# **Практические задачи разработки интерфейса программного обеспечения**

## Проектирование приложения

**Цель**

Проектирование и создание прототипа пользовательского интерфейса приложения.

**Задачи**

1. Выбрать и описать тему разрабатываемого приложения.
2. Определить объем задания по выбранной теме.
3. Ознакомиться с диаграммой вариантов использования UML.
4. Разработать диаграмму вариантов использования UML для приложения по выбранной тематике.
5. Построить сценарии вариантов.
6. Создать эскизы экранных форм.

**Описание задания**

Задание выполняется индивидуально каждым студентом. В начале работы необходимо определиться с тематикой разрабатываемого приложения. Выбранная тема должна быть интересна и понятна студенту. Разрабатываемое приложение должно обязательно содержать: механизм аутентификации; главное меню; контекстное меню и табличное отображение информации.

После анализа выбранной темы и определения функций приложения необходимо разработать диаграмму вариантов использования языка UML. Дополнительная информация о построении диаграммы находится в теоретической части.

Затем необходимо построить сценарии вариантов использования для более полного описания работы приложения. Сценарий варианта использования – текстовое описание, которое поясняет суть варианта использования. В лабораторной работе используется для определения элементов интерфейса, реализующих отдельные действия сценария. Предлагается оформить результаты в виде таблиц, примеры которых представлены в теоретической части.

В конце лабораторной работы необходимо подготовить эскизы экранных форм для пояснения предлагаемых интерфейсных решений. Эскизы могут быть сделаны вручную или с помощью программ, например, Visio.

**Теория и примеры реализаций**

1. Описание вариантов использования разрабатываемого предложения.

Диаграмма вариантов использования языка UML (Unified Modeling Language) может быть применена для формализации функциональных требований к системе, в т. ч. описания взаимодействия пользователей с проектируемой системой. Следует отметить, что изобразительных средств этой диаграммы недостаточно для подробного описания требований, поэтому ДВИ дополняют текстовыми сценариями. С их помощью можно уточнить или детализировать последовательность действий, совершаемых системой при выполнении ее вариантов использования.

Вариант использования представляет собой последовательность действий, выполняемых системой в ответ на событие, инициируемое некоторым внешним объектом (действующим лицом) [3].

На диаграмме вариантов использования языка UML рисуются актеры, варианты использования и связи (стрелки), которые их соединяют. Пример простейшей диаграммы вариантов использования приведен на рисунке 4.1.



Рис.4.1. Пример простейшей диаграммы вариантов использования

Актер (действующее лицо) – это роль, которую пользователь играет по отношению к системе [3]. При работе с программной системой все пользователи делятся на определенные группы. Каждая группа имеет определенные права, и может выполнять только разрешенные ей функции. Каждая такая группа описывается определенной ролью, например, клиент, администратор, оператор.

Связи, используемые в диаграммt вариантов использования, могут быть следующих четырех типов [2]:

* связь ассоциации – связь между вариантом использования и актером, показывающая возможность использования актером прецедента (сплошная линия);
* связь включения - связь между двумя вариантами использования, которая указывает, что некоторое заданное поведение для одного варианта использования включается в качестве составного компонента в последовательность поведения другого варианта использования (пунктирная линия со стрелкой и словом «include»);
* связь расширения - связь между двумя вариантами использования, которая указывает, что один из вариантов использования может присоединять к своему поведению некоторое дополнительное поведение, определенное для другого варианта использования (пунктирная линия со стрелкой и словом «extend»);
* связь обобщения – связь между элементами диаграммы, показывающая, что один элемент является частным случаем другого элемента.

На рисунке 4.2 показан пример диаграммы вариантов использования со связью обобщения.



Рис. 4.2. Пример диаграммы вариантов использования со связью обобщения

Более подробно о диаграммах вариантов использования можно прочитать в [1] и [2].

Следует отметить, что диаграммы вариантов использования не используется для функциональной декомпозиции системы. Придумывать названия для вариантов использования следует с т. з. пользователя, а не программиста.

Поскольку в лабораторной работе разрабатывается прототип ПИ, то следует дополнить каждую роль описанием по плану, приведенному ниже. Эта информация позволит предложить решения ПИ, которые наилучшим образом подойдут данной группе пользователей. Например, ПИ для новичков и опытных пользователей будет отличаться. Возможно, с разрабатываемым вами ПИ будут работать люди с ограниченными возможностями. Не обязательно использовать все пункты для каждой роли.

* требуемые знания о предметной области, приложении, прочите знания;
* умения, т.е. уровень мастерства в работе с системой;
* описание взаимодействия с приложением:
* частота нахождения в данной роли;
* регулярность;
* непрерывность;
* равномерность распределения работы во времени;
* интенсивность (скорость взаимодействия).
* описание используемой информации:
* каковы источники входящей информации, предоставляемой пользователем;
* объем информации;
* сложность информации.
* критерии практичности интерфейса;
* функциональная поддержка: особые требования к функциональности;
* практический риск: оценка потенциальных убытков при работе с системой;
* аппаратные ограничения (если есть);
* среда (возможно условия, в которых будет работать пользователь, накладывают какие-то ограничения на интерфейсные решения).

Рассмотрим пример создания диаграммы вариантов использования UML с помощью среды разработки Visual Studio. Заходим в среду разработки. В пункте меню выбираем «Архитектура» -> «Создать схему…».

В диалоговом окне «Добавление новой схемы» выбираем шаблон «Схема вариантов использования UML». Далее будет предложено создать проект модели, прописываем имя проекта «OK» (рис.4.3).

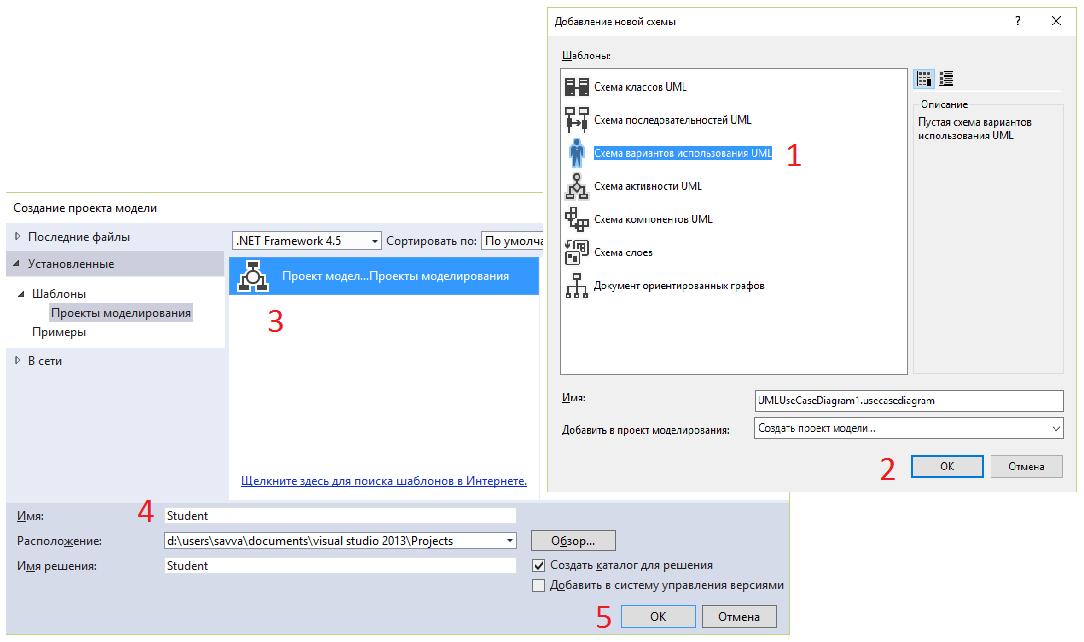


Рис.4.3.Создание шаблона UML-диаграммы в Visual Studio

В открывшемся окне необходимо открыть вкладку «панель инструментов», где содержаться весь необходимый инструментарий для выполнения задания по лабораторной работе.

Сценарий варианта использования – текстовое описание, которое поясняет суть варианта использования. В лабораторной работе используется для определения элементов интерфейса, реализующих отдельные действия сценария. Предлагается оформить результаты в виде следующих таблиц. (см. табл.4.1– 4.3).

Таблица 4.1

Аутентификация. Главный раздел

|  |  |
| --- | --- |
| **Вариант использования** | Ввести логин и пароль |
| **Роль** | Пользователь |
| **Цель** | Зайти в систему под своей учетной записью |
| **Краткое описание** | Пользователь запускает приложение. В появившейся форме вводит логин и пароль, затем нажимает кнопку. |
| **Тип** | Базовый |
| **Ссылки на другие варианты использования** | Нет |

После краткого описания варианта использования в следующей таблице необходимо привести более детальную последовательность действий (табл.4.2) при типичном ходе событий.

Таблица 4.2

Аутентификация. Раздел «Типичный ход событий»

|  |  |  |
| --- | --- | --- |
| **Типичный ход событий** | | |
| **Действия актера** | **Отклик системы** | **Элемент интерфейса** |
| 1. Пользователь запускает приложение  3. Пользователь вводит логин и пароль и нажимает кнопку Вход.  Исключение № 1: указаны неверно логин или пароль. | 2. Приложение показывает на экране форму для ввода имени пользователя и пароля.  4. Приложение закрывает форму авторизации и открывает главную форму приложения.  4.а. (см. исключения) | Поле ввода логина–элемент TextBox. Поле ввода пароля –элемент MaskedTextBox. Кнопки Вход и Отмена. |

В таблице4.3 описано исключение из типичного хода событий.

Таблица 4.3

Аутентификация. Раздел «Исключения»

|  |  |  |  |
| --- | --- | --- | --- |
| **Исключения** | | | |
|  | **Действия актера** | **Отклик системы** | **Элемент интерфейса** |
| 1. Указаны невер-но логин /пароль. | Пользователь вводит логин и пароль и нажимает кнопку Вход. | 4.а. На экране появляется сообщение «Логин или пароль введены неверно». | Стандартное диалоговое окно с текстом. |

Рассмотрим примерописания задания для системы тестирования. На рисунке4.4 показана диаграмма вариантов использования для этой системы.



Рис.4.4. Диаграмма вариантов использования для системы тестирования

Система тестирования прежде всего требуется следующим заинтересованным лицам:

* обучаемому (студенту);
* составителю тестов (преподавателю);
* преподавателю, принимающему экзамен;
* сотруднику деканата, осуществляющему контроль за успеваемостью;
* администратору сети и баз данных учебного учреждения.

На начальном этапе создания системы мы можем ограничиться только двумя важными для нас ролями действующих лиц:

* студент (тестируемый);
* администратор (он же преподаватель, он же составитель тестов).

Соответственно основные прецеденты (варианты использования) для нашей системы следующие:

Прецедент для студента:

* П1 — пройти тестирование.

Прецеденты для администратора:

* П2 — создать/изменить тест;
* ПЗ — просмотреть результаты тестирования;
* П4 — добавить/изменить пользователей и др.

Краткая форма описания содержит название варианта использования, его цель, действующих лиц, тип варианта использования (базовый, второстепенный или дополнительный) (см. табл.4.4).

Таблица 4.4

Краткое описание варианта использования

|  |  |
| --- | --- |
| **Название варианта использования** | Прохождение тестирования |
| **Роль** | Студент |
| **Цель** | Прохождение тестирования, получение оценки. |
| **Краткое описание** | Регистрация студента, запуск теста, выбор ответа или ввод ответа, завершение теста, получение оценки |
| **Тип варианта** | Базовый |

Подробная форма описания варианта использования представлена в табл.4.5.

Таблица 4.5

Подробное описание варианта использования прохождения теста

|  |  |
| --- | --- |
| **Действия исполнителя** | **Отклик системы** |
| 1. Студент вводит свои данные (ФИО, Группа), т.е. регистрируется в системе. | 2. Система создает на диске файл с результатом тестирования и предлагает выбрать тест. |
| 3. Студент выбирает тест. | 4. Система запускает тест. |
| 5. Студент последовательно отвечает на вопросы. | 6. Система регистрирует правильные и неправильные ответы. |
| 7. Студент завершает тестирование. | 8. Система подсчитывает процент правильных ответов. |
| 9. Студент ожидает результата. | 10. Система демонстрирует результат и предлагает сохранить его. |
| 11. Студент решает, сохранить результат или нет. | 12. Если выбрано сохранение, система записывает результат в файл. |
| 13. Студент завершает работу. | 14. Система завершает работу. |

Для защиты лабораторной работы необходимо представить отчет, содержащий цель работы, ход выполнения работы и выводы.

## Проектирование пользовательского интерфейса на этапе высокоуровневого проектирования

**Цель**

Получить практические навыки по проектированию пользовательского интерфейса на этапе высокоуровневого проектирования.

**Задачи**

1. Ознакомиться с возможностями создания форм в Visual Studio C#.
2. Разработать интерфейс приложения в соответствии с выбранной темой в первой лабораторной работе.
3. Убедиться, что разработанный интерфейс не содержит типичных проблем, возникающих на этапе разработки прототипа ПИ (см. раздел 1.4).

**Описание задания**

Основная задача разработать интерфейс приложения в соответствии с выбранной и описанной темой в первой лабораторной работе. В результате должно получиться полностью оформленное приложения (не отработка функционала, а именно отображение всех объектов, которые необходимы для решения поставленной задачи). Если в выбранной теме присутствует какие-то табличные элементы их необходимо оформить в виде DataGrigView и статический заполнить набором данных которые предположительно там будут находиться.

