Временные ряды и расчет зарплаты

Оглавление

[Аннотация 2](#_Toc443645646)

[Область применения 2](#_Toc443645647)

[Теория 3](#_Toc443645648)

[Терминология 3](#_Toc443645649)

[Период 4](#_Toc443645650)

[Временной ряд 5](#_Toc443645651)

[Практика 8](#_Toc443645652)

[Библиотека TimeLines 8](#_Toc443645653)

[Библиотека TimeLines.iCalGenerator 9](#_Toc443645654)

[Потокобезопасность 9](#_Toc443645655)

[Эффективность 9](#_Toc443645656)

[Документирование 10](#_Toc443645657)

[Примеры использования 10](#_Toc443645658)

[Модель расчета зарплаты 10](#_Toc443645659)

[Генерация графиков работы 11](#_Toc443645660)

[Генерация выходных 12](#_Toc443645661)

[Основной расчет 12](#_Toc443645662)

[Маркировка периодов фактически отработанного времени 13](#_Toc443645663)

[Работа в ночное время 14](#_Toc443645664)

[Подготовка данных для расчета 16](#_Toc443645665)

[Расчет начислений 17](#_Toc443645666)

[Конкуренция 19](#_Toc443645667)

[Заключение 20](#_Toc443645668)

# Аннотация

В настоящей статье описывается авторская библиотека для работы с временными рядами.

Библиотека разработана на C# и рассчитана на использование .Net Framework 4.0 и выше.

# Область применения

Область применения описываемой библиотеки – программирование различных задач обработки распределенных по времени данных, начиная c анализа показателей фондовых рынков и заканчивая расчетом заработной платы.

Именно расчет зарплаты послужил поводом для разработки описываемой библиотеки, и именно эта задача используется в настоящей статье для демонстрации возможностей временных рядов: описываются подходы к проектированию, рассматривается возможная реализация отдельных алгоритмов.

В расчете заработной платы есть всего два момента, представляющих некоторые алгоритмические трудности:

1. идентификация (маркировка) периодов фактически отработанного времени;
2. расчет оплаты по отдельным видам начислений при изменении в пределах расчетного периода каких-либо исходных параметров, например, оклада, графиков работы (и, как следствие, нормы времени), различных доплат, начисляемых пропорционально отработанному времени (вредные условия труда, замещение, совмещение профессий, увеличенный объем работы и пр.).

Первый пункт тесно связан с так называемым «табелированием» и (упрощенно) сводится к разбивке всего отработанного времени на отдельные периоды, соответствующие плановому времени работы, сверхурочным часам и работе в выходные/праздничные дни. Как правило, разработчики такой процесс стараются не автоматизировать, а отдают на откуп специально обученному пользователю (табельщику), от точности и правильности работы которого в конечном итоге и зависит корректность расчета зарплаты. Причина такого подхода кроется скорее всего в том, что при табелировании необходимо учитывать множество дополнительных, возможно, изменяющихся во времени, факторов: графики работы, система учета отработанного времени, предоставление дополнительных выходных и пр.

Вторая проблема не является чисто алгоритмической – это скорее проблема большого количества кода, который требуется для того, чтобы учесть все флюктуации входных данных. Здесь помогают как общие принципы борьбы со сложностью (декомпозиция, модульное тестирование и пр.), так и специфические предметно-ориентированные решения, например, расчет стоимости единицы времени с последующей разбивкой отработанного времени на промежутки с константным ее значением.

На самом деле обе указанные проблемы по своей сути связаны с необходимостью обработки меняющихся во времени данных и легко решаются с помощью описываемой в статье библиотеки временных рядов. На снимке экрана ниже представлена работа демонстрационной программы, использующей эту библиотеку и поставляемой автором в комплекте вместе с ней:

Хотя концепция временных рядов может достаточно легко восприниматься на интуитивном уровне, для лучшего понимания последующих примеров целесообразно рассмотреть некоторые теоретические основы.

# Теория

## Терминология

В рассматриваемой библиотеке используются два основных понятия:

1. период;
2. временной ряд.

Термин «период» интуитивно понятен и семантически соответствует определению [википедии](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B8%D0%BE%D0%B4).

