Задание №8.
События

Цели задания:

- Изучить механизм работы событий в C# - создание событий, зажигание событий, подписка и обработка событий в клиентском коде.

Механизм ***событий*** позволяет одной части системы уведомлять другую часть системы о каких-либо важных изменениях. Например, представим чат-приложение. Серверная часть приложения будет выполняться в бесконечном цикле, обращаясь к сети в ожидании сообщений от пользователей. Если сообщение по сети придёт, то серверная часть должна как-то сообщить пользовательскому интерфейсу о том, что нужно обновиться и показать новые данные пользователю. При этом, как мы знаем, серверная часть не должна иметь прямых ссылок на абстракции пользовательского интерфейса. То есть сервер не должен хранить ссылки на элементы пользовательского интерфейса в виде полей или свойств, и не должен вызывать какие-либо их методы. Реализовать подобную задачу можно с помощью событий.

События представляют собой, как правило, открытый список делегатов, который хранится в некотором объекте, и, во время выполнения внутренних алгоритмов объекта, происходит выполнение всех делегатов в этом списке. Если какие-либо объекты из клиентской части заинтересованы в том, чтобы узнать о начале выполнения события, они должны ***подписаться на событие*** – добавить свой метод-обработчик в список делегатов этого события.

Особенность события заключается в том, что выполнить список делегатов может только тот объект, внутри которого хранится событие. Преимуществом же события является то, что объект с событием не знает напрямую об объектах-подписчиках. На событие может подписаться любой клиент любого типа данных, но объект с событием не будет знать об их типах данных ничего. Таким образом, например, событие в серверной части программы, не будет знать ничего о классах пользовательского интерфейса.

Обработка событий

Как разработчики на платформе Windows Forms, вы уже успели поработать с событиями. Элементы управления – кнопки, списки, флажки, текстовые поля и т.п. – содержат множество событий. Клик мышью пользователем по кнопке, выбор пользователем другого пункта меню, нажатие клавиши пользователем в текстовом поле – всё это примеры событий. Когда вы пишите обработку действий пользователя в окне, на самом деле вы создаете специальные методы – ***обработчики событий***. Обработчики подписываются на событие элементов управления, код подписки выносится в файл формы *.designer.cs. Если мы создадите обработчик нажатия какой-либо кнопки, то перейдя в файл *.designer.cs вы обнаружите, что там появилась подобная строка:

```
//
// AddButton
//
this.AddButton.Dock = System.Windows.Forms.DockStyle.Fill;
this.AddButton.Location = new System.Drawing.Point(3, 3);
this.AddButton.Name = "AddButton";
this.AddButton.Size = new System.Drawing.Size(91, 35);
this.AddButton.TabIndex = 0;
this.AddButton.Text = "Add";
this.AddButton.UseVisualStyleBackColor = true;
this.AddButton.Click += new System.EventHandler(
                                    this.AddButton_Click);
```

Эта строка в конце и есть подписка на событие Click в кнопке AddButton. Подписку можно упростить до следующей записи:

```
this.AddButton.Click += AddButton_Click;
```

То есть механизм подписки на событие аналогичен групповой адресации делегатов [1]. На событие можно подписать любое количество методов, и они все начнут выполняться, если пользователь нажмет на кнопку в приложении.

Отписка от события выглядит так:

```
this.AddButton.Click -= AddButton_Click;
```

Есть важное правило при отписке от событий – вы должны быть уверены, что этот метод был подписан на событие. Если попытаться отписаться от события

методом, который не был подписан, это приведет к выбрасыванию исключения во время работы приложения.

Важно понять, что подписка на событие и обычный вызов метода – это разные понятия. В строке, где происходит подписка на событие, не происходит вызов обработчика. Здесь мы только указываем, что метод должен будет вызваться, когда произойдет какое-то событие. Вызов метода будет происходить в тот момент, когда событие произойдет, а не когда была сделана подписка. Поэтому достаточно один раз подписать метод-обработчик на событие, но количество выполнений этого метода будет зависеть от количества срабатываний события.