При разработке интерфейса обязательно необходимо предусмотреть использование и реализации следующих объектов:

* у приложения обязательно должно иметься меню включая в себя как минимум 3 пункта. Два из которых это «О программе …», «Выход»;



Рис.4.5 – пример оформления меню

* не менее 3 форм, с возможностью перехода между ними;
* в местах где требуется список элементов (объектов) использовать контейнер DataGrigView. В котором должен быть предусмотрено контекстное меню (например, на удаление, добавление и изменение);
* обязательно должна быть предусмотрена аутентификация пользователей.

В приложении должно быть минимум две роли: простые авторизированные пользователи и администраторы. У администратора должна быть возможность просматривать списки зарегистрированных пользователей, оформленных в виде DataGrigView, создавать нового пользователя, а также удалять уже существующих пользователей. Также администратор должен иметь возможность изменять пароли пользователей. Следует отменить, что в лабораторной работе № 2 необходимо спроектировать только интерфейсные формы для аутентификации и работы администратора. Их функционирование реализуется в лабораторной работе № 4.

**Теория и примеры реализаций**

Ниже приводится пример создания Windows Forms приложения в среде разработки Visual Studio 2015.

1. Запустить среду разработки Visual Studio.
2. Создать новое приложение, выбрав в меню File\New\Project. На экране появится следующая картинка (рис.4.6).

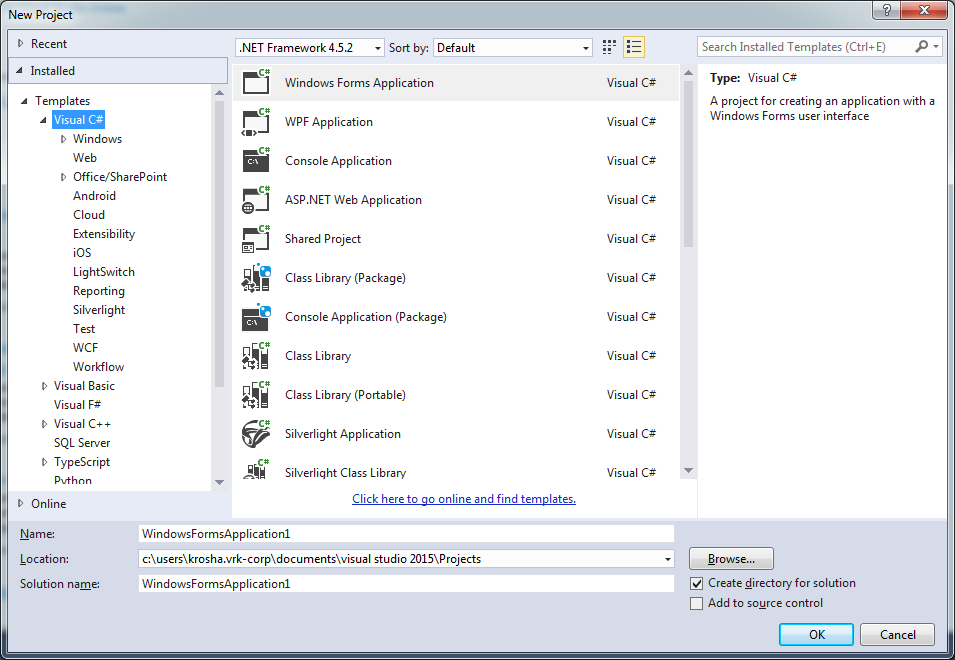


Рис.4.6. Форма создания нового приложения

На форме необходимо выбрать шаблон (Template) – Visual C#, Windows, Windows Forms Application. Затем указать название приложения, место на диске, куда оно будет сохранено, и нажать кнопку Ок. После этого Visual Studio создаст и откроет проект (см. рис.4.7).

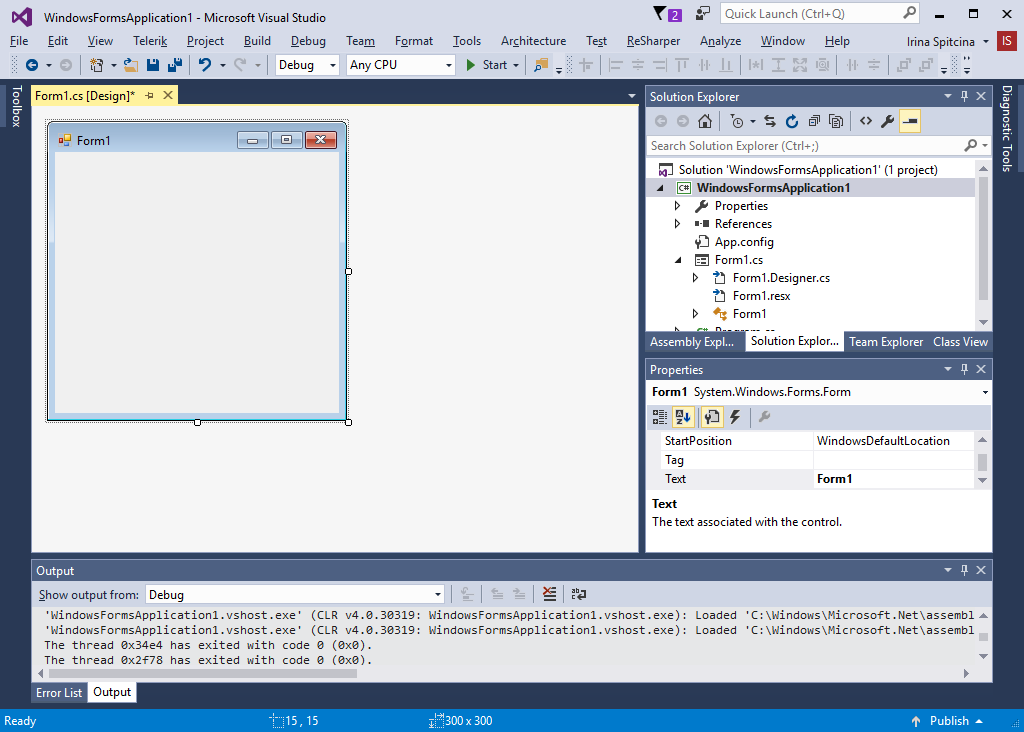


Рис.4.7. Форма созданного приложения

Рабочее окно состоит из следующих частей:

* панель инструментов (Toolbox) – располагается слева, содержит элементы управления, которые формируют пользовательский интерфейс (см. рис. 4.8). Вызвать ее также можно через меню View\Toolbox;

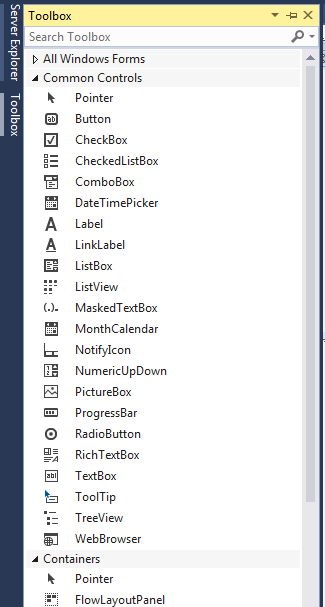


Рис.4.8. Панель инструментов (Toolbox)

* окно Solution Explorer (Обозреватель решений) – располагается справа, содержит информацию о составе проекта - узлы: Properties - настройки проекта, Links (Ссылки) - подключенные к проекту библиотеки, а также созданные и подключенные к проекту файлы исходных кодов (с расширением .cs) и подключенные к проекту формы (например, Form1) см. рис.4.9.

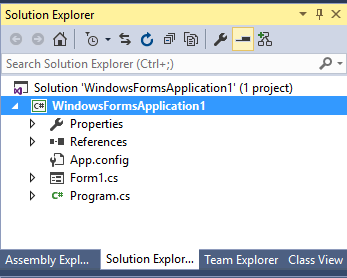


Рис.4.9 Окно Solution Explorer (Обозреватель решений)

* окно Properties (Свойства) – окно, позволяющее просмотреть и настроить свойства и методы выбранного элемент управления или формы (см. рис. 4.10).

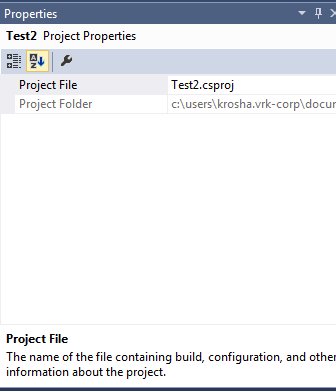


Рис.4.10. Окно Properties (Свойства)

1. На основе выбранного шаблона Visual Studio, создался каркас оконного приложения с главной формой Form1. Класс, ее описывающий, выглядит следующим образом (см. рис. 4.11).

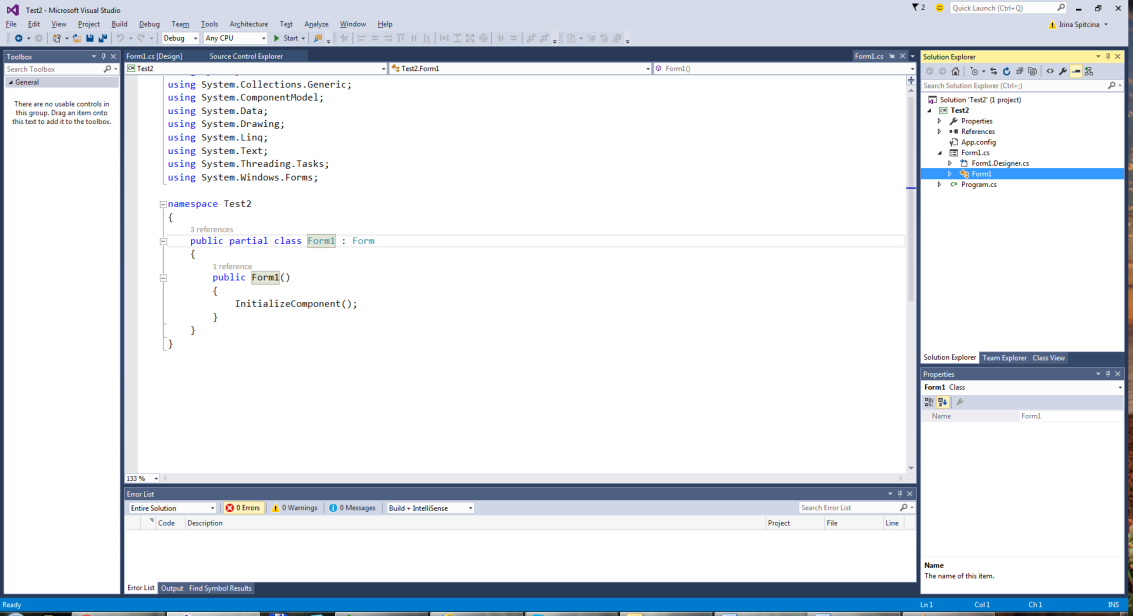


Рис.4.11. Описание класса Form1

Предлагаются следующие способы переключения между дизайном формы и описанием класса:

* выбрать в Solution Explorer Form1 (описание класса) или Form1.cs (вид формы) (см. рис.4.12);
* выбрать в меню View\Code (F7) либо View\Designer (Shift+F7).

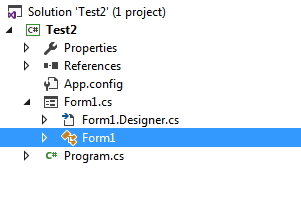


Рис.4.12. Выбор в Solution Explorer описание класса Form1

Создание оконных приложений сводится к созданию всех необходимых диалоговых окон, а также к размещению на них необходимых элементов. В дальнейшем необходимо прописать обработку событий, создаваемых пользователем, и настроить технические аспекты работы программы.

1. Приведем пример размещения кнопки на форме и написания обработчика события Click (нажатие кнопки).

Найти на панели инструментов (Toolbox) нужный элемент Button (кнопка) и переместить на форму (см. рис.4.13).

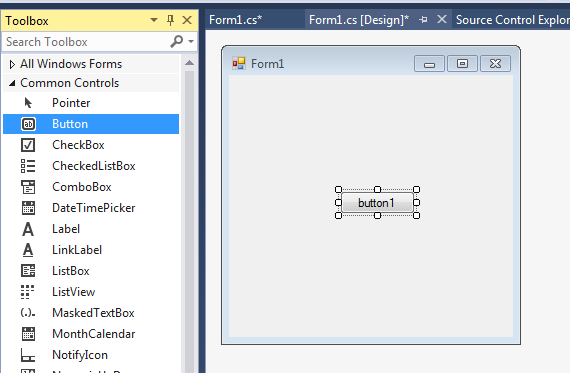


Рис.4.13. Размещение кнопки на форме

В окне Properties (Свойства) найти свойство Text и изменить его значение на «Закрыть» (см рис. 4.14).

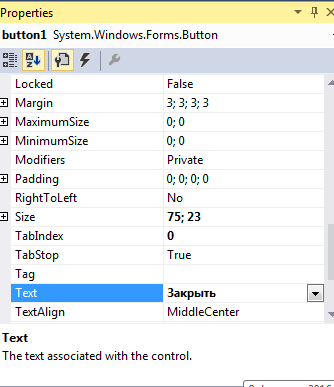


Рис.4.14 Изменение свойства Text элемента управления button1

В окне Properties (Свойства) переключиться на вкладку Event (события), нажав кнопку , и найти событие Click (см. рис.4.15).

