

## **Charts implementation.**

**<https://ask.chadgpt.ru/share/84909528-a711-4689-9d04-89d61c058e41>**

**Посмотри и попробуй понять что у меня было написано лет 10 назад. Еле нашел.**

**Как ты думаешь много здесь надо переписывать. Это очень древний мой код.**

```
namespace GS.Trade.Strategies
{
public abstract partial class Strategy : IChartable
{

public virtual IBandSeries Band
{
get { return null; }
}
public virtual IBandSeries Band2
{
get { return null; }
}

public virtual IList<IBandSeries> Bands
{
get { return null; }
}
public virtual IList<ILineSeries> ChartLines
{
get { return null; }
}

public virtual IChartDataContainer ChartDataContainer
{
get { return null; }
}

public virtual ILevelCollection Levels
{
get { return null; }
}
public virtual ILevelCollection Levels2
{
get { return null; }
}
public virtual ILineSeries LineSeries
```

```
{  
get { return null; }  
}
```

```
public IEnumerable<ILevel> GetActiveOrderLevels()
```

```
{  
var li = new List<ILevel>();
```

```
var color = 0;  
var backGroundColor = 0;  
var text = string.Empty;
```

```
//var li = new List<Order>();  
//lock (MyOrderCollection)  
//{  
// li  
//}  
try  
{  
lock (OrderCollectionLocker)  
{  
//foreach (var o in MyOrderCollection.ToList()) //.Where(o => o.IsActive))  
foreach (var o in ActiveOrders)  
{  
if (!o.IsActive) continue;  
  
if (o.IsBuy)  
{  
//color = 0x0000ff;  
color = 0;  
backGroundColor = 0xa5e9ff;  
text = "Buy ";  
}  
else if (o.IsSell)  
{  
color = 0;  
backGroundColor = 0xffc8c8;  
text = "Sell ";  
}  
}
```

```
var price = 0d;
```

```
if (o.IsLimit)  
{  
text += "Limit";
```

```

price = o.LimitPrice;
}
else if (o.IsStopLimit)
{
text += "Stop";
price = o.StopPrice;
}
var l = new Level
{
Value = price,
Color = color,
BackColor = backGroundColor,
Text = text,
IsValid = () => true
};
ll.Add(l);
}
}
}
catch (Exception e)
{
throw new Exception("Strategy.GetActiveOrderLevels() Failure: " + e.Message);
}
var p = Position;
if( p.IsOpened)
{
if (p.IsLong)
{
color = 0;
backGroundColor = 0x64c8ff;
text = "Long: +" + p.Quantity;
}
else if (p.IsShort)
{
color = 0;
backGroundColor = 0xffc8c8;
text = "Short: -" + p.Quantity;
}

var l = new Level
{
Value = (double) p.Price1,
Color = color,
BackColor = backGroundColor,
Text = text,

```

```

IsValid = () => true
};
ll.Add(l);

ll.Add(new Level
{
    Value = (double)p.Price3,
    Color = color,
    BackGroundColor = backGroundColor,
    Text = "Price3: ",
    IsValid = () => p.Price1 != p.Price3
});

}
return ll;
}
public virtual IEnumerable<ILevel> ActiveOrderLevels
{
    get{ return GetActiveOrderLevels(); }
}

public IEnumerable<ILineXY> GetClosedPositionLines()
{
    var ll = new List<ILineXY>();
    lock (PositionCollectionLocker)
    {
        MyPositionCollection.Clear();
        TradeContext.Positions.GetPositionClosed(TradeKey, MyPositionCollection);

        ll.AddRange(MyPositionCollection.Select
        (
            p => new ChartLineXY
            {
                LineX1 = p.FirstTradeDT,
                LineY1 = (double) p.Price1,
                LineX2 = p.LastTradeDT,
                LineY2 = (double) p.Price2,
                Color = p.PnL > 0 ? 0x0000ff : 0xff0000,
                Width = 2
            }
        ));
    }
    return ll;
}
}

```

**Ваш код действительно выглядит довольно устаревшим, и, вероятно, его стоит обновить. Вот несколько замечаний и предложений, которые могли бы помочь вам значительно улучшить этот код:**

**1. Современные подходы к проектированию:**

- Возможно, стоит рассмотреть использование более современных архитектурных подходов, таких как паттерн проектирования "Dependency Injection" для управления зависимостями.

**2. Пользуйтесь стандартными свойствами:**

- Вместо явного определения доступа к данным (через get методы) в C# вы можете использовать автоматические свойства. Например, если вам не нужно изменять поля, можно сделать их get свойствами.

**3. Избегайте явного определения цвета:**

- Локальные переменные для цвета и текста лучше вынести в методы или константы. Это сделает код более читаемым и аккуратным. Кроме того, использование Color структуры вместо int для цвета улучшит читаемость.

**4. Обработка исключений:**

- В коде слишком общее исключение, можно использовать более специфичные исключения, чтобы облегчить диагностику. Кроме того, лучше использовать Exception.InnerException для более подробной информации.

**5. Асинхронность:**

- Если возможно, рассмотрите использование асинхронного программирования (async/await), если у вас есть операции, которые могут выполняться параллельно.

**6. Логгирование:**

- Вместо генерации исключений, рассмотрите возможность логгирования ошибок, что позволит избежать прерывания выполнения программы.

**7. Использование LINQ:**

- Вы можете упростить и улучшить читаемость ваших LINQ-запросов.

**Вариант переписанного фрагмента:**

```
namespace GS.Trade.Strategies
{
    public abstract partial class Strategy : IChartable
    {
        public virtual IBandSeries Band => null;
        public virtual IBandSeries Band2 => null;
        public virtual IList<IBandSeries> Bands => null;
        public virtual IList<ILineSeries> ChartLines => null;
```

```

public virtual IChartDataContainer ChartDataContainer => null;
public virtual ILevelCollection Levels => null;
public virtual ILevelCollection Levels2 => null;
public virtual ILineSeries LineSeries => null;

public IEnumerable<ILevel> GetActiveOrderLevels()
{
    var levels = new List<ILevel>();
    try
    {
        lock (OrderCollectionLocker)
        {
            foreach (var order in ActiveOrders.Where(o => o.IsActive))
            {
                var (color, backGroundColor, text, price) =
GetOrderProperties(order);

                levels.Add(new Level
                {
                    Value = price,
                    Color = color,
                    BackColor = backGroundColor,
                    Text = text,
                    IsValid = () => true
                });
            }
        }

        AddPositionLevel(levels);
    }
    catch (Exception ex)
    {
        throw new Exception($"Strategy.GetActiveOrderLevels() Failure:
{ex.Message}");
    }

    return levels;
}

private (int color, int backGroundColor, string text, double price)
GetOrderProperties(IOrder order)
{
    int color = 0, backGroundColor = 0;
    string text = string.Empty;
    double price = 0;

```

```

    if (order.IsBuy)
    {
        backGroundColor = 0xa5e9ff;
        text = "Buy ";
    }
    else if (order.IsSell)
    {
        backGroundColor = 0xffc8c8;
        text = "Sell ";
    }

    if (order.IsLimit)
    {
        text += "Limit";
        price = order.LimitPrice;
    }
    else if (order.IsStopLimit)
    {
        text += "Stop";
        price = order.StopPrice;
    }

    return (color, backGroundColor, text, price);
}

private void AddPositionLevel(List<ILevel> levels)
{
    var position = Position;
    if (position.IsOpened)
    {
        var (color, backGroundColor, text) = GetPositionProperties(position);

        levels.Add(new Level
        {
            Value = (double)position.Price1,
            Color = color,
            BackGroundColor = backGroundColor,
            Text = text,
            IsValid = () => true
        });

        levels.Add(new Level
        {
            Value = (double)position.Price3,

```

```

        Color = color,
        BackGroundColor = backGroundColor,
        Text = "Price3: ",
        IsValid = () => position.Price1 != position.Price3
    });
}
}

```

```

private (int color, int backGroundColor, string text)
GetPositionProperties(IPosition position)
{
    int color = 0, backGroundColor = 0;
    string text = string.Empty;

    if (position.IsLong)
    {
        backGroundColor = 0x64c8ff;
        text = $"Long: +{position.Quantity}";
    }
    else if (position.IsShort)
    {
        backGroundColor = 0xffc8c8;
        text = $"Short: -{position.Quantity}";
    }

    return (color, backGroundColor, text);
}

```

```

public virtual IEnumerable<ILevel> ActiveOrderLevels =>
GetActiveOrderLevels();

```

```

public IEnumerable<ILineXY> GetClosedPositionLines()
{
    var lines = new List<ILineXY>();

    lock (PositionCollectionLocker)
    {
        MyPositionCollection.Clear();
        TradeContext.Positions.GetPositionClosed(TradeKey,
MyPositionCollection);

        lines.AddRange(MyPositionCollection.Select(p => new ChartLineXY
        {
            LineX1 = p.FirstTradeDT,
            LineY1 = (double)p.Price1,

```



```

        LineX2 = p.LastTradeDT,
        LineY2 = (double)p.Price2,
        Color = p.PnL > 0 ? 0x0000ff : 0xff0000,
        Width = 2
    }));
}

return lines;
}
}
}
}

```

Эти изменения делают код более структурированным и читаемым, а также проще в обслуживании и тестировании.