В стандартных классах .NET можно найти большое количество событий, не только в элементах пользовательского интерфейса. Windows Forms является самым наглядным примером использования событий. На событиях построено всё взаимодействие элементов между собой.

Попробуем создать собственное событие.

Создание событий

События основаны не делегатах – ссылках на функции. Например, у нас есть класс товара и мы хотим уведомлять другую часть системы в тех случаях, если у нас поменялась его стоимость:

```
public class Product
{
    private int _cost;

    public int Cost
    {
        get
        {
            return _cost;
        }
        set
        {
            if (value < 0)
            {
                throw new ArgumentException();
            }
```

```
            _cost = value;
        }
    }

    //...
}
```

В первую очередь, в класс нужно объявить делегат и добавить событие на основе делегата. Для начала сделаем событие на основе делегата без входных аргументов:

```
public class Product
{
    public delegate void ProductEvent();

    public event ProductEvent CostChanged;

    private int _cost;

    public int Cost
    {
        get
        {
            return _cost;
        }
        set
        {
            if (value < 0)
            {
                throw new ArgumentException();
            }

            _cost = value;
        }
    }

    //...
}
```

В одной строке мы объявляем делегат ProductEvent, который может хранить ссылки на любые методы без входных аргументов и возвращаемого значения.

В следующей строке мы объявляем само событие. Событие объявляется с указанием модификатора доступа, далее идет ключевое слово event, тип делегата и уникальное название события. Название события должно отражать суть события, формулируется в виде ответа на вопрос «Что произошло?» или «Что произойдет?»:

```
public event ProductEvent CostChanged;
```

В данном случае ProductEvent – это тип делегата, а CostChanged («стоимость изменилась») это название события.

События, как правило, делаются открытыми, чтобы обработчики из других классов могли подписаться. Однако, вы можете сделать защищенные и даже закрытые события, если это необходимо для вашей архитектуры.

Событие создано, но пока оно нигде зажигается. Добавим логику срабатывания события, после чего будут выполняться обработчики событий. Стоимость меняется в свойстве Cost, туда мы и добавим зажигание события:

```
public class Product
{
    public delegate void ProductEvent();

    public event ProductEvent CostChanged;

    private int _cost;

    public int Cost
    {
        get
        {
            return _cost;
        }
        set
        {
            if (value < 0)
            {
                throw new ArgumentException();
            }

            _cost = value;
```

```
            // проверка на null обязательна - это проверка,
            // что на событие кто-то подписан
            if (CostChanged != null)
                CostChanged();
        }
    }

    //...
}
```

Так наше событие называется CostChanged «стоимость изменилась» («Что произошло?»), то и зажигание события надо ставить после присвоения нового value в поле _cost. Перед тем как зажечь событие – вызвать событие как обычный метод – нужно обязательно выполнить проверку события на null. Событие имеет значение null, если на него никто не подписан, и попытка выполнить событие приведет к выбрасыванию исключения. Начиная с C# 7.0 проверка на null упростилась и теперь это можно записать в таком виде:

```
// Оператор ‘?’ выполняет проверку на null
// Invoke() вызывает событие
CostChanged?.Invoke();
```