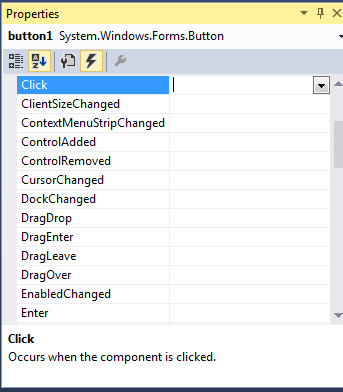


Рис.4.15. Вкладка Event

Двойным щелчком мыши на событии создать его обработчик (см. рис. 4.16).

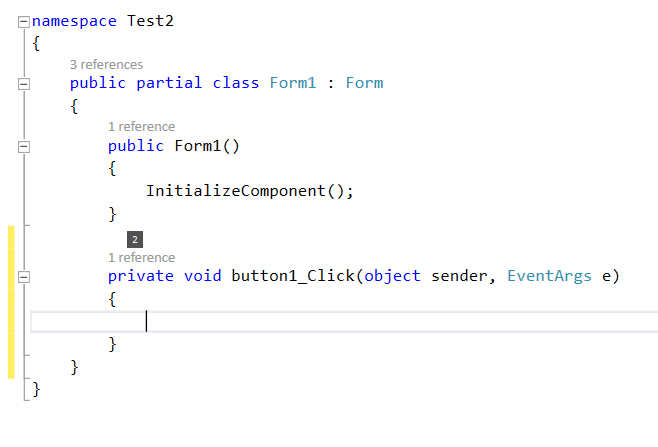


Рис.4.16. Обработчик события Click

Для закрытия формы вызовем метод Close().

Запустим наше приложение (F5) или кнопка Start (см. рис.4.17) и убедимся в работоспособности приложения.

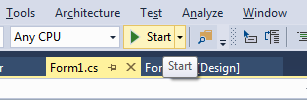


Рис.4.17 – Кнопка Start для запуска приложения

Ниже приводится пример перехода (открытия) форм.

Изначально приложение по умолчанию создает одну форму и при запуске выводит ее. Когда же появляется несколько форм, необходимо осуществить переход между ними.

Создаем вторую форму: в «обозревателе решений», нажимаем правой кнопкой мышки по проекту -> «Добавить» -> «Форма Windows» -> «OK».

На первой форме Form1, создаем кнопку, как это было описано ранее. В конструкторе кликаем дважды по кнопке «button1» (см. рис.4.18). При этом автоматический создастся событие «button1\_Click».

Для перехода необходимо создать экземпляр класса формы, которую мы хотим отобразить и вызвать метод отображения.

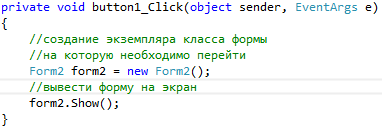


Рис. 4.18. Пример кода для открытия второй формы

В данном случае при запуске приложения и нажатии на кнопку button1, отобразиться вторая форма. Но формы будут обе активны, то есть работать можно будет и с первой формой и со второй. Иногда требуется при открытии второй формы первая блокировалась, в место метода Show() должен вызываться метод ShowDialog().

Пример использования компонента DataGridView.

Для отображения табличных данных обычно используется компонент DataGridView(см. рис.4.19).

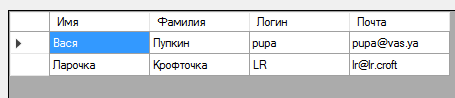


Рис. 4.19. DataGridView для таблицы студенты

Заполнить данными на данном этапе лучше всего из конструктора формы см. рис. 4.20.

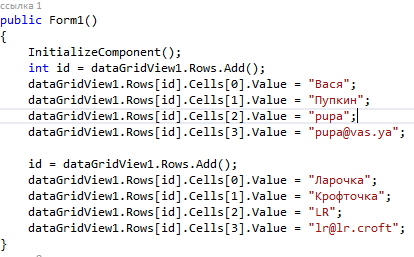


Рис.4.20. Заполнение DataGridView данными

Кратко опишем понятие *Высокоуровневое проектирование*.

Высокоуровневое проектирование — разработка на основе сценариев вариантов использования общей модели пользовательского интерфейса приложения с описанием структуры форм и переходов между ними.

Изучая пользовательские сценарии и диаграмму вариантов использования, можно выделить функциональные блоки приложения, а также операции, выполняемые пользователями в этих блоках, и объекты, над которыми эти операции делаются. После этого необходимо сгруппировать отдельные выделенные элементы, исходя из их логической связи. Таким образом, получается структура главного меню и других навигационных элементов (выпадающих меню, кнопок, инструментальных панелей).

При разработке меню следует придерживаться определенных стандартов. Это ускорит обучение пользователя работе с новым приложением за счет использования им уже сформированных ранее привычек. Перечислим некоторые рекомендации (см. рис.4.21):

* пункты меню, относящиеся к одной функциональной группе, следует отделять разделителем (черта или пустая строка);
* названия пунктов меню пишутся с большой буквы, желательно, чтобы название состояло максимум из двух слов;
* название меню, к которому относится каскадное меню, следует заканчивать стрелкой;
* в названиях меню можно использовать пиктограммы;
* недоступные пункты меню показывать другим цветом (обычно серым);
* можно делать невидимыми недоступные пункты меню.

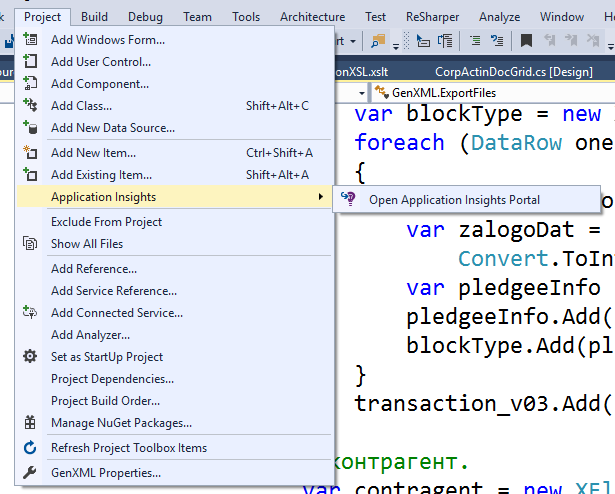


Рис.4.21. Пример оформления пунктов меню

В конце лабораторной работы следует убедиться, что разработанный прототип приложения:

* корректно отображается при разных условиях (размер экрана монитора, разрешение экрана);
* содержит корректно выбранные компоненты для ввода и отображения информации;
* содержит контекстные подсказки, контекстные меню;
* содержит информирующие и\или предупреждающие сообщения для пользователей;
* имеет несколько возможностей вызова функций главного меню (панель инструментов, «горячие» клавиши).

Дополнительную информацию о разработке можно почитать в [9].

Для защиты лабораторной работы необходимо представить отчет, содержащий цель работы, ход выполнения работы и выводы.

## Разработка функций приложения, позволяющих взаимодействовать с папками и файлами

**Цель**

Получить практические навыки по разработке функций приложения, позволяющих взаимодействовать с папками и файлами.

**Задачи**

1. Ознакомиться с классами, которые реализуют взаимодействия с папками и файлами.
2. Доработать приложение, добавив функцию записи действий пользователя: основные действия пользователя и возникшие сообщения об ошибках должны сохраняться в лог-файле. Структура лог-файла разрабатывается студентов самостоятельно.
3. Добавить функцию, которая позволить отображать дерево каталогов и файлов.
4. При размещении информации на форме использовать принцип Золотого сечения.

**Описание задания**

Данная лабораторная работа может выполняться как доработка функционала уже разработанного ранее приложения, но может быть реализована как самостоятельное приложение.

Необходимо разработать функцию приложения, позволяющую просматривать содержимое директории, осуществлять переход по директориям, а также предоставлять полную (вся которая только имеется) информацию о выделенном объекте (например, имя файла, дата создания, разрешения, владелец, расширение и т.д.).

Для любых текстовых документов и изображений реализовать вывод информации на экран, например, при выделении текстового документа с расширением txt в специально отведенное окно должно отобразиться информация (содержимое) документа, точно также с изображением.

Добавить в приложение настройку (с возможностью сохранения), которая позволит отображать выбранный файл в ассоциированном с ним приложении. Например, файл в расширение doc в приложение Word.

Продумать структуру лог-файла приложения. Сохранять все основные действия пользователя (выбор пунктов меню, нажатие на кнопки формы) и сообщения об ошибках в лог-файле.

**Теория и примеры реализаций**

В стандартном наборе компонент C# существую не визуальные компоненты **OpenFileDialog** и **SaveFileDialog**, с помощью которых создаются диалоги открытия и сохранения файла.

Ниже перечислены основные свойства OpenFileDialog и SaveFileDialog:

* свойство AddExtension – разрешает или запрещает автоматическое добавление расширение, указанное в свойстве DefaultExt;
* свойство DefaultExt – расширение, принятое по умолчанию для автоматического добавления к имени файла при AddExtension = true;
* свойство CheckFileExists – используется для получения или устанавки значения, указывающего отображать или нет диалоговое окно предупреждения, если пользователь указал в свойстве FileName имя файла, которого не существует в данной директории и нажал кнопку "Oткрыть" при невыбранном кликом мышки файле. При CheckFileExists = true вместо прерывания будет выдано сообщение, что такого файла нет и исключения не вызывается;
* свойство FileName – имя файла по умолчанию для выборки если была нажата кнопка OK и не выбран кликом мышки файл в окне диалога;
* свойство CheckPatchExists – используется для получения или установки значения, указывающего отображать или нет диалоговое окно предупреждения, если пользователь указал в свойстве FileName имя файла с несуществующим именем директории;
* свойства Filter, FilterIndex – фильтр для выбираемых файлов и индекс строки, отображаемой в окошечке "Имя файла".

Например: значение свойства заданного строкой Filter = “rtf файлы (\*.rtf)|\*.rtf|txt файлы(\*.txt)|\*.txt ” и при FilterIndex = 1.

Данные настройки позволит установить выбор только текстовых файлов в формате rtf или txt. В окошечке "Тип файла" будет только две строки (см. рис. 4.22). По умолчанию из начально будет установлен первый фильтр, то есть по rtf.

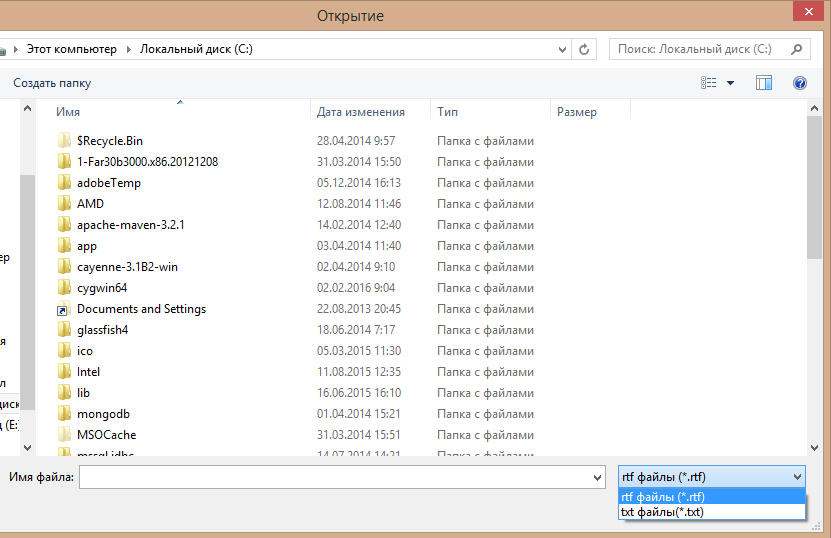


Рис.4.22. Окно компонента OpenFileDialog с фильтром для выбираемых файлов

* свойство InitialDirectory – директория, которая выбирается при старте OpenFileDialog;
* свойство MultiSelect – при значении true позволяет выбрать мышкой при нажатой кнопке Shift или Ctrl несколько файлов и сохранить их имена в свойстве FileNames в виде массива строк;
* свойство ReadOnlyChecked – при значении true позволяет открывать команде OpenFile выбранные файлы только в режиме чтения;
* свойство Title – заголовок диалогового окна.

Основной методкомпонетовOpenFileDialog и SaveFileDialog–метод ShowDialog(), который позволяет открыть диалоговое окно.

Для работы с каталогами файлов существует компонент **FolderBrowserDialog**, который обеспечивает просмотра, создание, и выбора рабочегокаталога.Перечислим Основные свойства компонента FolderBrowserDialog:

* свойство RootFolder задает одну из системных папок и смещает корень дерева просмотра на данную папку, делая недоступными для просмотра все другие папки выше выбранной. Так, задав в качестве свойства "Programs", доступными для просмотра будут только папка меню "Программы" со всеми ее вложениями;
* свойство SelectedPath позволяет задать папку, отображаемую при старте как выбранную. Это свойство не влияет на выбор при задании имени папки вне доступности, определяемой свойством RootFolder. Свойство имеет встроенный редактор (кнопочку с тремя точками, появляющуюся при выборе свойства, нажатие на которую вызывает редактор "Обзор папок"), позволяющий обычным для Windows способом выбрать или создать папку;
* свойство ShowNewFolderButton разрешает показ кнопки "Создать новую папку", а тем самым и создание папки из программы;
* свойство Description позволяет задать текст, который будет выведен над окном выбора папки.

