

# Описание API программы FinLab.MTS

---



версия 1.0.0.1 от 04.09.2010г.

## Содержание

1. Концепция программы FinLab.MTS. ....	4
2. Философия алгоритмов в FinLab.MTS. ....	5
3. Класс Algorithm.....	6
3.1. Интерфейс IAlgorithm.....	6
3.1.1. Свойства.....	6
3.1.2. Методы.....	7
3.2. Интерфейс TaskInfo.....	8
3.2.1. Свойства.....	8
3.3. Интерфейс Iinitializeable.....	9
3.3.1. Методы.....	9
3.4. Интерфейс IDisposable.....	9
3.4.1. Методы.....	9
3.5. Интерфейс IControlable.....	9
3.5.1. Методы.....	9
3.6. Защищенные (protected) свойства Algorithm.....	10
3.7. Защищенные (protected) методы Algorithm.....	11
3.7.1. Методы, доступные для переопределения (virtual).....	11
3.7.2. Вспомогательные методы.....	11
4. Как это работает.....	13
4.1. Инициализация алгоритма.....	13
4.2. Действия при обновлении стакана.....	13
4.3. Действия при сделке.....	14
5. Практическое руководство по созданию собственного алгоритма.....	14
5.1. Подготовка проекта.....	14
5.2. Пример алгоритма.....	15
5.3. Требования по разработке алгоритма.....	20
6. Приложения.....	20
6.1. Приложение №1. TaskParameterList<T>.....	20
6.2. Приложение №2. AddCommand.....	21
6.3. Приложение №3. ObjectAssociatedControl.....	22
6.4. Приложение №4. Informer.....	23
6.5. Приложение №5. ResultList.....	23
6.6. Приложение №6. IStakan.....	23
6.7. Приложение №7. Candellist.....	25
6.8. Приложение №8. TickList.....	25

6.8.1.	Общие свойства и методы класса .....	25
6.8.2.	Методы ускоренной выборки элементов .....	27
6.8.3.	Методы ускоренного вычисления .....	27
6.9.	Приложение №9. TaskMode .....	29
6.10.	Приложение №10. Trade .....	29

## 1. Концепция программы FinLab.MTS.

Программа FinLab.MTS позволяют пользователю свои собственные торговые стратегии, выходящие за рамки торговых стратегий, заложенных в FinLab.PairTrade.

Программа FinLab.MTS имеет специально разработанный для этих целей API-интерфейс.

Для понимания принципов работы программы FinLab.MTS необходимо рассмотреть два ключевых понятия:

- Пользовательская задача
- Алгоритм пользователя

**Пользовательская задача** – это задача (очень подробно о понятии «задача» рассказано в описании FinLab.PairTrade), в которой решения о выставлении и снятии заявок производятся согласно Алгоритму пользователя.

Пользовательская задача в общем случае рассчитана на парный трейдинг. В ней содержится «Первая нога» (фьючерс) и «Вторая нога» (спот).

Спекулятивный трейдинг в контексте пользовательской задачи – это трейдинг, при котором вторая нога или пустой стакан (Empty) или просто не учитывается.

Индексный трейдинг также возможен в том случае, когда спот является синтетическим финансовым инструментом, созданным из заготовки.

Ни в том, ни в другом случае, реализация алгоритма со стороны разработчика ни чем не отличается от общего случая. Более того, можно написать алгоритм, применимый одновременно для всех трех вариантов трейдинга.

**Пользовательская задача обладает всеми свойствами задачи как таковой. Ее можно запускать, останавливать, сохранять ее настройки в файл, загружать ее из файла, вносить в состав суперзадачи и т.д.**

**Алгоритм пользователя** в программе FinLab.MTS - это некая программная структура с четко определенным функционалом, предназначенная для анализа торговых данных и принятия торговых решений.

Никаких ограничений по количеству запускаемых пользовательских задач в программе нет. Все определяется производительностью конечного компьютера.

Пользовательская задача выступает посредником между алгоритмом пользователя и торговым терминалом. Пользовательская задача передает алгоритму пользователя **данные о котировках**, спрашивает алгоритм о **необходимости постановки или снятия заявки**, а так же информирует алгоритм о совершаемых задачей **сделках**.

Таким образом, при реализации своего алгоритма **пользователь избавляется от необходимости решать вопросы:**

- Совместимости с терминалом
- Обработки служебной информации

- Перекрытия открываемых позиций

А это означает, что:

- Любой алгоритм, написанный для программы FinLab.MTS будет работать с любым терминалом, поддерживаемым программой! Кроме того, такой алгоритм будет работать с любым финансовым инструментом на любой бирже.
- Алгоритм пользователя будет получать от программы только ту информацию, которая необходима для принятия торговых решений.

Более подробно о структуре алгоритма рассказано ниже.

## 2. Философия алгоритмов в FinLab.MTS.

С программной точки зрения алгоритм пользователя – это класс, наследующий абстрактный класс **Algorithm** из пространства имен **FinLab.MTS.Infinity** библиотеки FinLab.MTS.Infinity.dll.

При наследовании указанного класса пользователь должен переопределить ряд методов и свойств класса. Переопределяемые методы класса **Algorithm** отвечают за принятие решений о постановке или снятии заявок, а также обрабатывают данные о сделках, совершенных согласно пользовательского алгоритма.

Переопределяемые свойства класса **Algorithm** служат для отображения промежуточной расчётной информации в таблице главного окна программы.

Работа пользовательской задачи и алгоритма строится по следующему принципу:

**В любой момент времени в рынке может быть не более одной заявки на покупку и не более одной заявки на продажу.**

Этот принцип однозначно определяет последовательность работы пользовательской задачи:

- 1) При любом изменении на рынке пользовательская задача обращается к алгоритму
- 2) Если в рынке нет заявки на покупку, то пользовательская задача опрашивает алгоритм на предмет выставления новой заявки на покупку. Если алгоритм ничего не выдает в ответ, то программа ждет следующего изменения ситуации на рынке.
- 3) Если алгоритм выдает команду на выставление заявки, то программа передает эту команду в терминал и дожидается ответа терминала о результате выполнения этой команды.
- 4) Если команда не выполнена, и заявка не выставлена, то программа возвращается к пункту 1.
- 5) Если же команда выполнена, и заявка становится в очередь, то при следующем обновлении стакана задача опрашивает алгоритм на предмет снятие этой заявки. В случае отказа – задача возвращается в пункт 5, в случае необходимости снятия заявки – снимает заявку и возвращается в пункт 1.

- 6) Если заявка исполняется полностью, то задача возвращается к пункту 1.
- 7) Если происходит сделка, то задача оповещает алгоритм о совершенной сделке.

Процесс аналогичен для заявок на продажу.