Событие создано и зажигается при присвоении нового значения стоимости. Теперь мы можем проверить работу нашего события. Для этого создадим еще один класс и обработчик события:

```
public class Product
{
    public delegate void ProductEvent();

    public event ProductEvent CostChanged;

    private int _cost;

    public int Cost
    {
        get
        {
            return _cost;
        }
        set
```

```
        {
            if (value < 0)
            {
                throw new ArgumentException();
            }

            _cost = value;

            // проверка на null обязательна - это проверка,
            // что на событие кто-то подписан
            CostChanged?.Invoke();
        }
    }

    //...
}

public class Demo
{
    // наш обработчик, который должен срабатывать,
    // когда зажигается событие
    private void Product_CostChanged()
    {
        MessageBox.Show("Цена изменилась");
    }

    public void TestEvent()
    {
        // создаём объект продукта, на который подпишемся
        var product = new Product();

        // подписываемся на событие изменения цены
        product.CostChanged += Product_CostChanged;

        // меняем стоимость, чтобы спровоцировать
        // зажигание события
        product.Cost = 1000;
        product.Cost = 500;
        product.Cost = 700;

        // три изменения стоимости ->
        // три срабатывания обработчика ->
```

```
            // три показа всплывающего окна
        }
    }
```

В конце можно сделать отписку от события, но это необязательно – объект Product по завершению TestEvent() будет уничтожен, а вместе с ним и подписки на события.

При работе с событиями необходимо понимать, что подписка на событие одного объекта означает, что обработчик будет срабатывать на изменение цены только одного товара. Если вы хотите отслеживать изменения стоимости во всех товарах, необходимо подписываться на событие каждого товара.

Отлично, мы создали собственное событие, аналогичное событиям в элементах пользовательского интерфейса, и убедились, что оно работает. Теперь немного улучшим наш код.

Во-первых, при зажигании событий, таких как CostChanged, лучше делать проверку на то, что значение поля действительно меняется на новое:

```
set
{
    if (value < 0)
    {
        throw new ArgumentException();
    }

    if (_cost != value)
    {
        _cost = value;
        CostChanged?.Invoke();
    }
}
```

Простая проверка защищает программу от ложных срабатываний событий.

Во-вторых, мы можем передавать данные от события всем обработчикам. Например, сообщать им значение старой и новой стоимости. Для этого нам понадобится изменить тип делегата на делегат с аргументами:

```
public delegate void ProductEvent(int oldCost, int newCost);
```

После изменения типа делегата мы должны изменить вызов события и обработчик события. Начнем с события:

```
    if (_cost != value)
    {
        var oldCost = _cost;
        _cost = value;
        var newCost = _cost;
        CostChanged?.Invoke(oldCost, newCost);
    }
```

И теперь изменим обработчик в классе Demo:

```
private void Product_CostChanged(int oldCost, int newCost)
{
    var message = "Цена изменилась.\n" +
            $"Старое значение {oldCost}\n" +
            $"Новое значение {newCost}";
    MessageBox.Show(message);
}
```

Запустите пример и убедитесь, что исправленное событие работает, и теперь обработчик получает значения стоимости в качестве аргументов.

В-третьих, при создании событий разработчики используют встроенный делегат EventHandler<>. Аналогично делегатам Func<> и Action<>, EventHandler<> является обобщенным и может использоваться для создания событий с любыми аргументами. Логика EventHandler предполагает передачу в событие двух аргументов. Первый – object sender – это ссылка на сам объект, зажигающий событие. В качестве него достаточно передать ссылку this. Второй аргумент – это любые необходимые данные о событии (такие же, как старая и новая стоимость товара), которые должны быть упакованы в объект типа EventArgs или его наследник.

Перепишем наш код на использование стандартного делегата EventHandler. Сначала сделаем класс с данными события, который наследуется от EventArgs:

```
public class CostEventArgs : EventArgs
{
    public int OldCost { get; set; }

    public int NewCost { get; set; }
```

```
}
```

Сам по себе класс EventArgs не содержит каких-либо полезных данных, но его принято делать базовым классом для аргументов событий. Теперь удалим наш делегат и заменим его в объявлении события на EventHandler<>:

```
public event EventHandler<CostEventArgs> CostChanged;
```

Теперь для вызова события предварительно надо создать объект CostEventArgs и проинициализировать его значениями стоимости.

```
if (_cost != value)
{
    var args = new CostEventArgs()
        {
            OldCost = _cost
        }
    _cost = value;
    args.NewCost = _cost;
    CostChanged?.Invoke(this, args);
}
```