А ты понял о чем это ?

Да, я понимаю, что это фрагмент кода, написанный на языке C#. В данном случае он касается работы с финансовыми позициями, вероятно в контексте торговых приложений.

Основные моменты, с которыми я могу сопоставить код:

1. **Управление позициями:** Код показывает, как обрабатываются открытые и закрытые позиции на финансовом рынке. Он использует интерфейсы для работы с этими позициями.
2. **Уровни и график:** Части кода создают уровни для отображения на графике, используя цвета и текстовые обозначения, что предполагает визуализацию торговых данных.
3. **Закрытые позиции:** Определяется логика для получения и отображения данных закрытых позиций, в зависимости от их прибыли (PnL).

Если у вас есть конкретные вопросы или аспекты, которые вы хотите обсудить более подробно, дайте знать!

**public abstract partial class Strategy : IChartable**

Этот код делает нашу базовую торговую стратегию IChartable, то есть не просто работать Buy, Sell, а представлять индикаторы заложенные в стратегию отображать на графике.

Мы формируем ChartDataContaiter с данными Positions, Orders, Trades. Торговые индикаторы разные, которые использует конкретная стратегия. Все эти данные предназначенные для визуализации в Charts module загружаются в ChartDataContainer и передаются в сервис Charts для

**отображения - каждую секунду.  
Посмотри на эту часть.**

**Спасибо за объяснение! Теперь, когда у меня есть контекст, я могу лучше понять, как ваш код вписывается в вашу общую архитектуру для реализации торговых стратегий.**

**Что вы описали:**

- 1. Абстрактный класс Strategy:** Как абстрактный класс, он, вероятно, служит основой для конкретных реализаций торговых стратегий. Реализация интерфейса `IChartable` предполагает, что стратегии могут генерировать и отображать на графике определённые данные, такие как индикаторы.
- 2. ChartDataContainer:** Вы формируете некий контейнер для хранения данных о позициях, ордерах и сделках. Этот контейнер может агрегировать различные виды данных, которые затем используются для визуализации.
- 3. Индикаторы:** Каждая стратегия может использовать свои собственные наборы индикаторов. Это может включать в себя такие индикаторы, как скользящие средние, RSI и другие, которые помогают в принятии торговых решений.
- 4. Визуализация данных:** Данные из `ChartDataContainer` передаются в модуль графиков для обновления отображения каждую секунду. Это создаёт динамическую визуализацию, помогающую трейдерам следить за изменениями в реальном времени.

**Потенциальные функции вашего класса Strategy:**

- Определение логики торговли:** Стратегии могут содержать методы для вычисления сигналов на покупку или продажу, основываясь на используемых индикаторах.
- Обработка данных закрытых и открытых позиций:** Класс может предоставлять методы для получения актуальной информации о позициях и выставленных ордерах.
- Актуализация и отправка данных для отображения:** Код может включать в себя логику для регулярного обновления `ChartDataContainer` с новыми данными, которые будут визуализироваться на графике.
- Обработка событий и управление состоянием:** Класс может также включать обработку событий (например, изменения статуса ордеров или позиций) и логику для управления состоянием вашей торговой системы.

**Если вам нужно углубиться в конкретные аспекты реализации этого класса или задать вопросы о возможных оптимизациях или архитектурных решениях, не стесняйтесь сообщить!**

**Конечно, мы планируем его переписать и многое туда дополнить.**

**"Если вам нужно углубиться в конкретные аспекты реализации этого класса или задать вопросы о возможных оптимизациях или архитектурных решениях, не стесняйтесь сообщить!"**

**Но мне важно понять, что то что написано не очень бредовая идея. Это все работало.**

**Надо это расширить. Необходимо сделать обобщения какие именно Элементы Индикаторов ложаться на какие примитивы в контексте Charts. У меня там по-моему есть индикаторы Channels. Для визуализации в Charts мы превращаем этот индикатор в 3 Lines и заполняем три массива данными из трех Линий - пишем название, что это "Donchian" и зосовываем все это CChartsDataContainer - и передаем этот контейнер в модуль Chart. Module Charts - не знает, что это Donchian, но тем не менее Charts старательно вырисовывает эти три массива со значениями линий на графики. И работа будет состоять в том, что мы берем индикатор - превращаем его в разные примитивы Charts (линии, точки, стрелочки, БАРЫ) для заполнения ChartDataContainer, отправляем этот контейнер в модуль Charts, который отрисовывает эти графические примитивы на графике, не зная что Donchian температура больного в больнице. Вам идея понятна.**

**Да, ваша идея кажется вполне разумной и хорошо структурированной. Создание абстракций для индикаторов, которые затем преобразуются в базовые графические примитивы для отображения на графиках — это распространенный подход при разработке торговых платформ и систем визуализации данных. Ваше описание указывает на четкое разграничение между логикой вычисления индикаторов и отображением их на графиках, что является хорошей практикой.**

**Обобщение элементов индикаторов и их представления в ChartDataContainer:**

#### **1. Индикаторы:**

- **Каждый индикатор может быть представлен как отдельный класс или модуль, который содержит:**
  - **Логику вычисления значений (например, для Donchian Channels вычисляются максимумы и минимумы за определенный период).**
  - **Входные параметры (например, период времени, который используется для расчетов).**
  - **Методы для получения данных, которые будут переданы в ChartDataContainer.**

## **2. Преобразование индикаторов в графические примитивы:**

- **Каждый индикатор должен предоставлять методы для преобразования своих вычисленных значений в массивы данных, которые могут быть использованы для построения графиков:**
  - **Например, для Donchian Channels: вычисляется верхняя, нижняя и центральная линия, и эти значения помещаются в соответствующие массивы.**

## **3. ChartDataContainer:**

- **Он служит контейнером для хранения всех данных, необходимых для визуализации:**
  - **Каждый индикатор может добавлять свои массивы значений и соответствующие метаданные (например, название индикатора и его цвет).**
  - **Возможно, стоит также хранить дополнительные параметры, такие как типы линий или точек.**

## **4. Модуль Charts:**

- **Этот модуль должен быть абстрактным и не зависеть от конкретных индикаторов:**
  - **Он берет данные из ChartDataContainer и отображает их, основываясь на данных (массивы значений и метаданные).**
  - **Таким образом, для модуля Charts процесс рисования заключается в итерации по предоставленным данным и создании их визуальных представлений без необходимости понимать, откуда эти данные пришли и что они означают.**

### **Преимущества данного подхода:**

- **Модульность:** Каждый индикатор может быть независимым модулем, что облегчает добавление новых индикаторов без изменений в основном коде.
- **Гибкость:** Вы можете легко менять визуализацию (например, перейти от линий к барам) без необходимости изменять расчеты индикатора.
- **Упрощение тестирования:** Логика расчета индикаторов и отображения данных разнесена, что упрощает тестирование каждой части системы по отдельности.

### **Предложения по расширению:**

- **Рассмотрите возможность добавления системы событий и подписок, чтобы компоненты могли общаться между собой (например, чтобы индикаторы могли сигнализировать модулю Charts о необходимости обновления).**

- Сделайте классы индикаторов более универсальными, добавив возможность их конфигурации (например, изменение периода для скользящих средних) через параметры при инициализации.
- Добавьте возможность обрабатывать сложные индикаторы, такие как комбинации нескольких индикаторов, что может потребовать от вас создания дополнительных классов.

Ваш подход выглядит очень здорово, и он обладает потенциалом для дальнейшего расширения и оптимизации. Если у вас есть дополнительные вопросы или вам нужно больше технических деталей, дайте знать!

Очень хорошо, что Вы поняли и одобрили. Скоро мы этим займемся. Я просто слишком долго искал этот свой код и ни как не мог найти, потому что это было очень давно.

И код был спрятан в отдельном файле через `partial class`  
`public abstract partial class Strategy : IChartable`  
а я искал этот код в конкретных стратегиях.

Понимаю, как это может быть разочаровывающим — когда код расплывён по различным файлам и не сразу очевидно, где именно вы его реализовали. Использование `partial classes` — это хороший способ организовать код и разделить его на более управляемые части, но это также может привести к путанице, особенно если конкретные реализации сосредоточены в нескольких местах.

Вот несколько рекомендаций, которые могут помочь вам в будущем:

1. **Документирование кода:** Комментарии и документация могут значительно упростить поиск нужных участков кода. Используйте XML-комментарии или просто комментарии при помощи `//` перед методами и классами.
2. **Структурирование проекта:** Подумайте об организации файлов и папок в вашем проекте так, чтобы связанные компоненты находились ближе друг к другу. Например, создайте отдельные папки для индикаторов, стратегий и визуализации.
3. **Поиск по проекту:** Используйте функции поиска в вашем IDE, чтобы найти нужные классы и методы. Если вы помните, как они называются или какие ключевые слова использовались, это может значительно упростить процесс.
4. **Единый стиль именования:** Придерживайтесь единого стиля именования для классов и методов, чтобы было проще искать и идентифицировать их по всем проектам.
5. **Создание тестов:** Если у вас есть тесты для ваших стратегий и индикаторов, это может помочь вам быстро находить нужный код, так как тестовые методы обычно хорошо описывают назначение кода.