Таким образом:

- задача никогда не спросит алгоритм о необходимости выставить заявку в каком-либо направлении, если в этом направлении уже существует активная неисполненная заявка.
- Задача никогда не спросит алгоритм о необходимости снять заявку, если заявок нет.
- Задача никогда не передаст алгоритму «чужие» заявки и сделки.

5 шагов для создания собственного торгового робота

- 1) Создать класс, наследующий абстрактный класс Algorithm;
- 2) Переопределить необходимые методы и свойства;
- 3) Скомпилировать dll-ку и скопировать ее в папку программы FinLab.MTS ;
- 4) Запустить программу и создать новую пользовательскую задачу;
- 5) Выбрать подготовленный класс из списка и начать работу.

### 3. Класс Algorithm

Как уже было сказано, это базовый класс. Он реализует несколько интерфейсов, а также имеет ряд собственных свойств и методов.

```
namespace FinLab.MTS.Infinity
{
    [Serializable]
    public abstract class Algorithm : IDisposable, ITaskInfo, IControlable, IAlgorithm,
    IInitializeable
    {...}
}
```

Возможность сохранять и загружать задачу предъявляет к алгоритму требование **сериализуемости**. Эти требования описаны в разделе 5.3. Подробнее о сериализации можно узнать на сайте Майкрософт.

<http://msdn.microsoft.com/en-us/library/ms950721.aspx>

#### 3.1. Интерфейс IAlgorithm

##### 3.1.1. Свойства

```
TaskParameterList<bool> BoolParameters { get; set; }
TaskParameterList<double> DoubleParameters { get; set; }
```

Это коллекции булевых и вещественных параметров. Эти коллекции уже определены в классе Algorithm и доступны пользователю. Эти коллекции предназначены для пользова-

телей, которые по каким-то причинам не хотят писать UI настроек своего алгоритма. Параметры, содержащиеся в этих коллекциях, отображаются в окне «Список параметров» настроек пользовательской задачи (Руководство пользователя FinLab-PairTrade, стр. 55). Более подробно о работе этих коллекций будет рассказано в приложении №1.

```
Informer Informer { get; }
```

Экземпляр класса `Informer`. Необходим для отображения на главной форме программы той информации, которая не может быть отображена в основной таблице, т.е. информации, специфической для каждого пользовательского алгоритма. По умолчанию уже инициализирован, но может быть использован пользователем по своему усмотрению. Подробнее об этом классе будет рассказано в приложении №4.

### 3.1.2. Методы

**Все методы, описанные в интерфейсе `IAlgorithm`, обязательны к переопределению.**

1) Методы, отвечающие за выставление заявок

```
AddCommand CheckBuyDirection(IStakan futures, IStakan spot);  
AddCommand CheckSellDirection(IStakan futures, IStakan spot);
```

Параметры:

- `futures` – стакан финансового инструмента, являющегося фьючерсом
- `spot` – стакан финансового инструмента, являющегося спотом

Возвращаемое значение: экземпляр класса **`AddCommand`**, содержащий в себе всю необходимую для выставления заявки информацию или **`null`**, если заявку выставить не надо.

В этих методах алгоритм пользователя анализирует ситуацию на рынке, и, если принимает решение о выставлении заявки – создает новый экземпляр `AddCommand`, заполняет его поля (цену, количество и пр.) и передает его пользовательской задаче в виде возвращаемого значения. Эта команда затем передается в терминал и служит для выставления заявки.

Оба эти метода имеют идентичное назначение с тем только исключением, что первый отвечает за покупку, а второй за продажу.

Пользовательская задача, получив команду, проверяет ее направление и объем (чтобы не купить и не продать лишнего), запоминает ее и транслирует ее в терминал.

**Внимание!** Заявки выставляются только по фьючерсу. Пользователю достаточно заполнить только цену, количество и тип заявки (рыночная или лимитированная). Остальные значения в любом случае будут подставлены пользовательской задачей.

**Внимание!** Алгоритм пользователя сам ответственен за округление цены заявки до минимального шага цены фьючерса!

Более подробно о классе `AddCommand` рассказано в приложении №2, об интерфейсе `IStakan` – в приложении №6.

2) Методы, отвечающие за снятие заявки

```
bool DeleteOrderBuy(AddCommand addCommand, IStakan futures, IStakan spot);
bool DeleteOrderSell(AddCommand addCommand, IStakan futures, IStakan spot);
```

Параметры:

- addCommand – ранее поданная алгоритмом (через методы CheckBuyDirection или CheckSellDirection) команда на выставление заявки.
- futures – стакан финансового инструмента, являющегося фьючерсом
- spot – стакан финансового инструмента, являющегося спотом

Возвращаемое значение: **true**, если указанная заявка не удовлетворяет рыночной ситуации и подлежит снятию, в противном случае – **false**.

Все свойства addCommand в точности соответствуют реальной заявке, находящейся в текущий момент в рынке. Это означает, что если после выставления заявка была частично исполнена, то в addCommand будет указано количество лотов (контрактов), равное количеству лотов (контрактов), оставшемуся в заявке.

3) Метод, оповещающий алгоритм о сделке

```
void OnTrade(Trade trade);
```

Параметры:

- trade – экземпляр сделки, содержащий все необходимую информацию

Класс Trade описан в приложении №10.

## 3.2. Интерфейс ITaskInfo

### 3.2.1. Свойства

**Все свойства, описанные в интерфейсе ITaskInfo, обязательны к переопределению.**

```
double AveragePrice { get; }
```

Информативное свойство. Значение этого свойства отображается в главном окне программы в колонке Aver. Spread. Может быть любым значением.

```
int CurrentLotSizeBuy { get; }
int CurrentLotSizeSell { get; }
```

Свойства, влияющие на расчеты. Свойство означает количество контрактов, которое планируется купить и продать. В соответствии со значением этого свойства рассчитываются значения опорных бидов и опорных оферов фьючерса и спота, из которых с последствием получаются значения спрэдов Buy Market Spread, Sell Market Spread, Buy Limit Spread, Sell Limit Spread.

К примеру, алгоритм пользователя использует значения спрэдов, рассчитываемые пользовательской задачей. Если задача рассчитана на торговлю 1-2-3 контрактами, то погрешность расчета спрэда мала и ей можно пренебречь. Но если задача будет торговать 100-300 контрактами, то разница между спрэдом на 1 контракт и спрэдом на 100 контрактов может быть существенной. Алгоритм должен это учитывать с помощью этого свойства. Может быть любым положительным числом.

```
double TargetPriceBuy { get; }  
double TargetPriceSell { get; }
```

Информативные свойства. Значения этих свойств отображаются в главном окне программы в колонках Target Buy Price и Target Sell Price. Может быть любым значением.

### 3.3. Интерфейс `IInitializeable`

#### 3.3.1. Методы

**Все методы, описанные в интерфейсе `IInitializeable`, обязательны к переопределению.**

```
bool Initialize();
```

Метод используется для инициализации экземпляра `Algorithm`.

