySeeker.FindDay( start, 3 ); // same working week  
    Console.WriteLine( "DaySeeker(3): " + day1 );  
    // > DaySeeker(3): Freitag; 18.02.2011 | 0.23:59  
  
    Day day2 = daySeeker.FindDay( start, 4 ); // Saturday -> next Monday  
    Console.WriteLine( "DaySeeker(4): " + day2 );  
    // > DaySeeker(4): Montag; 21.02.2011 | 0.23:59  
  
    Day day3 = daySeeker.FindDay( start, 9 ); // Holidays -> next Monday  
    Console.WriteLine( "DaySeeker(9): " + day3 );  
    // > DaySeeker(9): Montag; 07.03.2011 | 0.23:59  
}  
// DaySeekerSample
```

Environmental Elements

Zeitdefinitionen und grundlegenden Berechnung sind in verschiedene Hilfsklassen aufgeteilt:

TimeSpec

Zeit- und Periodenkonstanten

YearHalfyear/

Aufzählungen für Halbjahre, Quartale, Monate und Wochentypen

YearQuarter/

YearMonth/

<code>YearWeekType</code>	
<code>TimeTool</code>	Operationen zur Bearbeitung von Datum- und Zeitwerten sowie für vordefinierte Zeitperioden
<code>TimeCompare</code>	Vergleichsfunktionen für Zeitperioden
<code>TimeFormatter</code>	Formattierung der Zeitperioden
<code>TimeTrim</code>	Stutzfunktionen für Zeitperioden
<code>Now</code>	Ermittlung des momentanen Zeitpunkt für verschiedene Zeitperioden wie z.B. der Startzeitpunkt des aktuellen Kalenderquartals
<code>Duration</code>	Ermittlung vordefinierter Zeiträume
<code>Time</code>	Die Zeitwerte eines Moments (<code>DateTime</code>)
<code>DateTimeSet</code>	Sortierte Liste von eindeutigen Zeitpunkten
<code>CalendarVisitor</code>	Abstrakte Basisklasse zur Iteration über Kalenderperioden
<code>TimeLine</code>	Berechnungstool für das Trennen und Zusammenführen von Zeiträumen

Bibliothek und Unit Tests

Die Bibliothek **Time Period** steht in drei Versionen zur Verfügungen:

- Library für .NET 2.0 inklusive Unit Tests
- Library für .NET for Silverlight 4
- Library für .NET for Windows Phone 7

Für die meisten Klassen bestehen NUnit Tests. Der Source Code ist bei allen Varianten derselbe (siehe weiter unten Composite Library Development), wobei die Unit Tests nur in der Bibliothek mit dem vollständigen .NET Framework verfügbar sind.

Die Erstellung von Tests für zeitbasierende Funktionen ist kein leichter Task, wirken sich doch verschiedene Faktoren auf den Zustand der Testobjekte aus:

- Verschiedene `Cultures` besitzen unterschiedliche Kalender
- Funktionen welche sich auf `DateTime.Now` beziehen, können bei verschiedenen Ausführungszeitpunkten zu abweichenden Testergebnissen führen
- Zeitberechnung – insbesondere mit Zeitperioden – führen zu einer Vielzahl von Sonderfällen

Es ist nicht verwunderlich, dass der Codeumfang der Unit Tests fast das Dreifache gegenüber der Bibliothek ist.