Для визуального отображения структуры каталогов и файлов можно использовать компоненты **ListView** и **TreeView**.

Теперь кратко ознакомимся с классами для работы с файлами из пространства имен System.IO.

Класс **File**– класс предоставляет статические методы для создания, копирования, удаления, перемещения и открытия одного файла, а также помогает при создании объектов FileStream.

Перечислим основные методы класса:

* Delete(stringpath) – удаляет файл c именем и в каталоге, определенным path;
* Exists(stringpath) – проверяет наличие файла в указанном каталоге - при наличии файла возвращает true;
* Create(stringpath), или Create (stringpath, intbufersize) – используется для создания файла c именем и в каталоге, определенным path и с указанным размером буфера. Методы возвращают объект типа FileStream. Исключение возникает если диск закрыт для записи;
* Open(string path, FileModemode, FileAccessacess, FileShareshare) – открывает файл по указанному пути, с указанным режимом чтения, записи и доступа для чтения - записи и указанной опцией совместного использования. При открытии создает объект типа FileStream. Обязательными параметрами являются только первые два;
* CreateText(string path) – создает файл, который будет открыт для записи. Метод возвращает объект типа StreamWriter, который может быть использован для записи текста в файл. Исключение возникает если диск закрыт для записи;
* OpenText(string path) – используется для открытия существующего текстового файла и создания объекта типа StreamReader;
* AppendText(string path) – создает объект типа StreamWriter, который можно использовать для добавления текста в кодировке UTF-8 в конец файла;
* Copy(string sourepath, string destpatch, bool overwrite) – копирует существующий файл по новому адресу. При overwrite = false перезапись файла при его наличии не происходит;
* Move(string sourepath, string destpatch) – переносит существующий файл по новому адресу.

Класс **Path**– класс выполняет операции для экземпляров класса String, содержащих сведения о пути к файлу или каталогу. Эти операции выполняются межплатформенным способом.

Перечислим основные методы класса:

* GetFileName(string path) - возвращает имя файла вместе с расширением, но без пути.
* GetFileNameWithoutExtension(string path) – возвращает имя файла без расширения и без пути;
* GetExtension(string path) – возвращает расширение имени файла;
* GetFullPath(string path) – возвращает полное имя файла;
* GetDirectoryName(string patch) – возвращает имя каталога для указанного файла.

Класс **Directory**– класс предоставляет статические методы для создания, перемещения и перечисления в каталогах и вложенных каталогах.

Перечислим основные методы класса:

* CreateDirectory(string path) – создает директорию с указанным названием;
* Exists(string path) – проверяет существование директории с указанным названием.

Класс **FileInfo** наследник абстрактного класса **FileSystemInfo** используется для создания, копирования, удаления, перемещения, и открытия файлов, и при использовании ряда методов создает объекты типа FileStream.

Класc **FileSystemInfo** содержит свойства и методы, общие для управления файлами и директориями.

Рассмотрим вопрос запуска стороннего приложения из своего для отображения выбранного файла в ассоциированном с ним приложении.

Метод **Process.Start(String)**– запускает ресурс процесса путем указания имени документа или файла приложения и связывает ресурс с новым компонентом Process. Можно использовать для открытия файла в ассоциированном приложении.

При решении вопроса о том, как лучше разместить на форме элементы ПИ можно использовать практические принципы построения ПИ [10]:

* золотое сечение;
* кошелек Миллера;
* принцип группировки;
* бритва Оккама или KISS;
* видимость отражает полезность;
* умное заимствование.

Принцип золотого сечения был открыт еще в древности. Золотое сечение – это такое пропорциональное деление отрезка на неравные части, при котором весь отрезок так относится к большей части, как сама большая часть относится к меньшей. Если принять весь отрезок за единицу, то отношение его частей будет равно 1,618.

Золотой прямоугольник– это фигура с длинной стороной, равной короткой умноженной на 1.618. Его можно использовать в макетах ПИ. Для этого необходимо узнать ширину и высоту формы, а затем поделить ее на части, пропорциональные золотому прямоугольнику. Далее его можно применять кнопкам, панелям, изображениям и тексту.

Принцип кошелька Миллера говорит о том, что при группировке элементов ПИ (пунктов меню, кнопок, закладок и т.п.) следует учитывать возможность человека без усилий запоминать семь сущностей. Следовательно, в одной группе не должно быть больше 7±2 элемента.

Принцип группировки означает, что при размещении элементов ПИ на форме необходимо располагать их осмысленными группами. Например, на рис. 4.21 пункты меню объединены в группы.

Принцип «Бритва Оккама» означает, что:

* любая функция приложения должна выполняться минимальным числом действий;
* логика этих действий должна быть очевидной для пользователя;
* движения курсора и даже глаз пользователя должны быть оптимизированы.

Принцип «Видимость отражает полезность» состоит в том, что наиболее часто используемые элементы управления ПИ должны находиться на переднем плане и быть легко доступны пользователю. Например, панели инструментов, которые меняются в зависимости от того, какие задачи решает пользователь.

Принцип «Умное заимствование» означает, что при разработке своего ПИ следует использовать широко распространенные приемы дизайна, которые уже стали де-факто стандартом. Пользователю легче будет знакомиться с вашей разработкой.

Для защиты лабораторной работы необходимо представить отчет, содержащий цель работы, ход выполнения работы и выводы.

## Работа с методами сериализации и десериализации объектов

**Цель**

Получить практические навыки работы с методами сериализации и десериализации объектов.

**Задачи**

1. Ознакомиться с методами сериализации и десериализации объектов.
2. Разработать приложение в соответствии с заданием сохраняя сериализованные данные в файле.
3. Реализовать механизм шифрования паролей пользователей с помощью алгоритмаMD5.
4. Провести анализ удобства выполнения одной функции разрабатываемого приложения с использованием закона Фитса.

**Описание задания**

В разделе 4.2 был показан процесс разработки формы для аутентификации пользователей приложения и работы администратора системы (просмотр, изменение и удаление информации о пользователях). В данном разделе показано, как завершить разработку этих функции приложения.

Разработанная система должна сохранять все данные о пользователях в файле в виде сериализованных объектов (списков объектов). Так же у администратора должна быть возможность удалять элементы из сохраненных данных. Например, администратор может удалять любого пользователя и всю информацию, связанную с ним.

Пароли должны храниться в зашифрованном виде. Предлагается для этого использовать хеш-функцию MD5.

В конце лабораторной работы необходимо выбрать какую-нибудь уже реализованную функцию приложения, например, просмотр данных. Оценить время выполнения пользователем этой функции, используя закон Фитса. Провести анализ полученных результатов. Предложить изменения ПИ, которые позволили бы сократить время работы пользователя. Например, заменить кнопки на контекстное меню или предложить выполнение функции другим способом.

**Теория и примеры реализаций**

Сериализация объекта – процесс сохранения состояния объекта в последовательность байт. Десериализация– процесс обратного восстановления объекта из сохраненной последовательности байт. Сериализация / десериализация – удачный механизм для передачи объектов.

Рассмотрим, как работает этот механизм. Пусть разработан некоторый класс*Students*. Код этого класса приведен на рис.4.23. Ключевое слово [Serializable] показывает, что класс Students – сериализуемый. Если есть поля, которые по какой-то причине нужно исключить из сериализации, их необходимо пометить ключевым словом [NonSerialized].

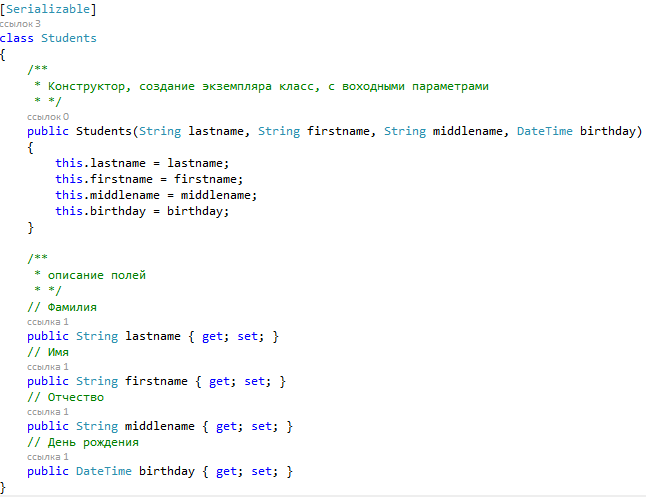


Рис. 4.23. Пример сереализуемого класса

Для наглядности создадим список элементов сериализованного класса Students, как показано на рис. 4.24.

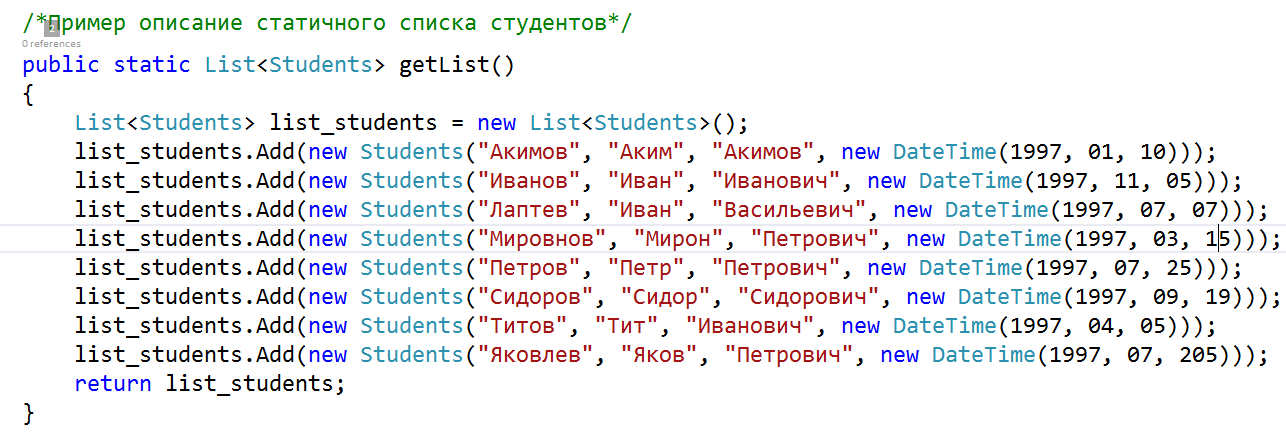


Рис. 4.24. Пример создание набора данных студентов для дальнейшего сохранения в файл

Для сохранения сериализованного объекта в файл напишем метод Save(), код которого приведен на рис. 4.25.

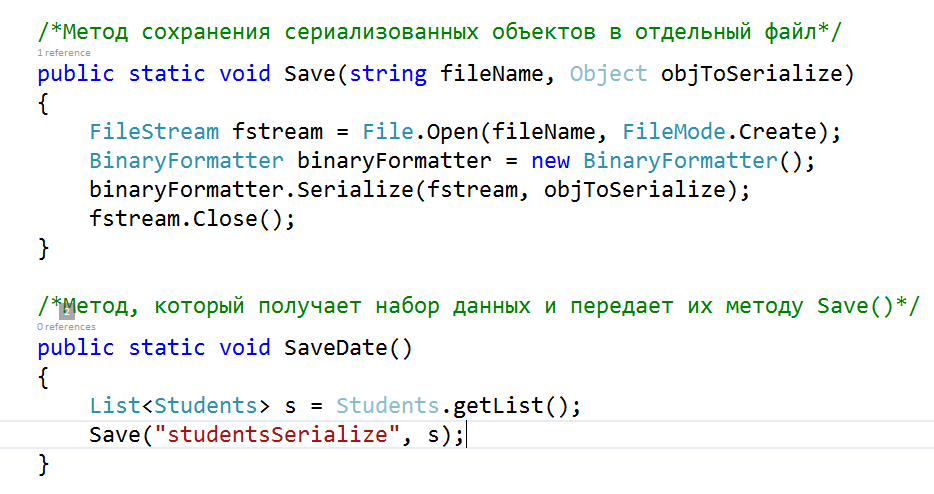


Рис. 4.25. Пример сохранения сериализованных объектов в файл

После выполнения метода SaveData() в том же каталоге, что и запускаемое приложение будет создан файл studentsSerialize сохраненным сериализованным объектом.

Для десериализации объекта из файла напишем метод ReadObjectFromFile(), код которого приведен на рис. 4.26.

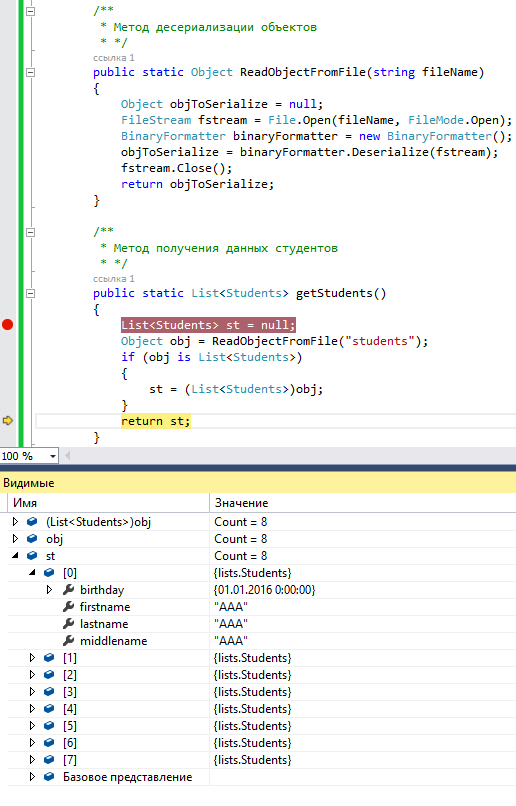


Рисунок 4.26 – Десериализация объекта из файла

В методе getStudents добавлена проверка типа объекта, который считывается из файла, чтобы избежать возникновения ошибки несоответствия типов. Ниже на рис. 4.26 показан десериализованный из файла объект.