Дело в том, что на момент создания экземпляра класса `Algorithm` (конструктор класса) весь функционал еще недоступен. Экземпляр еще не знает ничего ни о фьючерсе, ни о споте, ни о чем-либо другом. Попытка обращения к любому защищенному (`protected`) методу или свойству вызовет исключение `NullReferenceException`.

Метод `Initialize` вызывается пользовательской задачей в тот момент, когда передача всей информации от задачи к алгоритму завершена, и алгоритм может начинать процессы получения котировок, расчетов и т.п.

Возвращаемое значение: **true**, если инициализация прошла успешно, иначе **false**.

### 3.4. Интерфейс `IDisposable`

#### 3.4.1. Методы

**Все методы, описанные в интерфейсе `IDisposable`, обязательны к переопределению.**

```
void Dispose();
```

Метод, в котором выполняются операции по очистке используемых ресурсов при «умерщвлении» экземпляра `Algorithm`.

### 3.5. Интерфейс `IControlable`

#### 3.5.1. Методы

**Все методы, описанные в интерфейсе `IControlable`, обязательны к переопределению.**

```
ObjectAssociatedControl CreateControl();
```

Возвращаемое значение – экземпляр класса `ObjectAssociatedControl` – это контрол, содержащий пользовательский интерфейс изменения настроек алгоритма. Этот контрол отображается в окне «Пользовательский интерфейс» настроек пользовательской задачи (Руководство пользователя `FinLab-PairTrade`, стр. 57).

Подробнее о классе `ObjectAssociatedControl` будет рассказано в приложении №3.

### 3.6. Защищенные (protected) свойства Algorithm

```
IStakan Futures { get; }  
IStakan Spot { get; }
```

Полнофункциональные стаканы фьючерса и спота. Подробно об интерфейсе IStakan будет рассказано в приложении №6.

```
double KoefFutSpot { get; }
```

Коэффициент соответствия между спотом и фьючерсом. Равен количеству лотов (контрактов) спота, соответствующему 1 контракту фьючерса.

```
double LotSize { get; }
```

Размер лота спота. Т.е. количество единиц базового актива в 1 лоте спота.

```
int OpenedPosition { get; }
```

Текущая открытая позиция в задаче. Соответствует количеству открытой позиции по фьючерсу. Например, если куплено 10 контрактов, то OpenedPosition равна 10, если 20 контрактов продано, то OpenedPosition равна -20.

```
double AverageOpenedSpread { get; }
```

Средний спрэд по текущей открытой в задаче позиции.

```
int MaxBuy { get; }  
int MaxSell { get; }
```

Максимально разрешенная для задачи позиция в каждую сторону.

```
double PointPrice { get; }
```

Цена пункта фьючерса задаче. Также является ценой пункта спреда, поскольку любой спрэд в программе FinLab.MTS должен приводиться к фьючерсу.

```
int Timeout { get; }
```

Интервал в миллисекундах, с которым заполняется внутренний буфер значений спреда задачи.

```
ResultList Results { get; }
```

Набор результатов торговли. Через это свойство можно получить такие значения, как финансовый результат (реализованный и нереализованный), количество заявок и сделок, значения спреда и пр. Подробнее класс ResultList будет рассмотрен в приложении №5.

```
TaskMode TaskMode { get; }
```

Режим работы задачи. См. приложение №9.

```
TickList Ticks { get; }
```

Коллекция **тиков спреда** внутреннего буфера задачи. Эта коллекция заполняется значениями текущего спреда с интервалом Timeout.

Эта коллекция позволяет быстро и точно проводить практически любые математические расчеты, например, вычисления средней, средневзвешанной, дисперсии, корреляции, и т.д. Очень подробно о коллекции TickList рассказано в приложении №8.

### 3.7. Защищенные (protected) методы Algorithm

#### 3.7.1. Методы, доступные для переопределения (virtual)

```
Informer CreateInformer();
```

Возвращаемое значение: экземпляр `Informer`, который в последствии будет доступен через свойство `Informer`. Подробнее об этом классе будет рассказано в приложении №4.

Переопределение этого метода не является обязательным. Задача может использовать `Informer`, создаваемый по умолчанию.

```
void OnUpdateStakan(IStakan futures, IStakan spot);
```

Параметры:

- `futures` – стакан финансового инструмента, являющегося фьючерсом
- `spot` – стакан финансового инструмента, являющегося спотом

Данный метод вызывается при каждом обновлении стакана, как по фьючерсу, так и по споту.

Вызов этого метода предшествует вызовам методов `CheckBuyDirection`, `CheckSellDirection`, `DeleteOrderBuy`, `DeleteOrderSell`.

Переопределение этого метода не является обязательным.

#### 3.7.2. Вспомогательные методы

```
void InfoToLog(object message);
```

Параметры:

- `message` – объект сообщения для регистрации в логе

Метод, записывающий передаваемое сообщение в лог алгоритма. Лог алгоритма создается автоматически.

```
void InfoToLog(string format, params object[] args);
```

Параметры:

- `format` – формат строки
- `args` – список объектов для регистрации

Метод, записывающий в лог алгоритма сообщение в указанном формате. Работа метода по формированию строки вывода в лог аналогична методу:

```
string.Format(string format, params object[] args);
```

```
void ErrorToLog(object sender, Exception e);
```

Параметры:

- sender – источник ошибки
- e – «пойманное» исключение

Метод, регистрирующий в лог ошибки, возникновение которых возможно в результате работы алгоритма.

```
DateTime MSKTime(DateTime localTime);
```

Параметры:

- localTime – локальное время

Переводит любое локальное время в московское. Необходим для синхронизации времени в отличных от Московского часовых поясах.

```
DateTime MSKTime();
```

Метод, аналогичный вызову MSKTime(DateTime localTime) с параметром DateTime.Now.

```
double RoundPrice(double price);
```

Параметры:

- price – цена инструмента

Округляет любое значение цены до минимального шага цены фьючерса. Например, если фьючерс – RTS-9.10, а price = 142768, то результат будет равен 142770.

```
protected double Spread(double priceFutures, double priceSpot);
```

Параметры:

- priceFutures – цена фьючерса
- priceSpot – цена спота

Возвращаемое значение: значение спреда, соответствующего указанным ценам фьючерса и спота. Например, для пары GAZR-9.10 и GAZP при цене фьючерса = 16300 и цене спота = 162,87 возвращаемое значение будет равно 13.

```
double FuturesOrderLimitPrice(double spotBasisPrice, double targetSpread);
```

Параметры:

- spotBasisPrice – цена спота
- targetSpread – целевой спред

Возвращаемое значение – цена фьючерса, которая при заданном значении цены спота позволяет зафиксировать целевой спред. Например, для пары GAZR-9.10 и GAZP при цене спота = 162,87 и целевом спреде 27 возвращаемое значение будет равно 163014.

```
CandleList GetCandels(TimeSpan timeFrame);
```

Параметры:

- timeFrame – таймфрейм

Возвращаемое значение – автоматически обновляемая коллекция свечей **спрэда**. Строится по тиковым данным спреда, содержащимся во внутреннем буфере задачи. Полученная один раз, эта коллекция будет автоматически обновляться по мере поступления в буфер новых тиков. Повторный вызов метода с тем же параметром таймфрейма выдаст тот же самый экземпляр CandelList.