Термин «временные ряды», используемый в рассматриваемой библиотеке, отличается от [трактовки математической статистики](https://ru.wikipedia.org/wiki/%D0%92%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%BE%D0%B9_%D1%80%D1%8F%D0%B4). Можно было бы использовать термины «списки периодов» или «периодические списки», но первый не отражает упорядоченности периодов, а второй больше запутывает, чем объясняет, особенно если заглянуть в толковый словарь для уточнения смысла слова «[периодический](http://dic.academic.ru/dic.nsf/efremova/213040/%D0%9F%D0%B5%D1%80%D0%B8%D0%BE%D0%B4%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9)». Автор статьи будет благодарен, если кто-то сможет предложить более точное название. Пока же будем использовать термин «временные ряды» со следующей трактовкой:

*Временной ряд – набор (возможно, пустой) неперекрывающихся периодов, упорядоченных по времени от более ранних к более поздним (конец предыдущего периода меньше или равен началу последующего).*

Примечание: Один из рецензентов статьи, исходя из определения математического аналога для периода (см. ниже), предложил неплохой термин для обозначения временных рядов – «последовательность интервалов». Но в силу устоявшегося восприятия, по крайней мере, у автора и его коллег существующего термина «временные ряды», изменения ни в настоящей статье, ни в исходном коде самой библиотеки и поставляемых с ней примеров не делались.

## Период

Математический аналог периода: полуинтервал вида **[begin, end)**, где **begin** – начало периода, а **end** – конец периода (см. [статью в википедии](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%BC%D0%B5%D0%B6%D1%83%D1%82%D0%BE%D0%BA_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)#.D0.A2.D0.B8.D0.BF.D1.8B_.D0.BF.D1.80.D0.BE.D0.BC.D0.B5.D0.B6.D1.83.D1.82.D0.BA.D0.BE.D0.B2)[)](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%BC%D0%B5%D0%B6%D1%83%D1%82%D0%BE%D0%BA_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)). Т.е. начало периода – включительно, конец периода – исключительно.

Особенности периода в контексте рассматриваемой библиотеки:

* периоды вида **[start, end)**, где **start >= end**, считаются ошибочными и не используются;
* период вида **[DateTime.MinValue, DateTime.MaxValue)** считается бесконечным, т.е. эквивалентен **(-∞, +∞)**;
* период вида **[DateTime.MinValue, end)** считается бесконечным снизу, т.е. эквивалентен   
  **(-∞,end)** – имеет конец, но не имеет начала;
* период вида **[start, DateTime.MaxValue)** считается бесконечным сверху, т.е. эквивалентен **[start, +∞)** – имеет начало, но не имеет конца;
* с периодом могут быть ассоциированы произвольные данные; тип ассоциированных данных определяет тип периода (если данные не определены, то период не типизирован).

Из-за того, что время представляется в виде стандартной структуры DateTime, возникают соответствующие ограничения:

* максимальный диапазон времени ограничен значениями **DateTime.MinValue, DateTime.MaxValue**;
* обрабатываемое время дискретно с точностью 100 наносекунд.

Основные операции с периодами:

* определение длительности периода;
* изменение границ периода;
* смещение периода как разновидность изменения границ, когда начало и конец периода изменяются на одинаковую величину, т.е. длительность периода остается неизменной;
* вычитание из периода одного или нескольких других периодов.

Последняя операция может быть проиллюстрирована следующими рисунками (везде C = A – B):

|  |  |
| --- | --- |
|  |  |
|  |  |

## Временной ряд

В соответствии с [определением](#Определение_ряда) временной ряд является набором периодов и обладает следующими свойствами:

* временной ряд содержит 0..N периодов;
* периоды в контексте временного ряда упорядочены по времени от более ранних к более поздним;
* периоды в контексте временного ряда не перекрываются;
* с периодами могут быть ассоциированы данные разного типа, т.е. периоды могут быть разного типа.

Все операции с временными рядами формируют новый временной ряд, при этом ряды, участвующие в операциях в качестве операндов, не изменяются.

Основные операции с одним временным рядом (аналог унарных операций):

* привязка к периодам временного ряда ассоциированных данных;
* слияние сочлененных периодов A -> B:
* смещение периодов B = A + delta:

Операция полностью обратима, т.е. позволяет получить исходный временной ряд без потери данных:

A -> B = A + delta -> A = B – delta

* реверс B = not A:

По умолчанию при выполнении операции ряд B содержит нетипизированные периоды, т.е. данные, ассоциированные с периодами исходного ряда, теряются.

Как следствие, операция частично обратима, т.е. позволяет получить временной ряд, периоды которого совпадают с исходным временным рядом, но без ассоциированных с периодами данных:

A -> B = not A -> A = not B

Основные операции с несколькими временными рядами (аналог бинарных операций):

* логическое сложение (OR) – C = A or B:

Варианты формирования ассоциированных данных в периодах результата C:

* + все периоды C простые, без ассоциированных данных;
  + периоды 1, 3, 4 сохраняют данные периодов A, период 2 – данные периодов B;
  + периоды 1, 4 сохраняют данные периодов A, периоды 2, 3 – данные периодов B;
  + периоды 1, 4 сохраняют данные периодов A, период 2 – данные периодов B, для периода 3 данные формируются с помощью пользовательской функции f(A, B).
* объединение временных рядов – C = (A, B):

Операция фактически является частным случаем логического сложения временных рядов (OR), за исключением того, что перекрытие периодов в результате C рассматривается как ошибка.