Теперь рассмотрим вопрос шифрования паролей. Из соображений безопасности не стоит хранить пароли в открытом виде. Один из простых способов «шифрования» паролей – это применение хеш-функции MD5.

Хеш-функция – это математическое преобразование входного массива данных в определенной длины строку. Для MD5 длина получившейся строки равна 128 битам.

ВC# разработан абстрактный классMD5 (пространство имен System.Security.Cryptography), который реализует алгоритм MD5. Перечислим основные его методы:

* Create() – создает экземпляр реализации по умолчанию MD5 хэш-алгоритма;
* ComputeHash(Byte[]) – вычисляет хэш-значение для заданного массива байтов;
* Equals(Object) – определяет, равен ли заданный объект текущему объекту.

Следует отметить, что обратного не существует обратного преобразования получившейся строки в исходный массив данных. Для проверки пароля пользователя необходимо приводить вводимый пароль к хеш-значению и сравнивать с сохраненным в настройках.

Пример реализации метода шифрования и получившаяся строка приведены на рис. 4.27.

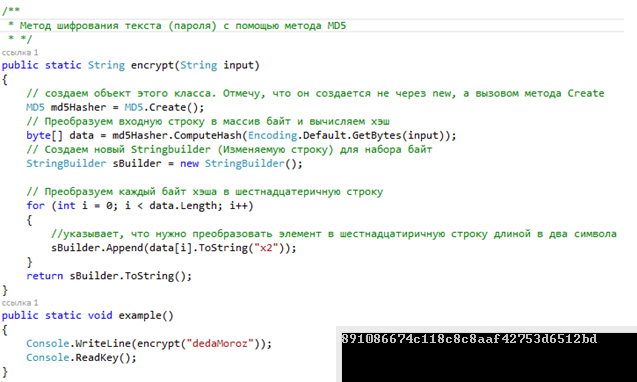


Рис. 4.27. Пример реализации функции шифрования по алгоритму MD5

Для анализа скорости работы пользователя в разрабатываемой системе можно использовать закон Фитса, который позволяет вычислить время, необходимое для того, чтобы достичь объекта. Закон Фитса может быть записан в виде следующей формулы:

*T* = LOG2(,

где *Т* – время достижения цели;

*D* – расстояние до цели;

*W* – размер цели вдоль линии перемещения курсора.

Таким образом, чем дальше находится объект от текущей позиции курсора или чем меньше размеры этого объекта, тем больше времени потребуется пользователю для перемещения к нему курсора.

Отметим, что не стоит всегда слепо использовать закон Фитса. Существуют ограничения, при которых он неприменим. Например, пользователь не всегда перемещает курс мыши к цели уверенно и по прямой траектории. Однако база, лежащая в основе этого закона, остается.

Для защиты лабораторной работы необходимо представить отчет, содержащий цель работы, ход выполнения работы и выводы.

## Документирование

**Цель**

Получить практические навыки работы по использованию возможностей приложений Microsoft Word и Excel для отображения информации из разрабатываемой информационной системы.

**Задачи**

1. Ознакомиться с возможностями создания документов MicrosoftWord и Excel с данными из приложения, разработанного в ходе предыдущих лабораторных работ.
2. Разработать функции формирования отчетов в приложениях Microsoft Word и Excel.
3. Провести эвристическую оценку разработанного приложения на основе эвристик Якоба Нильсена.

**Описание задания**

Необходимо в соответствии с выбранной тематикой разрабатываемого приложения реализовать функцию формирования отчетов в файлы word и excel. Например, для темы «Электронный деканат» в Word можно формировать «хвостовки», а в Excel - отчет, содержащий оценки студента. Дать возможность авторизованному пользователю из меню и из некоторого окна формировать отчеты.

**Теория и примеры реализаций**

1. Общие сведения об объектной модели Excel.

Для разработки решений, использующих MicrosoftOfficeExcel, необходимо взаимодействие с объектами, предоставляемыми объектной моделью Excel:

* Microsoft.Office.Interop.Excel.Application;
* Microsoft.Office.Interop.Excel.Workbook;
* Microsoft.Office.Interop.Excel.Worksheet;
* Microsoft.Office.Interop.Excel.Range.

Объектная модель Excel точно соответствует пользовательскому интерфейсу. Объект Application представляет приложение в целом, а каждый из объектов Workbook содержит коллекцию объектов Worksheet. Объект Range позволяет работать с отдельными ячейками или группой ячеек.

Для работы с этими объектами необходимо в раздел проекта References добавить сборку Microsoft.Office.Interop.Excel (см. рис. 4.28).

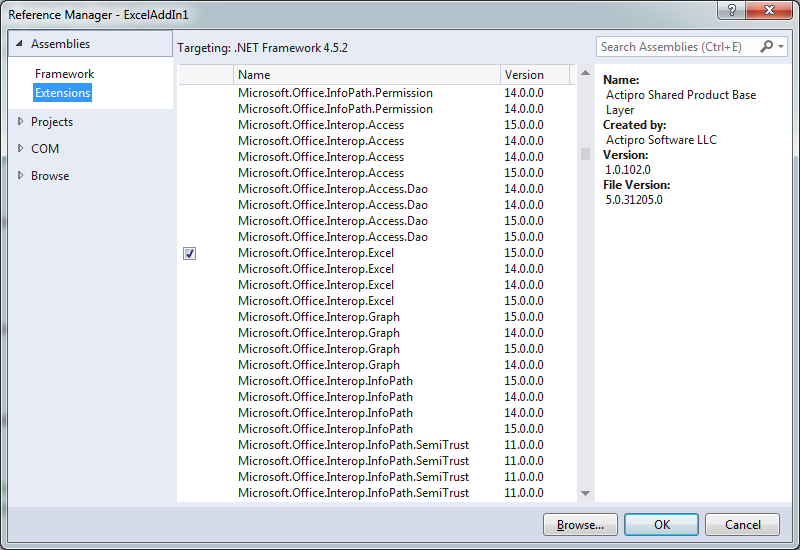


Рис.4.28. Добавление сборки Microsoft.Office.Interop.Excel в проект

Для запуска Excel необходимо выполнить следующие действия:

* подключить с псевдонимом пространство имен:

using Excel = Microsoft.Office.Interop.Excel;

* объявить переменную для работы с приложением:

private Excel.ApplicationexApp;

* загрузить в память новый экземпляр приложения:

exApp = new Excel.Application();

exApp.Visible = true;

Для работы с книгой Excel необходимо выполнить следующие действия:

* создать новую книгу, например, с одним листом:

exApp.SheetsInNewWorkbook = 1;

exApp.Workbooks.Add();

Свойство SheetsInNewWorkbook определяет количество листов, которые будут созданы в новой рабочей книге Excel.

Методу Add в качестве необязательного параметра можно передать имя шаблона рабочей книги.

* записать значение в текущую ячейку:

exApp.ActiveCell.Value = 1;

* объявить переменную для работы с листом:

private Excel.WorksheetexWrkSht;

* получить ссылку на лист 1:

exWrkSht = exApp.Workbooks[1].Worksheets.get\_Item(1);

* запись значения в конкретную ячейку:

exWrkSht.get\_Range("B1", "B1").Value = 1;

Метод Save() сохраняет книгу под именем, присваиваемым по умолчанию, в паке «Мои документы». Метод SaveAs () позволяет сохранить под указанным именем, в нужной папке, в нужном формате.

Для сохранения книги можно использовать следующий код:

exApp.Workbooks[1].Save();

exApp.Workbooks[1].SaveAs("c:\\Users\\1.xlsx");

Для открытия существующего файла есть метод Open():

exApp.Workbooks[1].Open("c:\\Users\\1.xlsx");

На рис. 4.29 показан пример кода для вывода информации в таблицу, вставка формулы «сумма по первому столбцу».

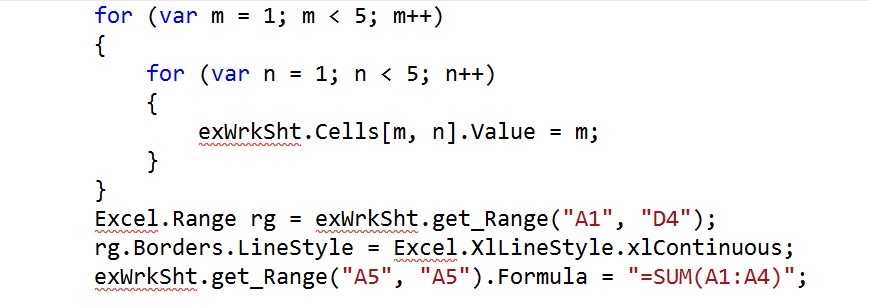
****

Рис. 4.29. Пример кода для работы с Excel

1. Общие сведения об объектной модели Word.

Для разработки решений, использующих MicrosoftOfficeWord, необходимо взаимодействие с объектами, предоставляемыми объектной моделью Word:

* Microsoft.Office.Interop.Word.Application
* Microsoft.Office.Interop.Word.Document,
* Microsoft.Office.Interop.Word.Selection,
* Microsoft.Office.Interop.Word.Bookmark,
* Microsoft.Office.Interop.Word.Range.

Объект Application представляет текущий экземпляр Word. Объект Application содержит объекты Document (документ), Selection (текущая выбранная область), Bookmark (закладка) и Range (диапазон текста). Каждый из этих объектов содержит множество методов и свойств, к которым можно обращаться для работы с объектом и взаимодействия с ним.

Иерархия этих объектов показана на рис.4.30.



Рис. 4.30. Объектная модель Word

Для работы с этими объектами необходимо в раздел проекта References добавить сборку Microsoft.Office.Interop.Word.

Для запуска Word необходимо выполнить следующие действия:

* подключить с псевдонимом пространство имен:

using Word = Microsoft.Office.Interop.Word;

* объявить переменную для работы с приложением:

private Word.ApplicationwdApp;

* загрузить в память новый экземпляр приложения:

wdApp = new Word.Application();

wdApp.Visible = true;

Для работы с документом Word необходимо выполнить следующие действия:

* создать новый документ:

wdApp.Documents.Add();

Методу Add в качестве необязательного параметра можно передать имя шаблона для создания нового документа.

* объявить переменную для работы с документом:

private Word.Documentdocum;

* получить ссылку на созданный документ можно по номеру или имени документа:

docum = wdApp.Documents.get\_Item(1);

* сохранение текущего документа.

Для сохранения документа можно использовать два метода Save() и SaveAs(). Метод Save() не имеет параметров, приводит к открытию стандартного диалога сохранения документа Word, который позволяет ввести название документа. В методе SaveAs() в качестве параметра необходимо указать имя файла, а также формат. Например:

docum.SaveAs("c:\\users\\doc2.docx");

Для открытия существующего документа используется метод Open().

Для вставки информации в документ можно использовать объект Selection, который всегда присутствует в документе.

wdApp.Selection.Text = "Hello Word";

Пример вставки таблицы в документ Word приведен на рис. 4.31.

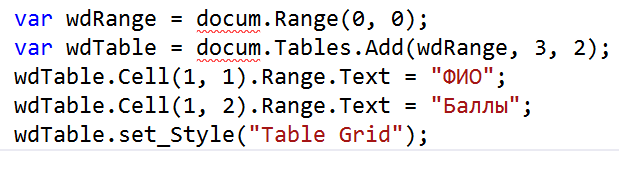


Рис. 4.31. Пример вставки таблицы в документ Word

Эвристическая оценка приложения позволяет выявить основные проблемы, с которыми скорее всего столкнётся пользователь. При этом интерфейс оценивают по некоторому набору критериев. В данной работе предлагается использовать эвристики Якоба Нильсена [10].Кратко рассмотрим их:

* во всех ли случаях пользователю понятно, что происходит в приложении;
* понятна ли пользователю терминология, которая используется в приложении (названия пунктов меню, формулировки сообщений и т. п.);
* может ли пользователь отменить свои действия;
* придерживались ли единообразия и стандартов при разработке ПИ;
* предотвращает ли ПИ ошибки пользователей;
* понятны ли пользователю его действия при работе с системой (подсказки, продуманность элементов ПИ);
* гибкость и эффективность реализации функций приложения для пользователей с разной подготовкой;
* придерживались ли эстетичного и минималистического дизайна;
* достаточное ли количество сообщений об ошибках и предупреждений, понятны ли их сообщения;
* наличие справочных материалов и документации по приложению.

В качестве экспертов предлагается выбрать двух студентов из вашей группы. Они должны изучить разработанный интерфейс при этом им не разрешается общаться и советоваться. Отчет эксперта должен содержать информацию о том, все ли критерии выполнены, и список выявленных ошибок и замечаний. Также эксперты могут добавить оценку каких-то критериев, не входящих в список эвристик. После этого, необходимо проанализировать полученные результаты на предмет выявления основных юзабилити-проблем, которые необходимо устранить. Анализ работы экспертов необходимо включить в отчет по лабораторной работе. Желательно привести рекомендации по исправлению самых существенных юзабилити-ошибок.