Подробно о коллекции CandelList будет рассказано в приложении №7.

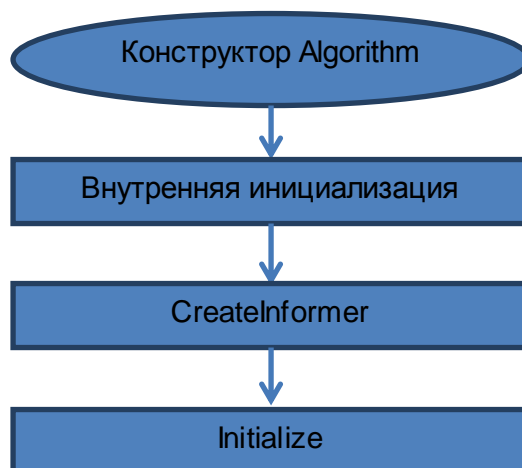
```
TickList GetFuturesAllTrades();  
TickList GetSpotAllTrades();
```

Возвращают коллекции тиков по фьючерсу и споту (список сделок, прошедших по фьючерсу или споту с начала торговой сессии). Очень подробно о коллекции TickList рассказано в приложении.

## 4. Как это работает

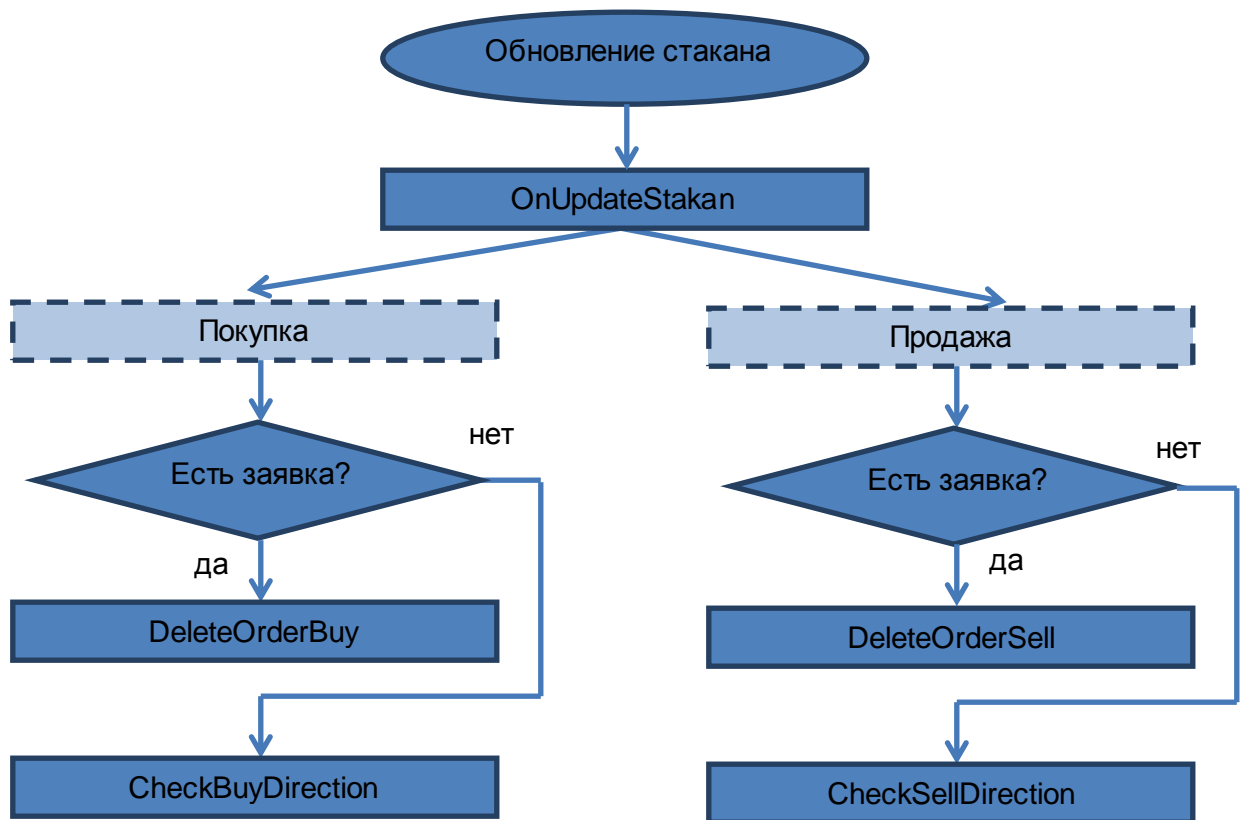
### 4.1. Инициализация алгоритма

Инициализация алгоритма проходит по следующей схеме:



### 4.2. Действия при обновлении стакана

Алгоритм при обновлении стакана действует по следующей схеме:



### 4.3. Действия при сделке

При поступлении сделки Algorithm вызывает метод OnTrade.

## 5. Практическое руководство по созданию собственного алгоритма

### 5.1. Подготовка проекта

Создайте в Microsoft Visual Studio (версии от 2008, либо другом редакторе, поддерживающем разработку под .Net Framework 3.5) новый проект Class Library для **.Net Framework 3.5**.

Добавьте к этому проекту следующие ссылки (References):

- System.Drawing
- System.Windows.Forms
- FinLab.TradeBase
- FinLab.MTS.Infinity

Создайте новый класс и унаследуйте его от класса FinLab.MTS.Infinity.Algorithm.

Теперь Вы можете сделать необходимые переопределения.

Скомпилируйте dll-ку и перенесите ее в папку с программой. Алгоритм готов к работе.

