Задание №2.
Композиция

Цели задания:

- Изучить принципы композиции классов.

В предыдущем задании мы изучили понятие использования. Использование является самой слабой связью между двумя объектами. Понятие сильной или слабой связи между классами выражает то, насколько изменения в одном классе приведут к необходимости изменений в другом классе. Использование – слабая связь, т.е. изменения в используемом классе чаще всего приводят к незначительным изменениям в использующем классе. Например, если мы изменили реализацию статического метода, то исправление вызова этого метода в программе, как правило, будет реализовать достаточно просто. Чуть больше изменений нам потребуется внести, если мы модифицируем класс, который используется в качестве входных или выходных аргументов какого-либо метода. Но эти изменения зачастую выполняются легко.

В рамках второго задания мы изучим более сильный вид связи – композицию. Композиция относится к более широкому понятию ***агрегирования*** – связи между двумя объектами как часть и целое. Другими словами, агрегирование подразумевает, что один объект является полем другого объекта. Примером агрегирования может служить дом и стены, человек и рука, машина и колеса и т.д. Стены являются частью дома – крупного, целого понятия, являющегося чем-то большим, чем просто сумма стен. Рука является частью человека, человек как объект-целое может управлять рукой – объектом-частью. Колеса являются частью машины, без которой машина не может выполнять свои функции.

В реальном мире любой объект можно разделить на составляющие. Каждая составляющая может быть самостоятельным объектом, однако их объединение в единое целое позволяет создать объекты с более сложным поведением, решающим и более сложные задачи. Так и в программировании мы можем составлять новые классы, где полями будут являться другие ранее написанные классы.

Агрегирование – более сильная связь, чем использование. Использование выражает обычное обращение одного объекта к другому. Например, когда вы обращаетесь к продавцу в магазине, чтобы он продал вам какой-нибудь товар, это общение можно рассматривать как использование – вы используете продавца (объект) для совершения покупки, вызывая его поведение, доступное для любого продавца (поведение, описывающее работу всех продавцов – то есть класс продавцов). В свою очередь агрегирование предполагает, что один объект не просто используется, а является ***частью*** другого объекта. Например, вы как человек можете использовать собственную руку (объект) для совершения различных действий. Также вы можете использовать молоток (объект) для различных действий. Но ваша связь с вашей рукой является более сильной, чем с молотком – рука является постоянной частью вас как объекта, в то время как молоток используется для решения задачи, а затем вы можете его оставить в ящике с инструментами, пока он снова не понадобится. Попытка избавится от руки может станет фатальным для вас, как объекта-целого, нежели попытка выкинуть молоток. Таким образом, агрегирование является более сильной связью с точки зрения смыслов.

С точки зрения программирования, попытка изменить класс, объекты которого являются полем другого класса, приведут к большему числу изменений в клиентском коде, чем попытка изменить класс, который просто используется в других классах. Хранение объекта как поля в классе подразумевает, что этот объект также часто будет вызываться в методах класса-контейнера. То есть агрегируемый объект зачастую используется более активно по сравнению с обычным использованием.

Агрегирование не означает физическое единство части и целого. Например, водитель и автомобиль. Водитель является владельцем автомобиля. Если мы будем создавать классы водителя и автомобиля, то в классе водителя мы можем объявить поле типа «Автомобиль», то есть с точки зрения программы, «Автомобиль» как объект является частью объекта «Водитель». Это логично, ведь без автомобиля пропадает сама суть понятия «Водитель». Можно сказать, что водитель агрегирует автомобиль, но в физическом мире это два отдельных объекта.

В свою очередь взаимодействие объектов как часть и целое (или, как иначе называют, часть и контейнер) может быть разным:

- Объект-часть может создаваться значительно позже по времени, чем объект-целое.
- Объект-целое может быть уничтожен, в то время как объекты-части продолжат свою работу.

- Объект-часть может быть обязательной или необязательной частью объекта-целого.
- Объекты-части могут взаимно заменяться во время работы объекта-целого.