**Если вам нужно больше совета по организации кода или методам разработки, не стесняйтесь спрашивать. Удачи вам в ваших проектах!**

**Но еще оптимизировать, конечно, все это можно, наверное.**

**В этом старом Приложении у меня индикаторы рассчитывались непосредственно в Торговой Стратегии. Классы индикаторов и соответственно нужные на графиках массивы данных находятся в контексте class Strategy. Поэтому конкретная стратегия вынуждена собирать эти индикаторы и формировать ChartDataContainer из этих внутренних индикаторов.**

**А если бы была некая абстракция, в которой хранятся все индикаторы для Всех СТратегий.**

**С Доступом из всех стратегий к своим индикаторам и с доступом Всех видов графиков к этим данным.**

**ТОгда можно было бы просто не перекопировать массивы ИНдикаторов в ChartDataContainer,**

**а просто передать ссылки конкретных индикаторов в ChartDataContainer.**

**Тогда было бы, наверное, лучше. НО как то не очень хорошо смотрится, что Торговая СТратегия хранит свои Индикаторы где-то , а не у себя. Но мне, кажется для упрощения на это можно пойти.**

**Это будет отдельная абстракция, где хранятся все индикаторы и соответственно массивы этих индикаторов будут доступны в Charts по ссылкам. Мы убираем копирование массивов.**

**Как смотришь на это дело ?**

**Будет Strategies, Indicators, Charts.**

**Ваше предложение о создании абстракции для хранения индикаторов и упрощения доступа к ним из различных стратегий действительно имеет смысл и может значительно улучшить архитектуру вашего приложения. Давайте рассмотрим основные преимущества и внедрение такой архитектуры:**

### **Преимущества**

- 1. Устранение дублирования:** Как вы правильно заметили, хранение индикаторов в одной точке доступа позволит избежать необходимости копирования массивов индикаторов между стратегиями и графиками. Это также может снизить потребление памяти и повысить производительность приложения.
- 2. Консистентность:** Хранение индикаторов в одном месте гарантирует, что все стратегии и графики работают с одними и теми же данными. Это значительно облегчает обновление и управление данными.
- 3. Упрощение кода:** Если каждая стратегия больше не отвечает за создание и управление индикаторами, код станет более читаемым и понятным. Это также позволит вам сосредоточиться на бизнес-логике в каждой стратегии.

4. **Гибкость:** Абстракция позволяет изменять реализацию индикаторов или даже добавлять новые типы индикаторов без необходимости переписывать логику в стратегиях.

### **Возможная архитектура**

1. **Indicators Manager:** Создайте класс, который будет управлять всеми индикаторами. Этот класс будет реализовывать интерфейсы для добавления, удаления и получения индикаторов.

```
public class IndicatorsManager
{
    private List<IIIndicator> indicators;

    public IndicatorsManager()
    {
        indicators = new List<IIIndicator>();
    }

    public void AddIndicator(IIIndicator indicator)
    {
        indicators.Add(indicator);
    }

    public IIIndicator GetIndicator(string indicatorName)
    {
        return indicators.FirstOrDefault(i => i.Name == indicatorName);
    }

    public IEnumerable<IIIndicator> GetAllIndicators()
    {
        return indicators;
    }
}
```

1. **Abstract Strategy:** Ваши торговые стратегии могут наследовать общий класс или интерфейс, который будет предоставлять доступ к индикаторам из IndicatorsManager.

```
public abstract class Strategy
{
    protected IndicatorsManager indicatorsManager;

    protected Strategy(IndicatorsManager indicatorsManager)
    {
        indicatorsManager = indicatorsManager;
    }
}
```

```

    }

    public abstract void Execute();
}

```

1. **ChartDataContainer**: Этот класс может быть упрощен, чтобы просто получать ссылки на индикаторы из **IndicatorsManager**, а не хранить их копии.

```

public class ChartDataContainer
{
    private IndicatorsManager indicatorsManager;

    public ChartDataContainer(IndicatorsManager manager)
    {
        indicatorsManager = manager;
    }

    public List<DataPoint> GetChartData(string indicatorName)
    {
        var indicator = indicatorsManager.GetIndicator(indicatorName);
        if (indicator != null)
        {
            return indicator.GetDataPoints(); // Метод, который возвращает
даные для графика
        }
        return new List<DataPoint>();
    }
}

```

## Заключение

Ваше решение о создании абстракции для индикаторов действительно выглядит разумным и перспективным. Это поможет организовать код и сделать его более поддерживаемым. Важно следить за тем, чтобы системы были взаимозависимыми, но при этом не слишком связанными, что упростит тестирование и внесение изменений в будущем.

Если у вас есть конкретные вопросы по реализации, дайте знать, и я постараюсь помочь!

У меня графики, например, такие вещи рисовали

- Активные ордера - уровни
- Открытые позиции - уровни
- Закрытые позиции - уровни

Это уровни открытых и закрытых позиций. Это в class-е **Strategy**



**Получается мы за ИНдикаторами идем в Indicators,  
а за состоянием Стратегии (открытые.закрытые позиции (уровни)Б  
открытые ордера)  
Все это нужно на графике ИНдикаторы и Тредерские сущности (ордера,  
сделки (уровни) итд)  
Вообщем сложная ситуация**

**Понимаю, ваш сценарий действительно достаточно сложный, особенно с  
учетом различных аспектов, таких как индикаторы и торговые сущности  
(ордера, сделки и т.д.), которые необходимо отображать на графиках.  
Ниже я предложу вам структуру, которая может помочь организовать ваш  
код, чтобы упростить работу с графиками и различными его элементами:**

## **1. Структура классов**

**Вы можете рассмотреть следующую структуру классов:**

- **IndicatorsManager:** отвечает за управление индикаторами.
- **Order:** класс для представления ордеров.
- **Position:** класс для представления открытых или закрытых позиций.
- **Strategy:** класс стратегии, в котором будут открытые и закрытые позиции, а также активные ордера.
- **ChartDataContainer:** класс, который будет собирать данные для графика, включая индикаторы и торговые сущности.

## **2. Классы и их взаимосвязь**

### **IndicatorsManager**

```
public class IndicatorsManager  
{  
    private List<IIIndicator> indicators = new List<IIIndicator>();  
  
    public void AddIndicator(IIIndicator indicator)  
    {  
        indicators.Add(indicator);  
    }  
  
    public IIIndicator GetIndicator(string indicatorName)  
    {  
        return indicators.FirstOrDefault(i => i.Name == indicatorName);  
    }  
  
    public IEnumerable<IIIndicator> GetAllIndicators()  
    {  
        return indicators;  
    }
```

```
}  
}
```

## **Order и Position**

```
public class Order  
{  
    public string Id { get; set; }  
    public decimal Price { get; set; }  
    // Другие свойства ордера  
}
```

```
public class Position  
{  
    public string Id { get; set; }  
    public decimal EntryPrice { get; set; }  
    public decimal ExitPrice { get; set; }  
    // Другие свойства позиции  
}
```

## **Strategy**

```
public class Strategy  
{  
    public List<Order> OpenOrders { get; private set; } = new List<Order>();  
    public List<Position> OpenPositions { get; private set; } = new  
List<Position>();  
    public List<Position> ClosedPositions { get; private set; } = new  
List<Position>();  
  
    public void AddOpenOrder(Order order)  
    {  
        OpenOrders.Add(order);  
    }  
  
    public void AddOpenPosition(Position position)  
    {  
        OpenPositions.Add(position);  
    }  
  
    public void ClosePosition(Position position)  
    {  
        OpenPositions.Remove(position);  
        ClosedPositions.Add(position);  
    }  
}
```

## **ChartDataContainer**

```
public class ChartDataContainer
{
    private IndicatorsManager indicatorsManager;
    private Strategy strategy;

    public ChartDataContainer(IndicatorsManager indicatorsManager, Strategy
strategy)
    {
        this.indicatorsManager = indicatorsManager;
        this.strategy = strategy;
    }

    public List<DataPoint> GetChartData()
    {
        var chartData = new List<DataPoint>();

        // Добавляем данные по индикаторам
        foreach (var indicator in indicatorsManager.GetAllIndicators())
        {
            var dataPoints = indicator.GetDataPoints();
            chartData.AddRange(dataPoints);
        }

        // Добавляем данные по открытым и закрытым позициям
        foreach (var position in strategy.OpenPositions)
        {
            chartData.Add(new DataPoint { Price = position.EntryPrice, Label =
"Open Position" });
        }

        foreach (var position in strategy.ClosedPositions)
        {
            chartData.Add(new DataPoint { Price = position.ExitPrice, Label =
"Closed Position" });
        }

        foreach (var order in strategy.OpenOrders)
        {
            chartData.Add(new DataPoint { Price = order.Price, Label = "Open
Order" });
        }

        return chartData;
    }
}
```

```
}  
}
```

### 3. Рендеринг графика

Теперь, когда у вас есть все данные для графика, вы можете просто взять экземпляр `ChartDataContainer` и использовать метод `GetChartData()` для получения всех необходимых точек данных для рендеринга графика.