Для защиты лабораторной работы необходимо представить отчет, содержащий цель работы, ход выполнения работы и выводы.

## Применение WPF для разработки интерфейса пользователя

**Цель**

Изучить возможности WPF в плане разработки интерфейса пользователя.

**Задачи**

1. Ознакомится с особенностями компоновки WPF.
2. Разработать формы с элементами управления.

**Описание задания**

Разработать интерфейс формы приложения на базе WPF с использованием элементов ПИ: ListBox, ListView, TextBox, ComboBox, Menu.

**Теория и примеры реализации**

WPF (Windows Presentation Foundation) – API-интерфейс, основанный на DirectX. Платформа WPF включает знакомые стандартные элементы управления, но она рисует каждый текст, контур и фон самостоятельно. В результате WPF может предоставить намного более мощные средства, которые позволяют изменить визуализацию любой части экранного содержимого [11].

Для создания простейшего wpf-приложения необходимо выбрать соответствующий шаблон (см. рис.4.32).

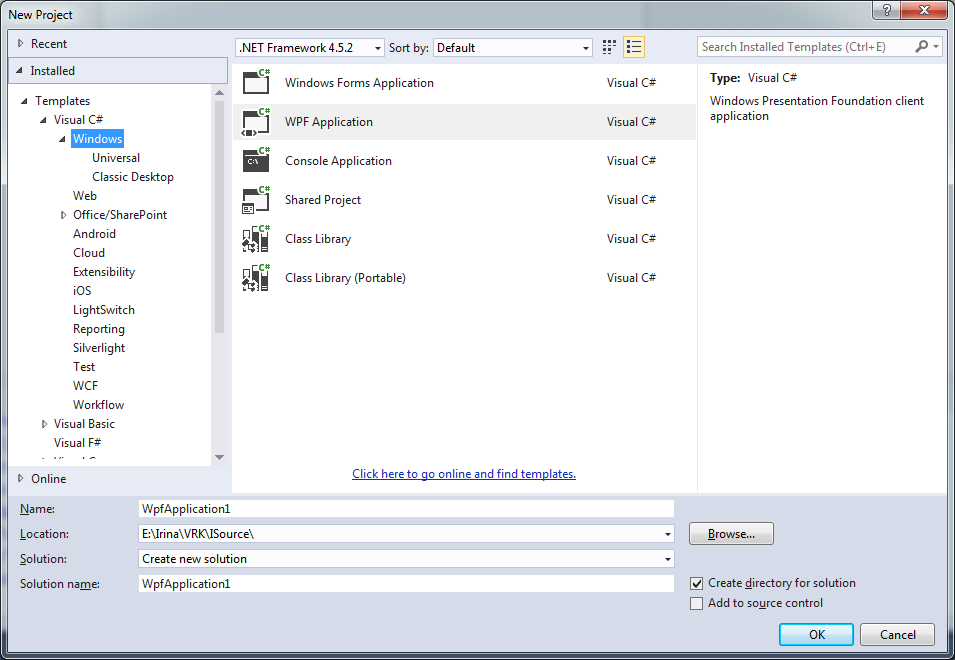


Рис.4.32. Выбор шаблона для WPF-приложения

В результате будет создано приложение, представляющее пустую форму. Рассмотрим более подробно получившееся приложение.

Файл App.xaml– этот файл XAML определяет приложение WPF и все его ресурсы. В этом файле также указывается начальный пользовательский интерфейс, который автоматически открывается при запуске приложения. В данном случае – это MainWindow.xaml, данный файл показан на рис. 4.33.

Файл MainWindow.xaml. – этот XAML-файл представляет главное окно приложения. Класс Window определяет свойства окна, такие как заголовок, размер и значок, и обрабатывает события, такие как открытие и закрытие окна.

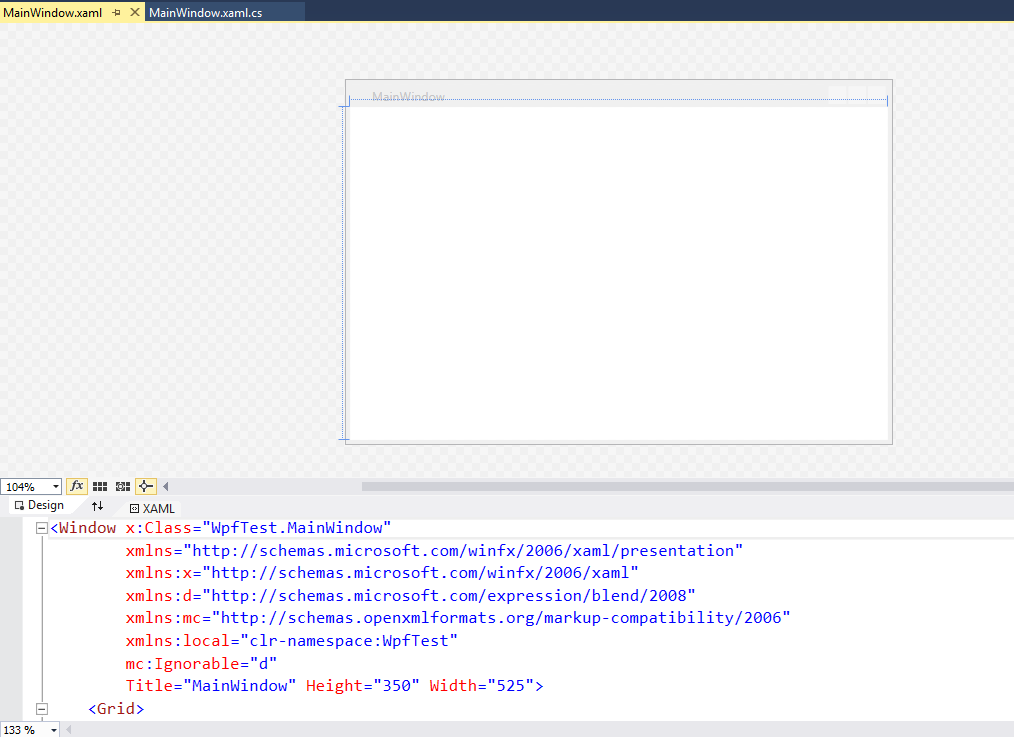


Рис. 4.33. Файл MainWindow.xaml

При построении пользовательских интерфейсов для WPF-приложений используется язык расширенной разметки приложений XAML (Extensible Application Markup Language). XAML-документ содержит разметку, описывающую внешний вид и поведение окна или страницы приложения, а связанные с ним файлы кода C# - логику приложения. Язык XAML обеспечивает разделение процесса дизайна приложения (графической части) и разработки бизнес-логики (программного кода) между дизайнерами и разработчиками. WPF XAML является подмножеством языка XML и позволяет описывать WPF-содержимое таких элементов как векторная графика, элементы управления и документы.

Каждый элемент XAML-документа отображается на некоторый экземпляр класса .NET. Имя такого элемента в точности соответствует имени класса. Например, элемент <Button> служит для WPF инструкцией для построения объекта класса Button;

Элементы XAML можно вкладывать друг в друга. Вложение элементов разметки обычно отображает вложенность элементов интерфейса;

Свойства класса определяются с помощью атрибутов или с помощью вложенных дескрипторов со специальным синтаксисом. Атрибуты элементов используются для задания свойств (Name, Height, Width и т.п.) и событий (Click, Load и т.д.) соответствующих объектов.

Для программного управления элементами управления, описанными в XAML-документе необходимо для элемента управления задать XAML атрибут Name. Так для задания имени элементу Button необходимо записать следующую разметку:

<ButtonName="grid">

</Button>

Простые свойства задаются в XAML-документе в соответствии со следующим синтаксисом:

ИмяСвойства ="значение"

Например, Name = "Button1"

При необходимости задать свойство, которое является полноценным объектом, используются сложные свойства в соответствии с синтаксисом "свойство-элемент":

Родитель.ИмяСвойства

**Компоновка**

При объявлении элемента управления непосредственно внутри окна, не имеющего панелей, он размещается в центре окна. Для организации содержимого окна в WPF используются разнообразные контейнеры. Все контейнеры компоновки WPF являются панелями, которые унаследованы от абстрактного класса System.Windows.Controls.Panel. Пространство имен System.Windows.Controls предлагает многочисленные панели. Перечислим основные:

* StackPanel–размещает элементы в горизонтальном или вертикальном стеке. Этот контейнер компоновки обычно используется в небольших разделах крупного и более сложного окна;
* WrapPanel–размещает элементы в последовательностях строк с переносом. В горизонтальной ориентации WrapPanel располагает элементы в строке слева направо, затем переходит к следующей строке. В вертикальной ориентации WrapPanel располагает элементы сверху вниз, используя дополнительные колонки для дополнения оставшихся элементов;
* DockPanel–выравнивает элементы по краю контейнера;
* Grid–выстраивает элементы в строки и колонки невидимой таблицы. Это один из наиболее гибких и широко используемых контейнеров компоновки;
* UniformGrid–помещает элементы в невидимую таблицу, устанавливая одинаковый размер для всех ячеек. Данный контейнер компоновки используется нечасто;
* Canvas–позволяет элементам позиционироваться абсолютно – по фиксированным координатам. Этот контейнер компоновки более всего похож на традиционный компоновщик WindowsForms, но не предусматривает средств привязки и стыковки. В результате это неподходящий выбор для окон переменного размера.

Компоновка определяется контейнером, но дочерние элементы тоже могут влиять на расположение на форме. Панели компоновки взаимодействуют со своими дочерними элементами через набор свойств компоновки, перечисленных ниже:

* HorizontalAlignment–определяет позиционирование дочернего элемента внутри контейнера компоновки, когда имеется дополнительное пространство по горизонтали. Доступные значения: Center, Left, Right или Stretch;
* VerticalAlignment–определяет позиционирование дочернего элемента внутри контейнера компоновки, когда имеется дополнительное пространство по вертикали. Доступные значения: Center, Top, Bottom или Stretch;
* Margin–добавляет некоторое пространство вокруг элемента. Свойство Margin– это экземпляр структуры System.Windows.Thickness, с отдельными компонентами для верхней, нижней, левой и правой граней;
* MinWidth, MinHeight–устанавливают минимальные размеры элемента. Если элемент слишком велик, чтобы поместиться в его контейнер компоновки, он будет усечен;
* MaxWidth, MaxHeight–устанавливают максимальные размеры элемента. Если контейнер имеет свободное пространство, элемент не будет увеличен сверх указанных пределов, даже если свойства HonzontalAlignment и VerticalAlignment установлены в Stretch;
* Width, Height–явно устанавливают размеры элемента. Эта установка переопределяет значение Stretch для свойств HorizontalAlignment и VerticalAlignment. Однако данный размер не будет установлен, если выходит за пределы, заданные в MinWidth, MinHeight, MaxWidth и MaxHeight.

Все эти свойства унаследованы от базового класса FrameworkElement и потому поддерживаются всеми графическими элементами, которые можно использовать в окне WPF.

**Элементы управления**

Элементы управления содержимым (Label, Button, CheckBox и RadioButton.) являются специализированным типом элементов управления, которые могут хранить некоторое содержимое – один или несколько элементов. Все элементы управления содержимым являются наследниками класса ContentControl.

Класс ContentControl наследуется от класса System.Windows.Control, который наделяет его и все дочерние классы базовыми характеристиками, которые:

* позволяют определять содержимое внутри элемента управления;
* позволяют определять порядок перехода с использованием клавиш табуляции;
* поддерживают рисование фона, переднего плана и рамки;
* поддерживают форматирование размера и шрифта текстового содержания.

Метка – Label является простейшим элементом управления содержимым. Метка поддерживает мнемонические команды – клавишу быстрого доступа, передавая фокус связанному с ней элементу управления. Для поддержки этой функции используют свойство Target, которому присваивают выражение привязки. В выражении привязки необходимо указать другой элемент управления, на который будет переходить фокус при нажатии клавиши быстрого доступа.

<Label Target="{Binding ElementName = txtA}">Выбор \_А</Label>

<TextBox Name="txtA">Выбортекста</TextBox>

Символ подчеркивания в тексте метки указывает на клавишу быстрого доступа. Все мнемонические команды работают при одновременном нажатии клавиши <Alt> и заданной клавиши быстрого доступа. В приведенном выше коде при нажатии комбинации <Alt+A> фокус перейдет на элемент управления TextBox с именем txtA.

Для классов Button, CheckBox и RadioButton определено событие Click.

Класс Button добавляет два свойства, доступные для записи: IsCancelи IsDefault.

При IsCancel *=* true кнопка будет работать как кнопка отмены окна и если нажать кнопку <Esc>, когда текущее окно находится в фокусе, то кнопка сработает.

Если IsDefault *=* true, то кнопка считается кнопкой по умолчанию.

Классы CheckBox и RadioButton имеют два состояния: нажата и отпущена. В классе имеются события Checked, Unchecked и Intermediate, которые генерируются при включении, выключении или переходе кнопки в неопределенное состояние.