Например, дом не может существовать без стен. При создании дома мы обязаны создать его стены – время жизни объектов совпадает. С другой стороны, водитель может сначала получить водительские права (стать объектом «Водитель»), но купить автомобиль значительно позже – объекты связываются между собой значительно позже своего создания.

Помещение агрегирует в себе мебель. Мебель зачастую является чем-то необходимым для эксплуатации помещения. Однако, если мы сломаем стул, помещение как объект не перестанет существовать. Кроме того, мы можем вынести всю мебель и занести другую. Помещение также связано агрегированием с мебелью, но помещение может существовать с разными объектами мебели или без мебели вовсе.

Поэтому, в программировании агрегирование различается двух типов – композиция и агрегация. ***Композиция*** – связь между двумя объектами «часть-целое», при котором время жизни объекта-части совпадает со временем жизни объекта-целого. Например, уже озвученный пример дома и стен является примером композиции – при создании дома мы обязаны создать стены, а при разрушении стен разрушается и объект дома. Другими словами, композиция – это такая разновидность агрегирования, когда объект-целое зависит от существования объекта-части. В то время как при агрегации время жизни объектов может отличаться (пример с помещением и мебелью).

В рамках второго задания мы изучим такой вид взаимодействия двух классов как композиция. Более простое объяснение понятия композиции – когда один объект является полем другого объекта

Это означает, что при создании объекта-целого одновременно создается и его объекты-части. В реальном жизни примером такой связи может быть дом и его стены. Стены являются обязательной частью дома. Для создания дома мы должны создать и его стены, а для разрушения дома мы должны стены разрушить.

Рассмотрим реализацию композиции в коде. Например, в программе с расписанием учебных занятий мы создадим класс учебных дисциплин Subject. Помимо названия дисциплины и количества часов, нам необходимо сохранить информацию о преподавателе дисциплины. Преподаватель как объект может описываться также отдельным классом с множеством полей. Чтобы сделать

однозначное соответствие дисциплины и её преподавателя, мы можем хранить объект преподавателя в классе Subject:

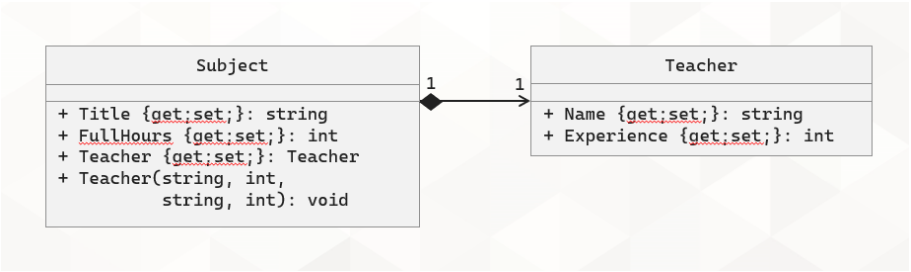
```
// Преподаватель
public class Teacher
{
   public string Name { get; set; }
   public int Experience { get; set; }

   public Teacher(string name, int experience)
   {
      Name = name;
      Experience = experience;
   }
};

// Дисциплина
public class Subject
{
   public string Title { get; set; }
   public int FullHours { get; set; }
   public Teacher Teacher { get; set; }

   public Subject(string title, int fullHours, string name, int
experience)
   {
      Title = title;
      FullHours = fullHours;
      Teacher = new Teacher(name, experience);
   }
};
```

В данном примере при создании объекта дисциплины у нас одновременно создается и новый объект преподавателя. Преподаватель является свойством (неявно – полем) класса дисциплины, то есть его неотъемлемой частью. Кроме того, в реальной предметной области, дисциплина не может существовать без преподавателя. Следовательно, дисциплина агрегирует объект преподавателя в качестве своего поля.

К сожалению, C# не позволяет строго контролировать время уничтожения объектов, но подразумевается, что при уничтожении объекта Subject уничтожится

и объект класса Teacher. Другими словами, время жизни объектов совпадает. А значит, дисциплина и преподавателем связаны конкретным видом агрегирования – композицией.