И последний шаг – изменение обработчика события:

```
private void Product_CostChanged(object sender,
                                 CostEventArgs args)
{
    var message = "Цена изменилась.\n" +
            $"Старое значение {args.OldCost}\n" +
            $"Новое значение {args.NewCost}";
    MessageBox.Show(message);
}
```

С помощью ссылки sender всегда можно определить объект, вызвавший событие в тех случаях, если один обработчик подписан на события разных объектов. Например, если обработчик подписан на события всех объектов товаров. Все остальные данные упакованы в отдельный класс. Такое решение требует создание отдельного класса, но это лучше, чем передача большого количества аргументов разных типов через запятую.

Полный пример кода:

```
public class CostEventArgs : EventArgs
{
    public int OldCost { get; set; }
    public int NewCost { get; set; }
}

public class Product
{
    public event EventHandler<CostEventArgs> CostChanged;

    private int _cost;

    public int Cost
    {
        get
        {
            return _cost;
        }
        set
        {
            if (value < 0)
            {
                throw new ArgumentException();
            }

            if (_cost != value)
            {
                var args = new CostEventArgs();
                args.OldCost = _cost;
                args.NewCost = value;
                _cost = value;
                CostChanged?.Invoke(this, args);
            }
        }
    }

    //...
}

public class Demo
{
    private void Product_CostChanged(object sender,
```

```
                                        CostEventArgs args)
    {
        var message = "Цена изменилась.\n" +
                $"Старое значение {args.OldCost}\n" +
                $"Новое значение {args.NewCost}";
        MessageBox.Show(message);
    }

    public void TestEvent()
    {
        // создаём объект продукта, на который подпишемся
        var product = new Product();

        // подписываемся на событие изменения цены
        product.CostChanged += Product_CostChanged;

        // меняем стоимость, чтобы спровоцировать
        // зажигание события
        product.Cost = 1000;
        product.Cost = 500;
        product.Cost = 700;
    }
}
```

Запустите пример и убедитесь, что код работает верно.

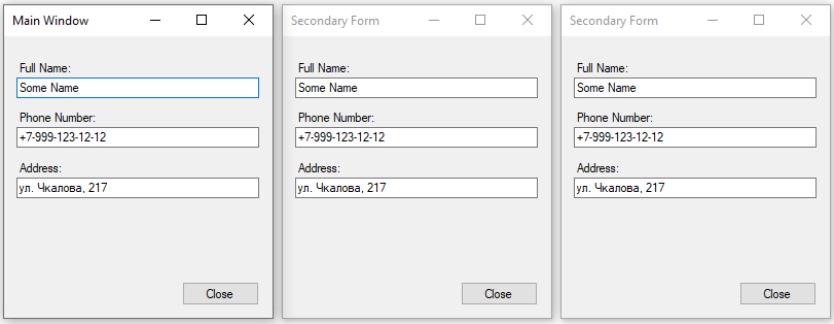
Аргументы события object sender и EventArgs args могут показаться вам знакомыми. Действительно, если мы посмотрите на обработчики любых событий в вашем пользовательском интерфейсе, вы увидите, что обработчики событий ButtonClick или SelectedIndexChanged своими аргументами ничем не отличаются от обработчика в примере выше. В большинстве случаев sender и EventArgs не используются, а потому и создавать класс-наследник для EventArgs смысла нет. Однако передача именно такого набора аргументов является своеобразным стандартом при разработке приложений.

События позволяют создавать целую систему уведомлений внутри приложения. Уведомления на действия пользователя, уведомления при изменении данных в бизнес-логике, уведомления о продолжительных процессах внутри метода и т.п.