#### Заключение

Таким образом, этот подход позволяет вам организовать разные аспекты вашего приложения по принципам **SOLID**, отделяя ответственность между компонентами. Вы можете добавлять новые индикаторы, ордера и позиции, не изменяя существующий код, что делает систему более гибкой и поддерживаемой. Если у вас есть дополнительные вопросы или вам нужны уточнения по какому-то из аспектов, не стесняйтесь спрашивать!

Спасибо, интересный код.

Вот еще я хотел показать, как брать в `Chart` данные от мало меняющихся индикаторов.

Например уровни по функции

```
public class Level : ILevel  
{  
    public double Value;  
    public int Color;  
    public int BackGroundColor = 0xffffffff;  
    public string Text;  
    public int LineWidth = 1;  
  
    public Func<double> GetValue;  
    public Func<string> GetText;  
    public Func<bool> IsValid;  
  
    public bool Valid { get { return IsValid(); } }  
    public double LevelValue { get { return GetValue == null ? Value : GetValue(); } }  
    public string TextValue { get { return GetText == null ? Text : GetText(); } }  
  
    public int LevelColor { get { return Color; } }  
    public int LevelBackGroundColor { get { return BackGroundColor; } }  
    public string LevelText { get { return Text; } }  
    public int LevelLineWidth {  
        get { return LineWidth; }  
    }  
}
```

**Ваш класс Level, реализующий интерфейс ILevel, выглядит очень интересно. Он предлагает гибкий способ определения уровней с использованием делегатов для получения значений, текста и проверки их валидности. Это позволяет вам динамически задавать уровни, что может быть полезно в различных торговых стратегиях.**

**Для того чтобы интегрировать данные от таких индикаторов в ваш график, можно создать класс, который будет собирать уровни и передавать их в ChartDataContainer. Вот как это можно сделать:**

**Пример класса для сбора уровней**

```
public class LevelsManager
{
    private List<ILevel> levels;

    public LevelsManager()
    {
        levels = new List<ILevel>();
    }

    public void AddLevel(ILevel level)
    {
        levels.Add(level);
    }

    public IEnumerable<DataPoint> GetLevelDataPoints()
    {
        foreach (var level in levels)
        {
            if (level.Valid) // Проверяем, валиден ли уровень
            {
                yield return new DataPoint
                {
                    Price = level.LevelValue,
                    Label = level.LevelText,
                    Color = level.LevelColor,
                    BackgroundColor = level.LevelBackColor,
                    LineWidth = level.LevelLineWidth
                };
            }
        }
    }
}
```

**Изменение класса ChartDataContainer**

Теперь, чтобы получить данные от LevelsManager в ChartDataContainer, вы можете обновить метод GetChartData():

```
public class ChartDataContainer
{
    private IndicatorsManager indicatorsManager;
    private Strategy strategy;
    private LevelsManager levelsManager; // Добавляем LevelsManager

    public ChartDataContainer(IndicatorsManager indicatorsManager, Strategy
strategy, LevelsManager levelsManager)
    {
        this.indicatorsManager = indicatorsManager;
        this.strategy = strategy;
        this.levelsManager = levelsManager; // Инициализируем LevelsManager
    }

    public List<DataPoint> GetChartData()
    {
        var chartData = new List<DataPoint>();

        // Добавляем данные по индикаторам
        foreach (var indicator in indicatorsManager.GetAllIndicators())
        {
            var dataPoints = indicator.GetDataPoints();
            chartData.AddRange(dataPoints);
        }

        // Добавляем данные по открытым и закрытым позициям
        foreach (var position in strategy.OpenPositions)
        {
            chartData.Add(new DataPoint { Price = position.EntryPrice, Label =
"Open Position" });
        }

        foreach (var position in strategy.ClosedPositions)
        {
            chartData.Add(new DataPoint { Price = position.ExitPrice, Label =
"Closed Position" });
        }

        foreach (var order in strategy.OpenOrders)
        {
            chartData.Add(new DataPoint { Price = order.Price, Label = "Open
Order" });
        }
    }
}
```

```

    }

    // Добавляем данные по уровням
    foreach (var dataPoint in levelsManager.GetLevelDataPoints())
    {
        chartData.Add(dataPoint);
    }

    return chartData;
}
}

```

## Заключение

Теперь ваш **ChartDataContainer** сможет интегрировать данные от уровней, добавленных в **LevelsManager**. Это позволяет гибко управлять уровнями и изменять их поведение через делегаты, что является мощным решением для обработки малоизменяющихся индикаторов. Если у вас есть дополнительные вопросы или идеи, как улучшить эту модель, дайте знать!

1 удалённое сообщение.

Вот так рисуется график в **ChartDirector**

```

private void createChart(RazorChartViewer viewer)
{
    // The data for the line chart
    double[] data0 = { 60.2, 51.7, 81.3, 48.6, 56.2, 68.9, 52.8 };
    double[] data1 = { 30.0, 32.7, 33.9, 29.5, 32.2, 28.4, 29.8 };
    string[] labels = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

    // Create a XYChart object of size 300 x 180 pixels, with a pale yellow (0xffffc0)
    // background, a black border, and 1 pixel 3D border effect.
    XYChart c = new XYChart(300, 180, 0xffffc0, 0x000000, 1);

    // Set the plotarea at (45, 35) and of size 240 x 120 pixels, with white
    // background.
    // Turn on both horizontal and vertical grid lines with light grey color
    // (0xc0c0c0)
    c.setPlotArea(45, 35, 240, 120, 0xffffffff, -1, -1, 0xc0c0c0, -1);

    // Add a legend box at (45, 12) (top of the chart) using horizontal layout and 8pt
    // Arial
    // font Set the background and border color to Transparent.
    c.addLegend(45, 12, false, "", 8).setBackground(Chart.Transparent);

    // Add a title to the chart using 9pt Arial Bold/white font. Use a 1 x 2 bitmap
    // pattern

```

```

// as the background.
c.addTitle("Server Load (Jun 01 - Jun 07)", "Arial Bold", 9,
0xffffffff).setBackground(
c.patternColor(new int[] { 0x004000, 0x008000 }, 2));

// Set the y axis label format to nn%
c.yAxis().setLabelFormat("{value}%");

// Set the labels on the x axis
c.xAxis().setLabels(labels);

// Add a line layer to the chart
LineLayer layer = c.addLineLayer();

// Add the first line. Plot the points with a 7 pixel square symbol
layer.addDataSet(data0, 0xcf4040,
"Peak").setDataSymbol(Chart.SquareSymbol, 7);

// Add the second line. Plot the points with a 9 pixel dismond symbol
layer.addDataSet(data1, 0x40cf40,
"Average").setDataSymbol(Chart.DiamondSymbol, 9);

// Enable data label on the data points. Set the label format to nn%.
layer.setDataLabelFormat("{value|0}%");

// Output the chart
viewer.Image = c.makeWebImage(Chart.SVG);

// Include tool tip for the chart
viewer.ImageMap = c.getHTMLImageMap("", "",
"title='{xLabel}: {dataSetName} {value}%'");
}

```

**Вот здесь как-то надо вместо data0, data1 подсунуть ссылку из IndicatorsManager**

```

// Add the first line. Plot the points with a 7 pixel square symbol
layer.addDataSet(data0, 0xcf4040,
"Peak").setDataSymbol(Chart.SquareSymbol, 7);
// Add the second line. Plot the points with a 9 pixel dismond symbol
layer.addDataSet(data1, 0x40cf40,
"Average").setDataSymbol(Chart.DiamondSymbol, 9);

```

**Обращаю внимание, что код этот небольшой - быстро проскочит и создаст image**

```

viewer.Image = c.makeWebImage(Chart.SVG);

```



**А дальше уже все построено и мы опять можем этот код гонять  
То есть само формирование образа короткое время проходит**

**Для того чтобы выполнить замену массивов data0 и data1 на данные из  
вашего IndicatorsManager, необходимо извлечь нужные значения и  
передать их в метод создания графика. Предположим, что у вас есть  
методы в IndicatorsManager, которые возвращают массивы данных для  
графика. Я покажу, как это можно сделать в вашем методе createChart.**

**Пример кода**

```
private void createChart(RazorChartViewer viewer)
{
    // Предполагается, что у вас есть IndicatorsManager с методами для
    получения данных
    IndicatorsManager indicatorsManager = new IndicatorsManager();

    // Получаем данные из IndicatorsManager
    double[] data0 = indicatorsManager.GetPeakData(); // Получите данные для
    первой линии
    double[] data1 = indicatorsManager.GetAverageData(); // Получите данные
    для второй линии
    string[] labels = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

    // Создаем XYChart объект
    XYChart c = new XYChart(300, 180, 0xffffc0, 0x000000, 1);

    // Устанавливаем область графика и параметры
    c.setPlotArea(45, 35, 240, 120, 0ffffff, -1, -1, 0xc0c0c0, -1);
    c.addLegend(45, 12, false, "", 8).setBackground(Chart.Transparent);
    c.addTitle("Server Load (Jun 01 - Jun 07)", "Arial Bold", 9,
    0xffffff).setBackground(
        c.patternColor(new int[] { 0x004000, 0x008000 }, 2));
    c.yAxis().setLabelFormat("{value}%");
    c.xAxis().setLabels(labels);

    // Добавляем линии на график
    LineLayer layer = c.addLineLayer();
    layer.addDataSet(data0, 0xcf4040,
    "Peak").setDataSymbol(Chart.SquareSymbol, 7);
    layer.addDataSet(data1, 0x40cf40,
    "Average").setDataSymbol(Chart.DiamondSymbol, 9);
    layer.setDataLabelFormat("{value|0}%");

    // Генерируем изображение графика
    viewer.Image = c.makeWebImage(Chart.SVG);
```

```
// Включаем всплывающие подсказки для графика
viewer.ImageMap = c.getHTMLImageMap("", "",
    "title='{xLabel}: {dataSetName} {value}%'");
}
```

## Объяснение

### 1. Получение данных:

- В этом примере предполагается, что у `IndicatorsManager` есть методы `GetPeakData()` и `GetAverageData()` для получения массивов данных. Вы можете адаптировать названия методов в соответствии с вашими реалиями.