На диаграмме классов связь композиции обозначается в виде сплошной незакрытой стрелки, направленной на композируемый класс. Со стороны композирующего класса на стрелке рисуется закрашенный ромб:

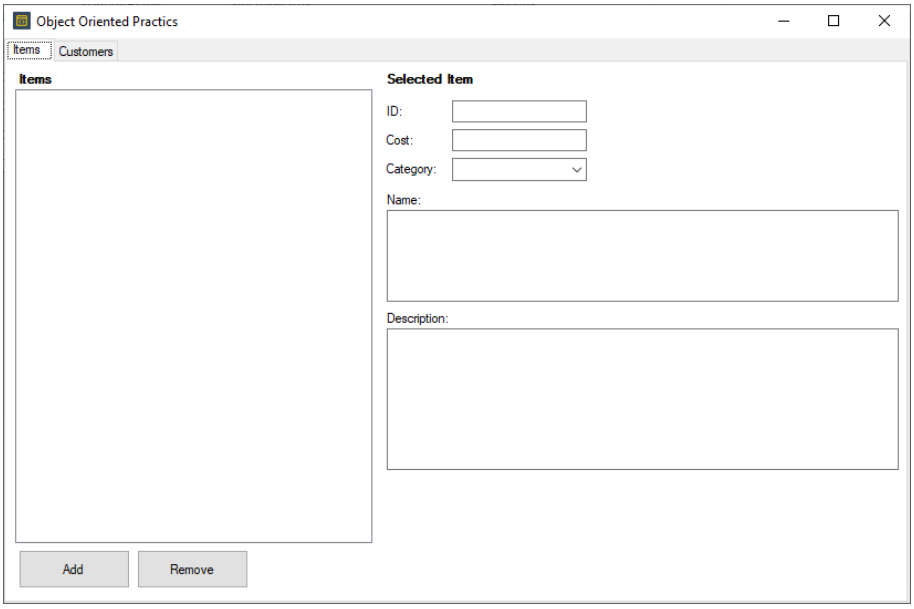


Также с обеих сторон связи указываются цифры «1» и «1», указывая, что на один объект дисциплины приходится один объект преподавателя.

Связь композиции между двумя объектами может отличаться количественно. Например, вариант «один ко одному» предполагает, что на один объект класса A приходится один объект класса B. В случае университетских дисциплин, каждая дисциплина связана с двумя преподавателями – лектором и практиком. А потому, в классе Subject вместо одного свойства Teacher можно сделать два поля или свойства Lecter и Assistant типа Teacher. В такой реализации каждой дисциплине будет соответствовать два объекта преподавателя, и связь будет равна «один ко двум». А если в классе Subject хранился бы массив или другая коллекция объектов класса Teacher, когда нам заранее неизвестно количество объектов в коллекции, можно было бы говорить о связи «один ко многим». В этом случае, кардинальность – количественное соотношение объектов – на диаграмме классов обозначалось не как «1 - 1», а как «1 - *» или «1 - n», где * и n обозначают любое произвольное количество объектов.

Композиция перечисления «один к одному»

1. Создать из ветки develop ветку Tasks/2_composition, перейти в новую ветку.
2. В проекте ObjectOrientedPractics добавить перечисление Category, описывающее категорию товара. В перечислении должно быть не менее 7 категорий товаров. Прописать xml-комментарий для перечисления.
3. Добавить в класс Item открытое автосвойство Category типа категории товара. Добавить xml-комментарий для нового свойства.
4. Добавить в конструктор класса Item дополнительный аргумент для инициализации нового свойства. Исправить xml-комментарий конструктора – добавить еще один тэг <param> для нового аргумента.
5. Теперь класс Item композирует категорию товара в варианте «1 к 1» - то есть на любой объект товара приходится одно значение категории. Необходимо добавить работу с категорией товара в пользовательский интерфейс.
6. На вкладку Items добавить выпадающий список (ComboBox) согласно макету. Переименовать элемент управления согласно требованиям RSDN:



7. В конструкторе класса ItemsTab добавить инициализацию выпадающего списка значениями перечисления (использовать метод Enum.GetValues(), аналогично заданиям прошлого семестра).