* логическое умножение (AND) – C = A and B:

Варианты формирования ассоциированных данных в периодах результата C:

* + все периоды C простые, без ассоциированных данных;
  + периоды 1, 2 сохраняют данные периодов A;
  + периоды 1, 2 сохраняют данные периодов B;
  + для периодов 1, 2 данные формируются с помощью пользовательской функции   
    f(A, B).
* вычитание – C = A - B:
* пользовательская обработка периодов временных рядов – C = {A, B}:

Наиболее универсальная операция, использующая пользовательскую функцию для формирования результатов на каждом из участков C1..C6, причем для каждого участка может формироваться 0..N периодов.

Все операции с несколькими временными рядами, за исключением двух последних, [коммутативны](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BC%D1%83%D1%82%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F) и [ассоциативны](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%BE%D1%86%D0%B8%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F), операция вычитания – только [ассоциативна](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%BE%D1%86%D0%B8%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F).

Дополнительная функциональность, упрощающая работу с временными рядами, – их генерация по различным правилам.

# Практика

## Библиотека TimeLines

Основная функциональность временных рядов реализована в виде библиотеки классов TimeLines:

Интерфейсы IPeriod и IPeriod<T> описывают нетипизированный и типизированный период соответственно.

Классы Period и Period<T> являются встроенными реализациями нетипизированного и типизированного периодов соответственно.

Класс PeriodComparerByBegin является простой реализацией сравнения периодов по их началу без учета ассоциированных данных.

Класс PeriodComparer является простой реализацией сравнения периодов по их границам без учета ассоциированных данных.

Статический класс PeriodUtils содержит методы расширения для интерфейса IPeriod, реализующие основные операции с периодами.

Статический класс TimeLineUtils содержит методы расширения для интерфейса IEnumerable<IPeriod>, реализующие основные операции с временными рядами.

Класс ListTimeLine является простой реализацией временного ряда в виде списка. В общем случае его использование не является необходимым, т.к. все операции с временными рядами выполняются в классе TimeLineUtils с интерфейсами IEnumerable<IPeriod>, т.е. в качестве временного ряда может выступать и список, и массив, и очередь, и т.д.

Класс TimeLine является легковесной оболочкой для временных рядов, реализующей наиболее часто используемые действия в виде операторов.

Класс Generator реализует некоторые методы генерации временных рядов по различным правилам.

## Библиотека TimeLines.iCalGenerator

Содержит класс Generator, реализующий генерацию временных рядов на основе календарей в формате iCalendar (.ics, .ical), который поддерживается большинством программ-календарей, например, Microsoft Outlook, Google Календарь, Apple iCal и т.д. Описание формата iCalendar представлено в RFC-5545.

Для реализации генератора используется свободно распространяемая библиотека [DDay.iCal](http://sourceforge.net/projects/dday-ical/).

## Потокобезопасность

Классы Period, Period<T>, PeriodComparer, PeriodComparerByBegin, ListTimeLine и TimeLine не являются потокобезопасными.

При использовании потокобезопасных реализаций IPeriod, IPeriod<T> и IEnumerable<IPeriod> методы расширений в классах PeriodUtils и TimeLineUtils, которые и реализуют основную функциональность временных рядов, становятся потокобезопасными.

## Эффективность

Реализация временных рядов выполнена в основном с использованием LINQ, что обеспечивает достаточно высокое быстродействие кода и эффективность использования памяти.

Все операции с временными рядами возвращают в качестве результата временные ряды в формате IEnumerable<IPeriod>, что позволяет использовать их в каскадных операциях, которые хорошо оптимизируются ядром LINQ. Это также дает дополнительный выигрыш в быстродействии.

Относительно медленным является создание новых периодов внутри различных операций, т.к. при этом может использоваться механизм рефлексии (см. метод PeriodUtils.CreateAnalogue). Для уменьшения потерь производительности применяются следующие меры:

1. для всех методов, требующих копирования периодов, предусмотрен параметр для передачи пользовательской функции копирования и лишь при его значении, равном null, используется метод PeriodUtils.CreateAnalogue. (см. например TimeLineUtils.Subtract);
2. если исходный период реализует интерфейс ICloneable (как например встроенные в библиотеку периоды Period и Period<T>), копирование периода внутри метода PeriodUtils.CreateAnalogue выполняется с помощью метода Clone этого интерфейса.