## 5.2. Пример алгоритма

Ниже представлен пример простого алгоритма. Этот алгоритм рассчитан на пару «Некий Фин. Инструмент – Empty». Принцип работы его прост: покупать, когда короткая средняя выше длинной, продавать, когда короткая средняя ниже длинной.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using FinLab.TradeBase;
using FinLab.MTS.Infinity;
using System.Drawing;
using System.Windows.Forms;

namespace Tester
{
    /// <summary>
    /// Алгоритм на 2 скользящих средних
    /// </summary>
    [Serializable]
    public sealed class MACrossAlgorithm : Algorithm
    {
        /// <summary>
        /// Период короткой скользящей средней
        /// </summary>
        public double ShortPeriod { get; set; }

        /// <summary>
        /// Период длинной скользящей средней
        /// </summary>
        public double LongPeriod { get; set; }

        /// <summary>
        /// Переменная для хранения значения короткой средней
        /// </summary>
        private double _ShortMA;

        /// <summary>
        /// Переменная для хранения значения длинной средней
        /// </summary>
        private double _LongMA;

        /// <summary>
        /// Конструктор. Здесь пустой.
        /// </summary>
        public MACrossAlgorithm() { }

        /// <summary>
        /// В качестве среднего значения будем возвращать значение короткой средней
        /// </summary>
        public override double AveragePrice
        {
            get { return _ShortMA; }
        }

        /// <summary>
        /// Размер лота будет 1
        /// </summary>
        public override int CurrentLotSizeBuy
        {
            get { return 1; }
        }
    }
}
```

```

/// <summary>
/// Размер лота будет 1
/// </summary>
public override int CurrentLotSizeSell
{
    get { return 1; }
}

/// <summary>
/// Этот параметр использовать не будем, поэтому оставляем 0
/// </summary>
public override double TargetPriceBuy
{
    get { return 0; }
}

/// <summary>
/// Этот параметр использовать не будем, поэтому оставляем 0
/// </summary>
public override double TargetPriceSell
{
    get { return 0; }
}

/// <summary>
/// Переопределяем Информер
/// </summary>
protected override Informer CreateInformer()
{
    //В данном контексте нам нужно видеть значения короткой и длинной средних,
    //а также режим - Покупка или Продажа
    //Поэтому создаем Информер с тремя столбцами
    return new Informer(3, new string[] { "Short MA", "Long MA", "Regim" });
}

/// <summary>
/// Тут пересчитываем значения средних
/// </summary>
/// <param name="futures">стакан фьючерса</param>
/// <param name="spot">чтакан спота - не используется</param>
public override void OnUpdateStakan(IStakan futures, IStakan spot)
{
    //Расчет средних с помощью быстрого метода ComputeAverage
    _ShortMA = this.Ticks.ComputeAverage(ShortPeriod);
    _LongMA = this.Ticks.ComputeAverage(LongPeriod);

    //Выводим на экран значения скользящих средних
    Informer.Values[0] = _ShortMA;
    Informer.Values[1] = _LongMA;

    //Выводим на экран Режим работы:
    //Покупка, если короткая над длинной, Продажа - если наоборот
    Informer.Values[2] = _ShortMA > _LongMA ? "Buy" :
        (_ShortMA < _LongMA ? "Sell" : "No");

    //В зависимости от режима работы выделяем его значение цветом
    Informer.Colors[2] = _ShortMA > _LongMA ? Color.GreenYellow :
        (_ShortMA < _LongMA ? Color.Pink : Color.White);

    //Записываем значения в лог
    this.InfoToLog("New MA Values are: Short = {0}; Long = {1}",
        _ShortMA, _LongMA);
}

/// <summary>

```

```

/// Метод, отвечающий за выставления заявок на покупку
/// </summary>
/// <param name="futures">стакан фьючерса</param>
/// <param name="spot">стакан спота</param>
/// <returns>Команду на заявку или null</returns>
public override AddCommand CheckBuyDirection(ISTakan futures, ISTakan spot)
{
    //Если короткая средняя над длинной,
    //то начинаем покупать по 1 контрактику
    //по рынку, т.е. по оферам
    if (_ShortMA > _LongMA)
        return new AddCommand()
        {
            Price = Futures.Ask,
            Quantity = 1,
            MktLimit = "L"
        };

    //Иначе ничего не делаем
    return null;
}

/// <summary>
/// Метод, отвечающий за выставления заявок на продажу
/// </summary>
/// <param name="futures">стакан фьючерса</param>
/// <param name="spot">стакан спота</param>
/// <returns>Команду на заявку или null</returns>
public override AddCommand CheckSellDirection(ISTakan futures, ISTakan spot)
{
    //Если короткая средняя под длинной,
    //то начинаем продавать по 1 контрактику
    //по рынку, т.е. по бидам
    if (_ShortMA < _LongMA)
        return new AddCommand()
        {
            Price = Futures.Bid,
            Quantity = 1,
            MktLimit = "L"
        };

    //Иначе ничего не делаем
    return null;
}

/// <summary>
/// Создает контрол настроек пользователя
/// </summary>
/// <returns>Контрол настроек пользователя</returns>
public override ObjectAssociatedControl CreateControl()
{
    return new ExampleControl1(this);
}

/// <summary>
/// Отвечает за снятие заявки на покупку
/// </summary>
/// <param name="addCommand">уже стоящая в рынке заявка</param>
/// <param name="futures">стакан фьючерса</param>
/// <param name="spot">стакан спота</param>
/// <returns>true, если заявку надо снять, иначе false</returns>
public override bool DeleteOrderBuy(AddCommand addCommand, ISTakan futures, ISTa-
kan spot)
{
    //снимаем заявку, если она не "лучшая" в стакане

```

```

        return addCommand.Price < futures.Bid;
    }

    /// <summary>
    /// Отвечает за снятие заявки на продажу
    /// </summary>
    /// <param name="addCommand">уже стоящая в рынке заявка</param>
    /// <param name="futures">стакан фьючерса</param>
    /// <param name="spot">стакан спота</param>
    /// <returns>true, если заявку надо снять, иначе false</returns>
    public override bool DeleteOrderSell(AddCommand addCommand, IStakan futures, IS-
takan spot)
    {
        //снимаем заявку, если она не "лучшая" в стакане
        return addCommand.Price > futures.Ask;
    }

    /// <summary>
    /// Метод, убивающий алгоритм. Тут пустой.
    /// </summary>
    public override void Dispose() { }

    /// <summary>
    /// Вызывается по приходу сделки. Тут пустой
    /// </summary>
    /// <param name="trade">Сделка</param>
    public override void OnTrade(Trade trade) { }

    /// <summary>
    /// Инициализирует алгоритм. Тут нечего инициализировать
    /// </summary>
    /// <returns>true</returns>
    public override bool Initialize()
    {
        return true;
    }

    /// <summary>
    /// Класс - контрол настроек пользователя
    /// </summary>
    private sealed class ExampleControl1 : ObjectAssociatedControl
    {
        private readonly MACrossAlgorithm _Item;

        private System.Windows.Forms.Label label1;
        private System.Windows.Forms.TextBox textBox_short;
        private System.Windows.Forms.Label label2;
        private System.Windows.Forms.TextBox textBox_long;

        /// <summary>
        /// Конструктор
        /// </summary>
        /// <param name="item">Экземпляр нашего алгоритма</param>
        public ExampleControl1(MACrossAlgorithm item)
        {
            InitializeComponent();
            _Item = item;