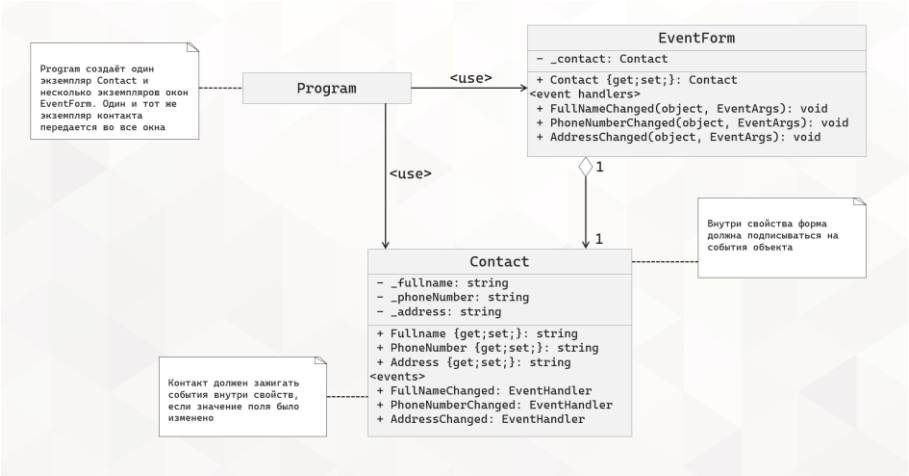
Синхронизация данных между несколькими окнами

Прежде чем внедрять события в наше приложение, сначала попрактикуемся на отдельных окнах.

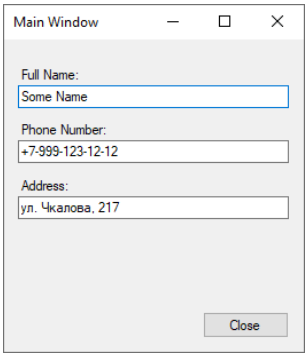
В отдельном решении необходимо разработать программу, которая позволяет в нескольких окнах выполнять редактирование данных контакта. При запуске программы перед пользователем должно появляться три одинаковых окна:



При редактировании данных в любом из окон, данные должны меняться во всех окнах. Для реализации подобного поведения, необходимо сделать один объект бизнес-логики (класс Contact), и передать его в каждую из трёх форм. Внутри класса Contact должны быть события, которые загораются на изменение свойств. Каждая из форм должна подписаться на события объекта контакта. Таким образом, если одна из форм изменит данные внутри контакта, в контакте должны загореться события и уведомить две других формы о том, что им нужно обновиться. Архитектура программы:



Верстка окна должна выглядеть следующим образом:



Особенности реализации

При работе с событиями важно учитывать несколько деталей:

1) **События внутри свойств должны зажигаться только в том случае, если значение поля действительно меняется**. То есть, если клиентский код пытается присвоить в свойство FullName точно такую же строку, то событие зажигаться не должно. В противном случае, обработка событий может уйти в бесконечную рекурсию: пользователь в одном из окон меняет имя. При изменении имени зажигается событие внутри класса Contact. В момент зажигания события три окна узнают об изменениях внутри Contact, обновляют данные в своих текстовых полях. При изменении текста в текстовых полях срабатывает событие Text-

Box_TextChanged, в обработчике которого происходит еще одно присвоение строки в свойство FullName контакта. Если при этом присвоении снова зажжется событие FullNameChanged, то обновление данных в контакте и окнах уйдет в бесконечную рекурсию.