На всех методах класса TimeLineUtils, возвращающих временные ряды и являющихся по сути именованными итераторами, оптимизирована проверка входных параметров: реальная проверка выполняется только один раз для всего итератора (пример – TimeLineUtils.Transform и TimeLineUtils.TransformInternal).

Кроме того, все методы, получающие на входе временные ряды, имеют дополнительный параметр checkTimeLine, который определяет, нужно ли при входе в метод проверять их на соответствие обязательным критериям временных рядов (упорядоченность периодов и отсутствие их перекрытий). Если разработчик уверен, что в его коде на вход соответствующего метода передается «правильный» временной ряд, то он может отключить дополнительную проверку, что также даст определенный выигрыш в быстродействии.

Принятые меры обеспечивают хорошую производительность временных рядов, которая подтверждается сравнительными тестами (см. раздел статьи [Конкуренция](#_Конкуренция)).

## Документирование

Отдельная документация для библиотек не составлялась, однако исходный код содержит комментарии, которых для использовавших временные ряды программистов пока было достаточно.

Вместе с библиотекой поставляются проекты, которые можно использовать в качестве примеров кодирования: демонстрационная программа (TimeLineTestApp), модульные (UnitTests) и сравнительные (ConsoleTests) тесты.

## Примеры использования

Рассмотренная выше концепция временных рядов может значительно упростить разработку различных задач, в частности, расчет зарплаты для повременной системы оплаты труда.

### Модель расчета зарплаты

Наше действующее законодательство в сфере оплаты труда требует от работодателя соблюдения достаточно общих принципов, смещая конкретику на уровень коллективных или персональных трудовых договоров. Так, например, работа в ночное время согласно Трудовому Кодексу РФ (Статья 154) «оплачивается в повышенном размере по сравнению с работой в нормальных условиях», но конкретный размер доплаты не указан. Более того, поскольку работодатель может использовать любые «модификации» оплаты труда, лишь бы они не ухудшали условия работников относительно действующего законодательства, то на практике встречаются очень разные варианты, например, «повышенная оплата за работу в ночное время рассчитывается на всю смену, если более половины смены приходится на ночное время», или еще запутаннее.

Как следствие, универсальную модель расчета зарплаты построить довольно сложно, если вообще практически возможно. Тем не менее, можно определить некоторую схему, следуя которой модель расчета зарплаты для конкретной организации можно построить достаточно просто.

Расчет зарплаты в терминах временных рядов сводится к расчету на отдельных периодах, которые получаются как результат пересечения нескольких исходных временных рядов:

* фактически отработанное время;
* изменение графиков работы;
* изменение оклада;
* другие временные ряды.

Некоторые исходные временные ряды (например, фактически отработанное время), состоят из нетипизированных периодов, другие содержат специфические данные (например, ряд изменений оклада – размер оклада на определенный период времени).

Конкретный перечень исходных временных рядов определяется с учетом специфики расчета зарплаты в конкретной организации.

Так, например, если в организации используется и обычный, и суммированный учет рабочего времени, то для расчета оплаты сверхурочной работы потребуется временной ряд изменений системы учета рабочего времени, т.к. она влияет в частности на расчет оплаты сверхурочных.

Или если, например, в организации предусмотрена доплата за замещение должностей (например, по формуле «за каждый день замещения в размере 50% от разницы в окладах»), то потребуется временной ряд, отражающий такие замещения.

При декомпозиции расчета на отдельные алгоритмы может быть полезным формирование промежуточных временных рядов.

Так, например, периоды временного ряда изменений графиков работы скорее всего будут содержать ссылки на соответствующие графики. Для использования же в расчетных алгоритмах удобнее использовать производный ряд «Плановое времени работы»:

Окончательный расчет зарплаты целесообразно выполнять в два этапа: сначала рассчитать в виде временного ряда повышающие коэффициенты на всех периодах отработанного времени, а суммы оплат рассчитать, как пересечение с временным рядом изменения оклада – ведь оклад тоже может изменяться.

В качестве примера далее рассмотрим реализацию отдельных алгоритмов.

### Генерация графиков работы

// 2-я смена – с 8:00 01.11.2015

IEnumerable<IPeriod> Shedule12\_2(DateTime monthPeriodStart, DateTime monthPeriodEnd)

{

return TimeLineUtils.Generate(

monthPeriodStart,

monthPeriodEnd,

TimeSpan.FromHours(12),

TimeSpan.FromHours(36)

);

}

Здесь все просто: генерируем временной ряд, первый период которого начинается с monthPeriodStart, последний период начинается не позже monthPeriodEnd, длительность периодов 12 часов, промежуток между началом текущего и началом следующего периода 36 часов.