8. Изменить логику вкладки таким образом, чтобы при выборе товара в общем списке товаров, выпадающий список показывал значение категории выбранного товара.

9. Для выпадающего списка создать обработчик события SelectedIndexChanged или SelectedItemChanged. Когда пользователь выбирает новое значение в выпадающем списке, выбранная категория должна присваиваться в категорию выбранного товара.

10. Запустить приложение, убедиться, что программа работает правильно – как новая функциональность, так и ранее реализованная.

11. Проверьте правильность именования всех новых объектов, переменных и методов.

12. Добавьте xml-комментарии для нового кода. Это: добавить комментарий для перечисления, добавить комментарий для автосвойства в классе Item, добавить комментарий для нового аргумента в конструкторе Item, добавить комментарии к полям или методам, которые вы, возможно, создали в классе ItemsTab.

13. На протяжении работы не забывайте делать фиксации в репозиторий и синхронизироваться.

> **Примечание: если ранее вы реализовывали сохранение пользовательских данных в программе, то после добавления нового свойства и изменений в конструкторе, ранее сохраненные данные могут перестать загружаться. Это нормальная ситуация, так как формат данных изменился. Просто удалите ранее сохраненный файл, и внесите необходимые изменения в механизм сохранения/загрузки данных. Если сохранение и загрузка реализовывались с использованием библиотеки Newtonsoft JSON.NET, то никаких изменений не потребуется.**

Композиция класса «один к одному»

1. В приложениях есть два подхода к представлению адресов: а) адрес представляется единой строкой; б) адрес представляется в виде набора отдельных строк – индекса, города, улицы и номера дома, номера квартиры. Первый подход более простой в реализации, однако неудобный с точки зрения валидации. Второй подход предполагает более сложный пользовательский интерфейс и создание новых классов для представления адреса, но позволяет сделать более простую проверку правильности адреса или его частей. Ранее в программе был реализован первый подход. В данном блоке задания необходимо реализовать второй подход работы с адресами, обеспечив их валидацию.
2. Создайте класс Address. В классе должны присутствовать поля:

a. _index – почтовый индекс, целое шестизначное число.
b. _country – страна/регион, строка, не более 50 символов.
c. _city – город (населенный пункт), строка, не более 50 символов.
d. _street – улица, строка, не более 100 символов.
e. _building – номер дома, строка, не более 10 символов.
f. _apartment – номер квартиры/помещения, не более 10 символов.

3. Необходимо реализовать свойства для указанных полей с обязательной валидацией. При желании поля _building и _apartment можно сделать числовыми, а не строковыми, но на практике их оставляют строковыми, так как номер дома или квартиры может содержать знаки и буквы - «10», «131/1» «251а» «12 стр.7», «47 кор.5» и т.д.

4. Добавить в класс конструктор по умолчанию и конструктор с аргументами для инициализации данных объекта.

5. Проверить оформление кода и добавить xml-комментарии.

6. Логику работы с новым классом проще вынести в отдельный класс. Для этого в папке View проекта создать подпапку Controls и добавить в неё новый элемент управления AddressControl.

7. Сверстайте элемент управления согласно макету:

8. Добавьте в AddressControl поле _address типа Address, а также открытое свойство для него. По умолчанию поле _address инициализируется пустым адресом.

9. Реализуйте логику элемента управления таким образом, что при присвоении извне некоторого объекта адреса в открытое свойство, элемент управления будет показывать данный адрес.

10. Реализуйте возможность редактирования объекта адреса с помощью элемента управления.

11. Реализуйте валидацию в элементе управления – если поле введено неправильно, то оно должно подсвечиваться красным цветом, а всплывающая подсказка должна показывать ошибку.