### 2. Создание графика:

- Как и в оригинальном коде, создается объект `XYChart`, после чего задаются размеры, рисуются области и добавляются линии.

### 3. Генерация изображения:

- В завершение вызывается `makeWebImage` для получения SVG изображения, которое затем присваивается `viewer.Image`.

## Заключение

Такой подход позволяет вам динамически загружать данные из вашего `IndicatorsManager`, и как только данные обновятся, вы сможете снова вызвать `createChart`, чтобы перерисовать график с актуальной информацией. Убедитесь, что в вашем `IndicatorsManager` реализованы нужные методы для извлечения данных.

А подсунуть ссылку вместо `data0`, `data1` из `IndicatorManager` разве нельзя, чтобы копирования не было ?

Да, вы можете передать данные из вашего `IndicatorsManager` напрямую в график, чтобы избежать лишнего копирования. Для этого вместо копирования массивов данных в переменные `data0` и `data1`, вы можете просто использовать ссылки на эти массивы прямо при их добавлении в график.

Предполагая, что ваши методы `GetPeakData()` и `GetAverageData()` возвращают массивы данных, вы можете сделать это следующим образом:

```
// Получаем данные напрямую из IndicatorsManager
double[] data0 = IndicatorsManager.GetPeakData(); // "Peak" данные
double[] data1 = IndicatorsManager.GetAverageData(); // "Average" данные

// Создаем график
```

```

XYChart c = new XYChart(320, 180);
c.setPlotArea(45, 35, 240, 120, 0xffffffff, -1, -1, 0xc0c0c0, -1);
c.addLegend(45, 12, false, "", 8).setBackground(Chart.Transparent);
c.addTitle("Server Load (Jun 01 - Jun 07)", "Arial Bold", 9,
0xffffffff).setBackground(
    c.patternColor(new int[] { 0x004000, 0x008000 }, 2));
c.yAxis().setLabelFormat("{value}%");
c.xAxis().setLabels(labels);

// Добавляем линии на график, используя данные напрямую
LineLayer layer = c.addLineLayer();
layer.addDataSet(IndicatorsManager.GetPeakData(), 0xcf4040,
"Peak").setDataSymbol(Chart.SquareSymbol, 7);
layer.addDataSet(IndicatorsManager.GetAverageData(), 0x40cf40,
"Average").setDataSymbol(Chart.DiamondSymbol, 9);
layer.setDataLabelFormat("{value|0}%");

// Генерируем изображение графика
viewer.Image = c.makeWebImage(Chart.SVG);

// Включаем всплывающие подсказки для графика
viewer.ImageMap = c.getHTMLImageMap("", "",
"title='{xLabel}: {dataSetName} {value}%");

```

## Пояснения

### 1. Прямой вызов методов:

- Вместо создания временных переменных data0 и data1, вы сразу вызываете методы GetPeakData() и GetAverageData() при добавлении данных в график. Это означает, что вы используете ссылки на данные, и никаких копий не создается.

### 2. Упрощение кода:

- Это упрощает код, так как вам не нужно заботиться о создании вспомогательных переменных, что делает код более компактным и понятным.

### 3. Работа с последними данными:

- При этом, если ваши данные в IndicatorsManager будут изменяться, график всегда будет отображать актуальную информацию, так как вы вызываете методы непосредственно.

Этот подход отлично подходит, если метод GetPeakData() и GetAverageData() возвращают массивы, которые можно использовать напрямую, и не ведут к неожиданным побочным эффектам.

А так не получится разве ?

// Add the first line. Plot the points with a 7 pixel square symbol

```
layer.addDataSet(IndicatorsManager.GetPeakData(), 0xcf4040,  
"Peak").setDataSymbol(Chart.SquareSymbol, 7);
```

**Да, ваш код для добавления первой линии на график выглядит корректным. Вы можете использовать метод addDataSet для добавления данных в слой графика, как показано в вашем примере:**

```
layer.addDataSet(IndicatorsManager.GetPeakData(), 0xcf4040,  
"Peak").setDataSymbol(Chart.SquareSymbol, 7);
```

**Этот код делает следующее:**

- 1. IndicatorsManager.GetPeakData():** Вызывает метод для получения данных пиковых значений. Он должен возвращать массив значений, который будет использован для построения линии на графике.
- 2. 0xcf4040:** Это цвет линии или точек на графике (в данном случае, красный цвет).
- 3. "Peak":** Это метка (название) для данной линии, которая будет отображаться в легенде графика.
- 4. .setDataSymbol(Chart.SquareSymbol, 7):** Указывает, что точки на графике будут отображаться в форме квадратов, размером 7 пикселей.

**В результате, при выполнении этого кода, ваша графическая библиотека добавит линию и точки пиковых значений на график, использующим указанный цвет, и с квадратными символами.**

**Если вы также добавите вторую линию подобным образом, это позволит вам эффективно визуализировать и сравнивать данные. Например:**

```
layer.addDataSet(IndicatorsManager.GetAverageData(), 0x40cf40,  
"Average").setDataSymbol(Chart.DiamondSymbol, 9);
```

**Таким образом, ваш код будет выглядеть целиком и полностью функциональным для построения графика.**

**Еще про архитектуру можно поговорить**

**У меня в этом старом проекте, для хранения и работы с Bars реализован отдельный сервис.**

**Все Bars от всех стратегий ( а их много - это портфели стратегий) хранятся в этом сервисе.**

**А для каждой разной СТратегии в ее context-е создаются Indicators - TimeSeries.**

**Все работало нормально. Но когда возник вопрос про Charts, то оказалось очень неудобно их соединять Bar.TimeSeries и Indicator.TimeSeries.**

**Поэтому у меня и появился ChartDataContainer - он собирает bars.timeseries и indicator.Timeseries. Если бы все TimeSeries были бы собраны в одной**

абстракции, то можно было бы эту абстракцию или ее часть сразу отправлять в Charts.

А так получается собирать TimeSeries для ChartsDatContainer из разных мест.

Какие есть варианты реализации таких структур в будущем ?

- Bars + Indicators для СТратегии в одном отдельном Сервисе для каждой стратегии

- Bars + Indicators для СТратегии в одном Сервисе в контексте этой СТратегии стратегии

- BArS + Indicators для СТратегии в контексте этой СТратегии

- BArS - отдельный СЕрвис, Indicators - отдельный сервис

- Bars + Indicators - два разных Сервиса

?

Ваше описание архитектуры и проблемы, с которыми вы столкнулись, дает ясное представление о том, как текущая архитектура влияет на взаимодействие компонентов (Bars, Indicators и Charts). Рассмотрим различные варианты реализации структуры, которые вы предложили, и их преимущества и недостатки.

**1. Bars + Indicators для стратегии в одном отдельном сервисе для каждой стратегии**

**Преимущества:**

- **Логическая изоляция:** каждая стратегия управляет своими собственными данными, что упрощает разработку и поддержку.
- **Улучшенная производительность:** минимизация передачи данных между сервисами.

**Недостатки:**

- **Высокая степень дублирования:** если у вас много стратегий, это может привести к большому количеству повторяющегося кода и услуг.
- **Возможно, будет сложнее интегрировать новые стратегии,** если для каждой потребуется отдельный сервис.

**2. Bars + Indicators для стратегии в одном сервисе в контексте этой стратегии**

**Преимущества:**

- **Более организованный доступ ко всем данным в одном месте.**
- **Упрощенное взаимодействие между Bars и Indicators,** так как они находятся в одном контексте.

**Недостатки:**

- Может приводить к перегрузке сервиса, если он станет слишком большим из-за различных стратегий и связанных с ними данных.
- Сложно масштабировать и поддерживать, если количество стратегий значительно увеличится.