### Генерация выходных

IEnumerable<IPeriod> GenerateWeekends(DateTime monthPeriodStart, DateTime monthPeriodEnd)

{

return TimeLineUtils.Generate(

monthPeriodStart,

monthPeriodEnd,

TimeSpan.FromDays(1),

TimeSpan.FromDays(1),

start => start.DayOfWeek == DayOfWeek.Saturday || start.DayOfWeek == DayOfWeek.Sunday

);

}

Этот пример аналогичен предыдущему, просто добавлено лямбда-выражение – дополнительный фильтр для исключения из результата отдельных генерируемых периодов.

### Основной расчет

#### Описание алгоритма

Последовательность расчета зарплаты (по одному работнику):

* с учетом изменений графиков работы построить плановые периоды работы;
* маркировать периоды фактически отработанного времени;
* на маркированных периодах: подготовить данные для расчета и рассчитать:
  + нормальные периоды;
  + доплату за работу в ночное время;
  + оплату сверхурочных;
  + оплату за работу в выходные/праздничные дни.

Принцип построения плановых периодов работы описан [выше](#PlanPeriodsBuilding) в качестве примера при рассмотрении модели расчета зарплаты.

Далее более подробно рассмотрен [алгоритм определения ночных часов](#_Работа_в_ночное) и другие алгоритмы основного расчета:

* [маркировка периодов фактически отработанного времени](#_Маркировка_периодов_фактически);
* [подготовка данных для расчета](#_Подготовка_данных_для);
* [расчет начислений](#_Расчет_начислений).

Необходимо отметить, что поскольку и подготовка данных для расчета, и окончательный расчет начислений выполняются на предварительно маркированных периодах, фактически можно совместить все указанные алгоритмы в одном, т.е. выполнять расчет начислений по ходу маркировки периодов.

Конкретные компоновочные решения зависят как от технических факторов (удобство кодирования, тестирования и отладки), так и от специфики предметной области. Так, в приведенном примере выбрана именно такая компоновка действий, потому что:

* доплата за работу в ночное время в бухгалтерском учете проходит как отдельный вид оплат;
* необходимо визуально отдельно отображать нормальные, ночные, сверхурочные и праздничные периоды;
* на отображаемых периодах должны выводиться расчетные данные.

#### Реализация алгоритма

/// <summary>

/// Основной расчет зарплаты

/// </summary>

public void Calculate()

{

// построить плановое время

var planTime = SheduleChanges.Transform(

p => TimeLineUtils.And(p.ToTimeLine(), (p as ShedulePeriod).Data));

// маркировка периодов фактически отработанного времени

var markPeriods = MarkPeriods\_2(WorkPeriods, planTime);

// подготовка данных для расчета - нормальные периоды

NormalPeriods = PrepareResultPeriods(PaymentFactors.Normal, markPeriods.Where(

p => (p as TimeLines.Period<PeriodType>).Data == PeriodType.Staff));

// расчет начислений - нормальные периоды

NormalPayment = CalculatePayment(NormalPeriods);

// подготовка данных для расчета - доплата за работу в ночное время

NightPeriods = PrepareResultPeriods(PaymentFactors.Night,

GetNightPeriods\_2(WorkPeriods, planTime, NightHours));

// расчет начислений - доплата за работу в ночное время

NightPayment = CalculatePayment(NightPeriods);

// подготовка данных для расчета - сверхурочные

OvertimePeriods = PrepareResultPeriods(PaymentFactors.Overtime, markPeriods.Where(

p => (p as TimeLines.Period<PeriodType>).Data == PeriodType.Overtime));

// расчет начислений - сверхурочные

OvertimePayment = CalculatePayment(OvertimePeriods);

// подготовка данных для расчета - работа в выходные/праздничные дни

FreedaysPeriods = PrepareResultPeriods(PaymentFactors.Freedays, markPeriods.Where(

p => (p as TimeLines.Period<PeriodType>).Data == PeriodType.Freedays));

// расчет начислений - работа в выходные/праздничные дни

FreedaysPayment = CalculatePayment(FreedaysPeriods);

}

### Маркировка периодов фактически отработанного времени

#### Описание алгоритма

Для корректного расчета размера оплаты фактически отработанное за расчетный период время необходимо разбить на 3 группы:

1. время, отработанное согласно графику работы – обычная оплата;
2. сверхурочная работа – оплачивается за первые два часа работы в полуторном размере, за последующие часы в двойном размере; если используется суммированный учет рабочего времени, то время сверхурочной работы суммируется за расчетный период, иначе берется за тот день, когда была переработка;
3. работа в выходной или нерабочий праздничный день – оплачивается в размере одинарной дневной или часовой ставки (части оклада (должностного оклада) за день или час работы) сверх оклада (должностного оклада), если работа в выходной или нерабочий праздничный день производилась в пределах месячной нормы рабочего времени, и в размере двойной дневной или часовой ставки (части оклада (должностного оклада) за день или час работы) сверх оклада (должностного оклада), если работа производилась сверх месячной нормы рабочего времени.