Для кнопки CheckBox включение элемента управления означает отметку в нем флажка. Свойство IsChecked, наследуемое от класса ToggleButton, может принимать три значения: true (включено), false (выключено), null (неопределено, которое отображается в виде затененного окна, и используется для промежуточного состояния). Пример XAML-описания трех кнопок CheckBox:

<CheckBoxHeight="16" Name="checkBox1"

Width="120" IsChecked="False"

ClickMode="Release">Выбор А</CheckBox>

<CheckBox Height="16" Name="checkBox2"

Width="120" IsChecked="True"

ClickMode="Press">Выбор Б</CheckBox>

<CheckBox Height="16" Name="checkBox3"

Width="120" IsChecked="{x:Null}"

ClickMode="Hover">Выбор В</CheckBox>

Для кнопки *RadioButton* добавлено свойство *GroupName*, которое позволяет управлять расположением переключателей в группе. Из группы можно выбрать только один переключатель.

<GroupBoxHeader="Группа радиокнопок" Height="100"

Name="groupBox1" Width="200">

<StackPanel>

<RadioButton Height="16" Name="radioButton2"

Width="120">Выбор Г</RadioButton>

<RadioButton Height="16" Name="radioButton1"

Width="120">Выбор Д</RadioButton>

<RadioButton Height="16" Name="radioButton3"

Width="120">Выбор Е</RadioButton>

</StackPanel>

</GroupBox>

Для вывода контекстного окна для элемента управления существует свойство ToolTip, например для кнопки:

<ButtonToolTip="Подсказка для кнопки А"></Button>

Элемент управления TextBox, как правило хранит одну строку текста:

<TextBoxName="txtA"> Выбор текста</TextBox>

Если необходимо создать многострочное представление, то свойству TextWrapping необходимо присвоить значение Wrap. Для многострочного элемента TextBox можно задать минимальное и максимальное количество строк, используя свойства MinLines и MaxLines.

Элементы управления ListBox и ComboBox являются элементами управления списками.

Для добавления элементов в ListBox можно вложить элементы ListBoxItem в элемент управления ListBox, как это показано для составления списка цветов (зеленый, голубой, желтый, красный):

<ListBox>

<ListBoxItem>Зеленый</ListBoxItem>

<ListBoxItem>Голубой</ListBoxItem>

<ListBoxItem>Желтый</ListBoxItem>

<ListBoxItem>Красный</ListBoxItem>

</ListBox>

Элемент управления ListBox хранит каждый вложенный объект в своей коллекции. При этом ListBoxItem может хранить не только строки, но и любой произвольный элемент.

Элемент управления ComboBox подобен ListBox.

**Стили**

Стили – это коллекция значений свойств, которые могут быть применены к элементу. Они позволяют определить общий набор характеристик форматирования и применять их по всему приложению для обеспечения согласованности. Стили в WPF могут устанавливать любое свойство зависимостей. Их можно применять для стандартизации визуального поведения каких-либо элементов. Стили WPF поддерживают триггеры, которые позволяют изменять стиль элемента при изменении других свойств. Стили позволяют использовать шаблоны для переопределения стандартного визуального представления элементов управления.

Стиль создается на базе класса Style со следующими свойствами:

* BasedOn – возвращает или задает определенный стиль, являющийся основой текущего стиля;
* Dispatcher – возвращает объект Dispatcher, с которым связан этот объект DispatcherObject;
* IsSealed – Возвращает значение, указывающее, доступен ли стиль только для чтения;
* Resources – Возвращает или задает коллекцию ресурсов, которые могут использоваться в области видимости данного стиля;
* Setters – возвращает коллекцию объектов Setter и EventSetter;
* TargetType – возвращает или задает тип, для которого предназначен данный стиль;
* Triggers – возвращает коллекцию объектов TriggerBase, применяющих значения свойств на основе заданных условий.

Например, в приложении необходимо многократно использовать кнопки, которые имеют светло-голубой фон и голубую рамку.

<Style x:Key="ButtonStyle" TargetType="Button">

<Setter Property="Background" Value="LightBlue"/>

<Setter Property="BorderBrush" Value="Blue" />

</Style>

Имя стилю задается атрибутом x:Key (ButtonStyle). Атрибут TargetType определяет тип, для которого будет применяться стиль (Button). Каждый объект Setter устанавливает в элементе одно свойство, которое обязательно должно быть свойством зависимостей. Установка свойств производится с помощью атрибутов Property, который определяет имя свойства, и Value, которое задает значение свойства.

В стиль можно добавить объект EventSetters, в котором привязывается события к определенному обработчику.

Триггеры позволяют вносить изменения в стиль при выполнении определенных условий. Добавим в стиль для кнопки ButtonStyle триггер, который будет формировать красный фон при наведении указателя мыши на кнопку.

<Style x:Key="ButtonStyle" TargetType="Button">

<Setter Property="Background" Value="LightBlue"/>

<Setter Property="BorderBrush" Value="Blue" />

<Style.Triggers>

<Trigger Property="IsMouseOver" Value="true">

<Setter Property="Background" Value="Red" />

</Trigger>

</Style.Triggers>

</Style>

В триггере должно быть указано идентифицирующее свойство (в нашем примере IsMouseOver), за которым должно вестись наблюдение, и значение, которого следует ожидать (в нашем примере true). Когда появляется необходимое значение, устанавливается свойство, которое определено объектом Setter. В приведенном примере при IsMouseOver = true, то есть наведении указателя мыши на кнопку, свойству Background присваивается значение Red, то есть фон кнопки становится красным. Когда указатель мыши покидает кнопку, условия срабатывания триггера нарушаются и фон кнопки переходит в прежнее состояние.

Кроме триггера, ожидающего изменение свойства, имеются триггеры события (EventTrigger), которые ожидают возникновения определенного события.

Пример создания главного меню:

<Menu>

<MenuItem Header="Файл" >

<MenuItem Header="Создать" ></MenuItem>

<MenuItem Header="Редактировать" ></MenuItem>

<MenuItem Header="Сохранить" ></MenuItem>

<MenuItem Header="Найти" />

<Separator></Separator>

<MenuItem Header="Удалить" ></MenuItem>

</MenuItem>

<MenuItem Header="Выход"></MenuItem>

</Menu>

Для защиты лабораторной работы необходимо представить отчет, содержащий цель работы, ход выполнения работы и выводы.

## Разработка WPF-приложения для работы с данными

**Цель**

Научиться разрабатывать WPF-приложения, взаимодействующие с данными.

**Задачи**

1. Ознакомится со способом представления и взаимодействия с данными - привязкой данных WPF
2. Изучить интерфейс INotifyPropertyChanged и конвертеры значений.
3. Разработать WPF-приложения, взаимодействующие с данными

**Описание задания**

Разработать приложение в технологии WPF, которое считывает информацию об объекте (например, студент) из структурированного файла (например, xml) и отображает его на форме с использованием не менее трех типов элементов управления. Изменения, внесенные пользователем, с помощью элементов управления, сохраняются обратно в файл.

**Теория и примеры реализаций**

Привязка данных представляет собой взаимодействие двух объектов: источника и приемника. Объект-приемник создает привязку к определенному свойству объекта-источника. При изменении объекта-источника, объект-приемник также будет изменяться [11].

Для определения привязки в xaml-файле используется следующее выражение:

{**Binding** **ElementName**=<Имя объекта-источника>, **Path** = <Свойство объекта-источника>}

Ниже приведен пример привязки. Компонент **TextBlox** с названием txt1TextBox , будет источником, а компонент **TextBlox** с названием txt2TextBox - приемником:

<TextBox x:Name="txt1TextBox">

<TextBox x:Name=" txt2TextBox " Text="{Binding ElementName= txt2TextBox, Path=Text}"/>

Привязка может быть установлена также в коде C#:

public MainWindow()

{

InitializeComponent();

Binding binding = new Binding();

binding.ElementName = "txt1TextBox"; // элемент-источник

binding.Path = new PropertyPath("Text"); // свойство элемента-источника

txt2TextBox.SetBinding(TextBox.TextProperty, binding); // установка привязки для элемента-приемника

}

Для удаления привязки можно воспользоваться классом **BindingOperations** и его методами:

* ClearAllBindings(DependencyObject) – удаляет все привязки для указанного объекта DependencyObject;
* ClearBinding(DependencyObject, DependencyProperty) – удаляет привязку из свойства DependencyProperty объекта DependencyObjec, если она существует;

Примеры.

BindingOperations.ClearBinding(txt2TextBox, TextBlox.TextProperty);

или

BindingOperations.ClearAllBindings(txt2TextBox);

Ниже приводятся некоторые свойства класса **Binding**:

* ElementName – определяет имя элемента, который будет использоваться в качестве источника привязки;
* IsAsync – определяет значение, которое указывает на использование асинхронного режима получения данных из объекта. По умолчанию равно False;
* Mode – определяет режим привязки;
* Path – определяет свойство объекта, к которому идет привязка;
* TargetNullValue – определяет значение по умолчанию, которое используется в случае, если значение привязанного свойства источника привязки равно null;
* RelativeSource – определяет источник привязки относительно положения текущего объекта;
* Source – определяет объект-источник привязки, если он не является элементом управления;
* XPath – определяет путь к xml-данным или XPath запрос, используется вместо свойства Path.

Ниже эти свойства рассматриваются подробнее.

Режимы привязки определяют возможность изменения данных в источнике и приемник. Для этого используется свойство **Mode** класса **Binding**. Оно может принимать одно из значений **BindingMode** (перечислимый тип):

* Default – по умолчанию (если меняется свойство TextBox.Text, то имеет значение TwoWay, в остальных случаях OneWay);
* OneWay – свойство объекта-приемника изменяется в случае изменения свойства объекта-источника, целесообразно использовать в случае, если объект-приемник доступен только для чтения;
* OneTime – свойство объекта-приемника устанавливается по свойству объекта-источника только один раз, например, при запуске приложения. Используется, когда источник является статичным, то есть не изменяется при работе приложения;
* TwoWay – оба объекта - приемник и источник могут изменять привязанные свойства друг друга, используется для просмотра и редактирования данных на форме;
* OneWayToSource – объект-приемник изменяет соответствующее свойство объекта-источника;

При использовании односторонней привязки изменения источника сразу же отображаются в соответствующем свойстве приемника. При использовании двусторонней привязки при изменении приемника свойство источника не изменяется мгновенно. Свойство **UpdateSourceTrigger** класса **Binding** определяет, как будет происходить обновление. Это свойство может принимать одно из следующих перечисленных ниже значений:

* Default – значение по умолчанию. Для большинства свойств разных компонент это значение PropertyChanged. Для свойства Text элемента TextBox это значение LostFocus.
* Explicit – источник обновляется только при вызове метода BindingExpression.UpdateSource();
* LostFocus – источник привязки обновляется каждый раз при потере фокуса приемником;
* PropertyChanged – источник привязки обновляется сразу же при изменении свойства в приемнике.

Если необходимо установить привязку к объекту, который не является элементом управления WPF, то для этого используется свойство **Source** класса **Binding**.

Свойство **TargetNullValue** класса **Binding** позволяет задать некоторое значение по умолчанию, если свойство в источнике привязки имеет значение null, то есть оно не определено

Свойство **RelativeSource** класса **Binding** используется для установки привязки относительно элемента-источника, который связан какими-нибудь отношениями с элементом-приемником. Например, элемент-источник может быть одним из внешних контейнеров для элемента-приемника или источник и приемник представляет собой один и тот же элемент.

Для установки этого свойства используется объект **RelativeSource**, который имеет свойство **Mode**. Это свойство задает способ привязки. Оно может принимать одно из значений перечисления **RelativeSourceMode**:

* Self – привязка осуществляется к свойству этого же элемента, то есть источник привязки также является и приемником привязки.
* FindAncestor– привязка осуществляется к свойству элемента-предка в последовательности родительских элементов элемента с привязкой.

Выше обсуждалось свойство **Source** для указания источника привязки. С помощью свойства **DataContext** можно установить область видимости, внутри которой у всех свойств с привязкой будет один источник привязки. Это свойство есть у всех элементов, которые наследуются от объекта **FrameworkElement**. Следует отметить, что источник, указанный через свойство **Source**, имеет приоритет над наследуемым контекстом данных.

Обсуждаемые выше режимы привязки не помогают решить проблему отображения изменений в объекте, который через **Binding** привязан к свойству визуального компонента. Для этого необходимо реализовать в его классе интерфейс **INotifyPropertyChanged**. Ниже приведен пример описания класса.

class Client: INotifyPropertyChanged

{

private string fullName;

private string shortName;

public event PropertyChangedEventHandler PropertyChanged;

public void RaisePropertyChanged(string propertyName)

{

if (PropertyChanged != null)

PropertyChanged(this, new

PropertyChangedEventArgs(propertyName));

}

public string FullName

{

get {returnfullName;}

set

{

fullName = value;

RaisePropertyChanged("FullName");

}

}

public string ShortName

{

get{returnshortName;}

set

{

shortName = value;

RaisePropertyChanged("ShortName");

}

}

}

Ниже показан пример кода для отображения свойств класса на форме.