            //записываем в textbox значения периодов
            this.textBox_short.Text = _Item.ShortPeriod.ToString();
            this.textBox_long.Text = _Item.LongPeriod.ToString();
        }

        /// <summary>
        /// Метод, передающий данные от пользователя в алгоритм

```

```

/// </summary>
public override void ApplyChanges()
{
    //считываем из textBox новые значения периодов
    _Item.ShortPeriod = double.Parse(this.textBox_short.Text);
    _Item.LongPeriod = double.Parse(this.textBox_long.Text);
}

/// <summary>
/// Инициализатор интерфейса
/// </summary>
private void InitializeComponent()
{
    this.label1 = new System.Windows.Forms.Label();
    this.textBox_short = new System.Windows.Forms.TextBox();
    this.label2 = new System.Windows.Forms.Label();
    this.textBox_long = new System.Windows.Forms.TextBox();
    this.SuspendLayout();
    //
    // label1
    //
    this.label1.AutoSize = true;
    this.label1.Location = new System.Drawing.Point(13, 16);
    this.label1.Name = "label1";
    this.label1.Size = new System.Drawing.Size(32, 13);
    this.label1.TabIndex = 0;
    this.label1.Text = "Short";
    //
    // textBox_short
    //
    this.textBox_short.Location = new System.Drawing.Point(51, 13);
    this.textBox_short.Name = "textBox_short";
    this.textBox_short.Size = new System.Drawing.Size(75, 20);
    this.textBox_short.TabIndex = 1;
    //
    // label2
    //
    this.label2.AutoSize = true;
    this.label2.Location = new System.Drawing.Point(13, 42);
    this.label2.Name = "label2";
    this.label2.Size = new System.Drawing.Size(31, 13);
    this.label2.TabIndex = 0;
    this.label2.Text = "Long";
    //
    // textBox_long
    //
    this.textBox_long.Location = new System.Drawing.Point(51, 39);
    this.textBox_long.Name = "textBox_long";
    this.textBox_long.Size = new System.Drawing.Size(75, 20);
    this.textBox_long.TabIndex = 1;
    //
    // UserControl1
    //
    this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.Controls.Add(this.textBox_long);
    this.Controls.Add(this.label2);
    this.Controls.Add(this.textBox_short);
    this.Controls.Add(this.label1);
    this.Name = "UserControl1";
    this.Size = new System.Drawing.Size(147, 76);
    this.ResumeLayout(false);
    this.PerformLayout();
}
}
}

```

```
}  
}
```

Это полностью рабочий алгоритм. Разработчики программы не несут ответственности за убытки, получение которых возможно при его использовании.

### 5.3. Требования по разработке алгоритма

- Помечайте класс Алгоритма атрибутом **[Serializable]**. В большинстве случаев все работает и без него, но Сериализация этого требует.
- Всегда создавайте конструктор **без параметров**, даже если он будет пустым. Сериализация этого требует.
- Все параметры, алгоритма, которые нужно сохранять, должны быть помечены как **public** и должны иметь разрешения для внешнего чтения и записи, например:

```
public double ShortPeriod;
```

или

```
public double ShortPeriod { get; set; }
```

В противном случае, возникнут проблемы при их сохранении в файл и чтении из файла.

- Не создавайте Параметров, зависящих друг от друга.
- Не создавайте сложных Параметров, зависящих от свойств фьючерса, спота и т.п. Для этих целей лучше использовать локальные переменные, значения которых задаются отдельно, например, в методе **Initialize**.
- Старайтесь по возможности самостоятельно обрабатывать исключения (Exception) по средствам конструкций try –catch-finally.
- Будь особенно внимательны при проработке интерфейса пользователя.

## 6. Приложения

### 6.1. Приложение №1. TaskParameterList<T>

Коллекция типизированных параметров. Тип T может быть любым, например, bool или double. Эта коллекция предназначена для хранения экземпляров TaskParameter<T>.

```
namespace FinLab.MTS.Infinity  
{  
    [Serializable]  
    [DoNotObfuscateType]  
    public class TaskParameter<T> : ITaskParameter<T>, ICloneable  
    {  
        public TaskParameter();  
        public TaskParameter(string Name, T Value);  
  
        public string Name { get; set; }  
    }  
}
```

```

        public T Value { get; set; }

        public object Clone();
    }
}

```

Как видно из описания, у экземпляра TaskParameter есть два свойства:

- Имя (Name) типа string (строка)
- Значение (Value) типа T

В коллекции TaskParameterList все элементы индексируются и могут быть получены пользователем по их имени через следующее свойство TaskParameterList:

```
public TaskParameter<T> this[string Name] { get; set; }
```

Наличие параметра с определенным именем может быть проверено с помощью метода:

```
public bool ContainsName(string name);
```

где name – имя искомого параметра.

Также коллекции открыты для добавления и удаления элементов посредством методов:

```

public void Add(TaskParameter<T> item);
public void Clear();
public bool Remove(string name);
public bool Remove(TaskParameter<T> item);

```

Методы добавления, удаления и получения элементов потокобезопасные (thread-safe).

## 6.2. Приложение №2. AddCommand

Класс команды. Содержит в себе информацию о параметрах заявки, планируемой к выставлению, такую как цена, количество, направление, тип (рыночная или лимитированная), инструмент и пр.

Свойства этого класса прозрачны.

```

public class AddCommand : ACommand
{
    public AddCommand();

    public string BuySell { get; set; }
    public string MktLimit { get; set; }
    public double Price { get; set; }
    public int Quantity { get; set; }
    public string SecCode { get; set; }
}

```

Изюминка этого класса заключается в том, что он не помечен атрибутом sealed. Пользователь может унаследовать этот класс и добавить к нему еще какую-либо информацию.

Для чего это нужно? Программа получает через метод **CheckBuyDirection** команду на выставление заявки на покупку. Положим, все прочие условия были соблюдены, и заявка была выставлена. Программа теперь будет опрашивать алгоритм на вопрос о снятии этой заявки. Происходит это посредством метода **DeleteOrderBuy**, в который в качестве параметра передается **та же самая команда**, которая была получена через метод **CheckBuyDirection**! Значит, может иметь место такой код:

```

/// <summary>
/// Класс, унаследованный от AddCommand
/// </summary>
private sealed class TypedAddCommand : AddCommand
{
    /// <summary>
    /// Команда метится неким типом, например, типом операции
    /// </summary>
    public CmdType Type { get; set; }

    public enum CmdType
    {
        No,
        OpenNewPosition,
        TakeProfit,
        StopLoss
    }
}

public override AddCommand CheckBuyDirection(IStakan futures, IStakan spot)
{
    //...какие-то действия.....
    //.....
    //Возвращается экземпляр TypedAddCommand
    return new TypedAddCommand()
    {
        Price = futures.Ask,
        Quantity = 1,
        MktLimit = "L",
        Type = TypedAddCommand.CmdType.OpenNewPosition
    };
}

public override bool DeleteOrderBuy(AddCommand addCommand, IStakan futures, IStakan spot)
{
    //Полученная команда тоже является экземпляром TypedAddCommand!!!
    if ((addCommand as TypedAddCommand).Type == TypedAddCommand.CmdType.OpenNewPosition)
        return true;

    return false;
}

```

Все это позволяет пользователю некоторым образом «метить» свои команды.

### 6.3. Приложение №3. ObjectAssociatedControl

Фактически – это обычный **UserControl**, которому добавили метод **ApplyChanges()**