Перефразируя:

1. стандартное время = реально отработанное время, совпадающее с плановым временем работы; или

Staff = realTime & planTime

1. сверхурочное время = реально отработанное время, совпадающее с рабочими днями и неплановым временем работы; или

Overtime = realTime & workDays & !planTime

1. работа в выходные = реально отработанное время, совпадающее с нерабочими днями, или

Freedays = realTime & !workDays

Примечание: Рассматриваемый пример упрощен, поэтому расчет коэффициентов для периодов второго и третьего типа выполняться не будет.

Понятие «рабочие дни» является производным от планового времени работы и отражает дни, в которые попадает хотя бы один период планового времени работы.

Поскольку полученные периоды между собой пересекаться не могут, можно составить из них общий временной ряд, чтобы, например, отобразить его пользователю в виде одного графика.

#### Реализация алгоритма

/// <summary>

/// Маркировка периодов реально отработанного времени в соответствии с плановым временем работы

/// </summary>

/// <param name="realTime">реально отработанное время</param>

/// <param name="planTime">плановое время работы</param>

/// <returns>[Period<PeriodType>]</returns>

TimeLine MarkPeriods(TimeLine realTime, TimeLine planTime)

{

using (var workDays = new TimeLine(planTime.Periods.ToDays()))

return

(realTime & planTime) \* PeriodType.Staff +

(realTime & !workDays) \* PeriodType.Freedays +

(realTime & workDays & !planTime) \* PeriodType.Overtime;

}

То же самое, функциональная форма без использования класса TimeLine:

IEnumerable<IPeriod> MarkPeriods\_2(IEnumerable<IPeriod> realTime, IEnumerable<IPeriod> planTime)

{

using (var workDays = new TimeLine(planTime.Periods.ToDays()))

{

return new TimeLines.TimeLine(TimeLineUtils.Join(

TimeLineUtils.And(realTime, planTime).SetData(PeriodType.Staff),

TimeLineUtils.And(realTime, workDays.Not()).SetData(PeriodType.Freedays),

TimeLineUtils.And(realTime, workDays, planTime.Not()).SetData(PeriodType.Overtime)));

}

}

### Работа в ночное время

#### Описание алгоритма

Доплата рассчитывается только для отработанного времени, совпадающего с плановым графиком работы.

#### Реализация алгоритма (вариант 1)

На входе – временные ряды маркированных периодов (см. [выше](#_Маркировка_периодов_фактически)) и ночного времени (периоды с 22:00 до 6:00 каждый день расчетного периода):

/// <summary>

/// Получить ночные периоды

/// </summary>

/// <param name="markedPeriods">временной ряд [Period<PeriodType>] маркированных периодов</param>

/// <param name="nightTime">ночное время</param>

/// <returns></returns>

TimeLine GetNightPeriods\_1(TimeLine markedPeriods, TimeLine nightTime)

{

return

nightTime &

from p in markedPeriods.Periods.Cast<IPeriod<PeriodType>>()

where p.Data == PeriodType.Staff

select p;

}

То же самое, функциональная форма без использования класса TimeLine:

IEnumerable<IPeriod> GetNightPeriods\_1(IEnumerable<IPeriod> markedPeriods, IEnumerable<IPeriod> nightTime)

{

return TimeLineUtils.And(

nighttime,

from p in markedPeriods.Periods.Cast<IPeriod<PeriodType>>()

where p.Data == PeriodType.Staff

select p);

}

Или нагляднее, но чуть медленнее из-за dynamic:

TimeLine GetNightPeriods\_1(TimeLine markedPeriods, TimeLine nightTime)

{

return

nightTime &

from p in markedPeriods.Periods

where ((dynamic)p).Data == PeriodType.Staff

select p;

}

То же самое, функциональная форма без использования класса TimeLine:

IEnumerable<IPeriod> GetNightPeriods\_1(IEnumerable<IPeriod> markedPeriods, IEnumerable<IPeriod> nightTime)

{

return TimeLineUtils.And(

nighttime,

from p in markedPeriods.Periods

where ((dynamic)p).Data == PeriodType.Staff

select p);

}

#### Реализация алгоритма (вариант 2)

Плюс предыдущей реализации алгоритма – эффективность. Но есть недостаток – для понимания алгоритма требуются пояснение что является «стандартным» временем.