<Label Content="Полное наименование" />

<TextBox x:Name="login" Grid.Column="1" Margin="0,5,0,5" MaxWidth="250" MinWidth="150" Text="{Binding Path=FullName, Mode=TwoWay}"/>

<Label Grid.Row="1" Grid.Column="0" Content="Краткое наименование" />

<TextBox x:Name="paswd" Grid.Row="1" Grid.Column="1" Margin="0,5,0,5" Padding="5" MaxWidth="250" MinWidth="150" Text="{Binding Path=ShortName, Mode=TwoWay}"/>

В некоторых случаях недостаточно просто использовать привязку, необходимо преобразовывать данные. Например, свойство класса может храниться в базе данных в виде кода, а на экране требуется показать понятное для пользователя значение. Либо источник и приемник привязки имеют соответствующие свойства несовместимые по типу. В этом случает необходимо использовать форматирование значений. В зависимости от типа визуального компонента для форматирования значений используются различные свойства:

* StringFormat – используется для класса Binding;
* ContentStringFormat – используется для классов ContentControl, ContentPresenter, TabControl;
* ItemStringFormat – используется для класса ItemsControl;
* HeaderStringFormat – используется для класса HeaderContentControl;
* ColumnHeaderStringFormat – используется для классов GridView, GridViewHeaderRowPresenter;
* SelectionBoxItemStringFormat–используется для классов ComboBox, RibbonComboBox.

Для того чтобы форматирование значений заработало, необходимо реализовать в конвертере значений интерфейс **System.Windows.Data.IValueConverter**. Этот интерфейс определяет два метода: **Convert()**, который преобразует пришедшее от привязки значение в тип, который понимается приемником привязки, и **ConvertBack()**, который выполняет обратное преобразование.

Эти методы имеют четыре параметра:

* object value – значение, которое надо преобразовать;
* Type targetType – тип, к которому надо преобразовать значение value;
* object parameter – вспомогательный параметр;
* CultureInfo culture – текущая культура приложения;

**Пример.**

Добавим в класс Client: еще одно свойство, значение которого будем отображать с помощью ComboBox.

class Client: INotifyPropertyChanged

{

private string fullName;

private string shortName;

private string indOrjur;

public string IndOrJur

{

get { return indOrjur; }

set

{

indOrjur = value;

RaisePropertyChanged("IndOrJur");

}

}

}

Определим конвертер:

public class IndOrJurConvert: IValueConverter

{

public object Convert(object value, Type targetType, object parameter,

CultureInfo culture)

{

if (value.ToString() == "Ф")

return 1;

else

return 0;

}

public object ConvertBack(object value, Type targetType,

object parameter, CultureInfo culture)

{

if (value.ToString() == "0")

return "Ю";

else return "Ф";

}

}

Укажем его в ресурсах окна:

<Window.Resources>

<local:IndOrJurConvert x:Key="IndOrJurConverter" />

</Window.Resources>

Определим ComboBox:

<ComboBox x:Name="comboBox" HorizontalAlignment="Left" Margin="9,0,0,0"

Grid.Row="2" VerticalAlignment="Top" Width="120"

SelectedIndex="{Binding Path=IndOrJur,

Converter={StaticResource IndOrJurConverter},

Mode=TwoWay}"/>

<TextBlock>юр. лицо</TextBlock>

<TextBlock>физ. лицо</TextBlock>

</ComboBox>

**Пример.**

Отобразим свойство IndOrJur с помощью RadioButtons.

<RadioButton x:Name="radioButton" GroupName="typClient" Content="юр. лицо" Grid.Column="1" HorizontalAlignment="Left"

Margin="13,3,0,0" Grid.Row="2" VerticalAlignment="Top"

IsChecked="{Binding Path=IndOrJur,

Converter={StaticResource IndOrJurConverterBool} ,ConverterParameter=0,

Mode=TwoWay}"/>

<RadioButton x:Name="radioButton1" GroupName="typClient" Content="физ. лицо" Grid.Column="1" HorizontalAlignment="Left"

Margin="117,3,0,0" Grid.Row="2" VerticalAlignment="Top"

IsChecked="{Binding Path=IndOrJur,

Converter={StaticResource IndOrJurConverterBool}, ConverterParameter=1,

Mode=TwoWay}"/>

Конвертер будет выглядеть следующим образом

public class IndOrJurConvertBool : IValueConverter

{

public object Convert(object value, Type targetType, object parameter,

CultureInfo culture)

{

if (parameter.ToString() == "1")

return (value.ToString() == "Ф");

else

return !(value.ToString() == "Ф");

}

public object ConvertBack(object value, Type targetType, object parameter,

CultureInfo culture)

{

if (parameter.ToString() == "1")

return ((bool)value? "Ф": "Ю");

else

return ((bool)value ? "Ю": "Ф");

}

}

Для защиты лабораторной работы необходимо представить отчет, содержащий цель работы, ход выполнения работы и выводы.

## Оценка эффективности пользовательского интерфейса по критерию скорости на основе модели GOMS

**Цель**

Изучить возможности оценки эффективности пользовательского интерфейса по критерию скорости на основе модели GOMS.

**Задачи**

1. Ознакомится с моделью GOMS.
2. Оценить скорость работы с разработанной ранее интерфейсной формой на основе модели GOMS.

**Описание задания**

Сравнить по модели GOMS эффективность интерфейса одной из разработанных форм системы при работе с помощью мыши и клавиатуры.

**Теория и примеры реализации**

GOMS это сокращение от английского Goals, Operators, Methods, and Selection Rules – Цели, Операторы, Методы и Правила выбора. Данный способ был предложен Кардом, Мораном и Ньюэллом в 1983 году. Моделирование GOMS позволяет оценить время выполнения конкретной операции при использовании данной модели ПИ [6].

При работе с моделью GOMS предполагается, что время, необходимое пользователю для выполнения определенной задачи с помощью ПИ, является суммой всех временных интервалов, за которые выполняются элементарные жесты, составляющие данную задачу. Очевидно, что для разных пользователей время выполнения каждого жеста может отличаться, но для сравнительного анализа различных интерфейсов по скорости их использования можно использовать средние значения, которые получены исследователями (см. табл. 4.6).

Таблица 4.6

Средние временные интервалы, требуемые для выполнения элементарных жестов

|  |  |  |
| --- | --- | --- |
| Операция, интервал | Название | Описание |
| *K* = 0.2 с | Нажатие клавиши | Время, необходимое для нажатия клавиши. |
| B = 0.1 с | Нажатие кнопки | Время, необходимое для нажатия на кнопку мыши |
| *P* = 1.1 с | Указание | Время, необходимое для указания на какую-то позицию на экране монитора. |
| *H* = 0.4 с | Перемещение | Время, необходимое для перемещения руки с клавиатуры на мышь или с мыши на клавиатуру. |
| *M* = 1.35 с | Ментальная подготовка | Время, необходимое для умственной подготовки к следующему шагу, т.е. время принятие решения о следующем действии. |
| *R* – 0.1 с | Ответ | Время ожидания ответа компьютера. Для базовых операций, таких как работа с меню, это время можно не учитывать, т.е. считать равным 0. |

Для вычисления времени, затрачиваемого на выполнение операции, с помощью модели GOMSсначала необходимо перечислить и записать последовательность производимых пользователем элементарных действий из списка жестов модели GOMS (см. табл. 4.6). Затем в эту последовательность необходимо добавить ментальные операции, т.е. умственные операции по подготовке к следующему действию. На следующем этапе следует исключить ментальные операции там, где последовательные действия не требуют времени на размышление. В таб. 4.7 предлагаются следующие правила расстановки.

Таблица 4.7

Правила расстановки ментальных операций

|  |  |
| --- | --- |
| Правило 0  Начальная расстановка операторов *M* | Операторы Mнеобходимо устанавливать:   * перед всеми операторами *K*; * перед всеми операторами *P*, предназначенными для выбора команд.   Перед операторами *P*, предназначенными для указания на аргументы выбранных команд, ставить оператор *M* не надо. |
| Правило 1  Удаление ожидаемых операторов *M* | Если оператор, следующий за оператором*M* является полностью ожидаемым с точки зрения предыдущего оператора, то этот оператор *M*может быть удален.  Например, если курсор мыши перемещается к нужному объекту с целью его выбора, т.е. пользователь нажмет на кнопку мыши, когда достигнет объект. Согласно правилу 0 эта последовательность разделяется ментальной операцией, затем, согласно данному правилу, эта ментальная операция убирается. |
| Правило 2  Удаление операторов M внутри когнитивных единиц | Когнитивной единицей является непрерывная последовательность вводимых символов, которые образовывают название команды или аргумент (например, имена файлов при сохранении, значения каких-то параметров для выполнения команды).  Если строка вида *MKMKMK* ...принадлежит когнитивной единице, то необходимо удалить все операторы *M*, кроме первого. |
| Правило 3  Удаление операторов *M* перед последовательными разделителями | Разделитель - это символ, которым обозначено начало или конец значимого фрагмента текста. Например, пробелы являются разделителями для большинства слов.  Если для выполнения команды требуется дополнительная информация (например, в случае, когда для перевода значения из метров в километры пользователю требуется указать это значение), эта информация называется аргументом данной команды.  Если оператор *K* означает лишний разделитель, стоящий в конце когнитивной единицы, то следует удалить находящийся перед ним оператор *M*. |
| Правило 4  Удаление операторов *M*, которые являются прерывателями команд | Постоянная строка – это последовательность символов, которая каждый раз вводится одинаково.  Если оператор *K* является разделителем, стоящим после постоянной строки, то стоящий перед ним оператор *M* удаляется.  Если оператор *K* является разделителем для строки аргументов или любой другой изменяемой строки, то оператор *M* перед ним сохраняется. |
| Правило 5  Удаление перекрывающих операторов *M* | Не следует учитывать любую часть оператора *M*, перекрывающую оператор *R*, который является задержкой из-за ожидания ответа системы. |

В конце, согласно полученной схеме подсчитывается суммарное время выполнения пользователем операции.

Пример использования модели GOMS.

На рис. 4.34 представлена интерфейсная форма для входа в систему.

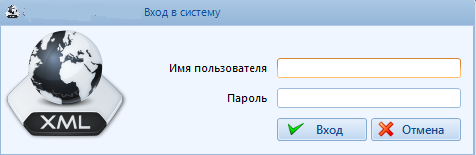


Рис. 4.34. Интерфейсная форма для входа в систему

Оценим эффективность работы с ней при использовании мыши для перемещения между элементами ПИ и кнопки Tabклавиатуры.

Действия пользователя в первом случае показаны в табл. 4.8.

Таблица 4.8

Действия пользователя (использовании мыши)

|  |  |
| --- | --- |
| **Очередное действие пользователя** | **Формируемая последовательность операций** |
| Ввод имени пользователя (8 символов) | *KKKKKKKK* |
| Перемещение руки на мышь | *KKKKKKKKH* |
| Указание на поле ввода пароля | *KKKKKKKKHP* |
| Ввод пароля (8 символов) | *KKKKKKKKHPKKKKKKKK* |
| Перемещение руки на мышь | *KKKKKKKKHPKKKKKKKKH* |
| Указание на кнопку Вход | *KKKKKKKKHPKKKKKKKKHP* |
| Нажатие кнопки Вход | *KKKKKKKKHPKKKKKKKKHP*B |

Затем расставляем ментальные операции. Их расстановка представлена в табл. 4.9.

Таблица 4.9

Расстановка ментальных операций в первом случае

|  |  |
| --- | --- |
| **Описание** | **Формируемая последовательность операций** |
| Начальная расстановка операторов *M* согласно правилу 0 (табл. 1). Имя пользователя и пароль – это аргументы команды. Внутри когнитивных единиц операторы *М* сразу не ставим согласно правилу 2. | *MKKKKKKKKHPMKKKKKKKKHP*B |
| Удаление ожидаемых операторов M по правилу 1 (табл. 1). | *MKKKKKKKKHPKKKKKKKKHP*B |
| Замена символов операторов соответствующими интервалами согласно табл. 2 и подсчет общего времени работы. | 1.35+ 0.2\*8 + 0.4 + 1.1 + 0.2\*8 + 0.4 + + 1.1 + 0.1 =7.65 с |

Действия пользователя во втором случае показаны в табл. 4.10.

Таблица 4.10

Действия пользователя (использовании только клавиатуры)

|  |  |
| --- | --- |
| **Очередное действие пользователя** | **Формируемая последовательность операций** |
| Ввод имени пользователя (8 символов) | *KKKKKKKK* |
| Нажатие на клавишу Tab (переход к следующему полю) | *KKKKKKKKK* |
| Ввод пароля (8 символов) | *KKKKKKKKKKKKKKKKK* |
| Нажатие на клавишу Tab (переход к кнопке Вход) | *KKKKKKKKKKKKKKKKKK* |
| Нажатие кнопки Вход | *KKKKKKKKKKKKKKKKKK*B |

Затем расставляем ментальные операции. Их расстановка представлена в табл. 4.11.

Таблица 4.11

Расстановка ментальных операций во втором случае

|  |  |
| --- | --- |
| **Описание** | **Формируемая последовательность операций** |
| Начальная расстановка операторов *M* согласно правилу 0 (табл. 1). Имя пользователя и пароль – это аргументы команды. Внутри когнитивных единиц операторы *М* сразу не ставим согласно правилу 2. | *MKKKKKKKKMKMKKKKKKKKMK*B |
| Удаление ожидаемых операторов M по правилу 1 (табл. 1). | *MKKKKKKKKKMKKKKKKKKK*B |
| Замена символов операторов соответствующими интервалами согласно табл. 2 и подсчет общего времени работы. | 1.35+ 0.2\*9 + 1.35 + 0.2\*9 + 0.1 = =6.4 с |

Анализ двух вариантов работы показывает, что работа с простой для ввода интерфейсной формой только с клавиатуры является более эффективной. Это следует учитывать при разработке ПИ.

Для защиты лабораторной работы необходимо представить отчет, содержащий цель работы, ход выполнения работы и выводы.