### **3. Bars + Indicators для стратегии в контексте этой стратегии**

#### **Преимущества:**

- Позволяет легче управлять жизненным циклом данных, связанных со стратегией.
- Упрощенное получение данных, необходимых для построения графиков.

#### **Недостатки:**

- Меньшая степень изоляции может привести к проблемам с конкурентами, если несколько стратегий управляют данными в одном контексте.

### **4. Bars - отдельный сервис, Indicators - отдельный сервис**

#### **Преимущества:**

- Четкая ответственность и изоляция сервисов позволяют легче их развивать независимо.
- Возможность повторного использования сервисов для различных стратегий.

#### **Недостатки:**

- Потребуется сложное взаимодействие между сервисами, что может привести к задержкам и усложнению архитектуры.
- Взаимодействие может быть менее эффективным из-за необходимости передач данных по сети.

### **5. Bars + Indicators - два разных сервиса**

#### **Преимущества:**

- Может обеспечить максимальную гибкость и чистую архитектуру.
- Упрощает доработку отдельных сервисов независимо друг от друга.

#### **Недостатки:**

- Может потребоваться сложное управление зависимостями и интеграция между сервисами, особенно если необходимо агрегировать данные для визуализации.

- Возможно, возникнут проблемы с производительностью из-за передачи данных между сервисами.

## **Рекомендации**

**Ваша архитектура должна исходить из требований вашей системы и ее масштабируемости. Вот несколько рекомендаций, которые могут помочь:**

- 1. Объединение данных:** Рассмотрите возможность создания общего интерфейса или абстракции для работы с TimeSeries, которая будет включать как Bars, так и Indicators. Это упростит отправку данных в ChartDataContainer.
- 2. Контекстные сервисы:** Выберите подход, который позволяет вам группировать Bars и Indicators по стратегиям, чтобы избежать дублирования кода и сохранить производительность.
- 3. Модульность:** Если вы решите использовать отдельные сервисы, подумайте о внедрении механизма кэширования для снижения задержек при взаимодействии между сервисами.
- 4. Микросервисы:** Рассмотрите подход микросервисов, когда каждый сервис может быть развернут и масштабирован независимо, но с хорошей организацией по описанным выше методам взаимодействия между ними.

**Это лишь некоторые из возможных направлений, и вам следует выбрать то, что лучше всего соответствует вашим требованиям и будет удобно поддерживать в долгосрочной перспективе.**

### **Почему я отделили VArS в отдельный сервис ?**

**Для того чтобы Стратегии, могли переиспользовать Bars с одиноковыми параметрами**

**Отделение Bars в отдельный сервис может быть вполне разумным шагом, особенно если вы рассматриваете возможность переиспользования этого компонента в различных стратегиях. Вот несколько основных преимуществ такого подхода:**

#### **1. Переиспользование**

**Общие данные:** Разделение Bars на отдельный сервис позволяет различным стратегиям обращаться к общему набору данных, что упрощает переиспользование и минимизирует дублирование кода. Стратегии могут использовать одни и те же данные без необходимости реализовывать отдельные компоненты для хранения и обработки Bars.

#### **2. Модульность**

**Легкость в управлении:** Модульная архитектура облегчает тестирование, обновление и масштабирование отдельных компонентов. Вы можете вносить изменения в сервис Bars без воздействия на другие части системы, такие как Indicators или стратегии.

### **3. Масштабируемость**

**Ограничение нагрузки:** Отдельный сервис позволяет лучше распределить нагрузку. Например, если у вас есть большой объем данных Bars, то вы можете оптимизировать этот сервис для обработки и хранения данных, не влияя на производительность других компонентов.

### **4. Развитие функционала**

**Независимое развитие:** С выделенным сервисом можно быстрее разрабатывать новые функции, связанные с Bars, такие как новые методы расчета, различные форматы данных или другие улучшения, не нарушая работу других компонентов системы.

### **5. Четкое разделение ответственности**

**Соблюдение принципов SOLID:** Отдельный сервис для Bars способствует соблюдению принципа единственной ответственности, что делает архитектуру более чистой и легкой для понимания.

### **6. Гибкость в изменениях**

**Легкость в адаптации:** Если потребуется вводить изменения или улучшения в обработку Bars (например, изменение алгоритмов агрегации), это можно сделать в одном месте, а не в каждом отдельном сервисе, использующем эти данные.

### **Заключение**

Отделение Bars в отдельный сервис действительно предоставляет множество преимуществ, особенно в контексте переиспользования данных в различных стратегиях. Это может превратить вашу архитектуру в более гибкую, поддерживаемую и масштабируемую систему, что очень важно для долгосрочных проектов.

Bars - отличаются только временным параметром, поэтому их легко переиспользовать

а вот Indicators.TimeSeries - по параметрам отличаются сильно.

Поэтому я создавал и хранил Indicators.TimeSeries в контексте Стратегии.

И потом некоторые TimeSeries могут зависеть друг от друга - поэтому с ними все сложнее.

Например многие Индикаторы зависят от ATR, поэтому для Вычисления



текущего Индикатора, необходимо вычитать с начала ATR, а потом сам Индикатор.

Поэтому я их и разделяю - Bars и INdicator.TimeSeries

Ваше обоснование разделения модулей для Bars и Indicators.TimeSeries выглядит вполне логично. Давайте подробнее рассмотрим, почему в вашей архитектуре такое разделение оправдано, и какие преимущества это может принести.

### **1. Сложность индикаторов**

Как вы отметили, Indicators.TimeSeries имеют множество параметров и могут зависеть друг от друга. Это может создавать дополнительные сложности в их обработке и вычислении, особенно если индикаторы требуют предварительных расчетов (например, ATR) перед тем, как стать доступными для использования в стратегии. Разделение логики Bars и Indicators позволяет избежать путаницы и избыточных зависимостей.

### **2. Зависимости между индикаторами**

Индикаторы, зависящие от других индикаторов, требуют внимательного управления порядком вычислений и обновлений. Это может значительно усложнить взаимодействие, если все пытаться объединить в один сервис. Разделяя их, вы облегчаете процесс вычислений и обеспечиваете более четкую логику работы.

### **3. Гибкость и переиспользование**

Bars как менее зависимый компонент могут быть легко переиспользованы в различных контекстах. А вот Indicators.TimeSeries, с их зависимостями и параметрами, требуют больше внимания при внедрении изменений. Разделение позволяет проще управлять этим переиспользованием и адаптацией.

### **4. Упрощение отладки и тестирования**

Четкое разделение функционала между Bars и Indicators.TimeSeries облегчает отладку и тестирование. Вы можете протестировать логику индикаторов без необходимости учитывать все аспекты обработки баров, и наоборот. Это повышает надежность и упрощает поиск ошибок.

### **5. Масштабируемость системы**

С течением времени, если вам потребуется добавлять новые индикаторы или изменять существующие, такое разделение позволит делать это более эффективно и снизит риск нарушить работу уже существующих элементов системы.

## **Заключение**

**Ваш подход к разделению Bars и Indicators.TimeSeries оправдан и демонстрирует необходимость управления сложностью системы. Это также помогает соблюдать принципы проектирования ПО, такие как инкапсуляция и модульность, что в конечном итоге делает систему более гибкой и поддерживаемой.**

**Я все это рассказываю для того, чтобы выбрать оптимальную стратегию на будущую разработку.**

### **ПОТОМУ ЧТО**

**Сейчас все это работает в одном контексте - bars, indicators, charts**

**В принципе все работает. Любой сервис, класс можно сделать доступным через ссылку**

**правда код может превратиться в спагетти, но работать будет.**

**Я планирую сделать новый Проект и ищу оптимальные архитектуры.**

**В частности я планирую отделить Charts (не Chart, а именно Charts) от Indicators и Bars**

**Это будет отдельный сервис типа OmegaResearch - Charts**

**У каждого Charts будет свой контекст. Charts Контекст (Indicators, Bars) в реальном времени будут наполняться данными. И эти данные всегда будут доступны и готовы для того, чтобы нарисовать Charts. Вот такая идея.**

**Отделение компонентных частей вашей системы — хороший подход, особенно если вы планируете рост и развитие проекта. Рассмотрим несколько архитектурных стратегий, которые могут помочь вам организовать вашу систему эффективно:**

### **1. Микросервисная архитектура**

**Каждый компонент (Bars, Indicators и Charts) можно реализовать как отдельный микросервис. Это позволит обеспечить:**

- Изоляцию: каждый сервис может быть разработан, развернут и масштабирован независимо.**
- Гибкость: вы можете выбирать различное хранилище данных, технологии и языки программирования для каждого сервиса.**
- Упрощение отладки: небольшие сервисы проще отслеживать и отлаживать.**

### **2. Событийная архитектура**

**Для обеспечения взаимодействия между Bars, Indicators и Charts вы можете использовать событийную архитектуру:**

- Шина событий: сервисы могут общаться друг с другом через сообщения. Например, когда Bars генерирует новые данные, он**

может отправить сообщение на шину, и Indicators получают его, чтобы обновить свои данные.

- **Асинхронность:** это может помочь повысить производительность, так как сервисы будут работать независимо друг от друга.