Если отказаться от использования предварительно маркированных периодов, то с некоторой несущественной потерей эффективности (из-за повторной обработки «сырых» периодов фактически отработанного времени) можно повысить наглядность алгоритма до его интуитивной прозрачности:

/// <summary>

/// Получить ночные периоды

/// </summary>

/// <param name="realTime">реально отработанное время</param>

/// <param name="planTime">плановое время работы</param>

/// <param name="nightTime">ночное время</param>

/// <returns></returns>

TimeLine GetNightPeriods\_2(TimeLine realTime, TimeLine planTime, TimeLine nightTime)

{

return realTime & planTime & nightTime;

}

То же самое, функциональная форма без использования класса TimeLine:

IEnumerable<IPeriod> GetNightPeriods\_2(

IEnumerable<IPeriod> realTime,

IEnumerable<IPeriod> planTime,

IEnumerable<IPeriod> nightTime)

{

return TimeLineUtils.And(realTime, planTime, nightTime);

}

### Подготовка данных для расчета

#### Описание алгоритма

Подготовка данных для расчета заключается в ассоциировании с каждым периодом данных следующей структуры:

/// <summary>

/// Данные для расчета начислений

/// </summary>

public class PaymentInfo

{

/// <summary>

/// Повышающий расчетный коэффициент

/// </summary>

public decimal Factor { get; set; }

/// <summary>

/// Оклад

/// </summary>

public decimal Salary { get; set; }

/// <summary>

/// Норма времени, часы

/// </summary>

public double NormHours { get; set; }

}

Для определения оклада учитывается временной ряд изменений оклада, нормы времени – изменения графиков работы.

#### Реализация алгоритма

/// <summary>

/// Подготовка данных для расчета

/// </summary>

/// <param name="factor">повышающий коэффициент</param>

/// <param name="periods">периоды</param>

/// <returns>[PaymentInfoPeriod]</returns>

TimeLineTestApp.TimeLine PrepareResultPeriods(decimal factor, IEnumerable<IPeriod> periods)

{

return new TimeLineTestApp.TimeLine(

TimeLines.TimeLineUtils.And(

new IEnumerable<IPeriod>[] { Salaries, SheduleChanges, periods },

(start, end, pp) => new PaymentInfoPeriod

{

Start = start,

End = end,

Data = new PaymentInfo

{

Factor = factor,

Salary = (pp[0] as SalaryPeriod).Data,

NormHours =

(pp[1] as ShedulePeriod).Data.SummaryDuration.TotalHours

}

}));

}

### Расчет начислений

Итак, после [подготовки расчета](#_Подготовка_данных_для_1) ассоциированные с периодами данные позволяют легко рассчитать начисления на каждом периоде:

/// <summary>

/// Период для отображения результатов расчета

/// </summary>

public class PaymentInfoPeriod : Period<PaymentInfo>

{

/// <summary>

/// Начислено за период

/// </summary>

public decimal Payment

{

get

{

return Data.Factor \* Data.Salary \*

(decimal)(this.Duration.TotalHours / Data.NormHours);

}

}

}

Самое простое для получения конечного результата – просуммировать полученные результаты по всем периодам:

/// <summary>

/// Расчет начислений - суммирование начислений по всем периодам

/// </summary>

/// <param name="resultTimeLine">[PaymentInfoPeriod]</param>

/// <returns></returns>

decimal CalculatePayment\_1(IEnumerable<IPeriod> resultTimeLine)

{

return resultTimeLine.Sum(p => (p as PaymentInfoPeriod).Payment);

}

Преимущество такого алгоритма – его простота. Однако с учетом большого объема вычислений при операциях с отдельными периодами «ручная» проверка результатов будет выполняться скорее всего по-другому, например, по каждому коду оплаты будет суммироваться длительность всех соответствующих периодов (*длительность\_периодов\_по\_коду*), а затем результат будет считаться по формуле:

*оплата\_по\_коду = длительность\_периодов\_по\_коду \* оклад / норма\_времени*

И здесь можно получить неожиданное расхождение результатов, например, в случае такого расчета при вычислениях с точностью 4 знака после запятой:

Расчетный месяц – декабрь 2015 года.

Норма времени при 40-часовой неделе – 183 часа (см. [производственный календарь на 2015 год](http://www.consultant.ru/law/ref/calendar/proizvodstvennye/2015/?utm_campaign=popular&utm_source=ya.direct&utm_medium=cpc&utm_content=calendar&yclid=5954172287136281463)).