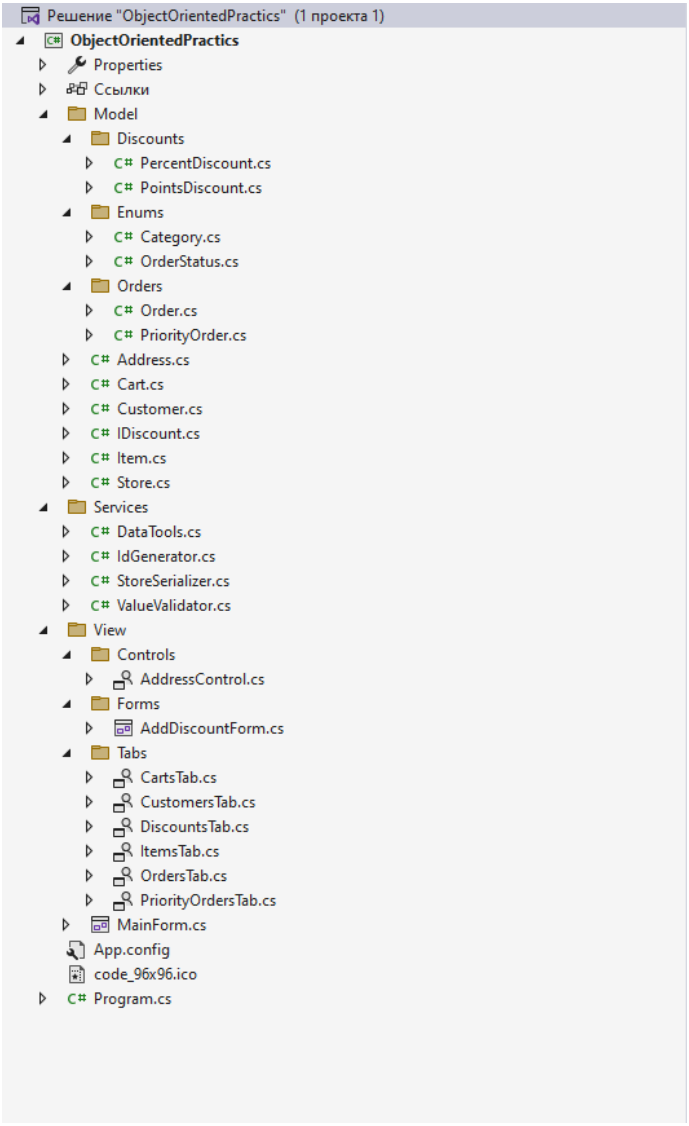
2) **Окно должно подписываться на события объекта внутри собственного свойства Contact**. Подписка выполняется только тогда, когда объект не равен null. При этом, если объект в поле меняется на новый объект, то сначала внутри свойства надо отписаться от событий старого объекта, и только затем подписаться на события нового. Отписка выполняется также только если поле не null.
3) **При уничтожении объекта окна (или его закрытии, что часто происходит непосредственно перед уничтожением объекта окна) должна происходить отписка от событий объекта**. Важный момент – несмотря на то, что контакт не хранит напрямую объекты окон, событие фактически является непрямой ссылкой на объекты окон. В случае языка C# проблема заключается в том, что объекты не уничтожаются сборщиком мусора, пока на них есть хоть одна ссылка в других объектах. Если не отписываться от событий контакта внутри окна перед его закрытием, то контакт будет хранить ссылки на окна, и окна не будут уничтожаться – до тех пор, пока не будет уничтожен объект контакта. В небольших приложениях это не страшно, но это лазейка для потенциальных утечек памяти, что может быть критично для enterprise-приложений.