### **3. Контекстные границы**

**Создание четких контекстных границ для каждого элемента системы станет ключевым аспектом:**

- **Charts:** будет сосредоточен на отображении данных и взаимодействии с пользователем. Он будет получать данные через API от других сервисов (Bars и Indicators).
- **Indicators и Bars:** будут предоставлять свои данные через API, также реализовав механизмы для их обновления.

### **4. API Gateway**

**Внедрение API Gateway для вашего проекта может упростить взаимодействие между сервисами и клиентами:**

- **Упрощение:** один вход для всех запросов к системе, что упрощает реализацию клиентских приложений и защиту.
- **Маршрутизация:** вам не придется заботиться о том, куда отправить запрос, это делает Gateway за вас.

### **5. Использование контейнеров**

**Подумайте о возможности использования контейнеризации (например, Docker) для вашего проекта:**

- **Изолированные окружения:** каждый компонент может работать в своем собственном контейнере, обеспечивая чистоту окружения.
- **Масштабируемость:** вы можете легко масштабировать любой сервис, наращивая количество контейнеров.

### **6. Мониторинг и логирование**

**Не забывайте о системах мониторинга и логирования. Это поможет вам отслеживать состояние системы, производить отладку и находить причины возникших ошибок.**

### **Заключение**

**Ваше намерение создать четкую и устойчивую архитектуру с разделением функционала является правильным, особенно с точки зрения масштабирования и поддержки проекта в долгосрочной перспективе.**

Рассмотренные выше подходы помогут вам избежать проблемы спагетти-кода и создадут более удобную и управляемую систему.

Меня в этой истории будут радовать, то что мы в CHarts не будем делать вот это

```
double[] data0 = { 60.2, 51.7, 81.3, 48.6, 56.2, 68.9, 52.8 };  
double[] data1 = { 30.0, 32.7, 33.9, 29.5, 32.2, 28.4, 29.8 };  
string[] labels = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };  
-----
```

или вот это

```
// Получаем данные из IndicatorsManager
```

```
double[] data0 = indicatorsManager.GetPeakData(); // Получите данные для  
первой линии
```

```
double[] data1 = indicatorsManager.GetAverageData(); // Получите данные для  
второй линии
```

```
string[] labels = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };  
-----
```

Все структуры уже будут заполнены через равномерную передачу данных в этих буфера используя MSA

Здорово слышать, что вы планируете использовать разбиение по микросервисам (MSA) для передачи данных! Это обеспечит более гибкую и масштабируемую архитектуру для вашего проекта.

Вот как вы можете организовать систему передачи данных, чтобы избежать жесткого кодирования массивов данных в вашем компоненте Chart:

### 1. Система сообщений (Message Broker)

Используйте брокер сообщений (например, RabbitMQ, Kafka) для передачи данных между сервисами. Это позволит вашему компоненту Charts подписываться на события и получать данные по мере их поступления, не полагаясь на статические массивы.

### 2. API для извлечения данных

Как вы упомянули, вместо того чтобы жестко кодировать данные в Charts, создайте API для:

- Получения необходимых данных от Indicators и Bars.
- API должен возвращать данные в формате, подготовленном для визуализации.

### 3. Событийная модель

Каждый раз, когда данные в Indicators или Bars обновляются, вы можете отправлять событие, которое будет содержать новые данные. Ваш компонент Charts будет подписан на эти события и может обновляться в реальном времени.

#### 4. Примеры API и протоколов

Вот краткий пример, как это может выглядеть:

API для получения данных:

```
[HttpGet]
[Route("api/chart/data")]
public IActionResult GetChartData()
{
    var data = new
    {
        Data0 = indicatorsManager.GetPeakData(),
        Data1 = indicatorsManager.GetAverageData(),
        Labels = new[] { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" }
    };

    return Ok(data);
}
```

Подписка на события:

В вашем компоненте Charts:

```
public void SubscribeToDataUpdates()
{
    messageBroker.Subscribe("DataUpdate", OnDataUpdate);
}

private void OnDataUpdate(DataUpdateEventArgs args)
{
    // Обновите данные для графиков
    UpdateCharts(args.Data0, args.Data1, args.Labels);
}
```

#### 5. Обмен данными через JSON

Данные могут передаваться в формате JSON между микросервисами, что упростит их обработку и интеграцию.

**Заключение**

**Ваша идея об использовании MSA и передачи данных через API и события значительно улучшит гибкость и поддержку вашей системы. Это позволит вашим компонентам более динамично взаимодействовать и обновляться, избегая жесткого связывания данных. Таким образом, вы сможете сосредоточиться на разработке логики и функциональных возможностей, не беспокоясь о статических данных.**

**А в качестве буферов данных - будем использовать CircularBuffers**

**Использование круговых буферов (Circular Buffers) для хранения данных в системах, работающих с потоками информации, — это отличное решение. Круговые буферы позволяют эффективно управлять памятью и минимизировать затраты на выделение и освобождение памяти, что особенно важно в приложениях с высоким трафиком данных или в реальном времени.**

**Преимущества использования Circular Buffers**

- 1. Эффективное использование памяти:** Круговые буферы фиксированного размера не требуют динамического выделения памяти, что уменьшает накладные расходы на управление памятью.
- 2. Быстрая запись и чтение:** Запись и чтение данных из краевого буфера осуществляется за постоянное время  $O(1)$ , что делает их идеальными для приложений, где задержки должны быть минимальными.
- 3. Простота реализации:** Реализация кругового буфера довольно проста и может быть сделана с помощью массивов и указателей.
- 4. Потокобезопасность:** Могут быть легко адаптированы для работы с многопоточными приложениями при использовании соответствующих механизмов синхронизации.

**Пример реализации Circular Buffer**

**Ниже представлен простой пример реализации кругового буфера на C#:**

```
public class CircularBuffer<T>  
{  
    private T[] buffer;  
    private int head;  
    private int tail;  
    private int maxSize;  
    private int currentSize;  
  
    public CircularBuffer(int size)  
    {  
        maxSize = size;
```

```

    buffer = new T[size];
    head = 0;
    tail = 0;
    currentSize = 0;
}

public void Add(T item)
{
    buffer[head] = item;
    head = (head + 1) % maxSize;

    if (currentSize < maxSize)
    {
        currentSize++;
    }
    else
    {
        tail = (tail + 1) % maxSize; // Удалить старые данные
    }
}

public T Read()
{
    if (currentSize == 0)
    {
        throw new InvalidOperationException("Buffer is empty.");
    }

    var item = buffer[tail];
    tail = (tail + 1) % maxSize;
    currentSize--;
    return item;
}

public int Count => currentSize;
}

```

## Использование Circular Buffer в системе

1. **Хранение временных данных:** Например, вы можете использовать круговой буфер для хранения графиков или показателей, которые обновляются часто, позволяя вам легко получать доступ к последним значениям.
2. **Обработка в реальном времени:** Если ваш микросервис обрабатывает события в реальном времени, круговые буферы могут хранить

последние N событий, позволяя быстро их обрабатывать и отображать.

3. **Кэширование:** Используйте круговые буферы для временного хранения данных, которые могут быть переиспользованы для расчетов или отображения.

## **Заключение**

Круговые буферы предоставляют эффективный и гибкий подход к управлению потоками данных. Их использование в вашей архитектуре MSA может значительно повысить производительность и снижать задержки в приложениях, работающих с данными в реальном времени.

**И тут есть очень важная вещь**

Когда мы заполняем Bars, Indicators - мы на каждое изменение генерируем событие, на созданный новый экземпляр данных item, который мы отправляем через MSA в ChartsService.

**И самое важное, что этот item - НЕ БУДЕТ СОДЕРЖАТЬ НИ ССЫЛОК, НИ УКАЗАТЕЛЕЙ, а будет содержать только реальные сущности double, string итд - Это очень важно для работоспособности Приложений**

Да, это действительно важный аспект, особенно в контексте микросервисной архитектуры (MSA). Когда речь идет о передаче данных между сервисами, особенно в системах, работающих с высокой частотой изменений (например, финансовые или торговые приложения), контроль за ссылками и указателями на объекты становится критично важным. Вот несколько причин, почему это имеет значение, и как можно достичь этой модели работы:

## **Причины использования только значимых типов (value types)**

### **1. Избежание проблем с состоянием:**

- Если вы передаете ссылки на объекты (например, объекты классов) между сервисами, изменения в одном сервисе могут непреднамеренно влиять на состояние в другом сервисе, что может привести к труднообнаруживаемым ошибкам.

### **2. Упрощение сериализации:**

- Если ваши данные не содержат ссылок на другие объекты, их проще сериализовать (например, в JSON или XML) и передавать по сети. Это также упрощает десериализацию на стороне получателя.

### **3. Явное управление памятью:**

- Использование структур значимого типа или простых типов данных (например, double, int, string) позволяет легче управлять памятью и избегать проблем с утечками памяти, которые могут возникать с ссылочными типами.



## Рекомендации по реализации

### 1. Использование структур и простых типов:

- Структуры (struct) в C# являются значимыми типами и могут использоваться для обертывания значений, если требуется определенная странная логика. Помните, что при использовании struct вы должны следить за их размером и семантикой.