```

public class ObjectAssociatedControl : UserControl, IApplyable
{
    public ObjectAssociatedControl();
    public virtual void ApplyChanges();
}

```

Это сделано для того, чтобы передавать информацию от графического интерфейса пользователя к алгоритму пользователя. Идея такая:

- Когда пользователь вызывает диалог настроек задачи, программа запрашивает у алгоритма этот контрол. Алгоритм создает новый экземпляр своего контрола, заполняет его информацией и передает программе.

- Пользователь производит изменения в настройках через графический интерфейс и нажимает ОК
- Программа вызывает метод ApplyChanges. В этом методе алгоритм должен получить от контроля новые настройки.

#### 6.4. Приложение №4. Informer

**Информер** – это класс, отвечающий за вывод на экран той информации, которая не может быть выведена в главной таблице программы.

**Область отображения Информера** – нижняя часть главного окна программы. Эта область может **скрываться** и **отображаться** нажатием длинной тонкой клавиши в нижней части главного окна.

```
public sealed class Informer
{
    public Informer(int count, string[] headers);

    public Color[] Colors { get; set; }
    public int Count { get; }
    public string[] Headers { get; }
    public object[] Values { get; set; }
}
```

Информер содержит в себе три списка.

- Список заголовков столбцов
- Список значений
- Список цветов для каждого значения

Список заголовков доступен только для чтения и задается в конструкторе класса. Список значений и список цветов доступны для редактирования, т.е. для вывода информации пользователю и для цветового выделения этой информации

#### 6.5. Приложение №5. ResultList

Класс, содержащий в себе промежуточные результаты торговли – тех результатов, которые отображаются в главной таблице программы.

**Внимание!** Использование этих значений в методах **CheckBuyDirection**, **CheckSellDirection**, **DeleteOrderBuy** и **DeleteOrderSell** **не рекомендуется**, поскольку их расчет производится **после** всех необходимых действий по выставлению и снятию заявок.

#### 6.6. Приложение №6. IStakan

Полнофункциональный стакан

```
public interface IStakan
{
    double Ask { get; } //лучший аск
    StakanRow[] Asks { get; } //список всех асков
    double Bid { get; } //лучший бид
    StakanRow[] Bids { get; } //список всех бидов
    int GlubinaBIDov { get; } //глубина бидов (кол-во строк)
    int GlubinaOFERov { get; } //глубина оферов (кол-во строк)
    DateTime LastUpdate { get; } //время последнего обновления
    double MinStep { get; set; } //мин. Шаг цены инструмента
}
```

```

double PrevAsk { get; } //предыдущий лучший аск
double PrevBid { get; } //предыдущий лучший бид
string SecCode { get; } //название инструмента

//событие обновления стакана
event StakanEvents_UpdateStakanEventHandler UpdateStakan;

double GetBasisAskPrice(long _Kontraktov, int _Zapas);
double GetBasisBidPrice(long _Kontraktov, int _Zapas);
int GetBuyVolumeAbovePrice(double price);
int GetSellVolumeUnderPrice(double price);
}

```

На методах этого класса следует остановиться поподробнее

```

double GetBasisAskPrice(long _Kontraktov, int _Zapas);
double GetBasisBidPrice(long _Kontraktov, int _Zapas);

```

Параметры:

- `_Kontraktov` – искомое количество контрактов
- `_Zapas` – коэффициент надежности (или запаса)

Возвращаемое значение: цена, по которой можно по рынку купить (в первом случае) или продать (во втором) заданный объем с заданным коэффициентом надежности. К примеру, для **100** контрактов с запасом **3** (т.е. в 3 раза больше) в приведенном ниже случае:

Цена ▾	Колич...	Собст...
146 905	47	
146 900	112	
146 895	79	
146 890	22	
146 885	29	
146 880	28	
146 875	26	
146 870	8	
146 865	2	
146 855	1	
146 850	245	
146 845	33	
146 835	78	
146 830	4	
146 825	6	
146 820	10	
146 815	57	
146 810	60	
146 805	33	
146 800	27	

цены будут равны:

```

GetBasisAskPrice(100, 3) = 146900
GetBasisBidPrice(100, 3) = 146835

```

Обратные методы

```

int GetBuyVolumeAbovePrice(double price);
int GetSellVolumeUnderPrice(double price);

```

Возвращают объем контрактов, стоящий за ценой. В указанном слчае было бы:

```

GetBuyVolumeAbovePrice(146835) = 356
GetSellVolumeUnderPrice(146900) = 307

```

## 6.7. Приложение №7. CandelList

Коллекция свечей, упорядоченная во времени.

Адаптированная под производительность потокобезопасная коллекция.

Ключевой особенностью коллекции является то, то она может строить свечи с любым тайм-фреймом, например, 6 минут 32 секунды.

Для того, чтобы в реальном времени строить свечи по тиковым данным, необходимо:

- Создать экземпляр CandelList с нужным таймфреймом

```
CandelList(TimeSpan timeFrame);
```

- Привязаться к ее событиям

```
event CandelEventHandler CandleClosed;  
event CandelEventHandler NewCandel;
```

- Добавлять в коллекцию тики, по которым она сама будет формировать нужные свечи

```
void Add(ITick tick);
```

Эта коллекция наследует класс List<Candel>, а это значит, она содержит весь функционал этого класса:

- Доступ к элементу коллекции по его индексу
- Получение количества элементов в коллекции
- Все возможности интерфейса IEnumerable<Candel>, доступные в .Net 3.5

Единственное исключение потоковой безопасности – это цикл foreach. Это связано с внутренней особенностью класса List<T>.

## 6.8. Приложение №8. TickList

### 6.8.1. Общие свойства и методы класса.

Коллекция тиков, упорядоченная во времени.

Адаптированная под производительность полностью потокобезопасная коллекция. Этот класс открывает перед пользователями практически безграничные возможности по проведению математических и статистических вычислений.

Коллекция реализует интерфейс IEnumerable<ITick> со всеми вытекающими отсюда возможностями.

Потоковая безопасность коллекции сохраняется при использовании цикла foreach.

Использование интерфейса ITick позволяет пользователю создавать свои классы, реализующие его, и использовать их с той же степенью надежности и скорости, и при этом добавлять в них свои данные.

Коллекция индексирована по номеру элемента. Т.е. каждый тик может быть получен по его номеру.