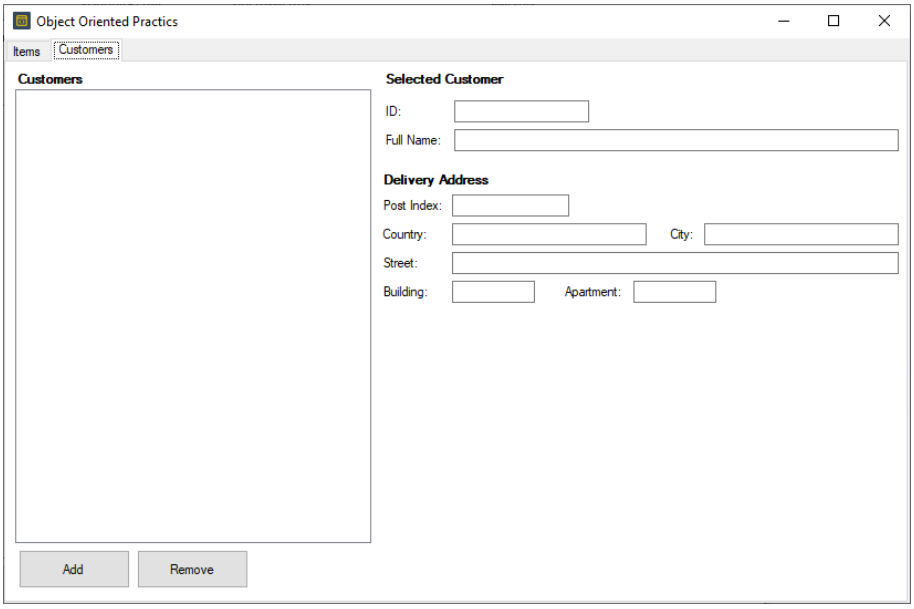
12. При попытке забрать объект адреса извне (через геттер открытого свойства), элемент управления должен гарантировать, что будут отданы все актуальные корректные данные.

13. Проверьте правильность работы элемента управления – для этого можно создать временную вкладку «AddressTab» в главном окне или новое окно. После успешной отладки, вкладку или окно можно будет удалить.

14. Если логика реализована правильно, проверьте правильность оформления кода и добавьте комментарии к коду.

15. Теперь, когда логика и элемент управления готовы, можно их внедрить в основную программу. Замените в классе Customer строковое поле _address на новое поле типа Address. Также сделайте изменения в свойстве и конструкторе класса.

16. На вкладке CustomersTab вместо текстового поля адреса разместите новый элемент управления AddressControl:



17. Исправьте логику CustomersTab таким образом, чтобы при выборе покупателя в общем списке, адрес из объекта Customer правильно передавался для отображения в AddressControl. Также убедитесь, что элемент управления AddressControl действительно позволяет редактировать адреса покупателей, и работает валидация.

18. Убедитесь, что программа работает корректно с новым элементом управления. Проверьте правильность оформления кода, проверьте и исправьте xml-комментарии у всех полей и методов, для которых вы изменили типы данных.

19. Сделайте фиксацию в репозиторий и синхронизируйтесь.

20. Теперь покупатели в нашей информационной системе имеют адрес доставки для товаров. Так как адрес является обязательными данными для любого покупателя в нашей системе, то можно смело сказать, что покупатель композирует в себе объект адреса.

Композиция класса «один ко многим»