Оклад – 20 000 руб.

Стоимость часа (для расчета по периодам) – 109,2896

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Отработано | | Расчет по периодам | | Расчет по общей длительности | | Разность |
| дни | часы | сумма | округление до копеек | сумма | округление до копеек |
| … |  |  |  |  |  |  |
| 8 | 64 | 6994,5344 | 6994,53 | 6994,5355 | 6994,54 | 0,01 |
| … |  |  |  |  |  |  |
| 11 | 88 | 9617,4848 | 9617,48 | 9617,4863 | 9617,49 | 0,01 |
| … |  |  |  |  |  |  |
| 21 | 168 | 18360,6528 | 18360,65 | 18360,6557 | 18360,66 | 0,01 |
| … |  |  |  |  |  |  |

Такое расхождение связано с ошибками округления при оперировании числами с ограниченной точностью, каковыми являются все числа в компьютере, в том числе вещественные. Дело в том, что операция деления с ограниченными по точности числами НЕ [дистрибутивна](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D1%81%D1%82%D1%80%D0%B8%D0%B1%D1%83%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D1%81%D1%82%D1%8C), т.е.:

*(y + z) / x ≠ y / x + z / x*

(левая часть неравенства соответствует «ручной» проверке расчетов, а правая – алгоритму расчета по периодам)

Причем ошибки округления возникают именно при расчете по периодам, т.е. при использовании предыдущего алгоритма расчета. Конечно, для компенсации этих ошибок можно использовать какой-либо алгоритм распределения – [такой](http://archival.ru/node/367), [такой](http://xpoint.ru/forums/programming/theory_algorythms/thread/36031.xhtml) или что-нибудь свое. Более того, на самом деле в практике расчета зарплаты точности decimal вполне достаточно, чтобы такого рода ошибки не возникали.

Тем не менее, вариант расчета по общей длительности периодов реализовать ненамного сложнее, чем суммирование результатов на периодах, зато он полностью тождественен алгоритму «ручной» проверки:

/// <summary>

/// Расчет начислений - суммирование длительности однотипных периодов, затем вычисление по их сумме

/// </summary>

/// <param name="resultTimeLine">[PaymentInfoPeriod]</param>

/// <returns></returns>

decimal CalculatePayment\_2(IEnumerable<IPeriod> resultTimeLine)

{

return

(

from PaymentInfoPeriod period in resultTimeLine

let data = period.Data

group period by new

{

Salary = data.Salary,

NormHours = data.NormHours,

Factor = data.Factor

} into g

select new

{

Salary = g.Key.Salary,

NormHours = g.Key.NormHours,

Factor = g.Key.Factor,

Duration = g.Sum(p => p.Duration().TotalHours)

}

).Sum(p => p.Factor \* p.Salary \* (decimal)(p.Duration / p.NormHours));

}

# Конкуренция

Автору не удалось найти полнофункциональный аналог для сравнительного тестирования. Что-то похожее – это [Time Period Library for .NET](http://www.codeproject.com/Articles/168662/Time-Period-Library-for-NET) – библиотека с сопровождающей ее подробной статьей, но, к сожалению, практически не документированным кодом. И хотя функциональность указанной библиотеки довольно запутана, в ней удалось найти некоторые аналоги операций с временными рядами. Результаты сравнительного тестирования отражены на скриншоте:

Эти же результаты представлены в таблице:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Функциональность** | **Time Period Library for .NET** | | **Временные ряды**  **(класс TimeLineUtils)** | | |
| **Класс** | **Время, мс** | **Метод** | **Время, мс** | **Улучшение, раз** |
| Вычисление разрывов между периодами (реверс) | TimeGapCalculator | 2941 | Not | 86 | 34,2 |
| Пересечение периодов (логическое сложение) | TimePeriodCombiner | 800 | Or | 15 | 53,3 |
| Пересечение периодов (логическое умножение) | TimePeriodIntersector | 787 | And | 14 | 56,2 |
| Вычитание периодов | TimePeriodSubtractor | 1697 | Subtract | 51 | 33,3 |
| Дополнительные операции с календарем | + | | - | | |
| Генерация временных рядов | - | | + | | |
| Привязка к периодам ассоциированных данных | - | | + | | |

Полученные результаты наглядно показывают преимущества библиотеки временных рядов как минимум в быстродействии. Практика же ее использования в реальных задачах подтвердила и удобство пользования.

# Заключение

В качестве заключения – спасибо, если осилили дочитать эту статью до конца.

Замечания, пожелания и предложения о доработках присылайте по адресу: [s.v.ivanov@bk.ru](mailto:s.v.ivanov@bk.ru) Иванову Сергею.