```
ITick this[int index] { get; }
```

Кроме того, доступно получение номера последнего элемента и общее количество элементов к коллекции:

```
int Count { get; }  
int LastIndex { get; }
```

Так же пользователь может получать последний элемент без использования его индекса

```
ITick Last { get; }
```

О появлении новых тиков можно узнавать, подписавшись на событие

```
event TickEventHandler NewTick;
```

Элементы в коллекцию могут быть добавлены в ее конец с помощью метода Add

```
void Add(ITick tick);
```

При этом, поскольку коллекция упорядочена по времени, если время добавляемого тика меньше, чем время последнего тика коллекции, то добавление тика не производится, тик игнорируется.

Во избежание хаоса удаление элементов из коллекции производится последовательно по одному, начиная с начала коллекции (т.е. самых старых).

```
void RemoveFirst();
```

Коллекция может быть полностью очищена.

```
void Clear();
```

Кроме того, поскольку TickList реализует интерфейс ICloneable, коллекция может быть клонирована

```
object Clone();
```

Возвращаемое значение – **новый экземпляр** коллекции тиков, содержание которого полностью совпадает с содержанием исходного.

Доступно так же получение индекса элемента

```
int IndexOf(ITick tick);
```

Параметры:

- tick– искомый тик

Возвращаемое значение – индекс тика или -1, если он не найден в коллекции.

**Внимание!** Поиск тика производится стандартными средствами .Net. Могут возникать проблемы, если в пользовательском классе, реализующем ITick, переопределен метод Equals. По умолчанию такой проблемы нет.

## 6.8.2. Методы ускоренной выборки элементов

Помимо стандартных средств выборки, доступных в .Net 3.5 в интерфейсе `IEnumerable<T>`, коллекция содержит метод ускоренной выборки элементов. Доступны следующие выборки:

- Выборка последних N элементов коллекции

```
IEnumerable<ITick> GetTicks(int count);
```

- Выборка элементов, время которых больше либо равно заданному

```
IEnumerable<ITick> GetTicks(DateTime endTime);
```

- Выборка элементов, удаленных во времени от последнего не более, чем на указанный интервал в миллисекундах

```
IEnumerable<ITick> GetTicks(double interval);
```

Все получаемые коллекции **потоковобезопасны** и **одноразовы**, т.е. они очищаются после прогона по ним.

## 6.8.3. Методы ускоренного вычисления

Как уже было сказано, в коллекции реализованы методы ускоренного вычисления различных математических выражений. Они работают бок о бок с методами ускоренного поиска. Гибкость и быстродействие этих методов достигается применением в них рекурсивного типа вычислений. Все эти методы основаны на следующем статическом методе:

```
static T Compute<T>(IEnumerable<ITick> ticks, Func<ITick, T, T> Fx, T defValue = null);
```

Параметры:

- `ticks` – некая коллекция тиков.
- `Fx` – функция, которая вычисляет на **N**-ом шаге значение по **N**-ому тикку и **(N-1)**-ому значению этой функции.
- `defValue` – значение типа `T`, которое передается в функцию `Fx` на 0-ом шаге.

Возвращаемое значение - значение типа `T` после пробега по всей коллекции.

Как видно из объявления, метод типизирован. Это позволяет за один «прогон» коллекции вычислять сразу несколько параметров.

Ключевым элементом этого метода является делегат `Func<ITick, T, T> Fx`. Фактически, это указатель на функцию вида

```
T SomeFunction(ITick tick, T prevValue);
```

Параметры:

- `tick` – текущий тик выборки
- `prevValue` – значение, полученное на предыдущем шаге вычисления.

Возвращаемое значение – значение, которое будет передано этой функции на следующем шаге вместе со следующим тиком.

Работа функции представлена на следующем простом примере: нужно вычислить суммарный объем последних 10 тиков, находящихся в коллекции.

```
static void VolumeLastTen(TickList someticks)
{
    IEnumerable<ITick> lastTen = someticks.GetTicks(10);

    Func<ITick, int, int> fx = (tick, vol) => vol + tick.Volume;

    int volume = TickList.Compute<int>(lastTen, fx, 0);
}
```

Более сложный пример: необходимо вычислить математическое ожидание и среднеквадратическое отклонение цены тиков в некоторой коллекции за последние 20 секунд.

```
private sealed class Temp
{
    public double Sum;
    public double SumKvadr;
    public int Count;
}

private static Func<ITick, Temp, Temp> Fx = (tick, temp) =>
{
    temp.Sum += tick.Price;
    temp.SumKvadr += tick.Price * tick.Price;
    temp.Count++;
};

private static void MO_SKO_Last20Sec(TickList someticks)
{
    IEnumerable<ITick> last20sec = someticks.GetTicks(20000.0);

    Temp temp = TickList.Compute<Temp>(last20sec, Fx, new Temp());

    double MO = temp.Sum / temp.Count;

    double dispersion = (temp.SumKvadr / temp.Count) - MO * MO;

    double SKO = Math.Sqrt(dispersion);
}
```

Этот метод статический. Его можно применять для любых коллекций тиков, не только TickList.

**Внимание!** Поточная безопасность для этого метода при применении других коллекций не гарантируется.

В коллекции TickList есть методы, доступные на уровне экземпляров, совмещающие в себе методы выборки и метод Compute. Эти методы потокобезопасны.

```
T Compute<T>(DateTime endTime, Func<ITick, T, T> Fx, T defValue = null);
T Compute<T>(double interval, Func<ITick, T, T> Fx, T defValue = null);
T Compute<T>(int count, Func<ITick, T, T> Fx, T defValue = null);
```

Кроме того, в классе TickList есть методы, быстро вычисляющие среднее и средневзвешанное значения цен тиков.

```
double ComputeAverage(DateTime endTime);
double ComputeAverage(double interval);
double ComputeAverage(int count);
```

```
double ComputeWeightedAverage(DateTime endTime);
double ComputeWeightedAverage(double interval);
double ComputeWeightedAverage(int count);
```

## 6.9. Приложение №9. TaskMode

Режим работы задачи. Задается пользователем в настройках.

```
public enum TaskMode
{
    tmNormal = 0,      //можно покупать и продавать
    tmBuy = 1,        //можно покупать и закрывать позицию
    tmSell = 2,       //можно продавать и закрывать позицию
    tmClose = 3,      //можно только закрывать позицию
    tmError = 4,      //не используется
    tmBuyOnly = 5,    //можно только покупать
    tmSellOnly = 6,   //можно только продавать
}
```

## 6.10. Приложение №10. Trade

Сделка со всей необходимой информацией.

```
public sealed class Trade
{
    public int Amount { get; }
    public string BuySell { get; }
    public DateTime DateTime { get; }
    public double Price { get; }
    public string SecCode { get; }
    public double Value { get; }
}
```