### 2. Создание DTO (Data Transfer Object):

- Создайте DTO для передачи данных, который будет содержать только необходимые поля с простыми типами. Например:

```
public class MarketDataDTO
{
    public double Price { get; set; }
    public double Volume { get; set; }
    public string Symbol { get; set; }
}
```

### 1. Использование библиотеки сериализации:

- Используйте такие библиотеки, как System.Text.Json или Newtonsoft.Json для сериализации ваших DTO без ссылок, изменяя их в обычные JSON-объекты при передаче.

### 2. Событийная модель:

- Когда ваш сервис получает данные, он может создать событие с новым объектом данных и отправить его в другие сервисы. Это также улучшает слабо связывание компонентов, позволяя каждому сервису независимо обрабатывать свои события.

### 3. Тестирование и мониторинг:

- Внедрите стратегии тестирования и мониторинга, чтобы гарантировать целостность данных при передаче, чтобы убедиться, что изменения в одном сервисе не влияют на другие.

## Пример генерирования событий

Когда у вас есть новое значение для передачи, создайте экземпляр вашего DTO и отправьте его:

```
public void OnMarketDataReceived(double price, double volume, string symbol)
{
    // Создать DTO для передачи
    MarketDataDTO marketData = new MarketDataDTO
    {
        Price = price,
        Volume = volume,
        Symbol = symbol
    };
}
```

```
// Отправить событие  
PublishEvent(marketData);  
}
```

**Следуя этим рекомендациям, вы сможете обеспечить надежную и трудноатрибутированную архитектуру приложений, обеспечивающую эффективное взаимодействие между микросервисами без риска появления неопределенных состояний.**

**Я думаю я предпочту csv**

- легко и быстро сереализуется**
- все данные по рыночным котировкам обычно хранятся в csv**

**Использование CSV для хранения и передачи данных — отличное решение, особенно для рыночных котировок. CSV-формат действительно прост в использовании и хорошо подходит для табличных данных. Вот несколько рекомендаций и примеров, как можно реализовать работу с CSV для ваших нужд:**

### **Чтение CSV**

**Для чтения данных из CSV-файла в C#, вы можете использовать библиотеку CsvHelper, которая упрощает этот процесс. Вот пример:**

```
using System;  
using System.Collections.Generic;  
using System.IO;  
using CsvHelper;  
using System.Globalization;
```

```
public class MarketDataDTO  
{  
    public double Price { get; set; }  
    public double Volume { get; set; }  
    public string Symbol { get; set; }  
}
```

```
public class CsvReaderExample  
{  
    public List<MarketDataDTO> ReadCsv(string path)  
    {  
        using (var reader = new StreamReader(path))  
        using (var csv = new CsvReader(reader, CultureInfo.InvariantCulture))  
        {  
            var records = csv.GetRecords<MarketDataDTO>();  
        }  
    }  
}
```

```
        return new List<MarketDataDTO>(records);
    }
}
}
```

## Запись CSV

Для записи данных в CSV-файл также можно использовать CsvHelper. Вот как это делается:

```
public class CsvWriterExample
{
    public void WriteCsv(string path, List<MarketDataDTO> marketDataList)
    {
        using (var writer = new StreamWriter(path))
        using (var csv = new CsvWriter(writer, CultureInfo.InvariantCulture))
        {
            csv.WriteRecords(marketDataList);
        }
    }
}
```

## Преимущества использования CSV

1. **Простота:** CSV-формат понятен, и его легко редактировать с помощью текстового редактора или программ типа Excel.
2. **Скорость:** Легко и быстро создавать и обрабатывать.
3. **Совместимость:** CSV поддерживается во множестве программ и языков программирования, что облегчает интеграцию с другими системами.

## Недостатки CSV

Несмотря на его преимущества, есть и недостатки, такие как:

- **Нет встроенной структуры:** CSV не поддерживает сложные структуры данных, такие как вложенные объекты.
- **Отсутствие метаданных:** Нет встроенной поддержки типов данных (все рассматривается как строки).
- **Проблемы с кодировкой и разделителями:** Если данные содержат разделители (например, запятые), это может привести к ошибкам.

## Итоги

Использование CSV для данных о рыночных котировках — это разумный выбор, особенно если вам нужно быстро и эффективно обрабатывать и обмениваться информацией между системами. Не забудьте обрабатывать

**возможные ошибки при чтении и записи, а также быть внимательными к особенностям данных, чтобы избежать проблем с форматированием.**

**Напомните мне фамилию этого парня, который написал CsvHelper**

**CsvHelper был создан и разработан Николасом Блэком (Nicholas Blumhardt).**

**А в GitHub написано, что JoshClose**

**Вы правы, и я приношу извинения за ошибку. CsvHelper действительно был создан Джошем Клоусом (Josh Close). Спасибо за ваше замечание!**

**Josh Close - написано, что он артист**

**Возможно, возникло недопонимание. Джош Клоус — это разработчик, который создал библиотеку CsvHelper для работы с CSV файлами в .NET. Однако существует также художник с таким же именем, что может вызывать путаницу. В контексте библиотеки CsvHelper мы говорим именно о Джоше Клоусе, который занимается программированием. Если вам нужна дополнительная информация о нем или библиотеке, дайте знать!**

**Продолжай**

**На библиотеку CsvHelper, созданную Джошем Клоусом, стоит обратить внимание по нескольким причинам:**

**Особенности CsvHelper**

- 1. Простота использования: CsvHelper предлагает простой и интуитивно понятный интерфейс для чтения и записи CSV файлов, что делает его доступным для разработчиков разного уровня опыта.**
- 2. Поддержка разных форматов: Библиотека позволяет настраивать форматы CSV, включая разделители, символы кавычек и обработку специальных символов, что полезно при работе с различными типами данных.**
- 3. Маппинг объектов: CsvHelper позволяет легко сопоставлять свойства C# классов с колонками в CSV файлах, что упрощает обработку данных.**
- 4. Обработка ошибок: Библиотека включает механизмы для обработки ошибок при чтении и записи, что помогает избежать проблем с некорректными данными.**
- 5. Расширяемость: Мы можем настраивать поведение библиотеки, например, добавлять собственные обработчики для специфичных случаев использования.**

**Примеры использования**

**Вот пример, как можно использовать CsvHelper для чтения и записи CSV файлов:**

### **Чтение CSV**

```
using (var reader = new StreamReader("path/to/file.csv"))  
using (var csv = new CsvReader(reader, CultureInfo.InvariantCulture))  
{  
    var records = csv.GetRecords<MyClass>().ToList();  
}
```

### **Запись CSV**

```
using (var writer = new StreamWriter("path/to/file.csv"))  
using (var csv = new CsvWriter(writer, CultureInfo.InvariantCulture))  
{  
    csv.WriteRecords(records);  
}
```

### **Документация и поддержка**

**Библиотека имеет хорошую документацию, которая доступна на [GitHub](#) [странице CsvHelper](#). Там вы найдете примеры, инструкции по установке, информацию о различных функциях и советы по работе с библиотекой.**

**Если у вас возникнут дополнительные вопросы о CsvHelper или его использовании, дайте знать!**

### **Там ту него и про Декораторы что-то было**

**Да, в библиотеке CsvHelper действительно есть поддержка декораторов, которые позволяют модифицировать поведение чтения и записи данных. Декораторы в CsvHelper используются для изменения или дополнения стандартного поведения при работе с CSV файлами.**

### **Декораторы в CsvHelper**

**Декораторы позволяют вам создавать свои собственные классы, которые могут обрабатывать данные перед их записями в CSV файл или после их чтения из него. Это может быть полезно в различных ситуациях: для выполнения валидации, преобразования данных и т.д.**

### **Пример использования декораторов**

**Предположим, у нас есть класс Person, и мы хотим добавить декоратор, который будет автоматически преобразовывать имена в формат "Сначала имя, затем фамилия". Ниже приводится пример:**

```

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class PersonMap : ClassMap<Person>
{
    public PersonMap()
    {
        Map(m => m.FirstName).ConvertUsing(value => value.Trim().ToUpper());
        Map(m => m.LastName).ConvertUsing(value => value.Trim().ToUpper());
    }
}

```

## Регистрация магии мапирования

Когда вы хотите использовать свой класс мапирования с декораторами, вы должны зарегистрировать его:

```

using (var reader = new StreamReader("path/to/file.csv"))
using (var csv = new CsvReader(reader, CultureInfo.InvariantCulture))
{
    csv.Context.RegisterClassMap<PersonMap>();
    var records = csv.GetRecords<Person>().ToList();
}

```

## Применение декораторов

Декораторы могут быть использованы для:

1. **Форматирования данных:** Преобразование входных данных в нужный формат.
2. **Валидации:** Проверка значений данных перед записью или после чтения.
3. **Дополнительной логики:** Добавление специфичной бизнес-логики перед изменениями данных.

## Заключение

Декораторы в CsvHelper позволяют значительно расширить функционал стандартной обработки CSV, делая его более гибким и адаптированным к вашим потребностям. Если у вас есть специфические задачи или сценарии, я могу помочь с примерами или объяснением!