Создание системы событий в приложении

1. В рамках этого задания мы добавим несколько свойств в классы логики и в классы пользовательского интерфейса.
2. Создать из ветки develop ветку Tasks/8_events, перейти в новую ветку.
3. Добавьте в класс Item события на изменение свойств NameChanged, CostChanged, InfoChanged на основе делегата EventHandler<>. Специальные аргументы в событии передавать не нужно, поэтому можно воспользоваться передачей стандартного объекта EventArgs. Убедитесь, что события работают.
4. Другой подход к реализации событий внутри классов – это создание общего события, которое будет зажигаться при изменении любого из свойств внутри класса. Такой подход годится, если нам нужно знать об изменениях внутри объекта, но при этом нам не важно, в каком именно свойстве произошли эти изменения. Добавьте в класс Address событие AddressChanged. Событие должно зажигаться при изменении любого из свойств – индекса, страны, города, улицы, здания или номера помещения. Событие также основано на EventHandler<EventArgs>. Убедитесь, что событие работает при изменении любого из свойств объекта.
5. Ранее в заданиях мы реализовали логику обновления данных на вкладках с помощью методов RefreshData() и события в главном окне SelectedTabChanged. В этом задании мы заменим использование события SelectedTabChanged на собственные события во вкладках.
6. Добавьте в класс ItemsTab событие ItemsChanged. Событие должно зажигаться, когда происходит добавление, удаление или редактирование товара.
7. В главном окне подпишитесь на событие ItemsTab.ItemsChanged и в обработчике события вызывайте методы RefreshData() для других вкладок программы.
8. При наличии событий в классах у разработчика появляется гибкость в разработке приложения. С одной стороны, он может подписать пользовательский интерфейс на события в бизнес-логике. В таком случае, пользовательский интерфейс сможет своевременно обновляться и всегда показывать актуальные данные. С другой стороны, одни части пользовательского интерфейса могут быть подписаны на события других элементов пользовательского интерфейса. Этот подход позволяет синхронизировать данные в разных частях приложения. И, как было сказано в начале, события могут быть организованы на сервере, чтобы уведомлять все клиентские приложения об изменении данных – такой подход позволяет создавать клиент-серверные приложения, например, чаты и мессенджеры. Однако, создавать события в классах нужно только тогда, когда вы видите в этом необходимость. Любое событие расширяет интерфейс класса, а интерфейс класса должен быть минимален, но достаточен.
9. Проверьте оформление кода, правильность именования, наличие комментариев. Сделайте коммит.

Проектная документация

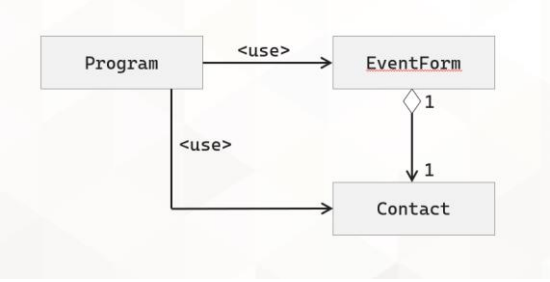
1. В результате выполнения всех заданий, структура проекта должна быть примерно следующей:



2. По результатам всех заданий обновите диаграмму классов. Диаграмма рисуется только для приложения. Рисовать диаграмму для промежуточного задания с тремя окнами не надо.

3. На диаграмме должны быть показаны все добавленные события – они обозначаются в области методов с специальным стереотипом <events>. Обозначение событий и обработчиков событий можно посмотреть в диаграмме классов промежуточного задания с тремя окнами. Сервисные классы с диаграммы можно убрать.

4. Отдельно нарисуйте сокращенную диаграмму классов, но уже для всех классов, включая классы пользовательского интерфейса. В сокращенной диаграмме классы указываются простым прямоугольником с названием, и без указания полей и методов. Цель сокращенной диаграммы показать связи между классами, не показывая их содержимого. Пример сокращенной диаграммы:



5. Диаграмму сохраните под названием «Practics8.*» в папке doc. Сохраните диаграмму как в формате программы, которую вы будете использовать, так и в формате png или jpeg, чтобы диаграмму можно было просмотреть онлайн через GitHub.

6. Выполнить сливание текущей ветки с веткой develop.

Дополнительные задания

1. В .NET есть встроенные коллекции, предоставляющие события – ObservableCollection [5]. В отличие от обычного класса List<>, ObservableCollection предоставляет событие CollectionChanged, позволяющее отслеживать изменения в коллекции – добавление, удаление, перестановку элементов. Переделайте классы Store, Cart, Order и Customer на использование ObservableCollection. Убедитесь, что при добавлении товаров в корзину или заказ, добавлении новых заказов покупателю, или новых товаров в магазин, события изменения коллекции действительно зажигаются.