1. В настоящий момент две вкладки нашего приложения хранят отдельные списки данных Items и Customers. Фактически, хранение в классе ItemsTab коллекции товаров само по себе является примером композиции «один ко многим» - на один объект ItemsTab приходится коллекция объектов класса Item, любое произвольное количество. Однако в приложениях, как правило, в бизнес-логике есть некоторый главный высокоуровневый класс, который предоставляет доступ ко всем пользовательским данным приложения. То есть, цель этого класса – композиция всех остальных данных приложения в себе.
2. В данном блоке заданий целью будет создание класса Store, который будет хранить данные всех товаров и покупателей, предоставляя их для вкладок ItemsTab и CustomersTab.
3. Создайте класс Store (магазин) с полями:
    a. _items – товары, тип List<Item>;
    b. _customers – покупатели, тип List<Customer>.
4. Для полей создайте свойства, а также конструктор класса (без параметров). В создании объектов класса, оба поля должны быть проинициализированы новыми пустыми списками, т.е. поля никогда не должны иметь значения null.
5. Проверьте правильность оформления кода, порядок членов класса и наличие xml-комментариев.
6. Класс Store композирует класс Item и класс Customer как «один ко многим» и является единой точкой доступа к пользовательским данным. Теперь необходимо модифицировать интерфейс на использование данного класса.
7. В классе ItemsTab добавьте открытое свойство Items типа List<Item>. Свойство должно возвращать или задавать список товаров вкладки, т.е. менять значение поля _items внутри вкладки, а также обновлять список товаров в элементе ListBox. Теперь, когда у вкладки появилось открытое свойство, возможности работы с данным элементов управления расширяются – главное окно может присваивать для отображения на вкладке разные списки товаров вместо работы всегда с одним и тем же списком.
8. Аналогично классу ItemsTab, сделайте открытое свойство в классе CustomersTab.
9. В классе главного окна добавьте поле _store и проинициализируйте его новым объектом.

10. В конструкторе главного окна после инициализации поля _store, сделайте присвоение списка товара из объекта Store во вкладку ItemsTab. Аналогично сделайте присвоение списка покупателей для вкладки CustomersTab.

11. Запустите программу и убедитесь, что программа работает верно. Проверьте правильность оформления кода и комментариев.

12. Сделайте фиксацию в репозитории и синхронизируйтесь.

13. С точки зрения пользователя, в нашей программе ничего не поменялось. Однако благодаря классу Store у нас появилось более четкое разделение обязанностей:

a. Теперь за инициализацию и хранение данных пользователя отвечает один класс Store. Главное окно хранит доступ к Store, отдавая на вкладке только нужный им для работы фрагмент данных.
b. Ранее каждая вкладка сама занималась инициализацией и хранением своих данных, но теперь вкладки только отображают переданные в них данные.
c. Если ранее вы реализовывали сохранение и загрузку данных программы, то вместо сериализации отдельных списков товаров и покупателей, вы можете сохранять объект типа Store. Другими словами, теперь будет выполняться сохранение всех пользовательских данных в виде одного объекта вместо отдельных несвязанных объектов.
d. В будущем, такие классы как Store становятся единой точкой доступа к базе данных. В программе нашего размера база данных пока не нужна, но закладывать правильную архитектуру для последующего расширения нужно уже сейчас.

> **Примечание: если ранее вы делали сохранение и загрузку данных пользователя в файл, то потребуется внести ряд изменений в программу. При появлении класса Store загрузка пользовательских данных должна выполняться в конструкторе класса MainForm, а не во вкладках ItemsTab и CustomersTab. То есть вкладки теперь ничего не знают о сериализации данных, что очень хорошо – сохранение данных это отдельная обязанность, и за её выполнение должна отвечать одна сущность, а не каждая часть пользовательского интерфейса.**
>
> **С другой стороны, теперь, когда данные хранятся в MainForm, вы можете сохранять данные только при закрытии программы, но не при каждом редактировании данных на вкладках. Это небольшое ограничение мы сможем обойти, когда изучим механизм событий.**

Проектная документация

1. По результатам всех заданий нарисуйте новую диаграмму классов.
2. На ней должны быть отражены новые перечисление и классы, а также связи между ними.
3. Для связей композиции на диаграмме обязательно подпишите ***кардинальность***, т.е. количественное соотношение объектов одного класса к другому. Обозначение композиции на диаграммах классов см. в книге Буча «Язык UML. Руководство пользователя».
4. Диаграмму сохраните под названием «Practics2.*» 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.
5. В результате выполнения всех заданий, структура проекта должна быть примерно следующей: