Документ подписан простой электронной подписью
Информация о владельце:
ФИО: Гарбар Олег Викторович
Должность: Заместитель директора по учебно-воспитательной работе
Дата подписания: 29.10.2021 14:00:23
Уникальный программный ключ:
5769a34aba1fca5ccbf44edc23bf8f452c6d4f04

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Индустриальный институт (филиал)
федерального государственного бюджетного образовательного учреждения
высшего образования «Югорский государственный университет»
(ИндИ (филиал) ФГБОУ ВО «ЮГУ»)

УТВЕРЖДАЮ
Заместитель директора по УВР
_____ Гарбар О.В.
«09» сентября 2021 г.

Методические указания
по выполнению практических работ
ПМ.01. РАЗРАБОТКА МОДУЛЕЙ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ КОМПЬЮТЕРНЫХ СИСТЕМ
МДК 01.01. РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

для специальности 09.02.07 Информационные системы и программирование

Нефтеюганск
2021

РАССМОТРЕНО: Предметной цикловой Комиссией специальных технических дисциплин Протокол №1 от 09.09.2021 _____Шарипова И.А.	СОГЛАСОВАНО: заседанием Методсовета протокол №1 от 16.09.2021 Председатель Методсовета _____Н.И. Савватеева

Методические указания по выполнению практических работ по МДК 01.01. Разработка программных модулей разработаны в соответствии с рабочей программой ПМ.01. Разработка модулей программного обеспечения для компьютерных систем по специальности 09.02.07 Информационные системы и программирование.

Разработчик: Чупракова И.В.., преподаватель ИндИ (филиала) ФГБОУ ВО «ЮГУ».

Пояснительная записка

Методические указания по выполнению лабораторных и практических работ по МДК 01.01. Разработка программных модулей разработаны в соответствии с рабочей программой профессионального модуля и предназначены для приобретения необходимых практических навыков и закрепления теоретических знаний, полученных обучающимися при изучении профессионального модуля, обобщения и систематизации знаний перед экзаменом.

Методические указания предназначены для обучающихся специальности 09.02.07 Информационные системы и программирование.

Освоение содержания МДК 01.01. Разработка программных модулей во время выполнения практических работ обеспечивает достижение обучающимися следующих **результатов:**

Код	Наименование общих компетенций
ОК 1.	Выбирать способы решения задач профессиональной деятельности, применительно к различным контекстам
ОК 2.	Осуществлять поиск, анализ и интерпретацию информации, необходимой для выполнения задач профессиональной деятельности.
ОК 3	Планировать и реализовывать собственное профессиональное и личностное развитие.
ОК 4	Планировать и реализовывать собственное профессиональное и личностное развитие.
ОК 5	Планировать и реализовывать собственное профессиональное и личностное развитие.
ОК 6	Проявлять гражданско-патриотическую позицию, демонстрировать осознанное поведение на основе традиционных общечеловеческих ценностей, применять стандарты антикоррупционного поведения.
ОК 7	Содействовать сохранению окружающей среды, ресурсосбережению, эффективно действовать в чрезвычайных ситуациях.
ОК 8	Использовать средства физической культуры для сохранения и укрепления здоровья в процессе профессиональной деятельности и поддержания необходимого уровня физической подготовленности
ОК 9	Использовать информационные технологии в профессиональной деятельности.
ОК 10	Пользоваться профессиональной документацией на государственном и иностранном языках
ОК 11	Использовать знания по финансовой грамотности, планировать предпринимательскую деятельность в профессиональной сфере
ПК 1.1.	Формировать алгоритмы разработки программных модулей в соответствии с техническим заданием
ПК 1.2.	Разрабатывать программные модули в соответствии с техническим заданием
ПК 1.3.	Выполнять отладку программных модулей с использованием специализированных программных средств
ПК 1.4.	Выполнять тестирование программных модулей
ПК 1.5.	Осуществлять рефакторинг и оптимизацию программного кода

В результате освоения профессионального модуля обучающийся должен:

Иметь практический опыт	В разработке кода программного продукта на основе готовой спецификации на уровне модуля; использовании инструментальных средств на этапе отладки программного продукта; проведении тестирования программного модуля по определенному сценарию; использовании инструментальных средств на этапе отладки программного продукта; разработке мобильных приложений
Уметь	осуществлять разработку кода программного модуля на языках низкого и высокого уровней; создавать программу по разработанному алгоритму как отдельный модуль; выполнять отладку и тестирование программы на уровне модуля; осуществлять разработку кода программного модуля на современных языках программирования; уметь выполнять оптимизацию и рефакторинг программного кода; оформлять документацию на программные средства
Знать	основные этапы разработки программного обеспечения; основные принципы технологии структурного и объектно-ориентированного программирования; способы оптимизации и приемы рефакторинга; основные принципы отладки и тестирования программных продуктов

В соответствии с рабочей программой по ПМ 01. Разработка модулей программного обеспечения для компьютерных систем, практические работы по МДК 01.01. Разработка программных модулей проводятся в пятом, шестом, седьмом семестре. Целесообразность данной группировки обусловлена необходимостью обобщения и систематизации знаний перед экзаменом.

Рабочая программа профессионального модуля предусматривает проведение практических работ МДК 01.01. Разработка программных модулей в объеме 122 часов.

Порядок выполнения практической и лабораторной работы

- записать название работы, ее цель в тетрадь;
- выполнить основные задания в соответствии с ходом работы;
- выполнить индивидуальные задания.

Рекомендации по оформлению практической и лабораторной работы

Задания выполняются обучающимися по шагам. Необходимо строго придерживаться порядка действий, описанного в практической работе

Результаты выполнения практических работ необходимо сохранять в своей папке на компьютере или USB – накопителе.

В случае пропуска занятий обучающийся осваивает материал самостоятельно в свободное от занятий время и сдает практическую работу с пояснениями о выполнении.

Критерии оценки практической и лабораторной работы

- наличие Цель выполняемой работы, выполнение более половины основных заданий (удовлетворительно);
- наличие Цель выполняемой работы, выполнение всех основных и более половины дополнительных заданий (хорошо);
- наличие Цель выполняемой работы, выполнение всех основных и индивидуальных заданий (отлично).

Перечень практических работ

№	Наименование разделов и тем профессионального модуля (ПМ)	Наименование лабораторных работ и практических занятий	Объем часов
	Раздел 1. Разработка программных модулей		
	МДК 01.01. Разработка программных модулей		
1.	**Тема 1.2. Оновные этапы разработки программного обеспечения**	Практическая работа № 1.1. Формирование алгоритмов разработки программных модулей в соответствии с техническим заданием	2
2.		Практическая работа № 1.2. Оформление документации на программные средства	2
3.	**Тема 1.3. Методы программирования**	Практическая работа № 1.3. Создание программ по разработанному алгоритму как отдельный модуль	2
4.		Практическая работа № 1.4. Разработка алгоритма поставленной задачи	2
5.		Практическая работа № 1.5. Реализация алгоритма поставленной задачи средствами автоматизированного проектирования	2
6.		Практическая работа № 1.6. Использование инструментальных средств на этапе отладки программного модуля	2
7.	**Тема 1.4. Структурное программирование**	Практическая работа № 1.7. Оценка сложности алгоритмов сортировки	2
8.		Практическая работа № 1.8. Оценка сложности алгоритмов поиска	2
9.		Практическая работа № 1.9. Оценка сложности рекурсивных алгоритмов	2
10.		Практическая работа № 1.10. Оценка сложности эвристических алгоритмов	2
11.	**Тема 1.5. Объектно-ориентированное программирование**	Практическая работа № 1.11. Работа с классами	2
12.		Практическая работа № 1.12. Перезагрузка методов	2
13.		Практическая работа № 1.13. Определение операций в классе	2
14.		Практическая работа № 1.14. Создание наследованных классов	2
15.		Практическая работа № 1.15. Работа с объектами через интерфейсы	2
16.		Практическая работа № 1.16. Использование стандартных интерфейсов	2
17.		Практическая работа № 1.17. Работа с типом данных структура	2
18.		Практическая работа № 1.18. Коллекции	2
19.		Практическая работа № 1.19. Параметризованные классы	2
20.		Практическая работа № 1.20. Использование регулярных выражений	2
21.		Практическая работа № 1.21. Операции со списками	2
22.	**Тема 1.6 . Паттерны проектирования**	Практическая работа № 1.22. Использование основных шаблонов	2
23.		Практическая работа № 1.23. Использование порождающих шаблонов	2
24.		Практическая работа № 1.24. Использование структурных шаблонов	2
25.		Практическая работа № 1.25. Использование поведенческих шаблонов	2
26.	**Тема 1.7. Событийно - управляемое программирование**	Практическая работа № 1.26. Разработка приложения с использованием текстовых компонентов	2
27.		Практическая работа № 1.27. Разработка приложения с несколькими формами	2
28.		Практическая работа № 1.28. Разработка приложения с не визуальными компонентами	2
29.		Практическая работа № 1.29. Разработка игрового приложения	2
30.		Практическая работа № 1.30. Разработка игрового приложения	2
31.		Практическая работа № 1.31. Разработка приложения с анимацией	2
32.	**Тема 1.8. Оптимизация и рефакторинг кода**	Практическая работа № 1.32. Оптимизация кода	2
33.		Практическая работа № 1.33. Рефакторинг кода	2

34.	**Тема 1.9. Разработка пользовательского интерфейса**	Практическая работа № 1.34. Пользовательская и программная модели интерфейса	2
35.		Практическая работа № 1.35. Разработка технического задания	2
36.		Практическая работа № 36. Разработка интерфейса пользователя	2
37.		Практическая работа № 1.37. Проектирование пользовательского интерфейса десктопного приложения	2
38.		Практическая работа № 1.38. Проектирование пользовательского интерфейса десктопного приложения	2
39.		Практическая работа № 1.39. Проектирование пользовательского интерфейса мобильного приложения	2
40.		Практическая работа № 1.40. Проектирование пользовательского интерфейса мобильного приложения	2
41.		Практическая работа № 1.41. Адаптивный веб-дизайн	2
42.		Практическая работа № 1.42. Адаптивный веб-дизайн	2
43.		Практическая работа № 1.43. Разработка протокола взаимодействия веб-сервисов	2
44.		Практическая работа № 1.44. Разработка REST API	2
45.		Практическая работа № 1.45. Разработка REST API	2
46.	**Тема 1.10. Основы ADO.Net**	Практическая работа № 1.46.Теоретические основы Технологии ado.Net	2
47.		Практическая работа № 1.47. Создание базы данных в среде MssqlServerManagement	2
48.		Практическая работа № 1.48. Создание базы данных в среде MssqlServerManagement	2
49.		Практическая работа № 1.49. Копирование и восстановление базы данных	2
50.		Практическая работа № 1.50. Разработка формы для работы с данными в среде VsualStudio без кода	2
51.		Практическая работа № 1.51. СозданиеSql- запросов	2
52.		Практическая работа № 1.52. СозданиеSql- запросов в среде sql Server Management Studio	2
53.		Практическая работа № 1.53. Программирование с помощью встроенного языка transact sql в Microsoft Sql Server	2
54.		Практическая работа № 1.54. Разработка формы работы с магазинами с использованием объекта Command	2
55.		Практическая работа № 1.55. Разработка формы работы с магазинами с использованием объекта Command	2
56.		Практическая работа № 1.56. Создание, удаление и редактирование данных в отсоединенной среде	2
57.		Практическая работа № 1.57. Быстрое создание пользовательского интерфейса посредством связывания с данными	2
58.		Практическая работа № 1.58. Безопасность вMssqlServer	2
59.		Практическая работа № 1.59. Создание отчетных форм для баз данных средствами MsVisualStudio	2
60.		Практическая работа № 1.60. Самостоятельная работа по автономным и подключенным объектам	2
61.		Практическая работа № 1.61. Самостоятельная работа по автономным и подключенным объектам	2
Итого			122

Практическая работа № 1.1. Формирование алгоритмов разработки программных модулей в соответствии с техническим заданием

Цель работы. Освоить процесс оформления спецификации требований к программному обеспечению.

Теоретическая часть.

Спецификация требований - это спецификация для определенного программного изделия, программы или набора программ, которые выполняют определенные функции в специфической среде. Спецификация требований может составляться одним или более представителями поставщика, одним или более представителями заказчика, или обоими. Предпочтительнее участие обоих.

Основными вопросами, которые должны рассматривать составитель (-ли) спецификации требований, являются следующие:

- ***Функциональные возможности***.

Каковы предполагаемые функции программного обеспечения?

- ***Внешние интерфейсы***.

Как программное обеспечение взаимодействуют с пользователями, аппаратными средствами системы, другими аппаратными средствами и другим программным обеспечением?

- ***Рабочие характеристики***.

Каково быстродействие, доступность, время отклика, время восстановления различных функций программного обеспечения и т.д.?

- ***Атрибуты***.

Каковы мобильность, правильность, удобство сопровождения, защищенность программного обеспечения и другие критерии?

- ***Проектные ограничения, налагаемые на реализацию изделия***.

Существуют ли требуемые стандарты на эффективном языке реализации, политика по сохранению целостности баз данных, ограничения ресурсов, операционная среда(-ы) и т.д.?

Спецификация требований к ПО:

- Должна правильно определять все требования к программному обеспечению. Причиной существования какого-либо требования к программному обеспечению может являться характер решаемой задачи или особая характеристика проекта.
- Не должна описывать детали разработки или реализации. Они должны быть описаны на этапе разработки проекта.
- Не должна налагать дополнительные ограничения на программное обеспечение. Эти ограничения надлежащим образом определяются в других документах, таких как план обеспечения качества программных средств.

Правильно составленная спецификация требований к ПО должна быть:

- ***Корректной;***

Спецификация требований к ПО является корректной, если, и только, если каждое требование, изложенное в ней, является требованием, которому должно удовлетворять программное обеспечение.

Не существует какого-либо средства или процедуры, которое гарантирует корректность. Спецификация требований к ПО должна сравниваться с любой качественной применимой спецификацией, такой как спецификация требований к системе, с другой документацией проекта и с другими применимыми стандартами, и должна гарантировать согласованность. В качестве альтернативы заказчик или пользователь может определить, правильно ли спецификация требований к ПО отражает фактические потребности.

- ***Однозначной;***

Спецификация требований к ПО является однозначной, если и только, если каждое изложенное в ней требование может интерпретироваться только однозначно. Как минимум, для этого требуется, чтобы каждая характеристика конечного продукта была описана с использованием одного уникального термина.

В тех случаях, когда термин, используемый в специфическом контексте, может иметь множественные значения, этот термин должен быть включен в глоссарий, в котором его значение описывается более конкретно.

Спецификация требований к ПО должна быть однозначной как для тех, кто составляет ее, так и для тех, кто ее использует. Однако, эти группы часто не имеют одинаковую квалификацию и, следовательно, не имеют тенденции к описанию требований к программному обеспечению одним и тем же образом. Способы представления требований, которые улучшают спецификацию требований для разработчика, могут оказаться неэффективными в том, что они уменьшают их понимание пользователем, и наоборот.

- ***Полной;***

Спецификация требований к ПО является полной, если и только, если она включает следующие элементы:

  - Все существенные требования, независимо от того, относятся ли они к функциональным возможностям, рабочим характеристикам, проектным ограничениям, атрибутам или внешним интерфейсам. В частности, должны быть подтверждены и обработаны любые внешние требования, налагаемые спецификацией системы.
  - Определение откликов программного обеспечения на все классы входных данных, которые могут быть реализованы, во всех возможных ситуациях. Следует заметить, что важно определить отклики, как на допустимые, так и недопустимые входные значения.
  - Полные обозначения и ссылки на все рисунки, таблицы и схемы в спецификации требований к ПО и определение всех терминов и единиц измерения.

- ***Непротиворечивой;***

Непротиворечивость обозначает внутреннюю непротиворечивость. Если спецификация требований к ПО не согласовывается с каким-то документом более высокого уровня, таким как, например, спецификации системных требований, то она является некорректной.

- ***Упорядоченной по ее значимости и/или устойчивости;***

Спецификация требований к ПО является упорядоченной по значимости и/или устойчивости, если каждое требование в ней имеет идентификатор, указывающий или значимость или устойчивость этого конкретного требования.

Как правило, все требования, которые касаются программного изделия, не являются важными в равной степени. Некоторые требования могут быть необходимыми, особенно для приложений, критичных в течение жизненного цикла, в то время как другие могут быть желательными.

Каждое требование в спецификации требований к ПО должно быть идентифицировано, чтобы сделать эти различия четкими и явными. Идентификация требований следующим образом помогает:

  - заказчикам более тщательно рассмотреть каждое требование, что часто позволяет разъяснить любые скрытые допущения, которые могут быть заключены в них.
  - разработчикам принять правильные проектные решения и приложить соответствующие усилия к различным составляющим программного изделия.

- ***Проверяемой;***

Спецификация требований к ПО является проверяемой, если и только, если каждое требование, изложенное в ней, может быть проверено. Требование являются проверяемым, если и только, если существует некий конечный эффективный процесс, используя который пользователь или машина могут убедиться, что программное изделие удовлетворяет этому требованию. В целом, любое неоднозначное требование не проверяемо.

Непроверяемые требования включают формулировки типа "работает хорошо", "хороший интерфейс с пользователем" и "обычно должно происходить". Эти требования не могут быть проверены, так как невозможно определить термины "хороший", "хорошо" или "обычно". Утверждение о том, что "программа никогда не должна зацикливаться" является непроверяемым, так как тестирование этого свойства теоретически невозможно.

Примером проверяемого утверждения является следующее:

Выходные данные программы должны вырабатываться в пределах 20 секунд в течение 60 % временного интервала события; и должны вырабатываться в пределах 30 секунд в течение 100 % временного интервала события.

Это утверждение может быть проверено, так как в нем используются конкретные термины и измеримые величины.

Если нельзя изобрести метод, чтобы определить, отвечает ли программное обеспечение определенному требованию, то это требование следует исключить или пересмотреть.

• ***Модифицируемой;***

Спецификация требований к ПО является модифицируемой, если и только, если ее структура и стиль таковы, что любые изменения требований могут быть выполнены легко, полностью и непротиворечивым образом при сохранении структуры и стиля. Как правило, модифицируемость требует, чтобы спецификация требований:

1) Имела связанную и легкую в использовании структуру с оглавлением, алфавитным указателем и явно выраженными перекрестными ссылками;

2) Не была избыточной (то есть, одно и то же требование не должно появляться в спецификации требований более чем в одном месте);

3) Выражала каждое требование раздельно, не смешивая его с другими требованиями.

Избыточность сама по себе не является ошибкой, но она легко может привести к появлению ошибок. Иногда избыточность может помочь сделать спецификацию требований более читаемой, но тогда могут возникнуть проблемы при модификации избыточного документа. Например, требование может быть изменено только в одном из тех мест, где оно появляется. Тогда спецификация требований становится противоречивой. Каждый раз, когда избыточность необходима, спецификация требований должна включать явные перекрестные ссылки, чтобы сделать ее модифицируемой.

• ***Отслеживаемой.***

Спецификация требований к ПО является отслеживаемой, если четко прослеживается источник каждого из ее требований и если она облегчает обращение к каждому из требований при дальнейшей разработке или модернизации документации. Рекомендуются следующие два типа отслеживаемости:

1. Обратная отслеживаемость (то есть, к предыдущим стадиям разработки).

Зависит от каждого требования, которое в явном виде ссылается на его источник в более ранних документах.

2. Прямая отслеживаемость (то есть, ко всем документы, порождаемым спецификацией требований.

Зависит от каждого требования в спецификации требований, имеющего однозначно определенное имя или номер ссылки.

Прямая отслеживаемость спецификации требований особенно важна, когда программное изделие вступает в стадию функционирования и сопровождения. По мере изменения кода и проектных документов необходимо иметь возможность определить полный набор требований, на которые могут повлиять эти изменения.

Задание.

1. Оформить внешнюю спецификацию к задаче.
2. Составить в виде блок-схемы алгоритм решения задачи.
3. Создать программу решения задачи на любом алгоритмическом языке программирования.
4. Отладить программу.
5. Составить отчет по практической работе.

Отчет по практической работе должен включать:

1. Внешнюю спецификацию.
2. Алгоритм решения задачи.
3. Текст программы на языке программирования.
4. Набор тестов для отладки программы.

<u>Задача:</u> Составить алгоритм и написать программу нахождения экстремального значения и/или его порядкового номера для заданных одномерных массивов (A[N], B[M], где N и M – размер массивов).

Варианты индивидуальных заданий.

1. Определить наименьшую среди сумм $\sum_{i=1}^{K}\left(A_i+\frac{1}{B_{K-i+1}}\right)$, ($K$=1,.., N) соответствующий номер K.
2. Определить две наибольшие по абсолютной величине разности A_i-A_{i-1}, где i=2..N, и соответствующие значения индекса i.
3. Определить наибольшее из отношений $\frac{A_i}{B_{N-i+1}}$, где i=1,...,N и соответствующий индекс i.

4. Определить наименьшее и наибольшее значения разности A_i - B_{N-i+1}, где i=1..N, и соответствующий индекс i.
5. Определить наибольшую среди сумм $\sum_{i=1}^{K} A_i$, $\sum_{i=1}^{K} B_i$, (K=1,..,N) и соответствующий номер K.
6. Определить два наименьших из значений $\sum_{i=1}^{K} (B_i^2 - B_i + K)$, ($K$=1,..,$M$) и соответствующие номера K.
7. Определить наименьшее из значений $\sum_{i=1}^{K} \left(\frac{1}{A_i}\right)$, ($K$=1,..,$N$) и соответствующий номер K.
8. Определить наименьшую среди сумм $\sum_{i=1}^{K} \left(\frac{1}{A_i}\right)$, $\sum_{i=1}^{K} B_i$, (K=1,..,N) и соответствующий номер K.
9. Определить наибольшее из произведений $\prod_{i=1}^{K} A_i, (K=1,[?][?][?],N)$, $\prod_{i=1}^{K} B_i, (K=1,[?][?][?],M)$.
10. Определить наибольшее из произведений $\prod_{i=1}^{K} A_i^2, (K=1,[?][?][?],N)$ и соответствующий номер K.
11. Определить наименьшее среди произведений $\prod_{i=1}^{K} \left(\frac{1}{A_i} - B_i\right)$, ($K$=1,..,$M$) и соответствующий номер K.
12. Определить наименьшую среди сумм $\sum_{i=1}^{K} A_i$ и $\sum_{i=1}^{K} \left(\frac{1}{B_i}\right)$, ($K$=1,..,$N$) и соответствующий номер K.
13. Определить два наибольших из абсолютных значений $\prod_{i=1}^{K} \left(\frac{A_i}{B_i}\right)$, $\sum_{i=1}^{K} (B_i + A_i)$, ($K$=1,..,$M$) и соответствующие номера K.
14. Определить наименьшее из значений $\sum_{i=1}^{K} (B_i^2 + B_i - K)$, ($K$=1,.., M) и соответствующий номер K.
15. Определить наибольшее по абсолютной величине из отношений $\frac{\sum_{i=1}^{K} A_i}{\prod_{i=1}^{K} A_i}$,($K$=1,..,$N$) и соответствующий номер K.
16. Определить наименьшее из значений $A_i^{B_i}$, (i=1,.., N) и соответствующий индекс i.
17. Определить два наибольших из произведений $\prod_{i=1}^{K} \left(\frac{1}{A_i^2}\right)$, ($K$=1,…,$N$) и соответствующий номер K.
18. Определить наименьшее из значений $\left(\sqrt{A_i} - A_i^3\right)$, ($i$=1,.., N) и номер соответствующего индекса i.

19. Определить наибольшее среди произведений $\prod_{i=1}^{K}(A_i+B_i)$, (K=1,..,M) и соответствующий номер K.
20. Определить наименьшее из значений $\prod_{i=1}^{K}(A_i+B_i)$, $\sum_{i=1}^{K}\left(\frac{B_i}{A_i}\right)$, ($K$=1,..,$M$) и соответствующий номер K.
21. Определить два наибольших из произведений $A_i * A_{N-i+1}$, (i=1,.., N) и соответствующие значения индекса i.
22. Определить наименьшее по абсолютной величине из значений $\prod_{i=1}^{K}\left(A_i^2-\frac{1}{B_i}\right)$ и $\sum_{i=1}^{K}\left(B_i+\frac{1}{A_i}\right)$, ($K$=1,..,$N$) и соответствующие номера K.
23. Определить наибольшее по абсолютной величине из значений $\prod_{i=1}^{K}(A_i-B_{i+1}^2)$, ($K$=1,..,$N$), и соответствующее значение K.
24. Определить два наименьших по абсолютной величине из значений $\prod_{i=1}^{K}(A_i-B_{i+1}^2)$, ($K$=1,..,$M$) и соответствующее значение K.
25. Определить наименьшее и наибольшее из произведений $\prod_{i=1}^{K}\left(A_i+\sqrt{B_i}\right)$, ($K$=1,..,$N$) и соответствующие значения K.

Практическая работа № 1.2. Оформление документации на программные средства

Цель работы приобретение практических навыков по разработке технологической проектной документации на программные средства различного назначения согласно требованиям стандартов ЕСПД.

Теоретический материал

ГОСТ 19.202. Спецификация. Требования к содержанию и оформлению

Настоящий стандарт устанавливает форму и порядок составления программного документа «Спецификация», определенного ГОСТ 19.101. Спецификация является основным программным документом для компонентов, применяемых самостоятельно, и для комплексов. Для компонентов, не имеющих спецификации, основным программным документом является «Текст программы».

Информационную часть (аннотацию и содержание) допускается в документ не включать.

Форма спецификации приведена на рис.1.

Спецификация в общем случае должна содержать разделы:

- *документация;*
- *комплексы;*
- *компоненты*.

Наименование каждого раздела указывают в виде заголовка в графе «Наименование». Для документов, выполненных печатным способом, заголовок подчеркивают.

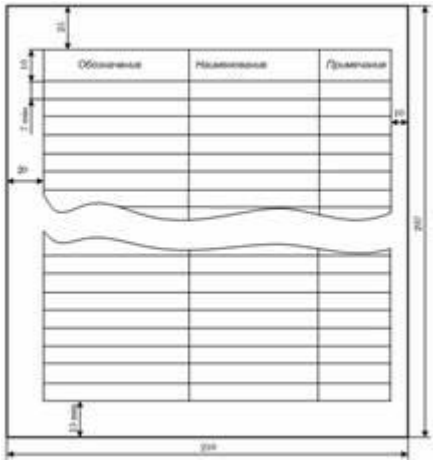


Рисунок 1 - Форма спецификации

В раздел *«Документация»* вносят программные документы на данную программу, кроме спецификации и технического задания, в порядке возрастания кода вида документа, входящего в обозначение. Далее записывают заимствованные программные документы. Запись их производится в порядке возрастания кодов предприятий-разработчиков и далее в порядке возрастания кода вида документа, входящего в обозначение.

После каждого раздела спецификации необходимо оставлять несколько свободных строк для дополнительных записей.

Графы спецификаций заполняют следующим образом:

- в графе «Обозначение» указывают:

в разделе «Документация» – обозначение записываемых документов программы;

в разделе «Комплексы» – обозначение спецификаций комплексов, входящих в данный комплекс;

в разделе «Компоненты» – обозначения основных программных документов компонентов.

- в графе «Наименование» указывают:

в разделе «Документация» – наименование и вид документа для документов на данную программу; полное наименование программы, наименование и вид документа для заимствованных документов;

в разделах «Комплексы» и «Компоненты» – полное наименование программы, наименование и вид документа.

- в графе «Примечание» указывают дополнительные сведения, относящиеся к записанным в спецификации программам.

В графе «Обозначение» запись производят в одну строку. В остальных графах спецификации записи допускаются в несколько строк.

ГОСТ 19.301. Программа и методика испытаний. Требования к содержанию, оформлению и контролю качества

Настоящий стандарт устанавливает требования к содержанию и оформлению программного документа «Программа и методика испытаний». Содержит номенклатуру показателей качества ПС, определяемых на основе результатов анализа данного документа, и методические указания по определению количественных значений показателей качества. Применяемость показателей качества и соответствующих им требований при разработке документа и контроле качества осуществляют в зависимости от принадлежности документируемой программы к конкретному подклассу ПС и устанавливают в соответствии с ГОСТ 28195. Наименования и обозначения показателей качества приведены по ГОСТ 28195.

1. Требования к содержанию

Составление информационной части (аннотации и содержания) является необязательным.

Документ *«Программа и методика испытаний»* должен содержать следующие разделы:

• объект испытаний;
• цель испытаний;
• требования к программе;
• требования к программной документации;
• средства и порядок испытаний;
• методы испытаний.

В зависимости от особенностей документа допускается вводить дополнительные разделы.

В разделе *«Объект испытаний»* указывают наименование, область применения и обозначение испытуемой программы.

В разделе *«Цель испытаний»* указывают цель проведения испытаний.

В разделе *«Требования к программе»* указывают требования, подлежащие проверке во время испытаний и заданные в ТЗ на программу, к которым относятся:

• требования устойчивости функционирования ПС при наличии ошибок во входных данных, а именно:

♦ контроль корректности входных данных;

♦ контроль принадлежности входных данных диапазону допустимых значений;

♦ контроль форматов входных данных;

♦ выдача диагностических сообщений пользователю при обнаружении ошибок во входных данных и предпринимаемые действия при обработке ошибок.

• требования возможности обработки ошибочных ситуаций;

• требования полноты обработки ошибочных ситуаций;

• требования к программе по восстановлению процесса выполнения в случае сбоя операционной системы, процессора, периферийных устройств, а именно:

♦ ведение системного журнала регистрации всех операций над данными;

♦ наличие средств получения копий выбранных частей данных, БД для последующего их восстановления;

♦ наличие средств восстановления для возврата БД или некоторых ее частей в первоначальное состояние;

♦ наличие средств, которые, используя системный журнал, устраняют в БД выполненные транзакции (единичная операция);

♦ наличие средств контроля, выявляющих нарушения и позволяющих отменять эффект выполнения предыдущей команды или нескольких предыдущих команд;

♦ наличие контрольных точек и средств, которые позволяют вернуться в последнюю контрольную точку вместо возвращения к началу транзакции;

♦ динамическое исключение неисправного устройства ввода–вывода из набора ресурсов ПС.

• требования к программе по восстановлению результатов при отказах процессора, операционной системы, которые включают в себя:

♦ восстановление вычислительного процесса и данных;

♦ восстановление данных, скопированных за некоторое время до момента сбоя;

♦ возможность повторного запуска ПС с последней контрольной точки.

• требования к тестированию программ;

• требования реализации диагностики всех граничных и аварийных ситуаций, которые создаются в процессе испытаний путем подбора входных данных;

• требования к динамическому тестированию программ;

• требования к статическому тестированию программ.

В разделе *«Требования к программной документации»* указывают состав программной документации, предъявляемой на испытания, и требования полноты и понятности изложения в документации информации:

• о назначении ПС;

• о принципах функционирования ПС;

• о взаимосвязи ПС с другими подсистемами;

• о входных и выходных данных;

• о действиях, относящихся к освоению работы с ПС (настройка, запуск, выполнение);

• о графическом представлении блок–схем, алгоритмов;

• о принятых соглашениях об использовании комментариев, символических имен переменных;

• о диагностических сообщениях, выдаваемых пользователю в ходе настройки, проверки и выполнения ПС;

• о наличии всех необходимых рисунков, формул, таблиц, которые должны содержать ту информацию, которая заложена в ссылке на нее.

В разделе *«Средства и порядок испытаний»* приводят:

• описание программной среды функционирования ПС, включающее в себя требования к:

♦ операционным системам и средствам их расширения;

♦ средствам управления базами данных;

♦ прочим ПС, используемым программой в процессе функционирования.

• описание программно–аппаратурной среды функционирования ПС, включающее в себя требования к:

♦ объему внутренней и внешней памяти, необходимому для функционирования ПС;

♦ периферийным устройствам;

♦ базовому программному обеспечению;

♦ другим техническим и программным средствам, используемым во время испытаний, а также порядку проведения испытаний.

• процедуры проверки соответствия программно–аппаратной среды функционирования предъявленным требованиям и порядок их выполнения;

• порядок выполнения процедур проверки корректности:

♦ функционирования программы на соответствие предъявленным требованиям;

♦ реализации всех основных функций;

♦ реализации всех частных функций.

В разделе *«Методы испытаний»* приводят описания используемых методов испытаний, в частности, описания тестов и способов проверок с указанием ожидаемых результатов испытаний (перечней тестовых примеров, контрольных распечаток тестовых примеров и т.п.).

2. Показатели качества, определяемые на основе результатов анализа раздела «Требования к программе»

Значение всех рассматриваемых показателей качества определяют на основе изучения и анализа раздела «Требования к программе» программного документа «Программа и методика испытаний» и материалов ТЗ.

При количественной оценке показателей качества ПС приняты следующие *общие правила*:

• если в ТЗ и в разделе «Требования к программе» отсутствуют требования какого–либо показателя качества, то этот показатель исключают из участия в экспертной оценке и ему присваивают значение 0;

• если требования реализованы или представлены в полном объеме, то этому показателю присваивают значение 1;

• если возможно вычислить значение показателя по формуле – приводится расчетная формула;

• если отсутствуют формулы для расчета значения показателя, то его значение определяют методом вычитания некоторых рекомендуемых «штрафов» из максимально возможной оценки за невыполнение некоторых требований.

Значение показателя *устойчивости функционирования* (Н0101) снижают при отсутствии требований:

• к контролю корректности входных данных – на 0,2;

• к контролю принадлежности входных данных диапазону допустимых значений – на 0,3;

• к контролю форматов входных данных – на 0,3;

• на выдачу диагностических сообщений об ошибке пользователю и предпринимаемые действия, связанные с обработкой возникшей ситуации при вводе ошибочных данных, – на 0,2.

Если требование возможности обработки ошибочных ситуаций (Н0102) существует в ТЗ или разделе «Требования к программе», то показателю Н0102 присваивают значение 1, в ином случае – значение 0.

Значение показателя *полноты обработки ошибочных ситуаций* (Н0103) определяют по формуле:

$$H0103 = 1 - \frac{N_н^о}{N_о^о} \qquad (1)$$

где $N_н^о$ – число необрабатываемых ошибочных ситуаций; $N_о^о$ – общее число ошибочных ситуаций при проведении эксперимента.

Требования по восстановлению процесса выполнения программы в случае сбоя операционной системы, процессора, внешних устройств (Н0201) могут содержать следующие требования:

• к фиксации и откату транзакций (создание файла отката) ПС;

• к целостности данных, т.е. наличие средств контроля и восстановления данных в случае нарушения целостности;

• на возможность запуска программы повторно с последней контрольной точки (т.е. наличие последней копии состояния ПС);

• на динамическое исключение неисправного устройства ввода–вывода из набора ресурсов ПС.

Если реализация ПС предполагает восстановление процесса выполнения в случае сбоя, то значение показателя определяют по формуле:

$$H0201 = \frac{N_p^{\text{в}}}{N_p^{\text{в}} + N_э^{\text{в}}} \qquad (2)$$

где $N_p^{\text{в}}$ – использованное в реализации число возможностей и средств по восстановлению процесса выполнения; $N_э^{\text{в}}$ – неиспользованное, по мнению эксперта, число возможностей и средств по восстановлению процесса выполнения.

Если реализуемый ПС алгоритм предполагает наличие требований к программе по восстановлению результатов выполнения при отказах процессора, операционной системы (Н0202), то значение показателя определяют по формуле:

$$H0202 = \frac{N_p^{\text{в}}}{N_p^{\text{в}} + N_э^{\text{в}}} \qquad (3)$$

где $N_p^{\text{в}}$ – использованное в реализации число возможностей и средств по восстановлению результатов выполнения; $N_э^{\text{в}}$ – неиспользованное, по мнению эксперта, число возможностей и средств по восстановлению результатов выполнения.

Значение показателя наличия требований к тестированию программ (С1702) определяют на основе изучений и анализа раздела «Требования к программе». В нем должны быть указаны все требования, для подтверждения, реализации которых используют методы тестирования, и приведены ссылки на фрагменты документа, содержащие описания соответствующих процедур тестирования. При наличии требований к тестированию программ в полном объеме показателю С1702 присваивают значение 1. При отсутствии какой–либо информации либо при наличии неточностей в ее описании значение показателя С1702 снижают на 0,1–0,2 за каждую погрешность вплоть до значения 0.

Если реализуемый ПС алгоритм предполагает реализации диагностики граничных и аварийных ситуаций (К1108), то значение показателя определяют по формуле:

$$K1108 = \frac{N_p^{\text{д}}}{N_p^{\text{д}} + N_э^{\text{д}}} \qquad (4)$$

где $N_p^{\text{д}}$ – использованное в реализации число диагностических проверок граничных и аварийных ситуаций; $N_э^{\text{д}}$ – неиспользованное, по мнению эксперта, число диагностических проверок граничных и аварийных ситуаций.

Значение показателя *наличия требований к динамическому тестированию программ* (К1301) определяют на основе изучений и анализа раздела «Требования к программе». В нем должны быть указаны все требования, для подтверждения, реализации которых используют методы динамического тестирования, и приведены ссылки на фрагменты документа, содержащие описания соответствующих процедур динамического тестирования. При наличии требований к динамическому тестированию программ в полном объеме показателю К1301 присваивают значение 1. При отсутствии какой–либо информации либо при наличии неточностей в ее описании значение показателя К1301 снижают на 0,1–0,2 за каждую погрешность вплоть до значения 0.

Значение показателя *наличия требований к статическому тестированию программ* (К1401) определяют на основе изучений и анализа раздела «Требования к программе». В нем должны быть указаны все требования, для подтверждения реализации которых используют методы статического тестирования, и приведены ссылки на фрагменты документа, содержащие описания соответствующих процедур статического тестирования. При наличии требований к статическому тестированию программ в полном объеме показателю К1401 присваивают значение 1. При отсутствии какой–либо информации либо при наличии неточностей в ее описании значение показателя К1401 снижают на 0,1–0,2 за каждую погрешность вплоть до значения 0.

3. Показатели качества, определяемые на основе результатов анализа раздела «Требования к программной документации»

Значение показателя полноты и понятности документации для освоения (У0201) определяют на основе изучения и анализа раздела «Требования к программной документации». Определяют полноту перечня документации, а также перечень требований, выполнение которых позволяет освоить документацию. Если документ содержит неполное описание документации, требуемой для освоения, значения показателя У0201 устанавливают от 0,1 до 0,8.

Значение показателя *наличия всех требуемых разделов* (У0604) определяют на основе изучения и анализа содержания всех разделов документа. Документ анализируют с точки зрения наличия в нем следующих разделов:

• объект испытаний;
• цель испытаний;
• требования к программе;
• требования к программной документации;
• средства и порядок испытаний;
• методы испытаний.

Если такие разделы существуют, то оценочному элементу присваивают значение 1. При отсутствии какого–либо раздела либо при наличии неточностей в его описании значение показателя У0604 снижают на 0,2–0,4 за каждую погрешность вплоть до значения 0.

Значение показателя *наличия всех рисунков, чертежей, формул, таблиц* (У0607) определяют на основе изучения и анализа содержания всех разделов документа, в которых должны быть даны ссылки на рисунки, чертежи, формулы, таблицы. Их наличие позволяет присвоить показателю значение 1. При отсутствии какого–либо рисунка, чертежа, формулы, таблицы либо при наличии неточностей в их описании значение показателя У0607 снижают на 0,2–0,4 за каждую погрешность вплоть до значения 0.

Значению показателя *правильности оформления титульных и заглавных листов документов* (К0703) присваивают значение 1, если лист утверждения и титульный лист оформлены в соответствии с ГОСТ 19.104. За каждое несоответствие стандарту значение показателя К0703 снижают на 0,2–0,4 вплоть до значения 0.

4. Показатели качества, определяемые на основе результатов анализа раздела «Средства и порядок испытаний»

Значению показателя *наличия описания программной среды функционирования ПС* (У0314) присваивают значение 1, если описание программной среды функционирования присутствует в необходимом объеме. При отсутствии одного из требований значение показателя У0314 снижают на 0,2–0,5. Если описание программной среды функционирования ПС отсутствует, то показателю У0314 присваивают значение 0.

Значение показателя *требуемого объема внутренней памяти* (Э0601) определяют на основе изучения и сопоставительного анализа раздела «Средства и порядок испытаний» документа «Программа и методика испытаний» и разделов «Используемые технические средства» и «Общие сведения» документа «Описание программы» (ГОСТ 19.402).

Если в разделе «Средства и порядок испытаний» данного документа указан требуемый для функционирования ПС объем оперативной памяти, который согласуется с соответствующими характеристиками всех ЭВМ, указанных в разделе «Используемые технические средства» документа «Описание программы» с учетом аналогичных требований со стороны программной среды и программного обеспечения, установленных в разделе «Общие сведения» того же документа, то показателю Э0601 присваивают значение 1. В остальных случаях значение показателя определяется по формуле:

$$Э0601 = 1 - \frac{N_н^п}{N_о^п} \qquad (5)$$

где $N_н^п$ – число несогласованных по требуемому объему внутренней памяти программно–аппаратных комбинаций среды функционирования ПС; $N_о^п$ – общее число возможных программно–аппаратных комбинаций среды функционирования ПС.

Значение показателя *требуемого объема внешней памяти* (Э0602) определяют на основе изучения и сопоставительного анализа раздела «Средства и порядок испытаний» документа «Программа и методика испытаний» и разделов «Используемые технические средства» и «Общие сведения» документа «Описание программы».

Если в разделе «Средства и порядок испытаний» данного документа указан требуемый для функционирования ПС объем внешней памяти, который согласуется с соответствующими характеристиками всех устройств, указанных в разделе «Используемые технические средства»

документа «Описание программы» с учетом аналогичных требований со стороны программной среды и программного обеспечения, установленных в разделе «Общие сведения» того же документа, то показателю Э0602 присваивают значение 1. В остальных случаях значение показателя определяется по формуле:

$$Э0602 = 1 - \frac{N_н^с}{N_о^с} \qquad (6)$$

где $N_н^с$ – число несогласованных по требуемому объему внешней памяти программно–аппаратных комбинаций среды функционирования ПС; $N_о^с$ – общее число возможных программно–аппаратных комбинаций среды функционирования ПС.

Значение показателя *требуемых периферийных устройств* (Э0703) определяют на основе изучения и сопоставительного анализа раздела «Средства и порядок испытаний» документа «Программа и методика испытаний» и раздела «Используемые технические средства» документа «Описание программы». Значение показателя определяют с точки зрения соответствия предъявленным требованиям, целесообразности и эффективности использования, по мнению эксперта, указанных в разделах периферийных устройств в процессе функционирования ПС. При отсутствии упущений показателю Э0703 присваивают значение 1. Значение показателя Э0703 при наличии упущений снижают на 0,2–0,5 по каждому случаю вплоть до значения 0.

Значение показателя *требуемого базового программного обеспечения* (Э0704) определяют на основе изучения и сопоставительного анализа раздела «Средства и порядок испытаний» документа «Программа и методика испытаний» и раздела «Общие сведения» документа «Описание программы». Значение показателя определяют с точки зрения соответствия предъявленным требованиям указанного в разделах базового программного обеспечения. При полном соответствии показателю Э0704 присваивают значение 1. Значение показателя Э0704 при наличии упущений снижают на 0,2–0,5 по каждому случаю вплоть до значения 0.

Значение показателя *отсутствия ошибок в описании действий пользователя* (К0803) определяют на основе изучения и анализа раздела «Средства и порядок испытаний». Если в описанных действиях, относящихся к загрузке, запуску, выполнению и завершению процедур проверки корректности функционирования программы на соответствие предъявленным требованиям, отсутствуют ошибки, то показателю К0803 присваивают значение от 0,8 до 1. За каждую ошибку в зависимости от тяжести последствий значение показателя К0803 снижают на 0,2–0,4 вплоть до значения 0.

Значение показателя *отсутствия ошибок в описании запуска* (К0804) определяют на основе изучения и анализа раздела «Средства и порядок испытаний». Если в описанных действиях, относящихся к запуску процедур проверки корректности функционирования программы на соответствие предъявленным требованиям, отсутствуют ошибки, то показателю К0804 присваивают значение от 0,8 до 1. За каждую ошибку в зависимости от тяжести последствий значение показателя К0804 снижают на 0,2–0,4 вплоть до значения 0.

Значение показателя *отсутствия ошибок в описании настройки* (К0806) определяют на основе изучения и анализа раздела «Средства и порядок испытаний». Если в описанных действиях, относящихся к настройке и запуску процедур проверки корректности функционирования программы на соответствие предъявленным требованиям, отсутствуют ошибки, то показателю К0806 присваивают значение от 0,8 до 1. За каждую ошибку в зависимости от тяжести последствий значение показателя К0806 снижают на 0,2–0,4 вплоть до значения 0.

Значение показателя *реализации всех основных функций* (К1102) определяют на основе изучения и анализа раздела «Средства и порядок испытаний». Если в разделе полностью описаны процедуры проверки корректности реализации всех основных функций, то показателю К1102 присваивают значение 1. При наличии упущений значение показателя К1102 снижают на 0,2–0,5 по каждому случаю вплоть до значения 0.

Значение показателя *реализации всех частных функций* (К1103) определяют на основе изучения и анализа раздела «Средства и порядок испытаний». Если в разделе полностью описаны процедуры проверки корректности реализации всех частных функций, то показателю К1103 присваивают значение 1. При наличии упущений значение показателя К1103 снижают на 0,2–0,5 по каждому случаю вплоть до значения 0.

5. Показатели качества, определяемые на основе результатов анализа раздела «Методы испытаний»

Значение показателя *наличия тестов для проверки допустимых значений входных данных* (Н0104) определяют на основе изучения и анализа раздела «Методы испытаний». Если в разделе тесты для проверки допустимых значений присутствуют и полностью охватывают все функции ПС, обеспечивающие проверку допустимых значений входных данных, то показателю Н0104 присваивают значение 1. При наличии упущений значение показателя Н0104 снижают на 0,2–0,5 по каждому случаю вплоть до значения 0.

Значение показателя *наличия описания способов проверки работоспособности программы* (К0114) определяют на основе изучения и анализа раздела «Методы испытаний». При полном и ясном описании способов проверки работоспособности программы с учетом всех предъявленных функциональных требований показателю К0114 присваивают значение

При наличии упущений значение показателя К0114 снижают на 0,2–0,5 по каждому случаю вплоть до значения 0.

ГОСТ 19.401. Текст программы. Требования к содержанию и оформлению

Аннотация и содержание не являются обязательными.

Основная часть документа должна состоять из текстов одного или нескольких разделов, которым даны наименования. Допускается вводить наименование также и для совокупности разделов. Каждый из этих разделов характеризуется одним из типов символической записи, например:

• символическая запись на исходном языке;
• символическая запись на промежуточных языках;
• символическое представление машинных кодов и т.п.

В символическую запись разделов рекомендуется включать комментарии, которые могут отражать, например, функциональное назначение, структуру.

ГОСТ 19.402. Описание программы

Составление информационной части (аннотации и содержания) является обязательным.

Описание программы должно содержать следующие разделы:

• общие сведения;
• функциональное назначение;
• описание логической структуры;
• используемые технические средства;
• вызов и загрузка;
• входные данные;
• выходные данные.

В зависимости от особенностей программы допускается вводить дополнительные разделы или объединять отдельные разделы.

В разделе «Общие сведения» должны быть указаны:

• обозначение и наименование программы;
• программное обеспечение, необходимое для функционирования программы;
• языки программирования, на которых написана программа.

В разделе «Функциональное назначение» должны быть указаны классы решаемых задач и/или назначение программы и сведения о функциональных ограничениях на применение.

В разделе «Описание логической структуры» должны быть указаны:

• алгоритм программы;
• используемые методы;
• структура программы с описанием функций составных частей и связи между ними;
• связи программы с другими программами.

Описание логической структуры программы выполняют с учетом текста программы на исходном языке.

В разделе «Используемые технические средства» должны быть указаны типы ЭВМ и устройств, которые используются при работе программы.

В разделе «Вызов и загрузка» должны быть указаны:

• способ вызова программы с соответствующего носителя данных;
• входные точки в программу.

Допускается указывать адреса загрузки, сведения об использовании оперативной памяти, объем программы.

В разделе «Входные данные» должны быть указаны:

• характер, организация и предварительная подготовка входных данных;
• формат, описание и способ кодирования входных данных.

В разделе «Выходные данные» должны быть указаны:

• характер и организация выходных данных;
• формат, описание и способ кодирования выходных данных.

Допускается содержание всех разделов иллюстрировать пояснительными примерами, таблицами, схемами, графиками.

В приложение к описанию программы допускается включать различные материалы, которые нецелесообразно включать в разделы описания.

ГОСТ 19.404. Пояснительная записка. Требования к содержанию и оформлению

Настоящий стандарт устанавливает требования к содержанию и оформлению программного документа «Пояснительная записка», входящего в состав документации на стадиях разработки эскизного и технического проекта программы.

Составление информационной части (аннотация и содержание) является необязательным.

Пояснительная записка должна содержать следующие разделы:

• введение;
• назначение и область применения;
• технические характеристики;
• ожидаемые технико-экономические показатели;
• источники, использованные при разработке.

В зависимости от особенностей документа отдельные разделы (подразделы) допускается объединять или вводить новые.

В разделе «Введение» указывают наименование программы и/или условное обозначение темы разработки, а также документы, на основании которых ведется разработка с указанием организации и даты утверждения.

В разделе «Назначение и область применения» указывают назначение программы, краткую характеристику области применения программы.

Раздел «Технические характеристики» должен содержать следующие подразделы:

• постановка задачи на разработку программы, описание применяемых математических методов и, при необходимости, описание допущений и ограничений, связанных с выбранным математическим материалом;
• описание алгоритма и/или функционирования программы с обоснованием выбора схемы алгоритма решения задачи, возможные взаимодействия программы с другими программами;
• описание и обоснование выбора метода организации входных и выходных данных;
• описание и обоснование выбора состава технических и программных средств на основании проведенных расчетов и/или анализов, распределение носителей данных, которые использует программа.

В разделе «Ожидаемые технико-экономические показатели» указывают технико-экономические показатели, обосновывающие выбранный вариант технического решения, а также, при необходимости, ожидаемые оперативные показатели.

В разделе «Источники, использованные при разработке» указывают перечень научно-технических публикаций, нормативно-технических документов и других научно-технических материалов, на которые есть ссылки в основном тексте.

В приложение к документу могут быть включены таблицы, обоснования, методики, расчеты и другие документы, использованные при разработке.

ГОСТ 19.502. Описание применения. Требования к содержанию и оформлению

Составление информационной части (аннотации и содержания) является обязательным.

Текст документа должен состоять из следующих разделов:

• назначение программы;
• условия применения;
• описание задачи;
• входные и выходные данные.

В зависимости от особенностей программы допускается вводить дополнительные разделы или объединять отдельные разделы.

В разделе «Назначение программы» указывают назначение, возможности программы, ее основные характеристики, ограничения, накладываемые на область применения программы.

В разделе «Условия применения» указываются условия, необходимые для выполнения программы (требования к необходимым для данной программы техническим средствам, и другим программам, общие характеристики входной и выходной информации, а также требования и условия организационного, технического и технологического характера и т.п.).

В разделе «Описание задачи» должны быть указаны определения задачи и методы ее решения.

В разделе «Входные и выходные данные» должны быть указаны сведения о входных и выходных данных.

В приложение к общему описанию могут быть включены справочные материалы (иллюстрации, таблицы, графики, примеры и т.п.).

Задания к практической работе

Предлагается разработать следующую технологическую документацию с соблюдением требований ЕСПД по их структуре и содержанию:

• описание программы;
• описание применения;
• пояснительная записка;
• программа и методика испытаний;
• спецификация.

В зависимости от предметной области и вида ПС, выданного в качестве задания, возможно изменение преподавателем состава и содержания технологической документации.

Пояснительная записка является достаточно объемным документом. Поэтому для сложных ПС по согласованию с преподавателем можно сократить объем некоторых разделов, но, описав при этом, по 1–2 объекта.

В документации обязательно должны быть приведены таблицы, схемы, иллюстрации, копии экранов, поясняющие положения документов.

Если в документации требуется привести блок-схемы алгоритмов, тексты программ, их требуется оформить согласно требованиям соответствующих стандартов ЕСПД.

1. Изучите и законспектируйте теоретический материал «Единая система программной документации», обратив особое внимание на следующие вопросы:

2. Ответить на контрольные вопросы.

3. Используя результаты практической работы № 10, последовательно разработать технологическую документацию на заданное ПС, выполнив требования ЕСПД к ее содержанию и оформлению.

4. Отчетом по практической работе является оформленная документация.

Контрольные вопросы

1. Требования к оформлению программных документов.
2. Требования к содержанию и оформлению технического задания.
3. Требования к содержанию и оформлению спецификации.
4. Требования к содержанию и оформлению программы и методики испытаний. Показатели качества программных средств, определяемые стандартом ГОСТ 19.301-2000.
5. Требования к содержанию и оформлению текста программы.
6. Требования к содержанию и оформлению описания программы.
7. Требования к содержанию и оформлению пояснительной записки.
8. Требования к содержанию и оформлению описания применения.
9. Требования к содержанию и оформлению руководств системного программиста, программиста, оператора и по техническому обслуживанию.

Практическая работа № 1.3. Создание программ по разработанному алгоритму как отдельный модуль

Цель работы: Создание программ по разработанному алгоритму как отдельный модуль

Теоретический материал:

Модуль (UNIT) – программная единица, текст которой компилируется независимо (автономно). Внутренняя структура модуля (тексты программ и т.д.) скрыта от пользователя.

Заголовок модуля

UNIT имя модуля;

Интерфейсная часть

INTERFACE начало раздела объявлений;

USES используемые при объявлении модули;

LABEL подраздел объявления доступных глобальных меток;

CONST подраздел объявления доступных глобальных констант;

TYPE подраздел объявления доступных глобальных типов;

VAR подраздел объявления доступных глобальных переменных;

PROCEDURE заголовки доступных процедур;

FUNCTION заголовки доступных функций;

Реализационная часть

IMPLEMENTATION начало раздела реализации;

USES используемые при реализации модули;

LABEL подраздел объявления скрытых глобальных меток;

CONST подраздел объявления скрытых глобальных констант;

TYPE подраздел объявления скрытых глобальных типов;

VAR подраздел объявления скрытых глобальных переменных;

PROCEDURE тела доступных и скрытых процедур;

FUNCTION тела доступных и скрытых функций;

Инициализационная часть

BEGIN

Операторы, которые выполняются при подключении модуля;

END.

1) Создайте модуль для вычисления факториала некоторого числа. В основной программе, не объявляя никаких переменных, осуществите вывод на экран факториалов чисел a и b.

2) Напишите программу для ввода некоторого числа X и вывода факториала этого числа. В программе должны использоваться два модуля: модуль для проверки, является ли число X целым и положительным, модуль для вычисления факториала (можно применить модуль, созданный для решения предыдущей задаче).

Ход работы:

В тетрадь оформите листинги программ и модулей с комментариями

1. Основная программа, согласно условию, будет содержать подключение модуля, ввод чисел a и b, вызов функции для каждого числа и вывод результата на экран. Текст программы можно представить так:

```
usees unitF; {подключение модуля}
begin
    writeln('введите два числа');
    readln(a,b); {ввод переменных, описанных в модуле}
    f1:=factor(a);{вызов функции, описанной в модуле; вычисление факториала от a}
    f2:=factor(b);      {вызов функции, описанной в модуле; вычисление факториала от b}
    writeln('факториал числа a = ', f1);
    writeln('факториал числа b = ', f2);
    readln;
end.
```

Модуль создаётся отдельно от основной программы. Его название должно совпадать с именем файла, в котором он записан (unitF). Для удобства использования модули сохраняют с расширениями pas и tpu, т.к. откомпилированный модуль нельзя исправить. В интерфейсной части модуля описываются переменные a, b типа integer и переменные f1, f2 типа longint (для записи результата используется тип с наибольшим диапазоном из целых типов), описывается функция factor. В реализационной части находится не только описание функции, но и сама функция. Инициализационная часть будет пустой. Текст модуля unitF (файлы unitF.tpu и unitF.pas):

```
unit unitF;   {заголовок модуля}
interface     {интерфейсная часть}
var a, b: integer; {описание переменных доступных из вызывающий модуль программ}
```

```
f1, f2: longint;
function factor(x: integer): longint; {описание доступной программам функции}
implementation    {реализационная часть}
function factor(x: integer): longint;    {тело функции}
var i: inttger;        {локальные переменные}
f: longint;
begin
f;=1;
for i:= 1 to x do              {цикл для вычисления факториала}
f:=f*i;
factor:=f;    {имени функции всегда присваивается результат}
end;   {завершение функции}
end.   {закрытие модуля, пустая инициализационная часть}
```

Задание 2. Решение этой задачи упрощается тем, что в предыдущей уже разработан модуль для вычисления факториала. Следовательно, необходимо создать ещё один модуль и исправить вызывающую программу. В модуле proverka находится функция: является ли число, от которого вычисляется факториал, положительным и целым, это проверяется так:

```
ost:=x-round(x);   {определение дробной части числа x}
if (x>=0) and (ost=0) then writeln('факториал числа = ',factor(round(x)))
else writeln('от данного числа не возможно найти факториал');
```

Если x неотрицательное и целое число, то вызывается функция для вычисления факториала из модуля unitf - factor.

Практическая работа № 1.4. Разработка алгоритма поставленной задачи

Цель работы: изучение метода разработки алгоритма поставленной задачи

Теоретически й материал

Алгоритм – это последовательность элементарных шагов, выполнение которой позволяет получать однозначный результат (не зависящий от того, кто выполнял эти шаги) или за конечное число шагов прийти к выводу о том, что решения не существует. [3].

Задача называется *алгоритмически неразрешимой*, если не существует машины, модели или алгоритма, которые ее бы решали.

Алгоритм может быть предназначен для выполнения его человеком или автоматическим устройством. *Создание алгоритма, пусть самого простого, - процесс творческий.* Другое дело – реализация уже имеющегося алгоритма, ее можно поручить субъекту или объекту, который не обязан вникать в существо дела, а возможно, и не способен его понять. Такой субъект или объект принято называть *формальным исполнителем*. Каждый алгоритм создается в расчете на вполне конкретного исполнителя. Совокупность действий (шагов) образует систему команд исполнителя. Алгоритм должен содержать только те действия, которые допустимы для данного исполнителя.

Чтобы алгоритм выполнил свое предназначение, его необходимо строить по определенным правилам.

Первое правило – необходимо задать множество объектов, с которыми будет работать алгоритм. *Формализованное* (в виде, удобном для записи, поиска и хранения в ПК) представление этих объектов носит название *данных*. Алгоритм приступает к работе с некоторым набором данных, которые называются *входными*, в результате своей работы выдает данные, которые называются *выходными*.

Второе правило – для работы алгоритма требуется память. В памяти размещаются входные, выходные и промежуточные данные. Поименованная ячейка памяти называется *переменной*. В теории алгоритмов размеры памяти не ограничиваются.

Третье правило – дискретность.

Четвертое правило – детерменированность. После каждого шага (действия) необходимо указывать, какой шаг выполняется следующим, либо дать команду остановки.

Пятое правило – сходимость (результативность). Алгоритм должен завершать работу после конечного числа шагов. При этом необходимо указать, что считается результатом работы алгоритма.

Виды алгоритмов

Виды алгоритмов как логико-математических средств:

механические – или детерминированные, жесткие, задают определенные действия в единственной и достоверной последовательности, обеспечивая однозначный и требуемый результат;

гибкие – дают последовательность нахождения решения задачи несколькими путями или способами, или это такие алгоритмы, в которых достижение результата однозначно не определено;

линейные – набор действий, выполняемых во времени последовательно, друг за другом;

разветвляющиеся – алгоритмы, содержащие хотя бы одно условие, в результате проверки которого программа переходит к одному из двух возможных шагов;

циклические – алгоритмы, предусматривающие многократное повторение одного и того же действия, но над новыми данными;

подчиненные (вспомогательные) – алгоритмы, ранее разработанные и целиком использованные при алгоритмизации задачи (обычно на их основе создаются подпрограммы).

Методы изображения алгоритмов

На практике распространены формы представления алгоритмов:

словесная - в виде последовательности записей на естественном языке*;*

графическая - в виде совокупности графических знаков*;*

псевдокоды – полуформализованное описание алгоритма на условном языке, включающем в себя как элементы языка программирования, так и фразы естественного языка, общепринятые математические обозначения и др.;

· *программная* – текст на языке программирования.

Выполнение алгоритма в виде блок-схемы перед программированием существенно облегчает процесс создания и отладки программы, определения форматов и перечня переменных, поиск ошибок, редактирование алгоритма в будущем.

Таблица 1 − *Знаки для изображения схем алгоритмов*

Обозначение (графическое изображение)	Название	Назначение	Наименование автофигуры в *Word*
Начало	Терминатор	Начало или завершение программы или подпрограммы	Знак завершения
A=B+C	Процесс	Обработка данных (вычисления, пересылки т.п.)	Процесс
B>C	Решение	Ветвления, выбор, итерационные и поисковые циклы	Решение
Ввод B, C	Данные	Операции ввода-вывода	Данные
Цикл i=1...10	Подготовка	Счетные циклы	Подготовка

Печать В и С	Документ	Вывод на бумагу	Документ
	Архив	Данные, хранящиеся в архиве или взятые из архива	-
	Документ	Документ, подготовленный вручную	-
Файл	Файл	Файл или база данных	Магнитный диск
Подпрограмма «Расчет 1»	Предопреде-ленный процесс	Вызов подпрограмм (процедур)	Типовой процесс
Отд. 118	Источник или приемник данных	Указание источника или приемника данных	-
12 «Ошибка!!!!»	Монитор	Вывод информации на экран	Дисплей
2	Соединитель	Маркировка разрывов линий	Узел
Стр.1 23	Соединитель	Маркировка разрывов линий	Ссылка на другую страницу
Комментарий к действию	Комментарий	Пояснения к действиям	Выноска
	Поток информации	Линии, связывающие блоки	Стрелка

В теории программирования доказано [1, 2], что для записи любого сложного алгоритма достаточно трех <u>*базовых структур*</u>: *следование* – последовательное выполнение действий (рис. 1,*а*); *ветвление* – соответствует выбору одного из двух вариантов действий (рис. 1,*б*); *цикл-пока* – определяет повторение действий, пока не будет нарушено условие, выполнение которого проверяется в начале цикла (рис. 2).

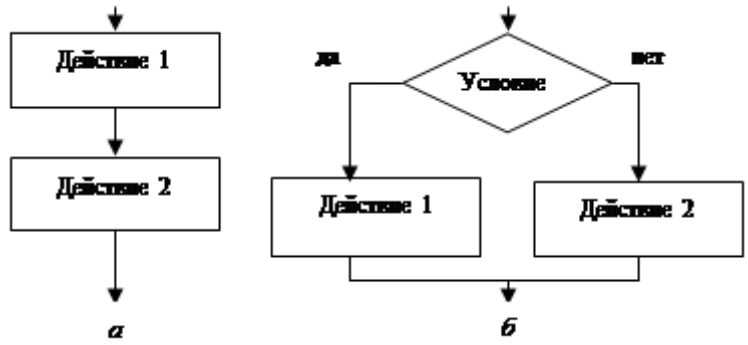


Рисунок 1 − Базовые алгоритмические структуры: *а*) следование, *б*) ветвление

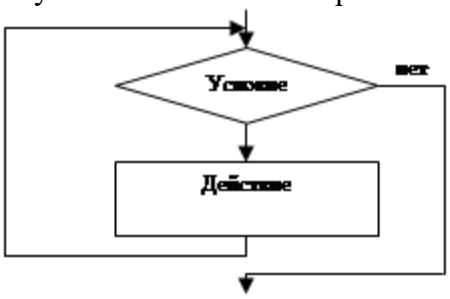


Рисунок 2 − Базовая структура: *цикл-пока*

На основе базовых структур строятся <u>*дополнительные структуры*</u> для изображения алгоритмов: выбор (рис. 3), цикл-до, счетный цикл.

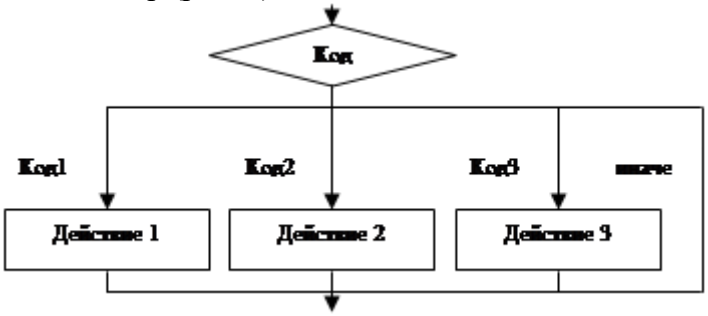


Рисунок 3 − Дополнительная структура «выбор» и реализация ее через базовые структуры

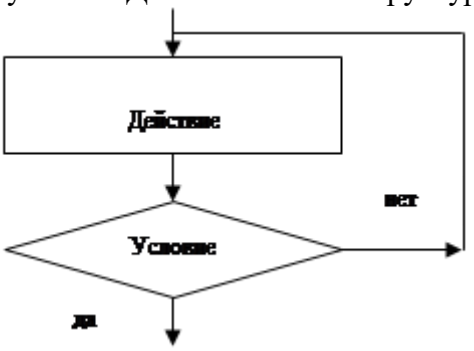


Рисунок 4 − Дополнительная структура: *цикл – до*

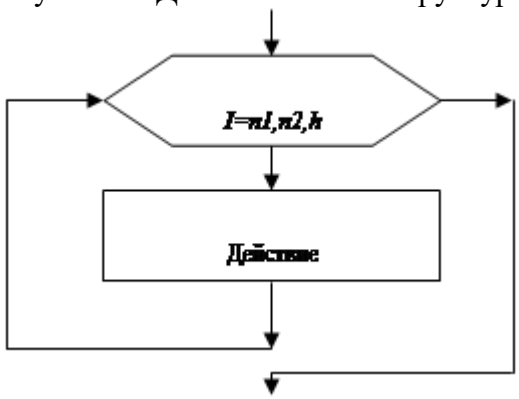


Рисунок 5 − Дополнительная структура: цикл с заданным числом повторений (*счетный цикл*).

На основе алгоритмов создается программное обеспечение (ПО) для решения прикладных задач.

Выражения

Все вычисления и другие преобразования данных в программе записываются в виде выражений. Обычно выражение включает несколько операций, которые выполняются в порядке их приоритетности (табл. 2).

Таблица 2 − *Приоритеты, присвоенные операциям*

Операции	Приоритет
(булево “не”)	
x(умножить),/(делить), *mod* (остаток от деления нацело), ∧ (булево “и”)	
+, -, ∨ (булево “или”)	
> , < , ≥ , ≤ , = (булево “равно”)	

Для изменения порядка выполнения операций используют круглые скобки. В выражениях допускается использование функций (табл. 3, рисунок 1). Им присваивается высший приоритет.

Таблица 3 − Функции SMath Studio из панели **Функции**

Математическая или словесная запись	Запись в SMathStudio	Математическая или словесная запись	Запись в SMathStudio
cosx	cos(x)	*sinx*	sin(x)
$\log_2 x$	$\log_2(x)$	*Определение знака числа*	sign(x)
lnx	ln(x)	*ctgx*	ctg(x)
tgx	tg(x)	*Производная*	$\frac{d}{d\bullet}\bullet$
e^x	exp(x)	*Интеграл*	$\int_{\bullet}^{\bullet}\bullet\, d\,\bullet$
Сумма ряда чисел (функция итерационного сложения)	$\sum_{\bullet=\bullet}^{\bullet}\bullet$	*Произведение ряда чисел (функция итерационного умножения)*	$\prod_{\bullet=\bullet}^{\bullet}\bullet$

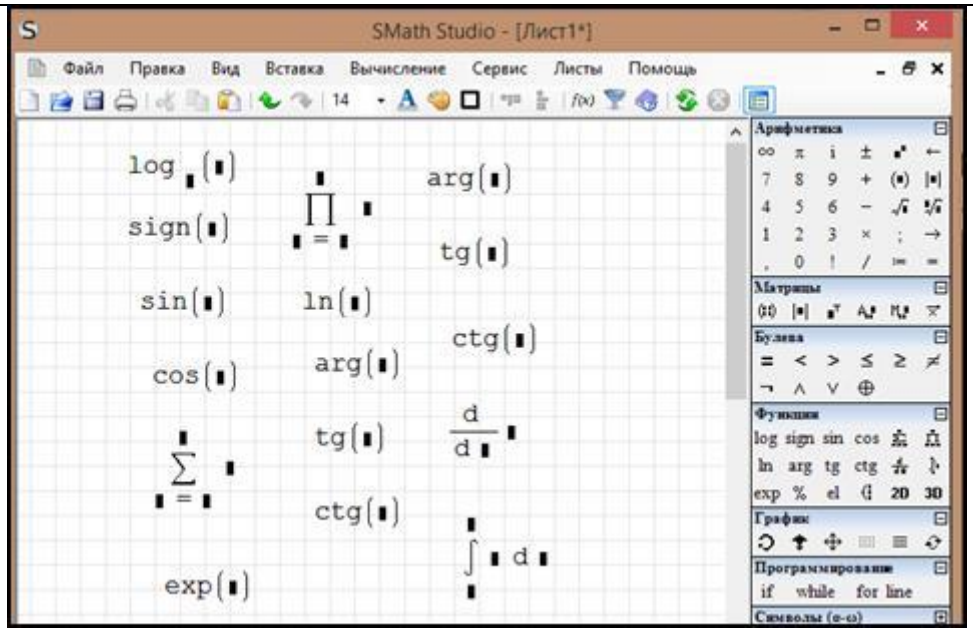


Рисунок 1 – Различные виды функций

Ход работы

*Этап 1. Постановка задачи.*Данрадиус окружности *R*. Вычислить длину окружности.

Этап 2. Анализ. Из математики известно, что длина окружности $l = 2\pi R$.

Этап 3. Проектирование и определение спецификаций. Составим линейный алгоритм для вычисления значения $l = 2\pi R$ (рис. 3). На этом же этапе рассчитаем тестовый пример (рис. 4). SMath Studio позволяет выполнять математические вычисления над данными имеющими размерность (в нашей задаче данные и результаты вычислений представлены в метрах).

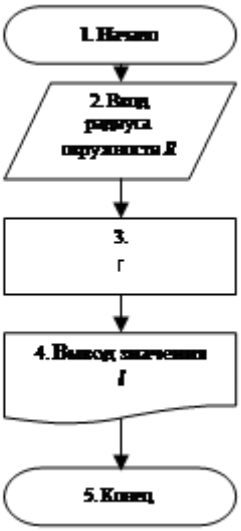


Рисунок 3 − Графический линейный алгоритм – следование - для расчета длины окружности.

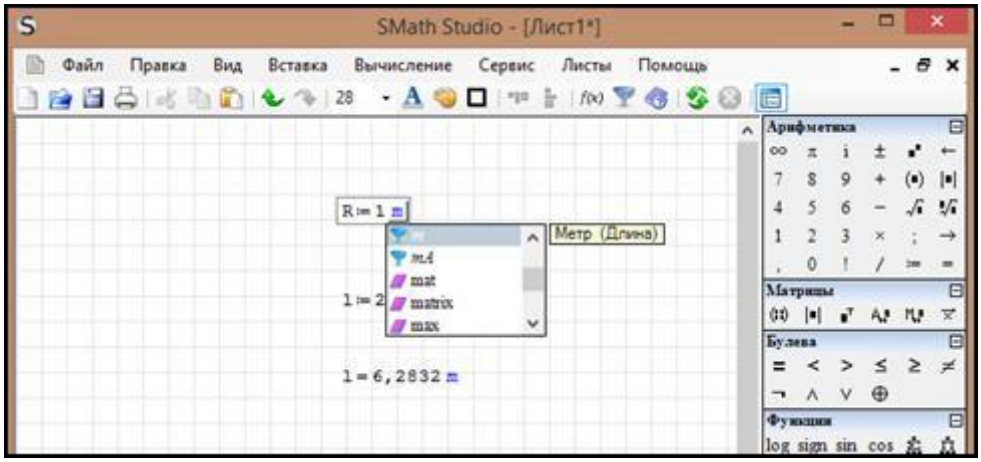


Рисунок 4 − Расчет тестового примера в SMath Studio с использованием единиц измерения

Примечание: важно следить за тем, чтобы все переменные и функции были *определены левее или выше тех выражений, где они используются.*

Таблица 4 − Спецификация к алгоритму и тестовый пример

№	Наименование	Обозначение в алгоритме	Обозначение в программе	Ед. изм.	Значение	Статус
	Радиус окружности	*R*	*R*	м		Входной параметр
	Длина окружности	*l*	*l*	м	6,283	Выходной параметр

Этап 4. Расчет тестового примера в SMathStudio (рисунок 5), результат заносится в таблицу 4.

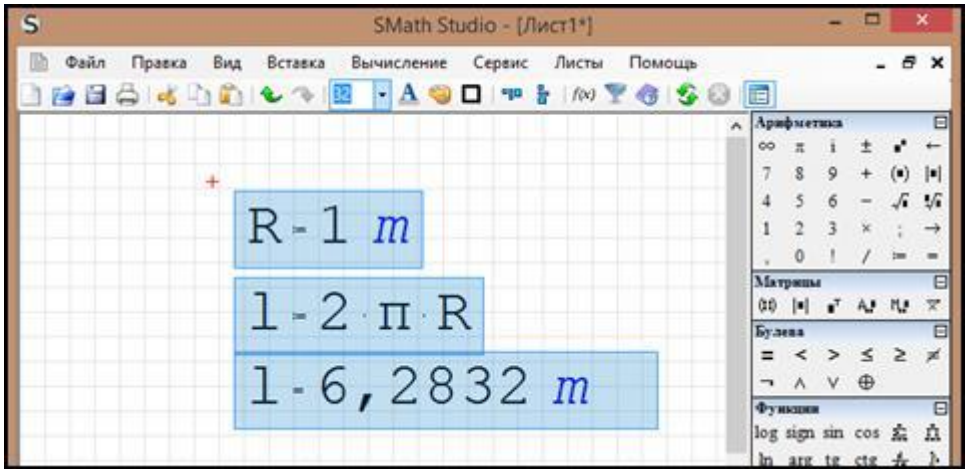


Рисунок 5 – Увеличение шрифта в SMath Studio

Список единиц измерения можно узнать с помощью кнопки главного меню (рис. 6).

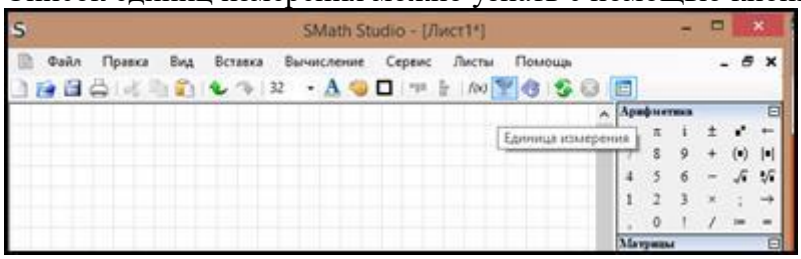


Рисунок 6 – Кнопка «Единица измерения»

Этап 5. Кодирование. Создаем и тестируем программу на PascalABC.

Примечание: PascalABC является свободно распространяемым программным продуктом.

Можно не устанавливать программу на ПК, а тестировать программы онлайн на сайте http://www.pascalabc.net/WDE/

```
PROGRAM Primer1;

VAR R: INTEGER; l: REAL;
BEGIN
R:=1; l:=2*PI*R; WRITELN('ДЛИНА ОКРУЖНОСТИ l=', l:4:2);
END.
```

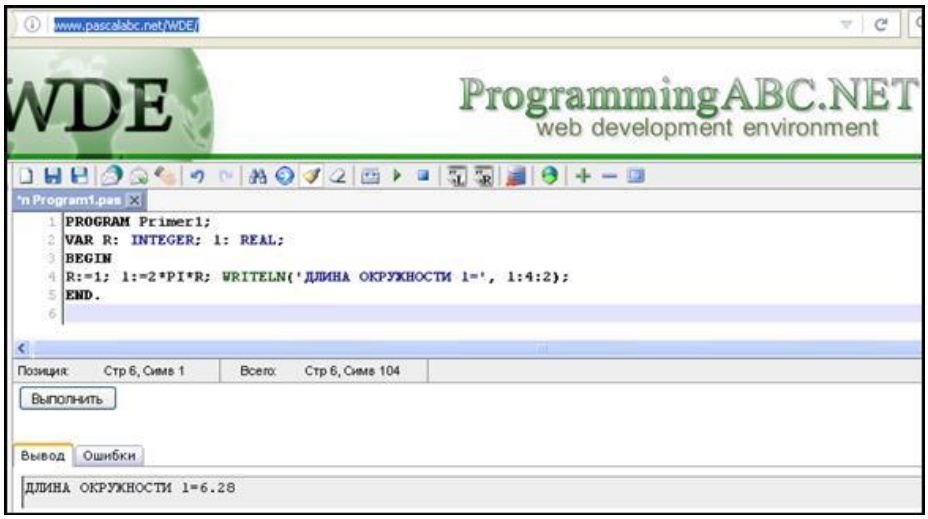


Рисунок 6.1 – Результат онлайн тестирования программы

Результат работы: разработан алгоритм и программа для решения поставленной задачи; рассчитан тестовый пример для проверки алгоритма и программы.

Практическая работа № 1.5. Реализация алгоритма поставленной задачи средствами автоматизированного проектирования

Цель работы: изучить способ разработки алгоритма поставленной задачи и реализация его средствами автоматизированного проектирования

Ход работы

Задание. Необходимо написать программу, которая будет выполнять действия на матрицами: умножения, сложения, вычитания, транспонирования. Программа должна решать введенные вручную матрицу в форму. Для удобства пользователя программа должна иметь интуитивно понятный интерфейс.

1. Выбор методов и разработка основных алгоритмов решения

В программе используется следующий алгоритм работы: в программе есть формы, в которые вводятся элементы матриц, элементы переводятся из String типа в Integer. Затем нужно нажать кнопку соответствующего действия. Выполняется алгоритм решения матриц и результат выводится в элемент DataGridView.

2. Для построения блок-схем использовалась программа Microsoft Office Visio 2013. С её помощью можно составлять различные диаграммы и схемы, в том числе, блок-схемы.

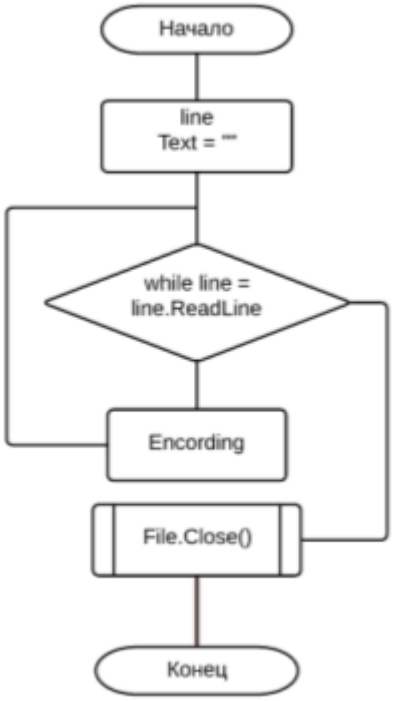


Рисунок 1.1 - Блок схема считывания и записи данных из записи в массив

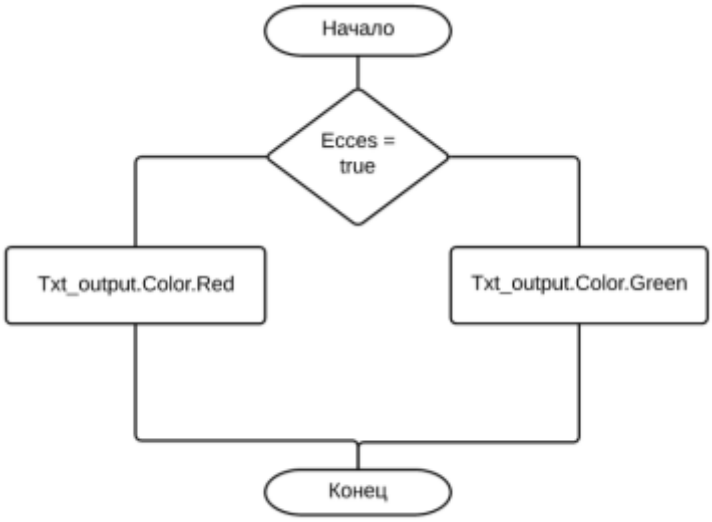


Рисунок 1.2 - Проверка на доступность для ввода

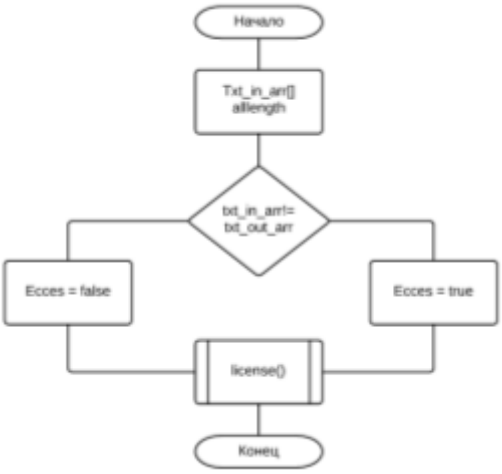


Рисунок 1.3 - Блок схема ввода данных в textbox и сравнения с существующим массивом

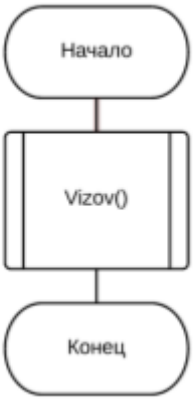


Рисунок 1.4 - Вызов метода Vizov с параметрами

3. Разработка кода программного продукта на основе готовой спецификации на уровне модуля

Калькулятор матриц реализован на языке программирования C# в среде программирования Microsoft Visual Studio Ultimate 2013. Выбор языка C# обусловлен тем, что он современный и популярный объектно-ориентированный язык программирования, а среда Microsoft Visual Studio Ultimate 2013 является мощным средством, позволяющим быстро создать программу, обладающую графическим оконным интерфейсом.

Макет окна представлен на рисунке 2.1

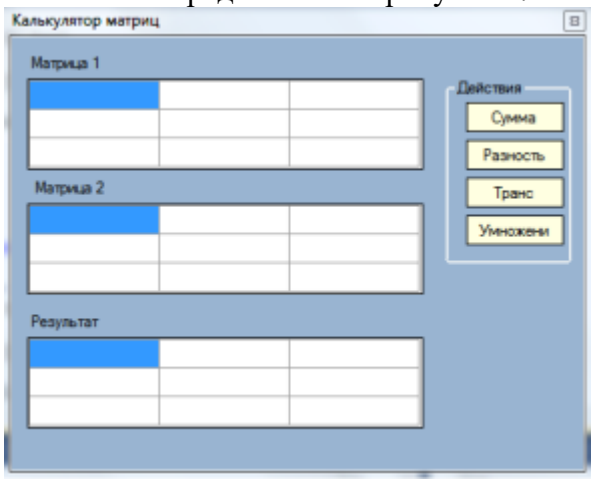


Рисунок 2.1 - Оконный интерфейс будущего приложения

На форме располагается 3 элемента DataGridView, в них будут размещаться матрицы. Так же 4 Button для выполнения действий над матрицами.

4. Использование инструментальных средств на этапе отладки программного модуля

При отладке программного продукта необходимо воспользоваться командой меню Отладка (рис. 3.1). В меню отладка существуют ряд команд, назначение которых представлено ниже.

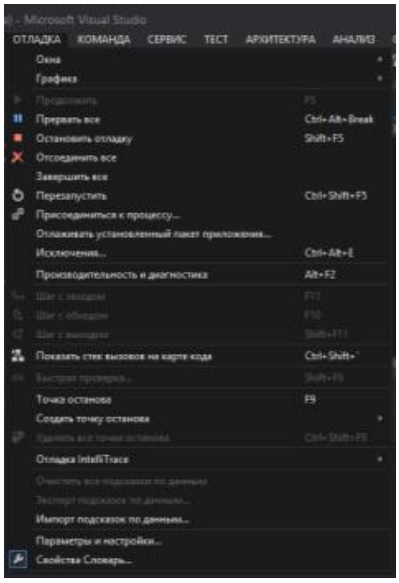


Рисунок 3.1 - Окно меню Отладка

5. Окна - открывает в интегрированной среде окно Точки останова, которое дает доступ ко всем точкам останова данного решения. Показывает в интегрированной среде окно Вывод.

6. Окно Вывод - это бегущий журнал множества сообщений, выдаваемых интегрированной средой, компилятором и отладчиком. Поэтому эта информация относится не только к сеансу отладки, а также открывает в интегрированной среде окно Интерпретация, которое позволяет выполнять команды:

- начать отладку - запускает приложение в режиме отладки;
- присоединиться к процессу - позволяет прикрепить отладчик к выполняющемуся процессу (исполняемому файлу). например, если запущено приложение без отладки, то можете потом прикрепиться к этому выполняющемуся процессу и начать отладку;
- исключения - открывает диалоговое окно Исключения, которое позволяет выбрать способ останова отладчика для каждого исключительного состояния;
- шаг с заходом - запускает приложение в режиме отладки. для большинства проектов выбор команды шаг с заходом означает вызов отладчика на первой выполняемой строке приложения. таким образом, можно войти в приложение с первой строки;
- шаг с обходом - когда вы не находитесь в сеансе отладки, то команда шаг с обходом просто запускает приложение точно так же, как это сделала бы кнопка run;
- точка останова - включает или выключает точку останова на текущей (активной) строке кода текстового редактора. эта опция неактивна, если в интегрированной среде нет активного кодового окна;
- создавать точку останова - активирует диалоговое окно создавать точку останова позволяющее указать имя функции, для которой необходимо создать точку останова;
- удалить все точки останова - удаляет все точки останова из текущего решения;
- очистить все подсказки по данным - деактивирует (без удаления) все точки останова текущего решения;
- параметры и настройки - Прерывать выполнение, когда исключения пересекают границу домена приложения или границу между управляемым и машинным кодом.

7. Проведение тестирования программного модуля по определенному сценарию

Оценочное тестирование, которое также называют «тестированием системы в целом» целью которого является тестирование программы на соответствие основным требованиям. Эта стадия тестирования особенно важна для программных продуктов. Включает следующие виды:

- тестирование удобства использования - последовательная проверка соответствия программного продукта и документации на него основным положениям технического задания;
- тестирование на предельных объемах - проверка работоспособности программы на максимально больших объемах данных, например, объемах текстов, таблиц, большом количестве файлов и т.п.;
- тестирование на предельных нагрузках - проверка выполнения программы на возможность обработки большого объема данных, поступивших в течение короткого времени;

- тестирование удобства эксплуатации - анализ психологических факторов, возникающих при работе с программным обеспечением; это тестирование позволяет определить, удобен ли интерфейс, не раздражает ли цветовое или звуковое сопровождение и т.п.;
- тестирование защиты - проверка защиты, например, от несанкционированного доступа к информации;
- тестирование производительности - определение пропускной способности при заданной конфигурации и нагрузке;
- тестирование требований к памяти - определение реальных потребностей в оперативной и внешней памяти;
- тестирование конфигурации оборудования - проверка работоспособности программного обеспечения на разном оборудовании;
- тестирование совместимости - проверка преемственности версий: в тех случаях, если очередная версия системы меняет форматы данных, она должна предусматривать специальные конвекторы, обеспечивающие возможность работы с файлами, созданными предыдущей версией системы;
- тестирование удобства установки - проверка удобства установки;
- тестирование надежности - проверка надежности с использованием математических моделей;
- тестирование восстановления - проверка восстановления программного обеспечения, например, системы, включающей базу данных, после сбоев оборудования и программы;
- тестирование удобства обслуживания - проверка средств обслуживания, включенных в программное обеспечение;
- тестирование документации - тщательная проверка документации, например, если документация содержит примеры, то их все необходимо попробовать;
- тестирование процедуры - проверка ручных процессов, предполагаемых в системе.

8. Естественно, целью всех этих проверок является поиск несоответствий техническому заданию. Считают, что только после выполнения всех видов тестирования программный продукт может быть представлен пользователю или к реализации. Однако на практике обычно выполняют не все виды оценочного тестирования, так как это очень дорого и трудоемко. Как правило, для каждого типа программного обеспечения выполняют те виды тестирования, которые являются для него наиболее важными. Так базы данных обязательно тестируют на предельных объемах, а системы реального времени - на предельных нагрузках.

Практическая работа № 1.6. Использование инструментальных средств на этапе отладки программного модуля

Цель работы: изучить инструментальные средства отлпдки программного модуля

Теоретический материал

Отладка ПС - это деятельность, направленная на обнаружение и исправление ошибок в ПС с использованием процессов выполнения его программ. *Тестирование* ПС - это процесс выполнения его программ на некотором наборе данных, для которого заранее известен результат применения или известны правила поведения этих программ. Указанный набор данных называется *тестовым* или просто *тестом*. Таким образом, отладку можно представить в виде многократного повторения трех процессов: тестирования, в результате которого может быть констатировано наличие в ПС ошибки, поиска места ошибки в программах и документации ПС и редактирования программ и документации с целью устранения обнаруженной ошибки. Другими словами:

Отладка = Тестирование + Поиск ошибок + Редактирование.

В зарубежной литературе отладку часто понимают [10.1-10.3] только как процесс поиска и исправления ошибок (без тестирования), факт наличия которых устанавливается при тестировании. Иногда тестирование и отладку считают синонимами [10.4,10.5]. В нашей стране в понятие отладки обычно включают и тестирование [10.6 -10.8], поэтому мы будем следовать сложившейся традиции. Впрочем, совместное рассмотрение в данной лекции этих процессов делает указанное разночтение не столь существенным. Следует, однако, отметить, что тестирование используется и как часть процесса аттестации ПС (см. лекцию 14).

Принципы и виды отладки программного средства

Успех отладки ПС в значительной степени предопределяет рациональная организация тестирования. При отладке ПС отыскиваются и устраняются, в основном, те ошибки, наличие которых в ПС устанавливается при тестировании. Как было уже отмечено, тестирование не может доказать правильность ПС [10.9], в лучшем случае оно может продемонстрировать наличие в нем ошибки. Другими словами, нельзя гарантировать, что тестированием ПС практически выполнимым набором тестов можно установить наличие каждой имеющейся в ПС ошибки. Поэтому возникает две задачи. Первая задача: подготовить такой набор тестов и применить к ним ПС, чтобы обнаружить в нем по возможности большее число ошибок. Однако чем дольше продолжается процесс тестирования (и отладки в целом), тем большей становится стоимость ПС. Отсюда вторая задача: определить момент окончания отладки ПС (или отдельной его компоненты). Признаком возможности окончания отладки является полнота охвата пропущенными через ПС тестами (т.е. тестами, к которым применено ПС) множества различных ситуаций, возникающих при выполнении программ ПС, и относительно редкое проявление ошибок в ПС на последнем отрезке процесса тестирования. Последнее определяется в соответствии с требуемой степенью надежности ПС, указанной в спецификации его качества.

Для оптимизации набора тестов, т.е. для подготовки такого набора тестов, который позволял бы при заданном их числе (или при заданном интервале времени, отведенном на тестирование) обнаруживать большее число ошибок в ПС, необходимо, во-первых, заранее планировать этот набор и, во-вторых, использовать рациональную стратегию планирования (проектирования [10.1]) тестов. Проектирование тестов можно начинать сразу же после завершения этапа внешнего описания ПС. Возможны разные подходы к выработке стратегии проектирования тестов, которые можно условно графически разместить (см. рис. 9.1) между следующими двумя крайними подходами [10.1]. Левый крайний подход заключается в том, что тесты проектируются только на основании изучения спецификаций ПС (внешнего описания, описания архитектуры и спецификации модулей). Строение модулей при этом никак не учитывается, т.е. они рассматриваются как черные ящики. Фактически такой подход требует полного перебора всех наборов входных данных, так как в противном случае некоторые участки программ ПС могут не работать при пропуске любого теста, а это значит, что содержащиеся в них ошибки не будут проявляться. Однако тестирование ПС полным множеством наборов входных данных практически неосуществимо. Правый крайний подход заключается в том, что тесты проектируются на основании изучения текстов программ с целью протестировать все пути выполнения каждой программ ПС. Если принять во внимание наличие в программах циклов с переменным числом повторений, то различных путей выполнения программ ПС может оказаться также чрезвычайно много, так что их тестирование также будет практически неосуществимо.

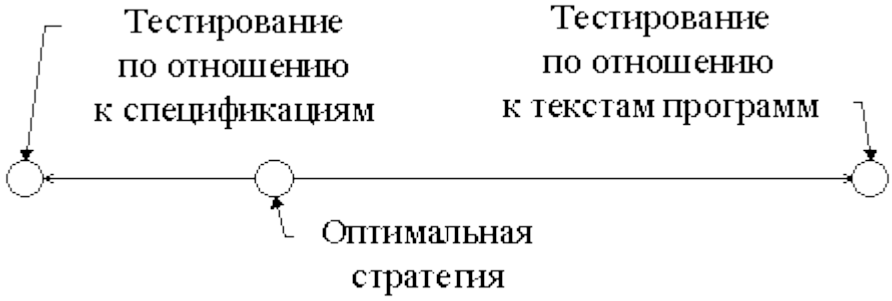


Рис. 10.1. Спектр подходов к проектированию тестов.

Оптимальная стратегия проектирования тестов расположена внутри интервала между этими крайними подходами, но ближе к левому краю. Она включает проектирование значительной части тестов по спецификациям, но она требует также проектирования некоторых тестов и по текстам программ. При этом в первом случае эта стратегия базируется на принципах:

- на каждую используемую функцию или возможность - хотя бы один тест,
- на каждую область и на каждую границу изменения какой-либо входной величины - хотя бы один тест,
- на каждую особую (исключительную) ситуацию, указанную в спецификациях, - хотя бы один тест.

Во втором случае эта стратегия базируется на принципе: каждая команда каждой программы ПС должна проработать хотя бы на одном тесте.

Оптимальную стратегию проектирования тестов можно конкретизировать на основании следующего принципа: для каждого программного документа (включая тексты программ),

входящего в состав ПС, должны проектироваться свои тесты с целью выявления в нем ошибок. Во всяком случае, этот принцип необходимо соблюдать в соответствии с определением ПС и содержанием понятия технологии программирования как технологии разработки надежных ПС (см. лекцию 1). В связи с этим Майерс даже определяет разные виды тестирования [10.1] в зависимости от вида программного документа, на основании которого строятся тесты. В нашей стране различаются [10.8] два основных вида отладки (включая тестирование): автономную и комплексную отладку ПС. *Автономная* отладка ПС означает последовательное раздельное тестирование различных частей программ, входящих в ПС, с поиском и исправлением в них фиксируемых при тестировании ошибок. Она фактически включает отладку каждого программного модуля и отладку сопряжения модулей. *Комплексная* отладка означает тестирование ПС в целом с поиском и исправлением фиксируемых при тестировании ошибок во всех документах (включая тексты программ ПС), относящихся к ПС в целом. К таким документам относятся определение требований к ПС, спецификация качества ПС, функциональная спецификация ПС, описание архитектуры ПС и тексты программ ПС.

Автономная отладка программного средства

При автономной отладке ПС каждый модуль на самом деле тестируется в некотором программном окружении, кроме случая, когда отлаживаемая программа состоит только из одного модуля. Это окружение состоит из других модулей, часть которых является модулями отлаживаемой программы, которые уже отлажены, а часть - модулями, управляющими отладкой (*отладочными* модулями, см. ниже). Таким образом, при автономной отладке тестируется всегда некоторая программа (*тестируемая программа*), построенная специально для тестирования отлаживаемого модуля. Эта программа лишь частично совпадает с отлаживаемой программой, кроме случая, когда отлаживается последний модуль отлаживаемой программы. В процессе автономной отладки ПС производится наращивание тестируемой программы отлаженными модулями: при переходе к отладке следующего модуля в его программное окружение добавляется последний отлаженный модуль. Такой процесс наращивания программного окружения отлаженными модулями называется *интеграцией* программы [10.1]. Отладочные модули, входящие в окружение отлаживаемого модуля, зависят от порядка, в каком отлаживаются модули этой программы, от того, какой модуль отлаживается и, возможно, от того, какой тест будет пропускаться.

При восходящем тестировании (см. лекцию 7) это окружение будет содержать только один отладочный модуль (кроме случая, когда отлаживается последний модуль отлаживаемой программы), который будет головным в тестируемой программе. Такой отладочный модуль называют *ведущим* (или драйвером [10.1]). Ведущий отладочный модуль подготавливает информационную среду для тестирования отлаживаемого модуля (т. е. формирует ее состояние, требуемое для тестирования этого модуля, в частности, путем ввода некоторых тестовых данных), осуществляет обращение к отлаживаемому модулю и после окончания его работы выдает необходимые сообщения. При отладке одного модуля для разных тестов могут составляться разные ведущие отладочные модули.

При нисходящем тестировании (см. лекцию 7) окружение отлаживаемого модуля в качестве отладочных модулей содержит *отладочные имитаторы* (заглушки) некоторых еще не отлаженных модулей. К таким модулям относятся, прежде всего, все модули, к которым может обращаться отлаживаемый модуль, а также еще не отлаженные модули, к которым могут обращаться уже отлаженные модули (включенные в это окружение). Некоторые из этих имитаторов при отладке одного модуля могут изменяться для разных тестов.

На практике в окружении отлаживаемого модуля могут содержаться отладочные модули обоих типов, если используется смешанная стратегия тестирования. Это связано с тем, что как восходящее, так и нисходящее тестирование имеет свои достоинства и свои недостатки.

К *достоинствам восходящего тестирования* относятся:

- простота подготовки тестов,
- возможность полной реализации плана тестирования модуля.

Это связано с тем, что тестовое состояние информационной среды готовится непосредственно перед обращением к отлаживаемому модулю (ведущим отладочным модулем).

Недостатками восходящего тестирования являются следующие его особенности:

- тестовые данные готовятся, как правило, не в той форме, которая рассчитана на пользователя (кроме случая, когда отлаживается последний, головной, модуль отлаживаемой программ);
- большой объем отладочного программирования (при отладке одного модуля приходится составлять много ведущих отладочных модулей, формирующих подходящее состояние информационной среды для разных тестов);
- необходимость специального тестирования сопряжения модулей.

К *достоинствам нисходящего тестирования* относятся следующие его особенности:

- большинство тестов готовится в форме, рассчитанной на пользователя;
- во многих случаях относительно небольшой объем отладочного программирования (имитаторы модулей, как правило, весьма просты и каждый пригоден для большого числа, нередко - для всех, тестов);
- отпадает необходимость тестирования сопряжения модулей.

Недостатком нисходящего тестирования является то, что тестовое состояние информационной среды перед обращением к отлаживаемому модулю готовится косвенно - оно является результатом применения уже отлаженных модулей к тестовым данным или данным, выдаваемым имитаторами. Это, во-первых, затрудняет подготовку тестов и требует высокой квалификации тестовика (разработчика тестов), а во-вторых, делает затруднительным или даже невозможным реализацию полного плана тестирования отлаживаемого модуля. Указанный недостаток иногда вынуждает разработчиков применять восходящее тестирование даже в случае нисходящей разработки. Однако чаще применяют некоторые модификации нисходящего тестирования, либо некоторую комбинацию нисходящего и восходящего тестирования. Исходя из того, что нисходящее тестирование, в принципе, является предпочтительным, остановимся на приемах, позволяющих в какой-то мере преодолеть указанные трудности.

Прежде всего, необходимо организовать отладку программы таким образом, чтобы как можно раньше были отлажены модули, осуществляющие ввод данных, - тогда тестовые данные можно готовить в форме, рассчитанной на пользователя, что существенно упростит подготовку последующих тестов. Далеко не всегда этот ввод осуществляется в головном модуле, поэтому приходится в первую очередь отлаживать цепочки модулей, ведущие к модулям, осуществляющим указанный ввод (ср. с методом целенаправленной конструктивной реализации в лекции 7). Пока модули, осуществляющие ввод данных, не отлажены, тестовые данные поставляются некоторыми имитаторами: они либо включаются в имитатор как его часть, либо вводятся этим имитатором.

При нисходящем тестировании некоторые состояния информационной среды, при которых требуется тестировать отлаживаемый модуль, могут не возникать при выполнении отлаживаемой программы ни при каких входных данных. В этих случаях можно было бы вообще не тестировать отлаживаемый модуль, так как обнаруживаемые при этом ошибки не будут проявляться при выполнении отлаживаемой программы ни при каких входных данных. Однако так поступать не рекомендуется, так как при изменениях отлаживаемой программы (например, при сопровождении ПС) не использованные для тестирования отлаживаемого модуля состояния информационной среды могут уже возникать, что требует дополнительного тестирования этого модуля (а этого при рациональной организации отладки можно было бы не делать, если сам данный модуль не изменялся). Для осуществления тестирования отлаживаемого модуля в указанных ситуациях иногда используют подходящие имитаторы, чтобы создать требуемое состояние информационной среды. Чаще же пользуются модифицированным вариантом нисходящего тестирования, при котором отлаживаемые модули перед их интеграцией предварительно тестируются отдельно (в этом случае в окружении отлаживаемого модуля появляется ведущий отладочный модуль, наряду с имитаторами модулей, к которым может обращаться отлаживаемый модуль). Однако, представляется более целесообразной другая модификация нисходящего тестирования: после завершения нисходящего тестирования отлаживаемого модуля для достижимых тестовых состояний информационной среды следует его отдельно протестировать для остальных требуемых состояний информационной среды.

Часто применяют также комбинацию восходящего и нисходящего тестирования, которую называют методом *сандвича* [10.1]. Сущность этого метода заключается в одновременном осуществлении как восходящего, так и нисходящего тестирования, пока эти два процесса тестирования не встретятся на каком-либо модуле где-то в середине структуры отлаживаемой

программы. Этот метод при разумном порядке тестирования позволяет воспользоваться достоинствами как восходящего, так и нисходящего тестирования, а также в значительной степени нейтрализовать их недостатки.

Весьма важным при автономной отладке является тестирование

сопряжения модулей. Дело в том, что спецификация каждого модуля программы, кроме головного, используется в этой программы в двух ситуациях: во-первых, при разработке текста (иногда говорят: тела) этого модуля и, во-вторых, при написании обращения к этому модулю в других модулях программы. И в том, и в другом случае в результате ошибки может быть нарушено требуемое соответствие заданной спецификации модуля. Такие ошибки требуется обнаруживать и устранять. Для этого и предназначено тестирование сопряжения модулей. При нисходящем тестировании тестирование сопряжения осуществляется попутно каждым пропускаемым тестом, что считают достоинством нисходящего тестирования. При восходящем тестировании обращение к отлаживаемому модулю производится не из модулей отлаживаемой программы, а из ведущего отладочного модуля. В связи с этим существует опасность, что последний модуль может приспособиться к некоторым "заблуждениям" отлаживаемого модуля. Поэтому, приступая (в процессе интеграции программы) к отладке нового модуля, приходится тестировать каждое обращение к ранее отлаженному модулю с целью обнаружения несогласованности этого обращения с телом соответствующего модуля (и не исключено, что виноват в этом ранее отлаженный модуль). Таким образом, приходится частично повторять в новых условиях тестирование ранее отлаженного модуля, при этом возникают те же трудности, что и при нисходящем тестировании.

Ход работы

Автономное тестирование модуля целесообразно осуществлять в четыре последовательно выполняемых шага [10.1].

1. На основании спецификации отлаживаемого модуля подготовьте тесты для каждой возможности и каждой ситуации, для каждой границы областей допустимых значений всех входных данных, для каждой области изменения данных, для каждой области недопустимых значений всех входных данных и каждого недопустимого условия.
2. Проверьте текст модуля, чтобы убедиться, что каждое направление любого разветвления будет пройдено хотя бы на одном тесте. Добавьте недостающие тесты.
3. Проверьте текст модуля, чтобы убедиться, что для каждого цикла существуют тесты, обеспечивающие, по крайней мере, три следующие ситуации: тело цикла не выполняется ни разу, тело цикла выполняется один раз и тело цикла выполняется максимальное число раз. Добавьте недостающие тесты.
4. Проверьте текст модуля, чтобы убедиться, что существуют тесты, проверяющие чувствительность к отдельным особым значениям входных данных. Добавьте недостающие тесты.

Комплексная отладка программного средства.

Как уже было сказано выше, при комплексной отладке тестируется ПС в целом, причем тесты готовятся по каждому из документов ПС [10.8]. Тестирование этих документов производится, как правило, в порядке, обратном их разработке. Исключение составляет лишь тестирование документации по применению, которая разрабатывается по внешнему описанию параллельно с разработкой текстов программ - это тестирование лучше производить после завершения тестирования внешнего описания. Тестирование при комплексной отладке представляет собой применение ПС к конкретным данным, которые в принципе могут возникнуть у пользователя (в частности, все тесты готовятся в форме, рассчитанной на пользователя), но, возможно, в моделируемой (а не в реальной) среде. Например, некоторые недоступные при комплексной отладке устройства ввода и вывода могут быть заменены их программными имитаторами.

1. Тестирование архитектуры ПС. Целью тестирования является поиск несоответствия между описанием архитектуры и совокупностью программ ПС. К моменту начала тестирования архитектуры ПС должна быть уже закончена автономная отладка каждой подсистемы. Ошибки реализации архитектуры могут быть связаны, прежде всего, с взаимодействием этих подсистем, в частности, с реализацией архитектурных функций (если они есть). Поэтому хотелось бы проверить все пути взаимодействия между подсистемами ПС. При этом желательно хотя бы протестировать все цепочки выполнения подсистем без повторного вхождения последних. Если

заданная архитектура представляет ПС в качестве малой системы из выделенных подсистем, то число таких цепочек будет вполне обозримо.

2. Тестирование внешних функций. Целью тестирования является поиск расхождений между функциональной спецификацией и совокупностью программ ПС. Несмотря на то, что все эти программы автономно уже отлажены, указанные расхождения могут быть, например, из-за несоответствия внутренних спецификаций программ и их модулей (на основании которых производилось автономное тестирование) функциональной спецификации ПС. Как правило, тестирование внешних функций производится так же, как и тестирование модулей на первом шаге, т.е. как черного ящика.

3. Тестирование качества ПС. Целью тестирования является поиск нарушений требований качества, сформулированных в спецификации качества ПС. Это наиболее трудный и наименее изученный вид тестирования. Ясно лишь, что далеко не каждый примитив качества ПС может быть испытан тестированием (об оценке качества ПС см. лекцию 14). Завершенность ПС проверяется уже при тестировании внешних функций. На данном этапе тестирование этого примитива качества может быть продолжено, если требуется получить какую-либо вероятностную оценку степени надежности ПС. Однако, методика такого тестирования еще требует своей разработки. Могут тестироваться такие примитивы качества, как точность, устойчивость, защищенность, временная эффективность, в какой-то мере - эффективность по памяти, эффективность по устройствам, расширяемость и, частично, независимость от устройств. Каждый из этих видов тестирования имеет свою специфику и заслуживает отдельного рассмотрения. Мы здесь ограничимся лишь их перечислением. Легкость применения ПС (критерий качества, включающий несколько примитивов качества, см. лекцию 4) оценивается при тестировании документации по применению ПС.

4. Тестирование документации по применению ПС. Целью тестирования является поиск несогласованности документации по применению и совокупностью программ ПС, а также выявление неудобств, возникающих при применении ПС. Этот этап непосредственно предшествует подключению пользователя к завершению разработки ПС (тестированию определения требований к ПС и аттестации ПС), поэтому весьма важно разработчикам сначала самим воспользоваться ПС так, как это будет делать пользователь [10.1]. Все тесты на этом этапе готовятся исключительно на основании только документации по применению ПС. Прежде всего, должны тестироваться возможности ПС как это делалось при тестировании внешних функций, но только на основании документации по применению. Должны быть протестированы все неясные места в документации, а также все примеры, использованные в документации. Далее тестируются наиболее трудные случаи применения ПС с целью обнаружить нарушение требований относительности легкости применения ПС.

5. Тестирование определения требований к ПС. Целью тестирования является выяснение, в какой мере ПС не соответствует предъявленному определению требований к нему. Особенность этого вида тестирования заключается в том, что его осуществляет организация-покупатель или организация-пользователь ПС [10.1] как один из путей преодоления барьера между разработчиком и пользователем (см. лекцию 3). Обычно это тестирование производится с помощью контрольных задач - типовых задач, для которых известен результат решения. В тех случаях, когда разрабатываемое ПС должно придти на смену другой версии ПС, которая решает хотя бы часть задач разрабатываемого ПС, тестирование производится путем решения общих задач с помощью как старого, так и нового ПС (с последующим сопоставлением полученных результатов). Иногда в качестве формы такого тестирования используют *опытную* эксплуатацию ПС - ограниченное применение нового ПС с анализом использования результатов в практической деятельности. По существу, этот вид тестирования во многом перекликается с испытанием ПС при его аттестации (см. лекцию 14), но выполняется до аттестации, а иногда и вместо аттестации.

Практическая работа № 1.7. Оценка сложности алгоритмов сортировки

Цель работы: получение навыков определения сложности алгоритмов сортировки.

Теоретический материал

Алгоритм сортировки

Алгоритм сортировки — это алгоритм для упорядочения элементов в списке. В случае, когда элемент списка имеет несколько полей, поле, служащее критерием порядка, называется ключом сортировки. На практике в качестве ключа часто выступает число, а в остальных полях хранятся какие-либо данные, никак не влияющие на работу алгоритма.

Оценка алгоритма сортировки

Алгоритмы сортировки оцениваются по скорости выполнения и эффективности использования памяти:

- **Время** — основной параметр, характеризующий быстродействие алгоритма. Называется также вычислительной сложностью. Для упорядочения важны *худшее*, *среднее* и *лучшее* поведение алгоритма в терминах мощности входного множества *A*. Если на вход алгоритму подаётся множество *A*, то обозначим $n = |A|$. Для типичного алгоритма хорошее поведение — это [1] и плохое поведение — это . Идеальное поведение для упорядочения — . Алгоритмы сортировки, использующие только абстрактную операцию сравнения ключей всегда нуждаются по меньшей мере в сравнениях. Тем не менее, существует алгоритм сортировки Хана (Yijie Han) с вычислительной сложностью , использующий тот факт, что пространство ключей ограничено (он чрезвычайно сложен, а за O-обозначением скрывается весьма большой коэффициент, что делает невозможным его применение в повседневной практике). Также существует понятие сортирующих сетей. Предполагая, что можно одновременно (например, при параллельном вычислении) проводить несколько сравнений, можно отсортировать *n* чисел за операций. При этом число *n* должно быть заранее известно;
- **Память** — ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. Как правило, эти алгоритмы требуют памяти. При оценке не учитывается место, которое занимает исходный массив и независящие от входной последовательности затраты, например, на хранение кода программы (так как всё это потребляет). Алгоритмы сортировки, не потребляющие дополнительной памяти, относят к *сортировкам на месте*.

Классификация алгоритмов сортировки

- **Устойчивость** (stability) — устойчивая сортировка не меняет взаимного расположения равных элементов.
- **Естественность поведения** — эффективность метода при обработке уже упорядоченных, или частично упорядоченных данных. Алгоритм ведёт себя естественно, если учитывает эту характеристику входной последовательности и работает лучше.
- **Использование операции сравнения.** Алгоритмы, использующие для сортировки сравнение элементов между собой, называются основанными на сравнениях. Минимальная трудоемкость *худшего случая* для этих алгоритмов составляет , но они отличаются гибкостью применения. Для специальных случаев (типов данных) существуют более эффективные алгоритмы.

Ещё одним важным свойством алгоритма является его сфера применения. Здесь основных типов упорядочения два:

- **Внутренняя сортировка** оперирует с массивами, целиком помещающимися в оперативной памяти с произвольным доступом к любой ячейке. Данные обычно упорядочиваются на том же месте, без дополнительных затрат.
  - В современных архитектурах персональных компьютеров широко применяется подкачка и кэширование памяти. Алгоритм сортировки должен хорошо сочетаться с применяемыми алгоритмами кэширования и подкачки.
- **Внешняя сортировка** оперирует с запоминающими устройствами большого объёма, но с доступом не произвольным, а последовательным (упорядочение файлов), т. е. в данный момент мы 'видим' только один элемент, а затраты на перемотку по сравнению с памятью неоправданно велики. Это накладывает некоторые дополнительные ограничения на алгоритм и приводит к специальным методам упорядочения, обычно использующим дополнительное дисковое пространство. Кроме того, доступ к данным на носителе производится намного медленнее, чем операции с оперативной памятью.
  - Доступ к носителю осуществляется последовательным образом: в каждый момент времени можно считать или записать только элемент, следующий за текущим.
  - Объём данных не позволяет им разместиться в ОЗУ.

Также алгоритмы классифицируются по:

- потребности в дополнительной памяти или её отсутствии
- потребности в знаниях о структуре данных, выходящих за рамки операции сравнения, или отсутствии таковой

Задание № 1

Напишите программу, которая выполняет следующие функции:

– Генерацию элементов массива с заданной размерностью;

– Сортировку массива каждым из 6 способов (пузырьковая сортировка, сортировка вставкой, сортировка выбором, быстрая сортировка, сортировка слиянием, шейкерная сортировка);

– Осуществляет подсчет времени сортировки и количества перестановок.

Задание № 2

Провести эксперимент сортировки массива со следующим количеством элементов: 1'000, 10'000 и 100'000. Для проведения эксперимента необходимо произвести по 5 запусков каждого алгоритма и выбрать наилучшее время. Выбранный наилучший результат занести в таблицу. Сортировку осуществлять с одним и тем же набором данных.

***Таблица 1**. Результаты эксперимента.*

Вид сортировки	**1'000**		**10'000**		**100'000**	
	Время	**Количество перестановок**	**Время**	**Количество перестановок**	**Время**	**Количество перестановок**
Bubble sort	0.003	76564	0.32	967782	33.389	9887983
Selection sort	0.002	990	0.256	9950	34.215	99509
Insertion sort	0.003	251411	0.375	25021276	40.468	1800935483
Quick sort	0.000	2733	0.001	41167	0.017	574566
Merge sort	0.001	9976	0.016	133616	0.237	1668928
Shaker sort	0.005	239124	0.76	24972810	82.974	1802548963

Задание № 3

Оцените сложность каждого алгоритма и сделайте вывод о наилучшем способе сортировки массива по каждой размерности массива. Свой ответ поясните.

С начала хотелось бы кратко описать каждый из представленных алгоритмов:

Сортировка пузырьком (от англ. Bubble sort) – данный алгоритм меняет местами два соседних элемента, если первый элемент массива больше второго. Так происходит до тех пор, пока алгоритм не обменяет местами все неотсортированные элементы.

Сложность настоящего алгоритма равно $O(n^2)$.

Сортировка выбором (от англ. Selection sort) – суть алгоритма заключается в проходе по массиву от начала до конца в поиске наименьшего элемента массива и перемещении его в начало.

Сложность настоящего алгоритма равно $O(n^2)$.

Сортировка вставками (от англ. Insertion sort) – алгоритм сортирует массив по мере прохождения по его элементам. На каждой итерации берется элемент и сравнивается с каждым элементом в уже отсортированной части массива, таким образом находя «свое место», после чего элемент вставляется на свою позицию. Так происходит до тех пор, пока алгоритм не пройдет по всему массиву. На выходе получается отсортированный массив.

Сложность настоящего алгоритма равно $O(n^2)$.

Быстрая сортировка (от англ. Quick sort) – суть алгоритма заключается в разделении массива на два под-массива, средней линией считается элемент, который находится в самом центре массива. В ходе работы алгоритма элементы, меньшие чем средний будут перемещены в лево, а большие в право. Такое же действие будет происходить рекурсивно и с под-массива, они будут разделяться на еще два под-массива до тех пор, пока не будет чего разделать (останется один элемент). На выходе получается отсортированный массив.

Среднее значение настоящего алгоритма равно $O(n \times \log n)$.

Сортировка слиянием (от англ. Merge sort) – этот алгоритм упорядочивает списки (или другие структуры данных, доступ к элементам, которых можно получать только последовательно) в определенном порядке. Слияние означает объединение двух и более

последовательностей в одну упорядоченную последовательность при помощи циклического выбора элементов, доступных в данный момент.

Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Затем их решения комбинируются, в результате чего массив упорядочивается.

Среднее значение настоящего алгоритма равно $O(n \times \log n)$.

Шейкерная сортировка (от англ. Cocktail sort) – она же сортировка перемешиванием, она же двунаправленная сортировка – по сути всего лишь оптимизированный алгоритм сортировки пузырьком (см. выше).

В ее основе также лежит сравнение двух соседних элементов. Единственное отличие состоит в том, что теперь это происходит в двух направлениях поочередно, постепенно сужая диапазон сортировки. В итоге за один проход в конец массива «всплывает» наибольший элемент из диапазона, а за следующий проход – в начало массива наименьший (при сортировке по возрастанию). Эти элементы можно больше не рассматривать и таким образом диапазон сужается с двух сторон.

Шейкерная сортировка работает быстрее, чем сортировка пузырьком, но все еще имеет сложность $O(n^2)$.

Оценить сложность алгоритмов нагляднее всего можно в виде таблицы и графика (для сравнения в ней представлены также некоторые другие виды сортировок):

***Таблица 2**. Сравнение временной сложности алгоритмов сортировки.*

Алгоритм	Структура данных	Временная сложность			Вспомогательные данные
		Лучшее	В среднем	В худшем	В худшем
Быстрая сортировка	Массив	O(n log(n))	O(n log(n))	O(n^2)	O(n)
Сортировка слиянием	Массив	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Пирамидальная сортировка	Массив	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Пузырьковая сортировка	Массив	O(n)	O(n^2)	O(n^2)	O(1)
Сортировка вставками	Массив	O(n)	O(n^2)	O(n^2)	O(1)
Сортировка выбором	Массив	O(n^2)	O(n^2)	O(n^2)	O(1)
Блочная сортировка	Массив	O(n+k)	O(n+k)	O(n^2)	O(nk)
Поразрядная сортировка	Массив	O(nk)	O(nk)	O(nk)	O(n+k)

Хорошо	Приемлемо	Плохо

Выводы: в результате проведенного эксперимента выяснилось, что наиболее эффективным из представленных алгоритмов является быстрая сортировка, которая полностью оправдывает свое название. Имея незначительную разницу во времени при упорядочивании небольших массивов (до 1'000 элементов), она проявляет весь свой потенциал при работе с большими массивами данных (свыше 10'000 элементов), что является ключевым преимуществом при работе в крупных проектах.

Сортировка слиянием при упорядочивании небольших массивов (до 1'000 элементов) демонстрирует хорошие показатели, однако при сортировке массивов среднего и большого объема (свыше 10'000 элементов) она показывает неудовлетворительные результаты.

Сортировки пузырьком, выбором и вставками имеют схожие алгоритмы работы и показывают неудовлетворительные результаты с массивами любой разрядности.

Самым медленным способом сортировки является шейкерная. Она показывает худшие результаты в массивах любой размерности и к работе с коммерческими проектами категорически не допускается.

Задание

Оформите отчет, который должен содержать:

– Описание алгоритмов сортировки в виде блок-схемы и ее описанием;

– Реализацию этих алгоритмов на языке программирования C++;

– Описание компьютера на котором проводился эксперимент (вид процессора, тактовая частота, количество ядер, количество оперативной памяти и т. д.);

– Результаты эксперимента в виде таблицы;

– Вывод по каждой размерности массива с указанием лучшей сортировки и пояснениями.

Практическая работа № 1.8. Оценка сложности алгоритмов поиска

Цель работы: Получение навыков построения алгоритмов с использованием пузырьковой сортировки.

Теоретический материал

Сортировка применяется во всех без исключения областях программирования, будь то базы данных или математические программы.

Практически каждый алгоритм сортировки можно разбить на три части:

сравнение, определяющее упорядоченность пары элементов;

перестановку, меняющую местами пару элементов;

собственно сортирующий алгоритм, который осуществляет сравнение и перестановку элементов до тех пор, сока все элементы множества не будут упорядочены. **Пузырьковая сортировка**

Самый простой алгоритм этой группы получил название пузырьковой сортировки. Не самый удачный алгоритм. Заключается в сравнении соседних элементов и, при необходимости их перестановке. При неубывающем упорядочении элементы "выплывают" как пузырьки каждый до своего уровня.

Ход работы

Описание алгоритма:

Используется два цикла. Внешний проходит N-1 раз, это гарантирует, что даже в худшем случае каждый элемент будет находиться на своем месте.

Исходный массив d, c, a, b.

В процессе работы программы массив будет изменяться следующим образом:

Расположим цифровой массив 4, 9, 7, 6, 2, 3 сверху вниз, от нулевого элемента - к последнему.

Идея метода: шаг сортировки состоит в проходе снизу вверх по массиву. По пути просматриваются пары соседних элементов. Если элементы некоторой пары находятся в неправильном порядке, то меняем их местами.

После нулевого прохода по массиву "вверху" оказывается самый "легкий" элемент - отсюда аналогия с пузырьком. Следующий проход делается до второго сверху элемента, таким образом второй по величине элемент поднимается на правильную позицию...

Делаем проходы по все **уменьшающейся нижней части массива до тех пор,** пока в ней не останется только один элемент. На этом сортировка заканчивается, так как последовательность упорядочена по возрастанию.

Массив представляет собой упорядоченную структуру однотипных данных, которые называются элементами массива.

Доступ к каждому элементу массива осуществляется с помощью индекса – в общем случае порядкового номера элемента в массиве.

Массивы могут быть как одномерными (адрес каждого элемента определяется значением одного индекса), так и многомерными (адрес каждого элемента определяется значением нескольких индексов).

Массивы занимают смежные ячейки памяти.

(Другими словами, элементы массива в памяти расположены последовательно друг за другом.) Ячейка с наименьшим адресом относится к первому элементу массива, а с наибольшим — к последнему. Предположим, мы используем массив a из семи элементов. После заполнения массив будет выглядеть следующим образом.

a[0] a[1] a[2] a[3] a[4] a[5] a[6]

5 1 2 4 6 3 9

Для работы с одномерными массивами применяются циклические алгоритмы.

Алгоритм, предусматривающий многократное повторение одного и того же действия над новыми данными, называется циклическим.

Тело цикла - это повторяющаяся последовательность действий (блок инструкций).

Цикл называется ***арифметическим***, если число повторений цикла известно заранее или может быть вычислено.

Цикл называется арифметическим, если число повторений цикла известно заранее или может быть вычислено.

Выражение 1 выполняется только один раз в начале цикла. Обычно оно определяет начальное значение параметра цикла (инициализирует параметр цикла). Выражение 2 — это условие выполнения цикла. Выражение 3 обычно определяет изменение параметра цикла. Блок инструкций — тело цикла, то есть инструкции, которые должны выполняться заданное количество раз..

Обратите внимание на то, что после вычисления выражения 3 происходит возврат к вычислению выражения 2 — проверке условия повторения цикла.

Пример

Заполнить массив десятью значениями 30,53,11,35,17,42,21,84,75,61. Отсортировать данную последовательность методом выбора.

Чтобы не загромождать блок-схему, вывод исходного и отсортированного массивов обозначен одним блоком «вывод» (надо помнить, что для вывода на экран массива используется цикл с параметром).

Результат работы алгоритма:

Исходный массив:

30 53 11 35 17 42 21 84 75 61

11 30 53 17 35 21 42 61 84 75

11 17 30 53 21 35 42 61 75 84

11 17 21 30 53 35 42 61 75 84

11 17 21 30 35 53 42 61 75 84

11 17 21 30 35 42 53 61 75 84

11 17 21 30 35 42 53 61 75 84

11 17 21 30 35 42 53 61 75 84

11 17 21 30 35 42 53 61 75 84

11 17 21 30 35 42 53 61 75 84

**********Отсортированный массив

11 17 21 30 35 42 53 61 75 84

Задание

Составить блок-схему алгоритма решения задачи.

1) Изменить процедуру сортировки так, чтобы сортировка производилась по убыванию элементов.

2) Проверить, является ли данная последовательность целых чисел упорядоченной по убыванию. Если нет, упорядочить ее.

3) Отсортировать четные элементы массива с помощью пузырьковой сортировки.

Дополнительные задания

1) Составьте алгоритм, упорядочивающий элементы массива, стоящие на нечетных местах, в

возрастающем порядке, а на четных - в убывающем.
2) В неупорядоченном массиве могут быть совпадающие элементы. Из каждой группы одинаковых элементов оставить только один, удалив остальные и «поджав» массив к его началу.
3) В массиве X(N) каждый элемент равен 0, 1 или 2. Переставить элементы массива так, чтобы сначала располагались все единицы, затем все двойки и, наконец, все нули (дополнительного массива не заводить).

Практическая работа № 1.9. Оценка сложности рекурсивных алгоритмов

Цель работы: получение навыков построения алгоритмов с использованием рекурсии.

Теоретический материал

При создании программы для решения сложной задачи программист обычно разбивает ее на подзадачи, чтобы для каждой такой подзадачи написать подпрограмму, а затем объединить все написанные подпрограммы в программу. Если подзадача достаточно сложная, то для решения может оказаться полезным разбить ее еще на более мелкие подзадачи и программировать каждую из них отдельно и т. д.

Блок-схема предопределенного процесса – обращение к подзадаче (в С++ к функции):

Рассмотрим задачу вычисления факториала числа N! = 1.2.3. . .N. Результатом будет одно число, поэтому лучше алгоритм оформить в виде функции.

Ее блок-схема показана на рисунке. Переменная К используется для накопления произведения и, поскольку 0! = 1 и1! = 1, то в блоке 2 ей сразу присваивается значение 1. Далее, если N>1, то в цикле, образованном блоками 4-5, накапливается искомое произведение и помещается в переменную К. В блоке 6 имя Fact функции получает значение вычисленного произведения из ячейки К. Для процедур действия, размещенного в блоке 6, не может быть, а для функций оно должно быть обязательно, поскольку иначе значение функции на выходе окажется неопределенным.

Обращение к функции в других алгоритмах (головных, процедурах, функциях) производится по ее имени.

При этом оно может входить в состав выражений. В качестве фактических параметров могут быть использованы как переменные, константы, так и целые выражения. Важно только, чтобы фактический параметр был совместим по типу с формальным, который содержится в заголовке описания алгоритма.

Пример использования функции Fact показан на рисунке. В операторе присваивания используется обращение к функции для N = 6. После передачи этого значения в алгоритм и вычислений внутри него результат будет сначала присвоен имени функции, т. е. переменной Fact, а затем в операторе присваивания - переменной L.

Часто бывает, что в процессе такого разбиения задача сводится к самой себе. Если при этом исходные данные становятся проще, то этот процесс можно продолжать до тех пор, пока исходные данные не окажутся настолько простыми, что решение задачи для них станет тривиальным.

Процедура или функция может содержать вызов других процедур или функций. В том числе процедура может вызвать саму себя. Никакого парадокса здесь нет – компьютер лишь последовательно выполняет встретившиеся ему в программе команды и, если встречается вызов процедуры, просто начинает выполнять эту процедуру. Без разницы, какая процедура дала команду это делать.

```
void Rec(int a){

if (a>0)Rec(a-1);

cout<<"\n a="<
}
```

Рассмотрим, что произойдет, если в основной программе поставить вызов, например, вида Rec(3).

```
void main(){clrscr();

int x;

cout<<"\nx=";cin>>x;

Rec(x);

getch(); }
```

Ниже представлена блок-схема, показывающая последовательность выполнения операторов.

Результат работы программы:

```
/*

x=3

a=0

a=1

a=2

a=3

*/
```

Процедура Rec вызывается с параметром a = 3. В ней содержится вызов процедуры Rec с параметром a = 2. Предыдущий вызов еще не завершился, поэтому можете представить себе, что создается еще одна процедура и до окончания ее работы первая свою работу не заканчивает. Процесс вызова заканчивается, когда параметр a = 0. В этот момент одновременно выполняются 4 экземпляра процедуры. Количество одновременно выполняемых процедур называют глубиной рекурсии.

Четвертая вызванная процедура (Rec(0)) напечатает число 0 и закончит свою работу. После этого управление возвращается к процедуре, которая ее вызвала (Rec(1)) и печатается число 1. И так далее пока не завершатся все процедуры. Результатом исходного вызова будет печать четырех чисел: 0, 1, 2, 3.

Еще один визуальный образ происходящего представлен на рисунке.

Выполнение процедуры Rec с параметром 3 состоит из выполнения процедуры Rec с параметром 2 и печати числа 3. В свою очередь выполнение процедуры Rec с параметром 2 состоит из выполнения процедуры Rec с параметром 1 и печати числа 2. И т. д.

Пример

Опишем рекурсивную функцию вычисления факториала. В функции один параметр — натуральное число п. Тривиальный случай — это вычисление значения 1!. Заметим, что верно следующее соотношение п! = п x (п — 1)!, поэтому можно свести задачу вычисления п! к решению той же задачи, но с другим, более "простым" параметром.

```
//rec03

//рекурсия вычисление факториала
```

```
#include

#include

long fact(int n) {

int f;

if (n==1) f=1;

else

f=n*fact(n-1);

return f;

}
void main(){

clrscr(); int a;

cout<<"a="; cin>>a;

cout<<"\n Factorial="<
getch();

}

/*

a=7

Factorial=5040

*/
```

Пусть n=3. Как будет выполняться вызов fact(n) ? На рисунке представлена схема выполнения вызова функции. Стрелки указывают порядок вычисления. Сначала происходит движение по стрелкам вниз, указывающее на то, что происходит временное прерывание выполнения текущего вызова, и переход к выполнению вызова более низкого уровня, пока не будет получен тривиальный случай. Затем происходит подъем вверх, что означает возобновление прерванных вызовов. Первым возобновится выполнение такого вызова, который был прерван последним.

Схема выполнения вызова рекурсивной функции

Задание 1

1. Составить алгоритм вычисления НОД двух чисел с помощью рекурсивной функции.

2. Составить алгоритм вычисления значения очередного числа Фибоначчи с помощью рекурсивной функции.

3. Напишите рекурсивную функцию, переворачивающую заданное натуральное число.

Задание 2

1. Напишите рекурсивную функцию, которая по заданным натуральным числам m и п выводит все различные представления числа n в виде суммы m натуральных слагаемых. Представления, различающиеся лишь порядком слагаемых, считаются одинаковыми. Напишите рекурсивную функцию, вычисляющую n!!.

2. Напишите рекурсивную функцию, определяющую количество единиц в двоичном представлении натурального числа.

3. Написать функцию сложения двух чисел, используя только прибавление единицы.
4. Написать функцию умножения двух чисел, используя только операцию сложения.
5. Проверить, является ли фрагмент строки с i-го по j-й символ палиндромом.
6. Подсчитать количество цифр в заданном числе.

Практическая работа № 1.10. Оценка сложности эвристических алгоритмов

Цель работы: получение навыков построения алгоритмов с использованием эвристических алгоритмов

Теоретический материал

Эвристические методы

Под эвристическими понимаются такие методы, правильность которых строго не доказывается. Они выглядят правдоподобными; кажется, что в большинстве случаев они должны давать верные решения. На уровне экспертной оценки алгоритма часто не удается придумать контрпример, доказывающий ошибочность или неуниверсальность метода. Это, разумеется, не является строгим обоснованием правильности метода. Тем не менее практика использования эвристических методов дает положительные результаты.

Эвристические методы разнообразны, поэтому нельзя описать какую-то общую схему их разработки. Чаще всего они применяются совместно с методами перебора для сокращения числа проверяемых вариантов. Некоторые варианты согласно выбранной эвристике считаются заведомо бесперспективными и не проверяются. Такой подход ускоряет работу алгоритма по сравнению с полным перебором. Платой за это является отсутствие гарантии того, что выбрано правильное или наилучшее из всех возможных решение.

Оценка сложности эвристических алгоритмов

Традиционно принято оценивать степень сложности алгоритма по объему используемых им основных ресурсов компьютера: процессорного времени и оперативной памяти. В связи с этим вводятся такие понятия, как временная сложность алгоритма и объемная сложность алгоритма.

Параметр временной сложности становится особенно важным для задач, предусматривающих интерактивный режим работы программы, или для задач управления в режиме реального времени. Часто программисту, составляющему программу управления каким-нибудь техническим устройством, приходится искать компромисс между точностью вычислений и временем работы программы. Как правило, повышение точности ведет к увеличению времени.

Объемная сложность программы становится критической, когда объем обрабатываемых данных оказывается на пределе объема оперативной памяти ЭВМ. На современных компьютерах острота этой проблемы снижается благодаря как росту объема ОЗУ, так и эффективному использованию многоуровневой системы запоминающих устройств. Программе оказывается доступной очень большая, практически неограниченная область памяти (виртуальная память). Недостаток основной памяти приводит лишь к некоторому замедлению работы из-за обменов с диском. Используются приемы, позволяющие минимизировать потери времени при таком обмене. Это использование кэш-памяти и аппаратного просмотра команд программы на требуемое число ходов вперед, что позволяет заблаговременно переносить с диска в основную память нужные значения. Исходя из сказанного можно заключить, что минимизация емкостной сложности не является первоочередной задачей. Поэтому в дальнейшем мы будем интересоваться в основном временной сложностью алгоритмов.

Время выполнения программы пропорционально числу исполняемых операций. Разумеется, в размерных единицах времени (секундах) оно зависит еще и от скорости работы процессора (тактовой частоты). Для того чтобы показатель временной сложности алгоритма был инвариантен относительно технических характеристик компьютера, его измеряют в относительных единицах. Обычно временная сложность оценивается числом выполняемых операций.

Как правило, временная сложность алгоритма зависит от исходных данных. Это может быть зависимость как от величины исходных данных, так и от их объема. Если обозначить значение параметра временной сложности алгоритма α

символом Tα, а буквой V обозначить некоторый числовой параметр, характеризующий исходные данные, то временную сложность можно представить как функцию Tα(V). Выбор параметра V зависит от решаемой задачи или от вида используемого алгоритма для решения данной задачи.

Ход работы

1. Оценим временную сложность алгоритма вычисления факториала целого положительного числа.

```
Function Factorial(x:Integer): Integer;
Var m,i: Integer;
Begin m:=l;
For i:=2 To x Do m:=ro*i;
Factorial:=m
End;
```

2. Подсчитаем общее число операций, выполняемых программой при данном значении x. Один раз выполняется оператор m:=1; тело цикла (в котором две операции: умножение и присваивание) выполняется x — 1 раз; один раз выполняется присваивание Factorial:=m. Если каждую из операций принять за единицу сложности, то временная сложность всего алгоритма будет 1 + 2 (x — 1) + 1 = 2x Отсюда понятно, что в качестве параметра следует принять значение x. Функция временной сложности получилась следующей:

Tα(V)=2V.

В этом случае можно сказать, что временная сложность зависит линейно от параметра данных — величины аргумента функции факториал.

3. Вычисление скалярного произведения двух векторов A = (a1, a2, …, ak), B = (b1, b2, …, bk).

```
AB:=0;
For i:=l To k Do AB:=AB+A[i]*B[i];
```

В этой задаче объем входных данных п = 2k. Количество выполняемых операций 1 + 3k = 1 + 3(n/2). Здесь можно взять V= k= п/2. Зависимости сложности алгоритма от значений элементов векторов A и B нет. Как и в предыдущем примере, здесь можно говорить о линейной зависимости временной сложности от параметра данных.

С параметром временной сложности алгоритма обычно связывают две теоретические проблемы. Первая состоит в поиске ответа на вопрос: до какого предела значения временной сложности можно дойти, совершенствуя алгоритм решения задачи? Этот предел зависит от самой задачи и, следовательно, является ее собственной характеристикой.

Вторая проблема связана с классификацией алгоритмов по временной сложности. Функция Tα(V) обычно растет с ростом V. Как быстро она растет? Существуют алгоритмы с линейной зависимостью Tα от V (как это было в рассмотренных нами примерах), с квадратичной зависимостью и с зависимостью более высоких степеней. Такие алгоритмы называются полиномиальными. А существуют алгоритмы, сложность которых растет быстрее любого полинома. Проблема, которую часто решают теоретики — исследователи алгоритмов, заключается в следующем вопросе: возможен ли для данной задачи полиномиальный алгоритм?

Практическая работа № 1.11. Работа с классами

Цель работы: получение практических навыков работы с классами и объектами

Теоретический материал

Описанием объекта является класс, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики. То есть некоторый шаблон - этот шаблон можно назвать классом. Конкретное воплощение этого шаблона может отличаться,

например, одни люди имеют одно имя, другие - другое имя. И реально существующий человек (фактически экземпляр данного класса) будет представлять объект этого класса.

По умолчанию проект консольного приложения уже содержит один класс Program, с которого и начинается выполнение программы.

По сути класс представляет новый тип, который определяется пользователем. Класс определяется с помощью ключевого слова class:

```
class Person
{

}
```

Где определяется класс? Класс можно определять внутри пространства имен, вне пространства имен, внутри другого класса. Как правило, классы помещаются в отдельные файлы. Но в данном случае поместим новый класс в файле, где располагается класс Program. То есть файл Program.cs будет выглядеть следующим образом:

```
using System;

namespace HelloApp
{
   class Person
   {

   }
   class Program
   {
      static void Main(string[] args)
      {

      }
   }
}
```

Вся функциональность класса представлена его членами - полями (полями называются переменные класса), свойствами, методами, событиями. Например, определим в классе Person поля и метод:

```
using System;

namespace HelloApp
{
   class Person
   {
      public string name; // имя
      public int age = 18;     // возраст

      public void GetInfo()
      {
         Console.WriteLine($"Имя: {name}  Возраст: {age}");
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         Person tom;
      }
   }
```

}

В данном случае класс Person представляет человека. Поле name хранит имя, а поле age - возраст человека. А метод GetInfo выводит все данные на консоль. Чтобы все данные были доступны вне класса Person переменные и метод определены с модификатором public. Поскольку поля фактически те же переменные, им можно присвоить начальные значения, как в случае выше, поле age инициализировано значением 18.

Так как класс представляет собой новый тип, то в программе мы можем определять переменные, которые представляют данный тип. Так, здесь в методе Main определена переменная tom, которая представляет класс Person. Но пока эта переменная не указывает ни на какой объект и по умолчанию она имеет значение null. Поэтому вначале необходимо создать объект класса Person.

Конструкторы

Кроме обычных методов в классах используются также и специальные методы, которые называются конструкторами. Конструкторы вызываются при создании нового объекта данного класса. Конструкторы выполняют инициализацию объекта.

Конструктор по умолчанию

Если в классе не определено ни одного конструктора, то для этого класса автоматически создается конструктор по умолчанию. Такой конструктор не имеет параметров и не имеет тела.

Выше класс Person не имеет никаких конструкторов. Поэтому для него автоматически создается конструктор по умолчанию. И мы можем использовать этот конструктор. В частности, создадим один объект класса Person:

```
class Person
{
   public string name; // имя
   public int age;     // возраст

   public void GetInfo()
   {
      Console.WriteLine($"Имя: {name}  Возраст: {age}");
   }
}
class Program
{
   static void Main(string[] args)
   {
      Person tom = new Person();
      tom.GetInfo();      // Имя: Возраст: 0

      tom.name = "Tom";
      tom.age = 34;
      tom.GetInfo();  // Имя: Tom Возраст: 34
      Console.ReadKey();
   }
}
```

Для создания объекта Person используется выражение new Person(). Оператор new выделяет память для объекта Person. И затем вызывается конструктор по умолчанию, который не принимает никаких параметров. В итоге после выполнения данного выражения в памяти будет выделен участок, где будут храниться все данные объекта Person. А переменная tom получит ссылку на созданный объект.

Если конструктор не инициализирует значения переменных объекта, то они получают значения по умолчанию. Для переменных числовых типов это число 0, а для типа string и классов - это значение null (то есть фактически отсутствие значения).

После создания объекта мы можем обратиться к переменным объекта Person через переменную tom и установить или получить их значения, например, tom.name = "Tom";.

Консольный вывод данной программы:

Имя: Возраст: 0

Имя: Tom Возраст: 34

Создание конструкторов

Выше для инициализации объекта использовался конструктор по умолчанию. Однако мы сами можем определить свои конструкторы:

```
class Person
{
   public string name;
   public int age;

   public Person() { name = "Неизвестно"; age = 18; }      // 1 конструктор

   public Person(string n) { name = n; age = 18; }         // 2 конструктор

   public Person(string n, int a) { name = n; age = a; }   // 3 конструктор

   public void GetInfo()
   {
      Console.WriteLine($"Имя: {name}  Возраст: {age}");
   }
}
```

Теперь в классе определено три конструктора, каждый из которых принимает различное количество параметров и устанавливает значения полей класса. Используем эти конструкторы:

```
static void Main(string[] args)
{
   Person tom = new Person();          // вызов 1-ого конструктора без параметров
   Person bob = new Person("Bob");     //вызов 2-ого конструктора с одним параметром
   Person sam = new Person("Sam", 25); // вызов 3-его конструктора с двумя параметрами


   bob.GetInfo();          // Имя: Bob  Возраст: 18
   tom.GetInfo();          // Имя: Неизвестно  Возраст: 18
   sam.GetInfo();          // Имя: Sam  Возраст: 25
}
```

Консольный вывод данной программы:

```
Имя: Неизвестно  Возраст: 18
Имя: Bob  Возраст: 18
Имя: Sam  Возраст: 25
```

При этом если в классе определены конструкторы, то при создании объекта необходимо использовать один из этих конструкторов.

Стоит отметить, что начиная с версии C# 9.0 мы можем сократить вызов конструктора, убрав из него название типа:

```
Person tom = new ();            // аналогично new Person();
Person bob = new ("Bob");       // аналогично new Person("Bob");
Person sam = new ("Sam", 25);   // аналогично new Person("Sam", 25);
```

Ключевое слово this

Ключевое слово this представляет ссылку на текущий экземпляр класса. В каких ситуациях оно нам может пригодиться? В примере выше определены три конструктора. Все три конструктора выполняют однотипные действия - устанавливают значения полей name и age. Но этих повторяющихся действий могло быть больше. И мы можем не дублировать функциональность конструкторов, а просто обращаться из одного конструктора к другому через ключевое слово this, передавая нужные значения для параметров:

```
class Person
{
   public string name;
```

```
    public int age;

    public Person() : this("Неизвестно")
    {
    }
    public Person(string name) : this(name, 18)
    {
    }
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
    public void GetInfo()
    {
        Console.WriteLine($"Имя: {name}  Возраст: {age}");
    }
}
```

В данном случае первый конструктор вызывает второй, а второй конструктор вызывает третий. По количеству и типу параметров компилятор узнает, какой именно конструктор вызывается. Например, во втором конструкторе:

```
public Person(string name) : this(name, 18)
{
}
```

идет обращение к третьему конструктору, которому передаются два значения. Причем в начале будет выполняться именно третий конструктор, и только потом код второго конструктора.

Также стоит отметить, что в третьем конструкторе параметры называются также, как и поля класса.

```
public Person(string name, int age)
{
    this.name = name;
    this.age = age;
}
```

И чтобы разграничить параметры и поля класса, к полям класса обращение идет через ключевое слово this. Так, в выражении this.name = name; первая часть this.name означает, что name - это поле текущего класса, а не название параметра name. Если бы у нас параметры и поля назывались по-разному, то использовать слово this было бы необязательно. Также через ключевое слово this можно обращаться к любому полю или методу.

Инициализаторы объектов

Для инициализации объектов классов можно применять инициализаторы. Инициализаторы представляют передачу в фигурных скобках значений доступным полям и свойствам объекта:

```
1
2
Person tom = new Person { name = "Tom", age=31 };
tom.GetInfo();          // Имя: Tom  Возраст: 31
```

С помощью инициализатора объектов можно присваивать значения всем доступным полям и свойствам объекта в момент создания без явного вызова конструктора.

При использовании инициализаторов следует учитывать следующие моменты:

С помощью инициализатора мы можем установить значения только доступных из внешнего кода полей и свойств объекта. Например, в примере выше поля name и age имеют модификатор доступа public, поэтому они доступны из любой части программы.

Инициализатор выполняется после конструктора, поэтому если и в конструкторе, и в инициализаторе устанавливаются значения одних и тех же полей и свойств, то значения, устанавливаемые в конструкторе, заменяются значениями из инициализатора.

Практическая работа № 1.12. Перезагрузка методов

Цель работы: изучить метод перезагрузки

Теоретический материал

Виртуальные и динамические методы могут быть перезагружаемыми. Перезагрузка нужна, чтобы произвести одинаковые или похожие действия с разнотипными данными. Статический метод перекрытия приводит к тому, что потомок «не видит» перекрытый родительский метод и может обращаться к нему только с помощью слова inherited, поэтому в Delphi используется перезагрузка, с помощью которой становятся видны одноименные методы как родителя, так и потомка.

Пример:

```
Type TFirst=class

i:Extended;

procedure SetData(x:Extended);

end;

TSecond=class(TFirst)

j:Integer;

procedure SetData(AValue:Integer);

end;

. . . . .

var T2:TSecond;

. . . . .

begin

T2.SetData (1.1); (1)

T2.SetData (1); (2)
```

В этом примере первый вызов метода SetData из объекта T2 с переменной =1.1 (не целая) вызовет ошибку! А второй вызов не приводит к ошибке, т.к. внутри объекта T2 статический метод с параметром типа Extended перекрыт одноименным методом с параметром типа Integer. Компилятор внутри T2 не признает параметр типа Extended.

Для доступа к методу SetData класса TFirst необходимо использовать служебное слово inherited. Например:

```
procedure TSecond.SetData;

begin

x :=1.1;
```

```
inherited SetData (x);

j:=x;

end;
```

Но нам нужно произвести схожие действия (1), (2) с разнотипными данными в строках одной программы.

В ходе компиляции при обращении к одному из одноименных методов компилятор проверяет тип и количество параметров обращения и на основе этой проверки выбирает нужный метод.

Overload – директива, позволяющая перезагрузить методы:

```
Type TFirst=class

i:Extended;

procedure SetData(x:Extended); Overload;

end;

TSecond=class(TFirst)

j:Integer;

procedure SetData(AValue:Integer); Overload;

end;

. . . . .

var T2:TSecond;

. . . . .
```

Теперь в программе можно использовать метод как родителя, так и потомка.

```
begin

T2.SetData (1.1);

T2.SetData (1);
```

Можно перезагрузить и виртуальный, и динамический методы. В этом случае надо добавить директиву reintroduce перед директивой overload.

Задача с использованием полиморфизма

Полиморфизм – это возможность использовать одинаковые имена для методов, входящих в различные классы. Концепция полиморфизма обеспечивает в случае применения метода к объекту использование именно того метода, который соответствует классу объекта.

Ход работы

Задание. Пусть определены 3 класса, один из которых является базовым для двух других:

```
Type
```

```
Tperson=class {базовый класс}

fname:string;

constructor Create(name:string);

function info:string; virtual;

end;

Tstud=class(Tperson) {класс– потомок}

fgr:integer; {поле для номера группы}

constructor Create(name:string;gr:integer);

function info:string; override;

end;

Tprof=class(Tperson) {класс– потомок}

fdep:string; ; {поле для названия кафедры}

constructor Create(name:string; dep:string);

function info:string; override;

end;
```

В каждом из этих классов определен метод info. В базовом классе при помощи директивы virtual метод info объявлен виртуальным. Это дает возможность классу–потомку произвести замену виртуального метода своим собственным. В каждом классе–потомке определен свой метод info, который замещает соответствующий метод родительского класса и отмечается директивой override.

Определим метод info для каждого класса индивидуально:

```
function Tperson.info:string;

begin

result:=’’;

end;

function Tstud.info:string;

begin

result:=fname+' gruppa '+inttostr(fgr);

end;

function Tprof.info:string;
```

```
begin

result:=fname+' department '+fdep;

end;
```

Далее в программе список всех людей можно представить массивом объектов класса Tperson. Отметим, что объект – указатель.

Список людей имеет вид:

```
list: array[1..szl] of Tperson; { szl –размер списка}
```

Объявить подобным образом список можно потому, что ОР позволяет присвоить указателю на родительский класс значение указателя на класс– потомок. Поэтому элементами массива list могут быть как объекты класса Tstud, так и объекты класса Tprof.

Вывод списка можно осуществить применением метода info к элементам массива, например:

```
St:= ’’;

for i:=1 to szl do

if list[i]<>nil then

St:=St+list[i].info+#13;

ShowMessage('Spisok:'+#13+St); { вывод в окно сообщения}
```

Во время работы программы каждый элемент массива может содержать как объект типа Tstud, так и объект типа Tprof.

Концепция полиморфизма обеспечивает применение к объекту именно того метода info, который соответствует типу объекта.

Напишем программу, которая использует объявления классов Tperson, Tstud, Tprof, формирует список студентов и преподавателей и выводит полученный список в окно сообщения. Будем использовать визуальное программирование.

Окно формы будет иметь вид:

GroupBox1—это компонент , объединяющий группу компонентов, связанных по смыслу. В данном случае он включает 2 зависимых переключателя – RadioButton1 и RadioButton2

Текст модуля кода программы:

```
unit Polimorfizm;
interface
uses
Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls;
Type
TForm1 = class(TForm)
Label1: TLabel;
Label2: TLabel;
Edit1: TEdit;
Edit2: TEdit;
GroupBox1: TGroupBox;
RadioButton1: TRadioButton;
RadioButton2: TRadioButton;
```

```
Button1: TButton;
Button2: TButton;
procedure Button1Click(Sender: TObject);
procedure Button2Click(Sender: TObject);
private
{ Private declarations }
public
{ Public declarations }
end;
type Tperson=class
fname:string;
constructor Create(name:string);
function info:string; virtual;
end;
Tstud=class(Tperson) {класс– потомок}
fgr:integer; {поле для номера группы}
constructor Create(name:string;gr:integer);
function info:string; override;
end;
Tprof=class(Tperson) {класс– потомок}
fdep:string; ; {поле для названия кафедры}
constructor Create(name:string; dep:string);
function info:string; override;
end;
Const szl=10;
Var
Form1: TForm1;
list:array[1..szl] of Tperson;
n:integer;
implementation {исполняемая часть}
{$R *.DFM}
constructor Tperson.Create(name:string); {описание конструктора}
begin
fname:=name;
end;
constructor Tstud.Create(name:string;gr:integer);
begin
inherited create(name);
fgr:=gr;
end;
constructor Tprof.Create(name:string;dep:string);
begin
inherited create(name);
fdep:=dep;
end;
function Tperson.info:string;
begin
result:=fname;
end;
function Tstud.info:string;
begin
result:=fname+' gruppa '+inttostr(fgr);
end;
function Tprof.info:string;
begin
result:=fname+' department '+fdep;
```

```
end;
procedure TForm1.Button1Click(Sender: TObject);
//процедура обработки нажатия на кнопку «Добавить»
begin
if n<szl
then
begin
if RadioButton1.Checked
then
list[n]:=Tstud.Create(Edit1.text, StrtoInt(Edit2.text))
else
list[n]:=Tprof.Create(Edit1.text, Edit2.text);
n:=n+1;
end
else
ShowMessage('Spisok zapolnen');
end;
prcedure TForm1.Button2Click(Sender: TObject);
//процедура обработки нажатия на кнопку «Список»
Var i:integer;
St:string;
Begin
for i:=1 to szl do

if list[i]<>nil then
St:=St+list[i].info+#13;
// list[i].info вызовет тот метод info, которому соответствует элемент
ShowMessage('Spisok:'+#13+St);
end;
end.
```

Процедура TForm1.Button1Click, которая запускается нажатием кнопки «Добавить» создает объект list[n] класса либо Tstud, либо Tprof.

Класс создаваемого объекта определяется состоянием переключателя RadioButton. Установка переключателя в положение Студент определяет класс Tstud, а в положение – Преподаватель определяет класс Tprof. Процедура TForm1.Button2Click, которая запускается нажатием кнопки Список (Button2) применяет метод info к каждому элементу массива list[i] как к объекту списка и формирует строку, представляющую весь итоговый список.

Виртуальность метода info обеспечивает применение к объекту именно того метода info, который соответствует типу объекта.

Отметим, что в данной программе результаты выводятся в окно сообщения процедурой ShowMessage. Например:

```
Showmessage('Spisok:'+#13+st);
Showmessage('spisok zapolnen');
```

Практическая работа № 1.13. Определение операций в классе

<u>Цель работы:</u> изучить способы определения операций в классе

<u>Теоретический материал</u>

Операции класса

C# позволяет переопределить действие большинства операций так, чтобы при использовании с объектами конкретного класса они выполняли заданные функции. Это дает возможность применять экземпляры собственных типов данных в составе выражений таким же образом, как стандартных, например:

```
MyObject a, Ь, c;
c = a + Ь; // используется операция сложения для класса MyObject
```

Определение собственных операций класса часто называют *перегрузкой операций.* Перегрузка обычно применяется для классов, описывающих математические или физические понятия, то есть таких классов, для которых семантика операций делает программу более понятной. Если назначение операции интуитивно не понятно с первого взгляда, перегружать такую операцию не рекомендуется.

Операции класса описываются **с** помощью методов специального вида*(функций-операций).* Перегрузка операций похожа на перегрузку обычных методов. Синтаксис операции:

[атрибуты] спецификаторы объявитель_операции тело

Атрибуты рассматриваются позже**,** в качестве*спецификаторов* одновременно используются ключевые словаpublicиstatic. Кроме того, операцию можно объявить как внешнюю (extern).

Объявитель операции содержит ключевое словоoperator, по которому и опознается описание операции в классе.*Тело* операции определяет действия, которые выполняются при использовании операции в выражении. Тело представляет собой блок, аналогичный телу других методов.

Новые обозначения для собственных операций вводить нельзя. Для операций класса сохраняются количество аргументов, приоритеты операций и правила ассоциации (справа налево или слева направо), используемые в стандартных типах данных.

При описании операций необходимо соблюдать следующие правила:

- операция должна быть описана как открытый статический метод класса (спецификаторы public static);
- параметры в операцию должны передаваться по значению (то есть не должны предваряться ключевыми словами refилиout);
- сигнатуры всех операций класса должны различаться;
- типы, используемые в операции, должны иметь не меньшие права доступа, чем сама операция (то есть должны быть доступны при использовании операции).

В C# существуют три вида операций класса: унарные, бинарные и операции преобразования типа.

Унарные операции

Можно определять в классе следующие *унарные операции***:**

+ - ! - ++ -- true false

Синтаксис объявителя унарной операции:

тип operator унарная_операция (параметр)

Примеры заголовков унарных операций:

public static int operator+ (MyObject m)

public static MyObject operator-- (MyObject m)

public static bool operator true(MyObject m)

*Параметр***,** передаваемый в операцию, должен иметь тип класса, для которого она определяется. Операция должна возвращать:

- для операций +,-,! и - величину любого типа;
- для операций ++и--величину типа класса, для которого она определяется;
- для операций trueиfalseвеличину типа bool.

Операции не должны изменять значение передаваемого им операнда. Операция, возвращающая величину типа класса, для которого она определяется, должна создать новый объект этого класса, выполнить с ним необходимые действия и передать его в качестве результата.

Префиксный и постфиксный инкременты не различаются (для них может существовать только одна реализация, которая вызывается в обоих случаях).

Операции trueиfalseобычно определяются для логических типов SQL, обладающих неопределенным состоянием, и не входят в число тем, рассматриваемых здесь.

В качестве примера усовершенствуем приведенный в листинге 1.17 классSafeArrayудля удобной и безопасной работы с массивом. В класс внесены следующие изменения:

- добавлен конструктор, позволяющий инициализировать массив обычным массивом или серией целочисленных значений произвольного размера;
- добавлена операция инкремента;
- добавлен вспомогательный метод Print вывода массива;

- изменена стратегия обработки ошибок выхода за границы массива;
- снято требование, чтобы элементы массива принимали значения в заданном диапазоне.

Ход работы

Задание1. Определить операции инкремента для класса SafeArray

```
using System;
namespace ConsoleApplication1
{
class SafeArray
{
public SafeArray( int size) // конструктор
{
a = new int[size];
length = size;
}
public SafeArray( params int[ ] arr) // новый конструктор
{
length = arr.length;
a = new int[length];
for( int i=0; i<length; ++i) a[ i ] = arr [ i ];
}
public static SafeArray operator++ ( SafeArray x ) // операция ++
{
SafeArray temp = new SafeArray( x.length );
for( int i = 0; i < x.length; ++i)
temp[ i ] = ++x.a[ i ];
return temp;
}
public int this[ int i ] // индексатор
{
get
{
if ( i >= 0 && i < length ) return a[i];
else throw new indexOutOfRangeException(); // исключение
}
set
{
if ( i >= 0 && i < length ) a[i] = value;
else throw new IndexOutOfRangeException(); // исключение
}
}
public void Print( string name ) // вывод на экран
{
Console.WriteLine( name + ":" );
for ( int i = 0; i < length; ++i )
Console.Write( "\t" + a[i] );
Console.WriteLine();
}
int[] a; // закрытый массив
int length; // закрытая размерность .
}
class Class1
{
static void Main()
{
try
{
```

```
SafeArray a1 = new SafeArray( 5, 2, -1, 1, -2 );
a1.Print( "Массив 1" );
a1++;
a1.Print( "Инкремент массива 1" );
}
catch ( Exception e ) // обработка исключения
{
Console.WriteLine( e.Message );
}
}
}
}
```

Бинарные операции

Можно определять в классе следующие бинарные операции:

+ - * / % & | ^ << >> == != > < >= <=

Синтаксис объявителя бинарной операции:

тип operator бинарная_операция (параметр1, параметр2)

Примеры заголовков бинарных операций:

```
public static MyObject operator + ( MyObject m1, MyObject m2 )
public static bool operator == ( MyObject m1, MyObject m2 )
```

Хотя бы один параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется. Операция может возвращать величину любого типа.

Операции ==и!=,>и<,>=и<= определяются только парами и обычно возвращают логическое значение. Чаще всего в классе определяют операции сравнения на равенство и неравенство для того, чтобы обеспечить сравнение объектов, а не их ссылок, как определено по умолчанию для ссылочных типов. Перегрузка операций отношения требует знания интерфейсов, поэтому она рассматривается позже.

Пример определения операции сложения для класса SafeArray, описанного в предыдущем разделе, приведен в листинге 1.20. В зависимости от операндов операция либо выполняет поэлементное сложение двух массивов, либо прибавляет значение операнда к каждому элементу массива.

Задание 2. Определение операции сложения для класса SafeArray

```
using System;
namespace ConsoleApplication1
{
class SafeArray
{
public SafeArray( int size )
{
a = new int [ size ];
length = size;
}
public SafeArray( params int [ ] arr )
{
length = arr.Length:
a = new int [ length ];
for ( int i = 0; i < length; ++i ) a[ i ] = arr [ i ];
}
public static SafeArray operator + ( SafeArray x, SafeArray y ) // +
{
int len = x.length < y.length ? x.length : y.length;
SafeArray temp = new SafeArray ( len );
for ( int i = 0; i < len; ++i ) temp[i] = x[i] + y[i];
return temp;
}
```



```
public static SafeArray operator + ( SafeArray x, int y ) // +
```

```
{
SafeArray temp = new SafeArray ( x.length );
for ( int i = 0; i < x.length; ++i ) temp[i] = x[i] + y;
return temp;
}
public static SafeArray operator + ( int x, SafeArray y )
{ // +
SafeArray temp = new SafeArray(y.length);
for ( int i = 0; i < y.length; ++i ) temp[i] = x + y[i];
return temp;
}
public static SafeArray operator ++ ( SafeArray x ) // ++
{
SafeArray temp = new SafeArray(x.length);
for ( int i = 0; i < x.length; ++i ) temp[i] = ++x.a[i];
return temp;
}
public int this[int i]
{ // [ ]
get
{
if ( i >= 0 && i < length ) return a[i];
else throw new IndexOutOfRangeException();
}
set
{
if ( i >= 0 && i < length ) a[i] = value;
else throw new IndexOutOfRangeException();
}
}
public void Print( string name )
{
Console.WriteLine( name + ":" );
for ( int i = 0; i < length; ++i )
Console.Write( "\t" + a [ I ] );
Console.WriteLine();
}
int[] a;
int length;
}
class Class1
{
static void Main()
{
try
{
SafeArray a1 = new SafeArray( 5, 2, -1, 1, -2 );
a1.Print( "Массив 1" );
SafeArray a2 - new SafeArrayC 1. 0, 3 );
a2.Print( "Массив 2" ); a1++;
SafeArray a3 = a1 + a2;
a3.Print( "Сумма массивов 1 и 2" );
a1 = a1 + 100; // 1
a1.Print( "Массив 1 + 100" );
a1 = 100 + a1; // 2
a1.PrintC "100 + массив 1" );
```

```
a2 += ++a2 +1; // 3 оторвать руки!
a2.Print( "++a2, a2 + a2 + 1" );
}
catch ( Exception e )
{
Console.WriteLine( e.Message );
}
}
}
}
```

Результат работы программы:

```
Массив 1:
5 2 -1 1 -2
Массив 2:
1 0 3
Сумма массивов 1 и 2:
7 3 3
Массив 1+100:
106 103 100 102 99
100 + массив 1:
206 203 200 202 199
++a2, a2 +a2 + 1:
5 3 9
```

Обратите внимание: чтобы обеспечить возможность сложения с константой, операция сложения перегружена два раза для случаев, когда константа является первым и вторым операндом (операторы 2 и 1).

Сложную операцию присваивания += (оператор 3) определять не требуется, да это и невозможно. При ее выполнении автоматически вызываются сначала операция сложения, а потом присваивания. В целом же оператор 3 демонстрирует недопустимую манеру программирования, поскольку результат его выполнения неочевиден.

ПРИМЕЧАНИЕ

В перегруженных методах для объектов применяется индексатор. Для повышения эффективности можно обратиться к закрытому полю-массиву и непосредственно, например: temp.a[i] = x + y.a[i].

Операции преобразования типа

Операции преобразования типа обеспечивают возможность явного и неявного преобразования между пользовательскими типами данных. Синтаксис объявителя операции преобразования типа:

```
implicit operator тип ( параметр ) // неявное преобразование
explicit operator тип ( параметр ) // явное преобразование
```

Эти операции выполняют преобразование из типа параметра в тип, указанный в заголовке операции. Одним из этих типов должен быть класс, для которого определяется операция. Таким образом, операции выполняют преобразование либо типа класса к другому типу, либо наоборот. Преобразуемые типы не должны быть связаны отношениями наследования. Примеры операций преобразования типа для класса Monster, описанного в разделе 6:

```
public static implicit operator int( Monster m )
{
return m.health;
}
public static explicit operator Monster( int h )
{
return new Monster( h, 100, "Fromlnt" );
}
```

Ниже приведены примеры использования этих преобразований в программе. Не надо искать в них смысл, они просто иллюстрируют синтаксис:


```
Monster Masha = new Monster( 200, 200, "Masha" );
```

Int i = Masha: // неявное преобразование

Masha = (Monster) 500; // явное преобразование

Неявное преобразование выполняется автоматически:

- при присваивании объекта переменной целевого типа, как в примере;
- при использовании объекта в выражении, содержащем переменные целевого типа;
- при передаче объекта в метод на место параметра целевого типа;
- при явном приведении типа.

+*Явное преобразование* выполняется при использовании операции приведения типа.

Все операции класса должны иметь разные сигнатуры. В отличие от других видов методов, для операций преобразования тип возвращаемого значения включается в сигнатуру, иначе нельзя было бы определять варианты преобразования данного типа в несколько других. Ключевые слова implicit иexplicitв сигнатуру не включаются, следовательно, для одного и того же преобразования нельзя определить одновременно явную и неявную версии.

*Неявное преобразование*следует определять так, чтобы при его выполнении не возникала потеря точности и не генерировались исключения. Если эти ситуации возможны, преобразование следует описать как явное.

Практическая работа № 1.14. Создание наследованных классов

Цель работы: изучить возможности наследования классов

Теоретические сведения

Язык C++ позволяет классу наследовать данные-элементы и функции-элементы одного или нескольких других классов. Новый класс называют производным классом. Класс, элементы которого наследуются производным классом, называется базовым классом. В свою очередь производный класс может служить базовым для другого класса. Наследование дает возможность заключить некоторое общее или схожее поведение различных объектов в одном базовом классе.

Наследование позволяет также изменить поведение существующего класса. Производный класс может переопределить некоторые функции-элементы базового, наследуя, тем не менее, основной объем свойств и атрибутов базового класса. Общий вид наследования: class Base { // ….. }; class Derived: <ключ доступа> Base { // …… };

Ключ доступа может быть private, protected, public. Если ключ не указан, то по умолчанию он принимается private. Наследование позволяет рассматривать целые иерархии классов и работать со всеми элементами одинаково, приводя их к базовому. Правила приведения следующие: Наследуемый класс всегда можно привести к базовому;

Базовый класс можно привести к наследуемому только если в действительности это объект наследуемого класса. Ошибки приведения базового класса к наследуемому отслеживаются программистом.

Доступ к элементам класса

При наследовании ключ доступа определяет уровень доступа к элементам базового класса внутри производного класса. В таблице описаны возможные варианты доступа.

Наследование Доступ в базовом классе Доступ в производном классе

public public protected private public protected private

protected public protected private protected protected private

private public protected private private private private

Конструкторы и деструкторы при наследовании

Конструкторы не наследуются. Если конструктор базового класса требует спецификации одного или нескольких параметров, конструктор производного класса должен вызывать базовый конструктор, используя список инициализации элементов. Пример 1.

```
#include <string>
class Base
{   public:
   Base(int, float);
};
class Derived: Base
```

```
{   public:
    Derived(char* lst, float amt);
};
    Derived:: Derived(char* lst, float amt) : Base(strlen(lst),amt)
    {   }
```

В деструкторе производного класса компилятор автоматически генерирует вызовы базовых деструкторов, поэтому для удаления объекта производного класса следует сделать деструктор в базовых классах виртуальным. Для вызова используется delete this либо operator delete.

Виртуальные функции

Функция-элемент может быть объявлена как virtual. Ключевое слово virtual предписывает компилятору генерировать некоторую дополнительную информацию о функции. Если функция переопределяется в производном классе и вызывается с указателем (или ссылкой) базового класса, ссылающимся на представитель производного класса, эта информация позволяет определить, какой из вариантов функции должен быть выбран: такой вызов будет адресован функции производного класса.

Для виртуальных функций существуют следующие правила: виртуальную функцию нельзя объявлять как static. спецификатор virtual необязателен при переопределении функции в производном классе. виртуальная функция должна быть определена в базовом классе и может быть переопределена в производном.

Ход работы

Задание. Написать программу с наследованием класса стек от класса массив.

```
#include <iostream.h>
#include <stdlib.h>
class massiv
{   int *num;
    int kol;
    public:
    massiv(int n);
    void print();
    virtual int kolich(){return kol;}
    void put(int k,int n){num[k]=n;}
    ~massiv(){delete num;}
};
massiv::massiv(int n)
{   num=new int[n];
    kol=n;
    for(int i=0;i<kol;i++)  num[i]=random(100)-50;
}
void massiv::print()
{   for(int i=0;i<kolich();i++)  cout<<num[i]<<" ";
    cout<<endl;
}
class stec:public massiv
{   int top;
    public:
    stec(int);
    virtual int kolich() {return top;}
    void pop(int k);
};
stec::stec(int n):massiv(n)
{   top=0;
}
void stec::pop(int k)
{   put(top++,k);  }
void main()
```

```
{   randomize();
    massiv a(10);
    a.print();
    stec b(10);
    b.pop(random(100)-50);
    b.pop(random(100)-50);
    b.pop(random(100)-50);
    b.print();
}
```

Задание 2. Разработать программу с использованием наследования классов, реализующую классы: графический объект; круг; квадрат. Используя виртуальные функции, не зная с объектом какого класса вы работаете, выведите на экран его размер и координаты.

Задание 3. Разработать программу с использованием наследования классов, реализующую классы: железнодорожный вагон; вагон для перевозки автомобилей; цистерна. Используя виртуальные функции, не зная с объектом какого класса вы работаете, выведите на экран его вес и количество единиц товара в вагоне.

Практическая работа № 1.15. Работа с объектами через интерфейсы

Цель работы: изучить способ создания и реализации интерфейса

Ход работы

Задание 1. Создание и реализация интерфейса

В этом упражнении Вы создадите интерфейс, определяющий поведение классов, которые будут его реализовывать.

Предполагается, что в библиотечной системе, разработанной в прошлой работе есть необходимость реализовать возможность оформления подписки на периодические издания. Включение этой функциональности в базовый класс **Item** не является правильным решением, так как к изданиям, оформляющим подписку не относятся, например книги. Поэтому необходимо создать интерфейс, объявляющий возможность оформления подписки и тогда классы, для которых предполагается данная функциональность должны будут реализовывать этот интерфейс.

Выполните подготовительные операции

Создайте папку Lab07 и скопируйте в нее решение MyClass, созданное в прошлом упражнении.

Создайте интерфейс IPubs с требуемой функциональностью

Откройте проект MyClass.sln в папке *install folder*\Labs\Lab07\.

Добавьте в проект новый интерфейс с именем **IPubs**: **Projects** (Проект) → **Add class** (Добавить класс). В окне **Добавление нового элемента** выберите **Интерфейс** и укажите его имя **IPubs**.

В интерфейсе **IPubs** объявите его функциональные члены – метод для проверки оформлена ли подписка на издание **Subs** и свойство **IfSubs** для

оформления подписки: interface IPubs

```
{
void Subs();
bool IfSubs { get; set;}
}
```

36

Реализуйте интерфейс в классе Magazine

Откройте класс **Magazine** и добавьте интерфейс в список наследования: class Magazine : Item, IPubs

```
{
Реализуйте свойство и метод, объявленные в интерфейсе: public bool IfSubs { get; set; }
public void Subs()
{
Console.WriteLine("Подписка на журнал \"{0}\": {1}." , title, IfSubs);
}
```

Протестируйте новую функциональность

В методе **Main** класса **Program** добавьте для уже имеющегося журнала **mag1** установку свойству **IfSubs** значения, устанавливающую подписку и

вызовите метод **Subs** для отображения информации о подписке:

Magazine mag1 = new Magazine("О природе", 5,"Земля и мы", 2014, 1235, true);

mag1.TakeItem();

mag1.Show();

mag1.IfSubs = true; mag1.Subs();

Постройте проект и исправьте ошибки, если это необходимо. Запустите и протестируйте программу.

Задание 2. Использование стандартных интерфейсов

В библиотеке классов .Net определено множество стандартных интерфейсов, задающих желаемую функциональность объектов. В этом упражнении Вы примените интерфейс **IComparable,** который задает метод сравнения объектов по принципу больше и меньше, что позволяет переопределить соответствующие операции в рамках класса, наследующего интерфейс **IComparable**.

Сравнение и дальнейшая сортировка будет реализована по полю **invNumber** –

Инвентарный номер.

Реализуйте наследование интерфейса IComparable

Добавьте в объявление абстрактного класса **Item** наследование интерфейса

IComparable:

abstract class Item : IComparable

{

. . .

Интерфейс **IComparable** определен в пространстве имен **System** и содержит единственный метод **CompareTo**, возвращающий результат сравнения двух объектов – текущего и переданного ему в качестве параметра. Реализация данного метода должна возвращать: 0 – если текущий объект и параметр равны, отрицательное число, если текущий объект меньше параметра и положительное число, если текущий объект больше параметра.

Добавьте в класс **Item** реализацию этого метода, причем сравнение реализуйте по полю **invNumber:**

int IComparable.CompareTo(object obj)

{

Item it = (Item)obj;

if (this.invNumber == it.invNumber) return 0; else if (this.invNumber > it.invNumber) return 1; else return -1;

}

Протестируйте использование новой функциональности

В методе **Main** класса **Program** создайте массив ссылок на абстрактный

базовый класс **Item:**

Item[] itmas = new Item[4];

Заполните массив созданными ранее книгами и журналом: itmas[0] = b1;

itmas[1] = b2; itmas[2] = b3; itmas[3] = mag1;

Отсортируйте массив с помощью статического метода **Sort** класса **Array**:

Array.Sort(itmas);

Отобразите весь список книг и журналов, используя полиморфный вызов метода **Show**:

Console.WriteLine("\nСортировка по инвентарному номеру"); foreach (Item x in itmas)

{

x.Show();

}

Постройте проект и исправьте ошибки, если это необходимо. Запустите и протестируйте программу. Информация о каждом элементе хранения должна выводиться согласно возрастанию инвентарных номеров.

Задание 3. Создание иерархии классов «Фигуры»

В этом упражнении требуется создать иерархию классов – геометрических фигур: треугольник, окружность и квадрат, которые являются производными классами общего класса **Shape**.

Класс треугольник реализован в упражнении 3 лабораторной работы 5. Классы окружность и квадрат определяются соответственно радиусом и стороной.

Реализуйте конструкторы, создающие объекты с заданным радиусом и стороной и методы, позволяющие:

вывести длины радиуса окружности и стороны квадрата на экран;

расчитать периметр и площадь фигур.

Общую функциональность фигур реализуйте в базовом классе Shape. Реализуйте дополнительную функциональность для треугольника и квадрата –вращение фигуры вокруг своего центра.

Метод вращения представьте в интерфейсе. Классы треугольник и квадрат должны реализовывать этот интерфейс.

Практическая работа № 1.16. Использование стандартных интерфейсов

Цель работы: изучить способы использования стандартных интерфейсоф

Ход работы

Интерфейс представляет некое описание типа, набор компонентов, который должен иметь тип данных. И, собственно, мы не можем создавать объекты интерфейса напрямую с помощью конструктора, как например, в классах:

```
IMovable m = new IMovable(); // ! Ошибка, так сделать нельзя
```

В конечном счете интерфейс предназначен для реализации в классах и структурах. Например, возьмем следующий интерфейс IMovable:

```
interface IMovable
{
   void Move();
}
```

Затем какой-нибудь класс или структура могут применить данный интерфейс:

```
// применение интерфейса в классе
class Person : IMovable
{
   public void Move()
   {
      Console.WriteLine("Человек идет");
   }
}
// применение интерфейса в структуре
struct Car : IMovable
{
   public void Move()
   {
      Console.WriteLine("Машина едет");
   }
}
```

При применении интерфейса, как и при наследовании после имени класса или структуры указывается двоеточие и затем идут названия применяемых интерфейсов. При этом класс должен реализовать все методы и свойства применяемых интерфейсов, если эти методы и свойства не имеют реализации по умолчанию.

Если методы и свойства интерфейса не имеют модификатора доступа, то по умолчанию они являются публичными, при реализации этих методов и свойств в классе и структуре к ним можно применять только модификатор public.

Применение интерфейса в программе:

```
using System;
```

```
2
3       namespace HelloApp
4       {
5          interface IMovable
6          {
7             void Move();
8          }
9          class Person : IMovable
10         {
11            public void Move()
12            {
13               Console.WriteLine("Человек идет");
14            }
15         }
16         struct Car : IMovable
17         {
18            public void Move()
19            {
20               Console.WriteLine("Машина едет");
21            }
22         }
23         class Program
24         {
25            static void Action(IMovable movable)
26            {
27               movable.Move();
28            }
29            static void Main(string[] args)
30            {
31               Person person = new Person();
32               Car car = new Car();
33               Action(person);
34               Action(car);
35               Console.Read();
36            }
37         }
38      }
```

В данной программе определен метод Action(), который в качестве параметра принимает объект интерфейса IMovable. На момент написания кода мы можем не знать, что это будет за объект - какой-то класс или структура. Единственное, в чем мы можем быть уверены, что этот объект обязательно реализует метод Move и мы можем вызвать этот метод.

Иными словами, интерфейс - это контракт, что какой-то определенный тип обязательно реализует некоторый функционал.

Реализация интерфейсов по умолчанию

Начиная с версии C# 8.0 интерфейсы поддерживают реализацию методов и свойств по умолчанию. Зачем это нужно? Допустим, у нас есть куча классов, которые реализуют некоторый интерфейс. Если мы добавим в этот интерфейс новый метод, то мы будем обязаны реализовать этот метод во всех классах, применяющих данный интерфейс. Иначе подобные классы просто не будут компилироваться. Теперь вместо реализации метода во всех классах нам достаточно определить его реализацию по умолчанию в интерфейсе. Если класс не реализует метод, будет применяться реализация по умолчанию.

```
1       class Program
2       {
3          static void Main(string[] args)
4          {
5             IMovable tom = new Person();
```

```
6            Car tesla = new Car();
7            tom.Move();     // Walking
8            tesla.Move();   // Driving
9         }
10      }
11
12      interface IMovable
13      {
14         void Move()
15         {
16            Console.WriteLine("Walking");
17         }
18      }
19      class Person : IMovable { }
20      class Car : IMovable
21      {
22         public void Move()
23         {
24            Console.WriteLine("Driving");
25         }
26      }
```

В данном случае интерфейс IMovable определяет реализацию по умолчанию для метода Move. Класс Person не реализует этот метод, поэтому он применяет реализацию по умолчанию в отличие от класса Car, который определяет свою реализацию для метода Move.

Стоит отметить, что хотя для объекта класса Person мы можем вызвать метод Move - ведь класс Person применяет интерфейс IMovable, тем не менее мы не можем написать так:

```
1      Person tom = new Person();
2      tom.Move();     // Ошибка - метод Move не определен в классе Person
```

Множественная реализация интерфейсов

Интерфейсы имеют еще одну важную функцию: в C# не поддерживается множественное наследование, то есть мы можем унаследовать класс только от одного класса, в отличие, скажем, от языка C++, где множественное наследование можно использовать. Интерфейсы позволяют частично обойти это ограничение, поскольку в C# класс может реализовать сразу несколько интерфейсов. Все реализуемые интерфейсы указываются через запятую:

```
1      myClass: myInterface1, myInterface2, myInterface3, ...
2      {
3
4      }
```

Рассмотрим на примере:

```
1      using System;
2
3      namespace HelloApp
4      {
5         interface IAccount
6         {
7            int CurrentSum { get; }  // Текущая сумма на счету
8            void Put(int sum);      // Положить деньги на счет
9            void Withdraw(int sum); // Взять со счета
10        }
11        interface IClient
12        {
13           string Name { get; set; }
14        }
15        class Client : IAccount, IClient
16        {
17           int _sum; // Переменная для хранения суммы
```

```
18          public string Name { get; set; }
19          public Client(string name, int sum)
20          {
21              Name = name;
22              _sum = sum;
23          }
24
25          public int CurrentSum { get { return _sum; } }
26
27          public void Put(int sum) {  _sum += sum; }
28
29          public void Withdraw(int sum)
30          {
31              if (_sum >= sum)
32              {
33                  _sum -= sum;
34              }
35          }
36      }
37      class Program
38      {
39          static void Main(string[] args)
40          {
41              Client client = new Client("Tom", 200);
42              client.Put(30);
43              Console.WriteLine(client.CurrentSum); //230
44              client.Withdraw(100);
45              Console.WriteLine(client.CurrentSum); //130
46              Console.Read();
47          }
48      }
49  }
```

В данном случае определены два интерфейса. Интерфейс IAccount определяет свойство CurrentSum для текущей суммы денег на счете и два метода Put и Withdraw для добавления денег на счет и изъятия денег. Интерфейс IClient определяет свойство для хранения имени клиента.

Обатите внимание, что свойства CurrentSum и Name в интерфейсах похожи на автосвойства, но это не автосвойства. При реализации мы можем развернуть их в полноценные свойства, либо же сделать автосвойствами.

Класс Client реализует оба интерфейса и затем применяется в программе.

Интерфейсы в преобразованиях типов

Все сказанное в отношении преобразования типов характерно и для интерфейсов. Поскольку класс Client реализует интерфейс IAccount, то переменная типа IAccount может хранить ссылку на объект типа Client:

```
1   // Все объекты Client являются объектами IAccount
2   IAccount account = new Client("Том", 200);
3   account.Put(200);
4   Console.WriteLine(account.CurrentSum); // 400
5   // Не все объекты IAccount являются объектами Client, необходимо явное приведение
6   Client client = (Client)account;
7   // Интерфейс IAccount не имеет свойства Name, необходимо явное приведение
8   string clientName = ((Client)account).Name;
```

Преобразование от класса к его интерфейсу, как и преобразование от производного типа к базовому, выполняется автоматически. Так как любой объект Client реализует интерфейс IAccount.

Обратное преобразование - от интерфейса к реализующему его классу будет аналогично преобразованию от базового класса к производному. Так как не каждый объект IAccount является

объектом Client (ведь интерфейс IAccount могут реализовать и другие классы), то для подобного преобразования необходима операция приведения типов. И если мы хотим обратиться к методам класса Client, которые не определены в интерфейсе IAccount, но являются частью класса Client, то нам надо явным образом выполнить преобразование типов: string clientName = ((Client)account).Name;

Практическая работа № 1.17. Работа с типом данных структура

Цель работы: изучение типа данных структура

Теоретический материал

Типы структур

Структуры по своей внутренней организации похожи на классы, они содержат набор полей и методов. Как правило, их используют для объявления типов, которые определяются только значениями полей и не имеют индивидуальности. Например, объекты, описывающие транзакции, несмотря на то, что значения их полей могут совпадать не будут тождественными, то есть нам их нужно уметь различать несмотря на внешнее сходство. А точки на геометрической плоскости, которые задаются двумя координатами, такой индивидуальности не имеют, и если координаты двух точек совпадают, то это значит, что речь идет об одной и той же точке. Именно для таких типов хорошо подходят структуры. Для их объявления используется ключевое слово struct:

```
struct Point
{
   public Point(double x, double y)
   {
      X = x;
      Y = y;
   }

   public double X {get;}
   public double Y {get;}
}
Point p1 = new Point(1,2);
Console.WriteLine($"({p1.X}, (p1.Y})");
```

Типы значений, допускающие null

Про типы значений, допускающих null см. ниже “Nullable-типы (нулевые типы) и операция ??”.

Типы значений кортежей

Кортежи используются для группировки данных, которые могут иметь разные типы в единую именованную сущность. Они являются объектами типа System.ValueTuple. Объявим кортеж, состоящий из двух элементов типа double:

```
(double, double) tp1 = (1.0, 2.0); // явное задание типов элементов кортежа
var tp2 = (8.1, 4.3); // использование var для объявления кортежа
```

Поля кортежа могут быть именованными:

```
(double X, double Y) tp3 = (3.2, 5.34);
var tp4 = (X: 1.2, Y: 3.4);
var X = 5.6;
var Y = 7.8;
var tp5 = (X, Y);
```

Более подробно про кортежи типов System.ValueTuple (тип-значение) и System.Tuple (ссылочный тип) будет рассказано в одном из следующих уроков.

Ссылочные типы

Переменные ссылочного типа располагаются в куче, за их уничтожение отвечает сборщик мусора, поэтому про них нельзя точно сказать, когда занимаемая ими память будет освобождена.

Переменная представляется в виде ссылки на соответствующее место в куче. Ссылочные типы являются наследниками от System.Object.

Типы классов

Классы являются наиболее фундаментальным элементов в системе типов C#. Тип System.Object, который является родительским для всех типов данных представляет собой класс. Из рассмотренных выше типов данных, класс больше всего похож на структуру, у них даже объявление похожи, только вместо ключевого слова struct нужно использовать class.

```
class Persone
{
   public Persone(string name, int age)
   {
      Name = name;
      Age = age;
   }
   public string Name {get;set;}
   public int Age {get;set;}
}
Persone persone1 = new Persone("John", 21);
Console.WriteLine($"Persone: Name: {persone1.Name}, Age: {persone1.Age}")");
```

Среди классов в C# можно выделить ряд классов, которые играю важную роль в языке, они перечислены в таблице ниже.

Класс	Описание
System.Object	Базовый класс для всех типов в C#
System.ValueType	Базовый класс для всех типов-значений
System.Enum	Базовый класс для всех перечислений
System.Array	Базовый класс для всех массивов
System.Delegate	Базовый класс для всех делегатов
System.Exception	Базовый класс для всех исключений
System.String	Класс, определяющий строкой тип данных

Типы интерфейсов

Интерфейс представляет собой набор методов, свойств, событий и индексаторов. До версии C# 8.0 интерфейс предполагал только декларацию (объявление) указанных выше элементов, начиная с 8.0, в рамках интерфейса можно располагать реализацию по умолчанию. Фактически интерфейс представляет собой контракт, а класс, который от него наследуется, реализует этот контракт.

Ход работы

1. Создадим интерфейс для описания человека, у которого есть два свойства имя: Name, и возраст: Age:

```
interface IPersone
{
   string Name {get;set;}
   int Age {get;set;}
}
```

2. Изменим объявление класса Persone, так, чтобы он представлял реализацию интерфейса IPersone:

```
class Persone: IPersone
{
//…
}
```

3. Объявим переменную типа IPersone:

```
IPersone persone2 = new Persone("Jim", 25);
Console.WriteLine($"Persone: Name: {persone2.Name}, Age: {persone2.Age}")");
```

Более подробно про интерфейсы будет рассказано в одном из следующих уроков.

Типы массивов

Массив – это структура данных, которая позволяет хранить один или более элементов. Массивы в C# делятся на одномерные и многомерные, среди последних наибольшее распространение получили двумерные массивы. Все массивы являются наследниками класса System.Array.

Создание и инициализация одномерного массива:

```
int[] nArr1 = new int[5];
nArr1[0] = 0;
nArr1[1] = 1;
nArr1[2] = 2;
nArr1[3] = 3;
nArr1[4] = 4;
```

Пример прямоугольного массива, в нем строки имеют одинаковую длину:

```
int[,] nMx = new int[2,2]; // прямоугольный массив
nMx[0,0]=0;
nMx[0,1]=1;
nMx[1,0]=2;
nMx[1,1]=3;
```

Пример зубчатого (jagged) массива, в нем строки могут иметь разную длину:

```
int[][] jg = new int[2][]; // зубчатый массив
jg[0] = new int[3];
jg[1] = new int[1];
```

Более подробно про массивы будет рассказано в одном из следующих уроков.

Типы делегатов

Делегаты являются аналогом указателей на функции из языков C / C++. Они используются в случаях, когда нужно передать некоторую функциональность как аргумент, перенаправлять вызовы и т.д.

Nullable-типы (нулевые типы) и операция ??

Объявление и инициализация Nullable-переменных

В работе с типами-значениями есть одна особенность, они не могут иметь значение null. При наличии любой из следующих строк кода, компиляция программы не будет выполнена:

```
int nv = null;
bool bv = null;
```

На практике, особенно при работе с базами данных, может возникнуть ситуация, когда в записи из таблицы пропущены несколько столбцов (нет данных), в этом случае, соответствующей переменной нужно будет присвоить значение null, но она может иметь тип int или double, что приведет к ошибке.

Можно объявить переменную с использованием символа ? после указания типа, тогда она станет nullable-переменной – переменной поддерживающей null-значение:

```
int? nv1 = null;
bool? bv1 = null;
```

Использование символа ? является синтаксическим сахаром для конструкции Nullable<T>, где T – это имя типа. Представленные выше примеры можно переписать так:

```
Nullable<int> nv1 = null;
Nullable<bool> bv1 = null;
```

Проверка на null. Работа с HasValue и Value

Для того чтобы проверить, что переменная имеет значение null можно воспользоваться оператором is с шаблоном типа:

```
bool? flagA = true;
if(flagA is bool valueOfFlag)
```

```
{
   Console.WriteLine("flagA is not null, value: {valueOfFlag}");
}
```

Также можно воспользоваться свойствами класса Nullable:

Nullable<T>.HasValue

Возвращает true если переменная имеет значение базового типа. То есть если она не null.

Nullable<T>.Value

Возвращает значение переменной если HasValue равно true, иначе выбрасывает исключение InvalidOperationException.

```
bool? flagB = false;
if(flagB.HasValue)
{
   Console.WriteLine("flagB is not null, value: {flagB.Value}");
}
```

Приведение Nullable-переменной к базовому типу

При работе с Nullable-переменными их нельзя напрямую присваивать переменным базового типа. Следующий код не будет скомпилирован:

```
double? nvd1 = 12.3;
double nvd2 = nvd1; // error
```

Для приведения Nullable-переменной к базовому типу можно воспользоваться явным приведением:

```
double nvd3 = (double) nvd1;
```

В этом случае следует помнить, что если значение Nullable-переменной равно null, то при выполнении данной операции будет выброшено исключение InvalidOperationException.

Второй вариант – это использование оператора ??, при этом нужно дополнительно задаться значением, которое будет присвоено переменной базового типа если в исходной лежит значение null:

```
double nvd4 = nvd1 ?? 0.0;
Console.WriteLine(nvd4);
bool? nvb1 = null;
bool nvb2 = nvb1 ?? false;
Console.WriteLine(nvb1);
Console.WriteLine(nvb2);
```

Второй вариант позволяет более лаконично обрабатывать ситуацию, когда вызов какого-то метода может возвращать null, а результат его работы нужно присвоить типу-значению, при этом заранее известно, какое значение нужно присвоить переменной в этой ситуации:

```
static int? GetValue(bool flag)
{
   if (flag == true)
      return 1000;
   else
      return null;
}
static void Main(string[] args)
{
   int test1 = GetValue(true) ?? 123;
   Console.WriteLine(test1);
   int test2 = GetValue(false) ?? 123;
   Console.WriteLine(test2);
}
```

Ключевое слово dynamic

Вначале статьи мы говорили о том, что есть языки со статической и динамической типизацией, C# – язык со статической типизацией, т.е. типы переменных определяются на этапе компиляции. Но в рамках платформы .NET есть возможность работать с Python и Ruby в реализациях IronPython и IronRuby, но это языки с динамической типизацией, в них тип определяется во время выполнения программы. Для того чтобы можно было в C# проекте работать с тем, что было создано в рамках IronPython (или IronRuby) начиная с C# 4, в языке появилось ключевое слово dynamic и среда DLR (Dynamic Language Runtime), благодаря которой можно создавать динамические объекты, тип которых будет определен на этапе выполнения программы, а не в процессе компиляции.

С помощью ключевого слова dynamic объявляются переменные, для которых нужно опустить проверку типов в процессе компиляции. Для этой переменной не производится присвоение типа из BCL (Base Class Library) – стандартной библиотеки классов .NET, фактически dynamic – это тип System.Object с дополнительным набором метаданных, они нужны для определения типа переменной в процессе выполнения (так называемое, позднее связывание).

Ниже приведены несколько примеров, на которых можно разобраться с тем, как работать с dynamic:

```
// Создадим переменную типа dynamic и проинициализируем ее double значением
dynamic dval1 = 12.3;
// Посмотрим на ее значение и тип
Console.WriteLine($"Value: {dval1}");
Console.WriteLine($"Type: {typeof(dval1)}"));
// Изменим значение переменной:
dval1 += 17;
Console.WriteLine($"Value: {dval1}");
Console.WriteLine($"Type: {typeof(dval1)}"));
// Присвоим переменной значение другого типа: bool
dval1 = true;
// Посмотрим на ее значение и тип
Console.WriteLine($"Value: {dval1}");
Console.WriteLine($"Type: {typeof(dval1)}"));
```

Как вы можете видеть значение и тип переменной dval1 менялись в процессе выполнения программы. При этом нужно помнить, что если вы присвоили переменной dynamic, какое-то значение, которое определило ее тип, а пытаетесь с ней работать как с переменной другого типа, то будет вызвано исключение:

```
dynamic dval2 = "hello"; // в переменной dval2 хранится строковое значение
Console.WriteLine($"Value: {dval2}");
Console.WriteLine($"Type: {typeof(dval2)}");
dval2 = 123; // теперь значение типа int
dval2 = dval2.ToUpper() // попытка вызвать на ней .ToUpper() приведет к ошибке
```

Оператор default

Оператор default создает значение по умолчанию для указанного типа, используется оно следующим образом: default(T), где T – это тип, для которого нужно создать соответствующее значение.

4. Объявим переменную типа int и присвоим ей значение по умолчанию с помощью new:

```
int n3 = new int();
Console.WriteLine($"Default int value: {n3}");
```

Тоже самое можно сделать с помощью оператора default:

```
int n4 = default(int);
Console.WriteLine($"Value of int that inited by default(T): {n4}");
```

Если C# может самостоятельно вывести тип, то можно воспользоваться не оператором, а литерой default, без явного указания типа:

```
int n5 = default;
Console.WriteLine($"Value of int that inited by default: {n5}");
```

Данный оператор полезен при разработке методов с обобщенным типом. Создадим метод, который выводит на консоль значение по умолчанию для типа переданного в нее аргумента:

```
static void PrintDefaultValue<T>(T val)
{
   Console.WriteLine($"Type of val: {val.GetType()}, default value: {default(T)}, current value: {val}");
}
```

Вызовем эту функцию:

```
static void Main(string[] args)
{
   PrintDefaultValue<int>(5);
   PrintDefaultValue<bool>(true);
}
```

Практическая работа № 1.18. Коллекции

Цель работы: Изучение инструмента коллекций

Теоретический материал

Коллекции являются одним из наиболее часто используемых инструментов в разработке программного обеспечения. В этом уроке мы познакомимся с пространством имен System.Collections.Generic, коллекциями List, Dictionary и типом Tuple.

Коллекции

Самым примитивным способом хранения объектов в C# является использование массивов. Одной из основных проблем, с которой столкнется разработчик следуя такому подходу, является то, что массивы не предоставляют инструментов для динамического изменения размера. В языке C# есть два пространства имен для работы со структурами данных:

System.Collections;

System.Collections.Generic.

Первое из них – System.Collections предоставляет структуры данных для хранения объектов типа Object. У этого решения есть две основных проблемы – это производительность и безопасность типов. В настоящее время не рекомендуется использовать объекты классов из System.Collections.

Для решения указанных выше проблем Microsoft были разработаны коллекции с обобщенными типами (их ещё называют дженерики), они расположены в пространстве имен System.Collections.Generic. Суть их заключается в том, что вы не просто создает объект класса List, но и указываете, объекты какого типа будут в нем храниться, делается это так: List<T>, где T может быть int, string, double или какой-то ваш собственный класс.

В рамках данного урока мы не будем подробно останавливаться на особенностях обобщенных типов, на текущий момент можете их воспринимать как псевдонимы, для реальных типов данных.

Коллекции в языке C#. Пространство имен System.Collections.Generic

Пространство System.Collections.Generic содержит большой набор коллекций, которые позволяют удобно и эффективно решать широкий круг задач. Ниже, в таблице, перечислены некоторые из обобщенных классов с указанием интерфейсов, которые они реализуют.

Обобщенный класс Основные интерфейсы Описание

List<T> ICollection<T>, IEnumerable<T>, IList<T> Список элементов с динамически изменяемым размером

Dictionary<TKey, TValue> ICollection<T>, IDictionary<TKey, TValue>,

IEnumerable<T>

Коллекция элементов связанных через уникальный ключ

Queue<T> ICollection, IEnumerable<T> Очередь – список, работающий по алгоритму FIFO

Stack<T> ICollection, IEnumerable<T> Стэк – список, работающий по алгоритму LIFO

SortedList<TKey,TValue> IComparer<T>, ICollection<KeyValuePair<TKey,TValue>>, IDictionary<TKey,TValue> Коллекция пар “ключ-значение”, упорядоченных по ключу

Ход работы

Класс List<T>

Коллекциями с класса List<T>. Эта коллекция является аналогом типизированного массива, который может динамически расширяться. В качестве типа можно указать любой встроенный либо пользовательский тип.

Задание 1. Создание объекта класса List<T>

Можно создать пустой список и добавить в него элементы позже, с помощью метода Add():

```
List<int> numsList = new List<int>();
numsList.Add(1);
```

Либо воспользоваться синтаксисом, позволяющем указать набор объектов, который будет храниться в списке:

```
List<int> nums = new List<int> {1, 2, 3, 4, 5};
var words = new List<string> {"one", "two", "three"};
```

Работа с объектами List<T>

Ниже приведены таблицы, в которых перечислены некоторые полезные свойства и методы класса List<T>. Более подробную информацию по методам и свойствам List<T> вы можете найти в официальной документации.

Свойства класса List<T>

Свойство Описание

Count Количество элементов в списке

Capacity Емкость списка – количество элементов, которое может вместить список без изменения размера

```
Console.WriteLine("Свойства");
Console.WriteLine($"- Count: nums.Count = {nums.Count}");
Console.WriteLine($"- Capacity: nums.Capacity = {nums.Capacity}");
```

Методы класса List<T>

Метод Описание

Add(T) Добавляет элемент к списку

BinarySearch(T) Выполняет поиск по списку

Clear() Очистка списка

Contains(T) Возвращает true, если список содержит указанный элемент

IndexOf(T) Возвращает индекс переданного элемента

ForEach(Action<T>) Выполняет указанное действие для всех элементов списка

Insert(Int32, T) Вставляет элемент в указанную позицию

Find(Predicate<T>) Осуществляет поиск первого элемент, для которого выполняется заданный предикат

Remove(T) Удаляет указанный элемент из списка

RemoveAt(Int32) Удаляет элемент из заданной позиции

Sort() Сортирует список

Reverse() Меняет порядок расположения элементов на противоположный

```
Console.WriteLine($"nums: {ListToString(nums)}");
nums.Add(6);
Console.WriteLine($"nums.Add(6): {ListToString(nums)}");
Console.WriteLine($"words.BinarySearch(\"two\"): {words.BinarySearch("two")}");
Console.WriteLine($"nums.Contains(10): {nums.Contains(10)}");
Console.WriteLine($"words.IndexOf(\"three\"): {words.IndexOf("three")}");
Console.WriteLine($"nums.ForEach(v => v * 10)");
nums.ForEach(v => Console.Write($"{v} => "));
nums.Insert(3, 7);
```

```
Console.WriteLine($"nums.Insert(3, 7): {ListToString(nums)}");
Console.WriteLine($"words.Find(v => v.Length == 3): {words.Find(v => v.Length == 3)}");
words.Remove("two");
Console.WriteLine($"words.Remove(\"two\"): {ListToString(words)}");
```

Код метода ListToString:

```
static private string ListToString<T>(List<T> list) =>
"{" + string.Join(", ", list.ToArray()) + "}";
```

Далее приведен пример работы со списком, в котором хранятся объекты пользовательского типа. Создадим класс Player, имеющий свойства: Name и Skill.

```
class Player
{
   public string Name { get; set; }
   public string Skill { get; set; }
}
```

Задание 2. Создадим список игроков и выполним с ним ряд действий:

```
Console.WriteLine("Работа с пользовательским типом");
List<Player> players = new List<Player> {
   new Player { Name = "Psy", Skill = "Monster"},
   new Player { Name = "Kubik", Skill = "Soldier"},
   new Player { Name = "Triver", Skill = "Middle"},
   new Player { Name = "Terminator", Skill = "Very High"}
};
Console.WriteLine("Количество элементов в players:{0}", players.Count);
//Добавим новый элемент списка players
players.Insert(1, new Player { Name = "Butterfly", Skill = "flutter like a butterfly, pity like a bee"});
//Посмотрим на все элементы списка
players.ForEach(p => Console.WriteLine($"{p.Name}, skill: {p.Skill}"));
```

Класс Dictionary<TKey,TValue>

Класс Dictionary реализует структуру данных Отображение, которую иногда называют Словарь или Ассоциативный массив. Идея довольно проста: в обычном массиве доступ к данным мы получаем через целочисленный индекс, в словаре используется ключ, который может быть числом, строкой или любым другим типом данных, который реализует метод GetHashCode(). При добавлении нового объекта в такую коллекцию для него указывается уникальный ключ, который используется для последующего доступа к нему.

Задание 3. Создание объекта класса Dictionary

Пустой словарь:

```
var dict = new Dictionary<string, int>();
```

Словарь с набором элементов:

```
var prodPrice = new Dictionary<string, double>()
{
   ["bread"] = 23.3,
   ["apple"] = 45.2
};
Console.WriteLine($"bread price: {prodPrice["bread"]}");
```

Задание 4. Работа с объектами Dictionary

Рассмотрим некоторые из свойств и методов класса Dictionary<TKey, TValue>. Полное описание возможностей этого класса вы можете найти на официальной странице Microsoft.

Свойства класса Dictionary

Свойство	Описание
Count	Количество объектов в словаре

Keys Ключи словаря
Values Значения элементов словаря
Console.WriteLine("Свойства");
Console.WriteLine($"Словарь prodPrice: {DictToString(prodPrice)}");
Console.WriteLine($"Count: {prodPrice.Count}");
Console.WriteLine($"Keys: {ListToString(prodPrice.Keys.ToList<string>())}");
Console.WriteLine($"Values: {ListToString(prodPrice.Values.ToList<double>())}");
Методы класса Dictionary
Метод Описание
Add(TKey, TValue) Добавляет в словарь элемент с заданным ключом и значением
Clear() Удаляет из словаря все ключи и значения
ContainsValue(TValue) Проверяет наличие в словаре указанного значения
ContainsKey(TKey) Проверяет наличие в словаре указанного ключа
GetEnumerator() Возвращает перечислитель для перебора элементов словаря
Remove(TKey) Удаляет элемент с указанным ключом
TryAdd(TKey, TValue) Метод, реализующий попытку добавить в словарь элемент с заданным ключом и значением
TryGetValue(TKey, TValue) Метод, реализующий попытку получить значение по заданному ключу
prodPrice.Add("tomate", 11.2);
Console.WriteLine($"Словарь prodPrice: {DictToString(prodPrice)}");
var isExistValue = prodPrice.ContainsValue(11.2);
Console.WriteLine($"isExistValue = {isExistValue}");
var isExistKey = prodPrice.ContainsKey("tomate");
Console.WriteLine($"isExistKey = {isExistKey}");
prodPrice.Remove("bread");
Console.WriteLine($"Словарь prodPrice: {DictToString(prodPrice)}");
var isOrangeAdded = prodPrice.TryAdd("orange", 20.1);
Console.WriteLine($"isOrangeAdded = {isOrangeAdded}");
double orangePrice;
var isPriceGetted = prodPrice.TryGetValue("orange", out orangePrice);
Console.WriteLine($"isPriceGetted = {isPriceGetted}");
Console.WriteLine($"orangePrice = {orangePrice}");
prodPrice.Clear();
Console.WriteLine($"Словарь prodPrice: {DictToString(prodPrice)}");
Кортежи Tuple и ValueTuple

Относительно недавним нововведением в языке C# (начиная с C# 7) являются кортежи. Кортежем называют структуру данных типа Tuple или ValueTuple (чуть ниже мы рассмотрим различия между ними), которые позволяют группировать объекты разных типов друг с другом. С практической точки зрения они являются удобным способом возврата из метода нескольких значений – это наиболее частый вариант использования кортежей.

Различия между Tuple и ValueTuple приведены в таблице ниже.

Tuple ValueTuple
Ссылочный тип Тип значение
Неизменяемый тип Изменяемый тип
Элементы данных – это свойства Элементы данных – это поля
Создание кортежей
Рассмотрим несколько вариантов создания кортежей.
Создание кортежа без явного и с явным указанием имен полей:
(string, int) p1 = ("John", 21);
(string Name, int Age) p2 = ("Mary", 23);

При этом для доступа к элементам кортежа в первом варианте используются свойства Item с числом, указывающем на порядок элемента, во втором – заданные пользователем имена:

Console.WriteLine($"p1: Name: {p1.Item1}, Age: {p1.Item2}");

Console.WriteLine($"p1: Name: {p2.Name}, Age: {p2.Age}");

Возможны следующие способы создания кортежей с явным заданием имен:

var p3 = (Name: "Alex", Age: 24);

var Name = "Lynda";

var Age = 25;

var p4 = (Name, Age);

Console.WriteLine($"p3: Name: {p3.Name}, Age: {p3.Age}");

Console.WriteLine($"p4: Name: {p4.Name}, Age: {p4.Age}");

При этом возможность обращаться через свойства Item1 и Item2 для созданных выше переменных остается:

Console.WriteLine($"p3: Name: {p3.Item1}, Age: {p3.Item2}");

Console.WriteLine($"p4: Name: {p4.Item1}, Age: {p4.Item2}");

Работа с кортежами

Как было сказано в начале раздела, кортежи можно возвращать в качестве результата работы метода. Пример метода, который сравнивает длину переданной строки с некоторым порогом и возвращает соответствующее bool-значение и целое число – длину строки:

static (bool isLonger, int count) LongerThenLimit(string str, int limit) =>

str.Length > limit ? (true, str.Length) : (false, str.Length);

Кортежи можно присваивать друг другу, при этом необходимо, чтобы соблюдались следующие условия:

количество элементов в обоих кортежах одинаковое;

типы соответствующих элементов совпадают, либо могут быть приведены друг к другу.

var p5 = ("Jane", 26);

(string, int) p6 = p5;

Console.WriteLine($"p6: Name: {p6.Item1}, Age: {p6.Item2}");

Операцию присваивания можно использовать для деструкции кортежа.

(string name, int age) = p5;

Console.WriteLine($"Name: {name}, Age: {age}");

Исходный код примеров из этой статьи можете скачать из нашего github-репозитория.

Практическая работа № 1.19. Параметризованные классы

Цель работы: изучить механизм параметрического полиморфизма на основе создания и использования параметризованных классов.

Ход работы

Задание. Ввести код программы, проанализировать результат.

Пример программы

//класс вещество, содержащий температуру и массу

class *Substance* {

double Mass;

double Temperature;

public:

//требуется перегрузка оператора сложения для класса Substance, так как данная операция совершается в классе Matrix. Без данной перегрузки компилятор выдаст ошибку

Substance* operator** + (Substance*** c) {

Substance res;

res.Mass = ***this***->Mass + c.Mass;

res.Temperature = (this->Temperature*this->Mass +

c.Temperature*c.Mass) / (this->Mass + c.Mass);

return res;

}

//так как метод rand_val вызывается из класса Matrix, то данный метод должен обязательно присутствовать в классе Substance

static *Substance* rand_val() {

Substance res;

res.Mass = (rand() % 2000) / 100.0 + 5;

```
res.Temperature = (rand() % 2000) / 100.0 + 10;
return res;
}
```

//так как метод to_str вызывается из класса Matrix, то данный метод должен обязательно присутствовать в классе Substance

```
void to_str(char* bufer) {
sprintf(bufer, "M=%.2lf;T=%.2lf", Mass, Temperature);
}
};
```

//определение шаблонного класса Matrix, который позволяет хранить матрицу из объектов, которые имеются тип T

```
template <class T>
class Matrix{
```

//определение массива элементов типа T

```
T m[4][4];
public:
Matrix() {
for (int i = 0; i < 4; i++)
for (int j = 0; j < 4; j++)
m[i][j] = T::rand_val();
}
```

//запись T::rand_val() в шаблонном классе предполагает наличие у класса T публичного статического метода rand_val. Программа не может быть собрана, если у класса T нет данного метода.

```
void print() {
char *bufer = new char[100];
printf("Matrix\n");
for (int i = 0; i < 4; i++) {
for (int j = 0; j < 4; j++) {
m[i][j].to_str(bufer);
printf("%s\t", bufer);
}
printf("\n");
}
printf("\n");
delete bufer;
}
```

//запись m[i][j].to_str(bufer) для элемента (он имеет тип T) массива m предполагает наличие метода to_str у класса T. Программа не может быть собрана, если у класса T нет данного метода.

```
Matrix operator + (Matrix &b) {
Matrix<T> res;
for (int i = 0; i < 4; i++)
for (int j = 0; j < 4; j++)
res.m[i][j] = this->m[i][j] + b.m[i][j];
return res;
}
```

//операция сложения между экземплярами класса T (это - this->m[i][j] и b.m[i][j]) предполагает наличие у класса T перегруженного оператора сложения. Программа не может быть собрана, если у класса T нет данного перегруженного оператора.

```
};
```

//класс Substance имеет статический метод rand_val(), метод to_str() и перегруженный оператор сложения, поэтому он может быть использован в качестве параметра шаблонного класса Matrix.

```
void main() {
Matrix<Substance> m1;
m1.print();
Matrix<Substance> m2;
m2.print();
Matrix<Substance> m3 = m1 + m2;
m3.print();
}
```

Практическая работа № 1.20. Использование регулярных выражений

Цель работы: изучить регулярные выражения

Теоретический материал

Регулярные выражения представляют собой язык описания текстовых шаблонов. Регулярные выражения содержат образцы символов, входящих в искомое текстовое выражение, и конструкции, определяемые специальными символами (метасимволами).

Метасимволы, используемые в регулярных выражениях

^ начало строки

$ конец строки

[] любой символ, заключенный в квадратные скобки; чтобы задать диапазон символов, в квадратных скобках указываются через дефис первый и последний символы диапазона

[^] любой символ, кроме символов, заданных в квадратных скобках

. любой отдельный символ

\ отменяет специальное значение следующего за ним метасимвола

* указывает, что предыдущий шаблон встречается 0 или более раз

\{n\} указывает, что предыдущий шаблон встречается ровно n раз

\{n,\} указывает, что предыдущий шаблон встречается не менее n раз

\{,n\} указывает, что предыдущий шаблон встречается не более n раз

\{n,m\} указывает, что предыдущий шаблон встречается не менее n и не более m раз

Примеры регулярных выражений

^the ищутся строки, начинающиеся с буквосочетания "the"

be$ ищутся строки, заканчивающиеся буквосочетанием "be"

[Ss]igna[lL] ищутся строки, содержащие буквосочетания: "signal", "Signal", "signaL" или "SignaL"

\. ищутся строки, содержащие точку

^...th ищутся строки, содержащие символы "th" в 4-й и 5-й позициях

^.*\{53\}th ищутся строки, содержащие символы "th" в 54-й и 55-й позициях

^.*\{10,30\}th ищутся строки, содержащие символы "th" в любых позициях между 11-й и 31-й

^.....$ ищутся строки, состоящие из 5 любых символов

^t.*e$ ищутся строки, начинающиеся с буквы "t" и заканчивающиеся буквой "e"

[0-9][a-z] ищутся строки, содержащие комбинацию: цифра-прописная буква

[^123] ищутся строки, не содержащие цифр "1" или "2" или "3"

Функции для работы с регулярными выражениями:

1) *boolereg(stringpattern, stringstring [, arrayregs])* – ищет в строке string соответствие регулярному выражению, заданному в шаблоне pattern.

2) *stringereg_replace(stringpattern, stringreplacement, stringstring)* – заменяет найденный в строке string шаблон pattern на строку replacement и, если соответствие было найдено, возвращает модифицированную строку.

3) *booleregi (stringpattern, stringstring[, arrayregs])* – идентична функции ereg, за исключением того, что она игнорирует регистр.

4) *arraysplit (stringpattern, stringstring [, intlimit])* – возвращает массив строк, которые представляют собой подстроки строки string, образованные в результате разделения строки string на подстроки в соответствии с регулярным выражением pattern.

5) *arrayspliti (stringpattern, stringstring [, intlimit])* - аналогична функции split, за исключением того, что является нечувствительной к регистру.

6) *stringeregi_replace (stringpattern, stringreplacement, stringstring)* – аналогична функции ereg_replace, за исключением того, что она является нечувствительной к регистру.

Ход работы

1. Составьте регулярное выражение для проверки корректности заполнения адреса электронной почты.

2. Создайте web-страницу, содержащую четыре поля (имя, адрес электронной почты, пароль и подтверждение пароля) и кнопку отправки данных.

Ход работы (решение сохраните в отдельную папку):

1. В первом задании нужно только составить выражение без его проверки на компьютере. В регулярном выражении для проверки адреса электронной почты необходимо учесть то, что: а) в имени пользователя могут присутствовать буквы нижнего и верхнего регистров, цифры, знаки подчеркивания, минуса и точки; б) для проверки разделителя между именем пользователя и именем домена в выражение требуется добавить +@; в) доменное имя может содержать две или три латинские буквы. Все три шага нужно объединить в одно выражение при помощи плюса.

2. К выполнению второго задания предъявляется следующие требования:

a. Поля и кнопка должны располагаться сверху вниз;

b. В имени могут содержаться только латинские буквы и цифры;

c. Адрес электронной почты проверяется в соответствие с регулярным выражением, составленным в предыдущем задании;

d. Пароль и подтверждение пароля должно отображаться знаками «*». Для этого указывается type=password. Конечно, значения обоих полей должны совпадать;

e. Вся форма и каждое поле в отдельности проверяется на пустоту при помощи функции empty или isset. Для каждого пустого поля выводится соответствующее сообщение красным цветом, например, «не введён адрес»;

f. Из первых двух полей удалите обратные слеши и тэги;

Как только форма заполнена абсолютно корректно на той же web-странице (где находится форма) выводится сообщение «всё в порядке».

Практическая работа № 1.21. Операции со списками

Цель работы: изучение операций со списками

Теоретический материал

Основные операции

Список – структура данных, в которой каждый элемент (узел) хранит информацию, а также ссылку на следующий элемент. Последний элемент списка ссылается на NULL.

Для нас односвязный список полезен тем, что

- Он очень просто устроен и все алгоритмы интуитивно понятны
- Односвязный список – хорошее упражнение для работы с указателями
- Его очень просто визаулизировать, это позволяет "в картинках" объяснить алгоритм
- Несмотря на свою простоту, односвязные списки часто используются в программировании, так что это не пустое упражнение.
- Эта структуру данных можно определить рекурсивно, и она часто используется в рекурсивных алгоритмах.

Для простоты рассмотрим односвязный список, который хранит целочисленное значение.

Односвязный список

Односвязный список состоит из узлов. Каждый узел содержит значение и указатель на следующий узел, поэтому представим его в качестве структуры

?

```
typedef struct Node {
   int value;
   struct Node *next;
} Node;
```

Чтобы не писать каждый раз struct мы определили новый тип. Теперь наша задача написать функцию, которая бы собирала список из значений, которые мы ей передаём. Стандартное имя функции – push, она должна получать в качестве аргумента значение, которое вставит в список. Новое значение будет вставляться в начало списка. Каждый новый элемент списка мы должны создавать на куче. Следовательно, нам будет удобно иметь один указатель на первый элемент списка.

?

```
Node *head = NULL;
```

Вначале списка нет и указатель ссылается на NULL. Для добавления нового узла необходимо

- Выделить под него память.
- Задать ему значение
- Сделать так, чтобы он ссылался на предыдущий элемент (или на NULL, если его не было)
- Перекинуть указатель head на новый узел.

1) Создаём новый узел

Создали новый узел, на который ссылается локальная переменная tmp

2) Присваиваем ему значение

Присвоили ему значение

3) Присваиваем указателю tmp адрес предыдущего узла

Перекинули указатель tmp на предыдущий узел

4) Присваиваем указателю head адрес нового узла

Перекинули указатель head на вновь созданный узел tmp

5) После выхода из функции переменная tmp будет уничтожена. Получим список, в который будет вставлен новый элемент.

Новый узел добавлен

?

```
void push(Node **head, int data) {
   Node *tmp = (Node*) malloc(sizeof(Node));
   tmp->value = data;
   tmp->next = (*head);
   (*head) = tmp;
}
```

Так как указатель head изменяется, то необходимо передавать указатель на указатель.

Теперь напишем функцию pop: она удаляет элемент, на который указывает head и возвращает его значение.

Если мы перекинем указатель head на следующий элемент, то мы потеряем адрес первого и не сможем его удалить и тем более вернуть его значения. Для этого необходимо сначала создать локальную переменную, которая будет хранить адрес первого элемента

Локальная переменная хранит адрес первого узла

Уже после этого можно удалить первый элемент и вернуть его значение

Перекинули указатель head на следующий элемент и удалили узел

?

```
int pop(Node **head) {
   Node* prev = NULL;
   int val;
   if (head == NULL) {
      exit(-1);
   }
   prev = (*head);
   val = prev->value;
   (*head) = (*head)->next;
   free(prev);
```

```
    return val;
}
```

Не забываем, что необходимо проверить на NULL голову.

Таким образом, мы реализовали две операции push и pop, которые позволяют теперь использовать односвязный список как стек. Теперь добавим ещё две операции - pushBack (её ещё принято называть shift или enqueue), которая добавляет новый элемент в конец списка, и функцию popBack (unshift, или dequeue), которая удаляет последний элемент списка и возвращает его значение.

Ход работы

Необходимо реализовать функции getLast, которая возвращает указатель на последний элемент списка, и nth, которая возвращает указатель на n-й элемент списка.

Так как мы знаем адрес только первого элемента, то единственным способом добраться до n-го будет последовательный перебор всех элементов списка. Для того, чтобы получить следующий элемент, нужно перейти к нему через указатель next текущего узла

?

```
Node* getNth(Node* head, int n) {
    int counter = 0;
    while (counter < n && head) {
        head = head->next;
        counter++;
    }
    return head;
}
```

Переходя на следующий элемент не забываем проверять, существует ли он. Вполне возможно, что был указан номер, который больше размера списка. Функция вернёт в таком случае NULL. Сложность операции O(n), и это одна из проблем односвязного списка.

Для нахождение последнего элемента будем передирать друг за другом элементы до тех пор, пока указатель next одного из элементов не станет равным NULL

?

```
Node* getLast(Node *head) {
    if (head == NULL) {
        return NULL;
    }
    while (head->next) {
        head = head->next;
    }
    return head;
}
```

Теперь добавим ещё две операции - pushBack (её ещё принято называть shift или enqueue), которая добавляет новый элемент в конец списка, и функцию popBack (unshift, или dequeue), которая удаляет последний элемент списка и возвращает его значение.

Для вставки нового элемента в конец сначала получаем указатель на последний элемент, затем создаём новый элемент, присваиваем ему значение и перекидываем указатель next старого элемента на новый

?

```
void pushBack(Node *head, int value) {
    Node *last = getLast(head);
    Node *tmp = (Node*) malloc(sizeof(Node));
    tmp->value = value;
    tmp->next = NULL;
    last->next = tmp;
}
```

Односвязный список хранит адрес только следующего элемента. Если мы хотим удалить последний элемент, то необходимо изменить указатель next предпоследнего элемента. Для этого нам понадобится функция getLastButOne, возвращающая указатель на предпоследний элемент.

?

```
Node* getLastButOne(Node* head) {
   if (head == NULL) {
      exit(-2);
   }
   if (head->next == NULL) {
      return NULL;
   }
   while (head->next->next) {
      head = head->next;
   }
   return head;
}
```

Функция должна работать и тогда, когда список состоит всего из одного элемента. Вот теперь есть возможность удалить последний элемент.

?

```
void popBack(Node **head) {
   Node *lastbn = NULL;
   //Получили NULL
   if (!head) {
      exit(-1);
   }
   //Список пуст
   if (!(*head)) {
      exit(-1);
   }
   lastbn = getLastButOne(*head);
   //Если в списке один элемент
   if (lastbn == NULL) {
      free(*head);
      *head = NULL;
   } else {
      free(lastbn->next);
      lastbn->next = NULL;
   }
}
```

Удаление последнего элемента и вставка в конец имеют сложность O(n).

Можно написать алгоритм проще. Будем использовать два указателя. Один – текущий узел, второй – предыдущий. Тогда можно обойтись без вызова функции getLastButOne:

?

```
int popBack(Node **head) {
   Node *pFwd = NULL;  //текущий узел
   Node *pBwd = NULL;  //предыдущий узел
   //Получили NULL
   if (!head) {
      exit(-1);
   }
   //Список пуст
   if (!(*head)) {
      exit(-1);
   }

   pFwd = *head;
   while (pFwd->next) {
      pBwd = pFwd;
      pFwd = pFwd->next;
   }
```

```
    if (pBwd == NULL) {
        free(*head);
        *head = NULL;
    } else {
        free(pFwd->next);
        pBwd->next = NULL;
    }
}
```

Теперь напишем функцию insert, которая вставляет на n-е место новое значение. Для вставки, сначала нужно будет пройти до нужного узла, потом создать новый элемент и поменять указатели. Если мы вставляем в конец, то указатель next нового узла будет указывать на NULL, иначе на следующий элемент

?

```
void insert(Node *head, unsigned n, int val) {
    unsigned i = 0;
    Node *tmp = NULL;
    //Находим нужный элемент. Если вышли за пределы списка, то выходим из цикла,
    //ошибка выбрасываться не будет, произойдёт вставка в конец
    while (i < n && head->next) {
        head = head->next;
        i++;
    }
    tmp = (Node*) malloc(sizeof(Node));
    tmp->value = val;
    //Если это не последний элемент, то next перекидываем на следующий узел
    if (head->next) {
        tmp->next = head->next;
    //иначе на NULL
    } else {
        tmp->next = NULL;
    }
    head->next = tmp;
}
```

Покажем на рисунке последовательность действий

Создали новый узел и присвоили ему значение

После этого делаем так, чтобы новый элемент ссылался на следующий после n-го

Теперь значение next нового узла хранит адрес того же узла, что и элемент, на который ссылается head

Перекидываем указатель next n-го элемента на вновь созданный узел

Теперь узел, адрес которого хранит head, указывает на новый узел tmp

Функция удаления элемента списка похожа на вставку. Сначала получаем указатель на элемент, стоящий до удаляемого, потом перекидываем ссылку на следующий элемент за удаляемым, потом удаляем элемент.

?

```
int deleteNth(Node **head, int n) {
    if (n == 0) {
        return pop(head);
    } else {
        Node *prev = getNth(*head, n-1);
        Node *elm  = prev->next;
        int val = elm->value;

        prev->next = elm->next;
        free(elm);
        return val;
```

```
    }
}
```

Рассмотрим то же самое в картинках. Сначала находим адреса удаляемого элемента и того, который стоит перед ним

Для удаления узла, на который ссылается elm необходим предыдущий узел, адрес которого хранит prev

После чего прокидываем указатель next дальше, а сам элемент удаляем.

Прекидываем указатель на следующий за удалённым узел и освобождаем память

Кроме создания списка необходимо его удаление. Так как самая быстрая функция у нас этот pop, то для удаления будем последовательно выталкивать элементы из списка.

?

```
void deleteList(Node **head) {
    while ((*head)->next) {
        pop(head);
        *head = (*head)->next;
    }
    free(*head);
}
```

Вызов pop можно заменить на тело функции и убрать ненужные проверки и возврат значения

?

```
void deleteList(Node **head) {
    Node* prev = NULL;
    while ((*head)->next) {
        prev = (*head);
        (*head) = (*head)->next;
        free(prev);
    }
    free(*head);
}
```

Осталось написать несколько вспомогательных функций, которые упростят и ускорят работу. Первая - создать список из массива. Так как операция push имеет минимальную сложность, то вставлять будем именно с её помощью. Так как вставка произойдёт задом наперёд, то массив будем обходить с конца к началу:

?

```
void fromArray(Node **head, int *arr, size_t size) {
    size_t i = size - 1;
    if (arr == NULL || size == 0) {
        return;
    }
    do {
        push(head, arr[i]);
    } while(i--!=0);
}
```

И обратная функция, которая возвратит массив элементов, хранящихся в списке. Так как мы будем создавать массив динамически, то сначала определим его размер, а только потом запихнём туда значения.

?

```
int* toArray(const Node *head) {
    int leng = length(head);
    int *values = (int*) malloc(leng*sizeof(int));
    while (head) {
        values[--leng] = head->value;
        head = head->next;
    }
    return values;
```

```
}
```

И ещё одна функция, которая будет печатать содержимое списка

?

```
void printLinkedList(const Node *head) {
   while (head) {
      printf("%d ", head->value);
      head = head->next;
   }
   printf("\n");
}
```

Теперь можно провести проверку и посмотреть, как работает односвязный список

?

```
void main() {
   Node* head = NULL;
   int arr[] = {1,2,3,4,5,6,7,8,9,10};
   //Создаём список из массива
   fromArray(&head, arr, 10);

   printLinkedList(head);

   //Вставляем узел со значением 333 после 4-го элемента (станет пятым)
   insert(head, 4, 333);
   printLinkedList(head);

   pushBack(head, 11);
   pushBack(head, 12);
   pushBack(head, 13);
   pushBack(head, 14);
   printLinkedList(head);

   printf("%d\n", pop(&head));
   printf("%d\n", popBack(&head));

   printLinkedList(head);
   //Удаляем пятый элемент (индексация с нуля)
   deleteNth(&head, 4);
   printLinkedList(head);
   deleteList(&head);

   getch();
}
```

Практическая работа № 1.22. Использование основных шаблонов

Цель работы: область применения основных шаблонов

Теоретический материал

Любая структура данных и алгоритмы имеют дополнительную ценность, если они могут хранить и работать с данными различных типов. Такая универсальность (или что одно и то же, независимость от данных) может быть достигнута в Си++ различными способами:

· в обычном Си (см. **9.3**) «переход от типа к типу» и абстрагирование от конкретного хранимого типа возможно на основе преобразования типов указателей и использования указателя **void*,** олицетворяющего «память вообще». Совместно с механизмом динамического связывания возможно создание алгоритмов, в которых фрагмент, ответственный за работу с конкретным типом данных, передается через *указатель на функцию;*

· в Си++ на основе **полиморфизма** (виртуальных функций) возможно создание интерфейсных классов, способных объединять единым механизмом доступа различные классы. Если эти классы являются «обертками» известных типов данных, то независимость от типов хранимых данных можно обеспечить ссылками на интерфейсный класс.

Однако приведенные выше способы не обеспечивают **синтаксической** совместимости, т.е. они реализуются как технологические приемы, а не как элементы языка. По аналогии с переопределением операций (см. **10.3**) хотелось бы использование естественного синтаксиса, где вместо **int, double** и т.п. фигурировало бы абстрактное обозначение типа, например, **T.**

Такое средство, позволяющее создавать заготовку функции или целого класса, в котором вместо конкретного имени типа данных будет фигурировать его символическое обозначение, называется **шаблоном**. В первом приближении смоделировать шаблон в обычном Си можно с использованием директив препроцессора для подстановки имен – **define.** Обозначив именами **T** и **N** тип и размерность массива, можно создать класс с использованием этих имен везде, где это необходимо.

```
//-------------------------------------------------------105-01.cpp
// Имитация шаблона в обычном Си
#define T int                // Параметры заданы через подстановку имен
#define N 100               // Тип элементов и размерность массива
struct Array{
        T Data[N];
        int k;                      // Текущее кол-во элементов
        Array(){ k=0; }             // Конструктор - массив пуст
        void add(T &v){             // Добавление элемента
                if (k>=N) return;
                Data[k++]=v;}
        T remove(int m){ // Удаление элемента по номеру
                T foo;
                if (m>=k) return foo;
                foo=Data[m];
                for(int i=m;i<k-1;i++)
                        Data[i]=Data[i+1];
                k--; return foo; }
};
void main(){
        Array A;
        int B[10]={6,2,8,3,56,7,89,5,7,9};
        for (int i=0;i<10;i++) A.add(B[i]);
        cout << A.remove(2) << endl; }
```

Чтобы применить ее для другого типа данных, нужно отредактировать директиву **define,** заменив, например, имя **int** на **double.** Но все же основным недостатком этой модели является однократность ее использования. В Си++ текстовая заготовка класса позволяет из одного описания создавать множество классов, отличающихся типом используемых объектов и размерностями данных.

Итак, шаблон можно определить как *текстовую заготовку определения класса с параметром, обозначающим тип используемых внутри переменных.* Сразу же, не дожидаясь описания синтаксиса, отметим особенности трансляции шаблона:

· шаблон является описанием *группы классов,* отличающихся используемым типом данных;

· при создании (определении) объекта шаблонного класса указывается тип данных, для которого он создается;

· при определении объекта шаблонного класса конкретный тип подставляется в шаблон вместо параметра и создается текстовая заготовка *экземпляра класса,* которая транслируется и дает собственный оригинальный программный код;

· и только затем происходит трансляция определения самого объекта.

Самое главное, в отличие от обычных объектов, объект шаблонного класса требует отдельного экземпляра класса для того типа, который обозначен в объекте.

Замечание: при чтении транслятором шаблона заголовка класса и шаблонов встроенных в него функций и методов их трансляция **не производится.** Это происходит в другое время: при трансляции определения объекта шаблонного класса генерируется текстовая заготовка экземпляра класса. Отсюда некоторый нюансы:

· чтобы проверить, как транслируется шаблон, нужно описать хотя бы один объект шаблонного класса;

· весь шаблон, в том числе и шаблоны методов, следует размещать в заголовочном файле проекта;

· на каждый тип данных – параметр шаблона создается отдельный класс и в сегменте команд размещается программный код его методов.

Следующим примером попробуем «убить двух зайцев». Во-первых, пояснить довольно витиеватый синтаксис шаблона, а во-вторых, выделить особенности реализации структур данных и использованием технологии ООП. Основной принцип шаблона, добавление к имени класса «довеска» в виде имени – параметра (например, **vector<T>**). Это имя обозначает внутренний тип данных, который может использоваться в любом месте класса: как указатель, ссылка, формальный параметр, результат, локальная или статическая переменная. Во всем остальном шаблон не отличается от обычного класса. Само имя шаблона (**vector**) теперь обозначает не один класс, а группу классов, отличающихся внутренним типом данных.

```
//-----------------------------------------------------105-02.cpp
//----- Шаблон СД - динамический массив указателей
template <class T> class vector{
        int sz,n;                       // Размерность ДМУ и кол-ко элементов
        T **obj;                  // Массив указателей на параметризованные
public:                           // объекты типа T
        T *operator[](int);         // оператор [int] возвращает указатель на
                              // параметризованный объект типа T
        operator int();           // Возвращает текущее количество указателей
        int append(T*);           // Добавление указателя на объект типа T
        int index(T*);            // Поиск индекса хранимого объекта
        vector(int);              // Конструктор
        ~vector(){ delete []obj; } // Деструктор
        };
```

Данный шаблон может использоваться для порождения объектов-векторов, каждый из которых хранит указатели объекты определенного типа. Имя класса при этом составляется из имени шаблона **vector** и имени типа данных (класса), который подставляется вместо параметра **T.**

```
        vector<int>       a;
        vector<double>  b;
        extern class      time;
        vector<time>     c;
```

При определении каждого вектора с новым типом объектов транслятором генерируется описание нового класса по заданному шаблону (естественно, неявно в процессе трансляции). Например, для типа **int** транслятор получит.

```
class vector<int>{
        int sz,n;                       // Размерность ДМУ и кол-ко элементов
        int **obj;                // Массив указателей на параметризованные
public:                           // объекты типа T
        int *operator[](int);       // оператор [int] возвращает указатель на
                              // параметризованный объект типа T
        operator int();           // Возвращает текущее количество указателей
        int append(int *);        // Добавление указателя на объект типа T
```

```
int index(int *);            // Поиск индекса хранимого объекта
vector(int);                 // Конструктор
~vector(){ delete []obj; } // Деструктор
};
```

Обратите внимание, что это *иллюстрация* принципа подстановки, а не фрагмент программы с синтаксисом Си++. Далее следует утверждение типа «масло масляное»: встроенные функции (методы) шаблонного класса – есть шаблонные функции. Это означает, что методы класса, включенные в шаблон, также должны «заготовками» с тем же самым параметром, то есть генерироваться для каждого нового типа данных. То же самое касается переопределяемых операторов.

```
//-----------------------------------------------------105-02.cpp
template <class T> vector<T>::operator int()
{ return n; }
template <class T> T* vector<T>::operator[](int k){
        return k>=n ? NULL : obj[k]; }
template <class T> int vector<T>::index(T *pobj){
        for ( int i=0; i<n; i++)
                if (pobj == obj[i]) return i;
        return -1;}
template <class T> vector<T>::vector(int sz0){
        sz=sz0; n=0; obj=new T*[sz]; }
template <class T> int vector<T>::append(T *pobj){
 if (n>=sz) return 0;
 obj[n++]=pobj;
 return 1;}
```

Приведенный пример касается только методов, «вынесенных» из заголовка класса. Для каждого из них пишется отдельный шаблон, а сам класс фигурирует в нем под именем вида **vector<T>.** Возможность непосредственного определения методов в заголовке шаблонного класса (inline-методов) также остается.

Шаблоны могут иметь также и параметры-константы, которые используются для статического определения размерностей внутренних структур данных. Кроме того, шаблон может использоваться для размещения не только указателей на параметризованные объекты, но и самих объектов. В качестве примера рассмотрим шаблон для построения циклической очереди ограниченного размера на основе статического массива (см. **6.1**), хранящей непосредственно сами объекты (значения, а не указатели).

```
//-----------------------------------------------------105-03.cpp
//------- Шаблон с параметром-константой
template <class T,int size> class FIFO{
        int fst,lst;                    // Индексы начала и конца очереди
        T queue[size];                  // Массив объектов класса T размерности size
public:
        T from();                       // Функции включения-исключения
        int into(T);                    //
        FIFO(){ fst=lst=0; }            // Конструктор
        };
template <class T,int size> T FIFO<T,size>::from(){
        T work=0;
        if (fst !=lst){
                work = queue[lst++];
                if (lst==size) lst=0;
                }
        return work;}
template <class T,int size> int FIFO<T,size>::into(T obj) {
```

```
if ((fst+1)%size==lst)
        return 0;                    // Проверка на переполнение
queue[fst++] = obj;
if (fst==size) fst=0;
return 1; }
```

Объекты такого шаблонного класса при определении имеют два параметра: тип данных и константу – статическую размерность.

```
struct x {…};
FIFO<double,100>        a;
FIFO<int,20>            b;
FIFO<x,50>              c;
```

Особенности разработки шаблонов структур данных

Шаблон является исключительно средством подстановки, он ничего не меняет в ни в существе класса структуры данных, для которой строится шаблон, ни в принципах передачи объектов класса – параметров шаблона. Во-первых, это касается способа хранения объектов в структуре данных: она может содержать указатели, а может и сами объекты (копии, значения):

- если шаблон хранит указатели на объекты, то он не касается проблем корректного копирования объектов и «не отвечает» за их создание и уничтожение. Деструктор шаблона обязан уничтожить динамические компоненты структуры данных (динамические массивы указателей, элементы списка), но он обычно не уничтожает хранимые объекты;
- если шаблон хранит сами объекты, то он «должен быть уверен» в корректном копировании объектов при их записи и чтении из структуры данных (конструктор копирования о переопределение присваивания для объектов, содержащих динамические данные). При разрушении структуры данных разрушаются и копии хранимых в ней объектов.

Во-вторых, шаблон может использовать ряд стандартных операций по отношению к типу данных – параметру шаблона: *присваивание (копирование), сравнение*, а также *ввод и вывод* в стандартные потоки. При использовании шаблона с параметром – не базовым типом, а классом, необходимо следить, чтобы эти операции были в нем переопределены.

Шаблоны классов списков

Для представления списка или дерева необходимы две сущности: элементы списка (вершины дерева), связанные между собой и заголовок – указатель на первый элемент списка (корневую вершину). В технологии ООП есть два равноправных решения:

- разрабатывается один класс, объекты которого играют разную роль в процессе работы класса. Первый объект – заголовок, создается программой (статически или динамически), доступен извне и не содержит данных (по крайней мере, в момент конструирования). Остальные объекты, содержащие данные, создаются динамически методами, работающими с первым объектом. Этот вариант имеет некоторые тонкости, связанные с тем, что программа должна уметь различать, где она работает с элементом списка, а где с элементом-заголовком;
- разрабатывается два класса – класс элементов списка и класс списка как такового, содержащего, как минимум, его заголовок. Объекты первого (вспомогательного) класса пользователем (программой, работающей с классом) не создаются. Они все – динамические, и их порождают методы второго (основного) класса;

Ход работы

Рассмотрим первый вариант на примере шаблона односвязного списка. Коллизий, связанных с распознаванием заголовочного элемента и элемента, содержащего данные, не возникает. Все методы, описанные в заголовке класса, применяются только по отношению к заголовочному элементу (в контексте все время фигурирует установка на первый элемент списка в виде **p=next**, либо установка на «предыдущий»-заголовочный в виде **p=this**). Часть простых методов реализована в самом заголовке класса, в них для указателей на элементы списка используется сокращенный контекст вида **list *p**, хотя класс **list** является шаблонным.

```
//-----------------------------------------------------105-04.cpp
template <class T> class list{
        list<T> *next;                  // Указатель на следующий в списке
```

```
        T data;                         // Элемент списка хранит сам объект
        list(T&  v)  { data=v; next=NULL;}  //  Скрытый  конструктор  для  элементов  с
данными
    public:  list() { next=NULL; }              // Конструктор для элемента - заголовка
        ~list(){                            // Деструктор рекурсивный
            if (next!=NULL) delete next;
            }                               // рекурсивное удаление следующего
        void front(T& v){                   // Включение в начало
            list *q=new list(v);
            q->next=next; next=q;
            }
        void end(T &v){                     // Включение в конец
            list *p,*q=new list(v);         // Дойти до последнего
            for(p=this;p->nezt!=NULL;p=p->next);
            p->next=q;                      // Следующий за последним - новый
            }
        void insert(T&,int);                // Включение по логическому номеру
        void insert(T&);                    // Включение с сохранением порядка
        T remove(int);                      // Извлечение по ЛН
        operator int(){                     // Приведение к int = размер списка
            list *q; int n;
            for(q=next,n=0; q!=NULL; n++,q=q->next);
            return n;
            }
        friend ostream &operator<<(ostream&, list<T>&);
        friend istream &operator>>(istream&, list<T>&);
    };                                      // Переопределенный вывод в поток
```

Единственная коллизия возникает в конструкторе и деструкторе. Для заголовочного элемента используется конструктор без параметров. Методы списка, создавая динамические элементы – объекты того же класса, используют закрытый конструктор с параметром – значением, хранимым в элементе списка. В методе удаления элемента из списка кроме логического исключения выбранного элемента из цепочки у него обнуляется указатель на следующего (**p->next=NULL**). Это делается для того, чтобы исключить рекурсивный вызов деструктора при удалении одного элемента списка.

Специфика односвязного списка проявляется в реализации методов вставки по номеру, с сохранением порядка и удаления. Текущий указатель в цикле ссылается на предыдущий элемент списка по отношению к искомому (поэтому, например, используется выражение **q->next->data** для сравнения значения в искомом элементе).

```
//-------------------------------------------------------105-04.cpp
template <class T> void list<T>::insert(T &newdata,int n){
        list<T> *p,*q;
        p=new list<T>(newdata);
        for (q=this; q->next!=NULL && n!=0; n--,q=q->next);
        p->next=q-> next;               // Вставка после текущего
        q->next=p; }                    // Отсчет от заголовка
//-------------------------------------------------------------------------------------
template <class T> void list<T>::insert(T &newdata){
 list<T> *p,*q;
 p=new list<T>(newdata);                // Сравнивать СО СЛЕДУЮЩИМ
 for (q=this; q->next!=NULL && newdata>q->next->data; q=q->next);
 p->next =q->next;                      // Вставка ПОСЛЕ текущего
 q->next=p;}
//-------------------------------------------------------------------------------------
template <class T> T list<T>::remove(int n){
        list<T> *q,*p;                  // Указатель на ПРЕДЫДУЩИЙ
```

```
        for (q=this; q->next!=NULL && n!=0; n--,q=q->next);
        T val;
        if (q->next==NULL) return val;      // Такого нет
        val=q->next->data;
        p=q->next; q->next=p->next;     // Исключить СЛЕДУЮЩИЙ из цепочки
        p->next=NULL;                   // Для исключения рекурсивного деструктора
        delete p;
        return val;}
```

Переопределенные операции ввода-вывода в стандартные потоки поддерживают саморазворачивающийся симметричный формат представления данных, использующий счетчик элементов. При вводе в соответствии с прочитанным значением счетчика многократно читается объект класса **T,** значение которого каждый раз включается в список.

```
//-----------------------------------------------------105-04.cpp
template <class T> ostream &operator<<(ostream &O, list<T> &R){
        list<T> *q;
        O << (int)R << endl;
        for (q=R.next; q!=NULL; q=q->next) O << q->data << endl;
        return O; }
template <class T> istream &operator>>(ostream &I, list<T> &R){
        T val; int n;
        I >> n;
        while(n--!=0){ I >> val; R.front(val); }
        return I; }
```

Практическая работа № 1.23. Использование порождающих шаблонов

Цель работы: ознакомиться с порождающими шаблонами

Теоретический материал

Порождающие шаблоны

Абстрактная фабрика — порождающий шаблон проектирования, позволяющий изменять поведение системы, варьируя создаваемыми объектами, при этом сохраняя интерфейсы. Он позволяет создавать целые группы взаимосвязанных объектов, которые, будучи созданными одной фабрикой, реализуют общее поведение. Шаблон реализуется созданием абстрактного класса Factory, который представляет собой интерфейс для создания компонентов системы (например, для оконного интерфейса он может создавать окна и кнопки). Затем пишутся классы, реализующие этот интерфейс.

Этот шаблон предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

Фабричный метод (Factory Method)

Фабричный метод — порождающий шаблон проектирования, предоставляющий подклассам интерфейс для создания экземпляров некоторого класса. В момент создания наследники могут определить, какой класс создавать. Иными словами, Фабрика делегирует создание объектов наследникам

родительского класса. Это позволяет использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне.

Шаблон определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу делегировать создание подклассов. Используется, когда:

- классу заранее неизвестно, объекты каких подклассов ему нужно создавать.
- класс спроектирован так, чтобы объекты, которые он создаёт, специфицировались подклассами.
- класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и планируется локализовать знание о том, какой класс принимает эти обязанности на себя.

Структура:

Product — продукт; определяет интерфейс объектов, создаваемых абстрактным методом. ConcreteProduct — конкретный продукт, реализует интерфейс Product.

Creator — создатель; объявляет фабричный метод, который возвращает объект типа Product. Может также содержать реализацию этого метода «по умолчанию»; может вызывать фабричный метод для создания объекта типа Product.

ConcreteCreator — конкретный создатель; переопределяет фабричный метод таким образом, чтобы он создавал и возвращал объект класса ConcreteProduct.

Одиночка (Singleton)

Одиночка — порождающий шаблон проектирования, гарантирующий, что в однопоточном приложении будет единственный экземпляр класса с глобальной точкой доступа.

Существенно то, что можно пользоваться именно экземпляром класса, так как при этом во многих случаях становится доступной более широкая функциональность.

Глобальный «одинокий» объект — именно объект, а не набор процедур, не привязанных ни к какому объекту — бывает нужен:

- если используется существующая объектно-ориентированная библиотека;
- если есть шансы, что один объект когда-нибудь превратится в несколько;
- если интерфейс объекта (например, игрового мира) слишком сложен, и не стоит засорять основное пространство имён большим количеством функций;
- если, в зависимости от каких-нибудь условий и настроек, создаётся один из нескольких объектов. Например, в зависимости от того, ведётся лог или нет, создаётся или настоящий объект, пишущий в файл, или «заглушка», ничего не делающая.

Поведенческие шаблоны

Стратегия (Strategy)

Стратегия — поведенческий шаблон проектирования, предназначенный для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости. Это позволяет выбирать алгоритм путем определения соответствующего класса. Шаблон Strategy позволяет менять выбранный алгоритм независимо от объектов-клиентов, которые его используют.

Проблема: по типу клиента (или по типу обрабатываемых данных) выбрать подходящий алгоритм, который следует применить. Если используется правило, которое не подвержено изменениям, нет необходимости обращаться к шаблону «стратегия».

Решение: отделение процедуры выбора алгоритма от его реализации. Это позволяет сделать выбор на основании контекста.

Класс Strategy определяет, как будут использоваться различные алгоритмы. Конкретные классы ConcreteStrategy реализуют эти различные алгоритмы. Класс Context использует конкретные классы ConcreteStrategy посредством ссылки на конкретный тип абстрактного класса Strategy. Классы Strategy и Context взаимодействуют с целью реализации выбранного алгоритма (в некоторых случаях классу Strategy требуется посылать запросы классу Context). Класс Context пересылает классу Strategy запрос, поступивший от его класса-клиента.

Наблюдатель (Observer)

Наблюдатель — поведенческий шаблон проектирования. Создает механизм у класса, который позволяет получать экземпляру объекта этого класса оповещения от других объектов об изменении их состояния, тем самым наблюдая за ними.

Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.

При реализации шаблона «наблюдатель» обычно используются следующие классы:

- Observable — интерфейс, определяющий методы для добавления, удаления и оповещения наблюдателей;
- Observer — интерфейс, с помощью которого наблюдатель получает оповещение;
- ConcreteObservable — конкретный класс, который реализует интерфейс Observable;
- ConcreteObserver — конкретный класс, который реализует интерфейс Observer.

Шаблон «наблюдатель» применяется в тех случаях, когда система обладает следующими свойствами:

- существует, как минимум, один объект, рассылающий сообщения;

●имеется не менее одного получателя сообщений, причём их количество и состав могут изменяться во время работы приложения;

●нет надобности очень сильно связывать взаимодействующие объекты, что полезно для повторного использования.

Данный шаблон часто применяют в ситуациях, в которых отправителя сообщений не интересует, что делают получатели с предоставленной им информацией.

Команда (Command)

Команда — поведенческий шаблон проектирования, используемый при объектно-ориентированном программировании, представляющий действие. Объект команды заключает в себе само действие и его параметры.

Паттерн обеспечивает обработку команды в виде объекта, что позволяет сохранять её, передавать в качестве параметра методам, а также возвращать её в виде результата, как и любой другой объект.

Паттерн Command преобразовывает запрос на выполнение действия в отдельный объект-команду. Такая инкапсуляция позволяет передавать эти действия другим функциям и объектам в качестве параметра, приказывая им выполнить запрошенную операцию. Команда – это объект, поэтому над ней допустимы любые операции, что и над объектом.

Интерфейс командного объекта определяется абстрактным базовым классом Command и в самом простом случае имеет единственный метод execute(). Производные классы определяют получателя

запроса (указатель на объект-получатель) и необходимую для выполнения операцию (метод этого объекта). Метод execute() подклассов Command просто вызывает нужную операцию получателя.

Впаттерне Command может быть до трех участников:

●Клиент, создающий экземпляр командного объекта.

●Инициатор запроса, использующий командный объект.

●Получатель запроса.

Сначала клиент создает объект ConcreteCommand, конфигурируя его получателем запроса. Этот объект также доступен инициатору. Инициатор использует его при отправке запроса, вызывая метод execute(). Этот алгоритм напоминает работу функции обратного вызова в процедурном программировании

– функция регистрируется, чтобы быть вызванной позднее.

Паттерн Command отделяет объект, инициирующий операцию, от объекта, который знает, как ее выполнить. Единственное, что должен знать инициатор, это как отправить команду. Это придает системе гибкость: позволяет осуществлять динамическую замену команд, использовать сложные составные команды, осуществлять отмену операций.

Ход работы

1. Нарисовать в PlantUML диаграмму классов реализуемой программы. (проектирование)
2. Реализовать программу на Java. (реализация)

Для каждого из шаблонов, предложенных в вариантах можно найти пример реализации UMLсхемы и кода в приложенной книге “Паттерны проектирования”. Также указана глава, где подробно описан данный шаблон.

Шаблон “Стратегия”. Проект “Принтеры”. В проекте должны быть реализованы разные модели принтеров, которые выполняют разные виды печати.

Шаблон “Наблюдатель”. Проект “Оповещение постов ГАИ”. В проекте должна быть реализована отправка сообщений всем постам ГАИ.

Шаблон “Декоратор”. Проект “Универсальная электронная карта”. В проекте должна быть реализована универсальная электронная карта, в которой есть функции паспорта, страхового полиса, банковской карты и т. д.

Практическая работа № 1.24. Использование структурных шаблонов

Цель работы: познакомиться с понятием структурных шаблонов

Теоретические сведения

Структура – это объединение одного либо более объектов (переменных, массивов, указателей, других структур). Как и массив, она представляет собой совокупность данных, но отличается от него тем, что к ее элементам необходимо обращаться по имени, и ее различные элементы не обязательно должны принадлежать одному типу.

Структуры удобно использовать там, где разнообразные данные, относящиеся к одному и тому же объекту, необходимо объединять. Например, ученика средней школы характеризуют следующие данные: фамилия, имя, дата рождения, класс, возраст.

Объявление структуры осуществляется с помощью ключевого слова struct, за которым следует ее тип, список элементов, заключенных в фигурные скобки. Ее можно представить в следующем общем виде:

struct тип {тип элемента 1 имя элемента 1;

тип элемента n имя элемента n; };

Именем элемента может быть любой идентификатор. В одной строке можно записывать через запятую несколько идентификаторов одного типа.

Например:

struct date { int day;

int month;

int year;} ;

Русские буквы использовать в идентификаторе в языке СИ нельзя.

Следом за фигурной скобкой, заканчивающей список элементов, могут записываться переменные данного типа, например:

struct date {…} a, b, c;

При этом выделяется соответствующая память.

Выведенное имя типа можно использовать для объявления записи, например: struct date day;. Теперь переменная day имеет тип date.

Разрешается вкладывать структуры одна на другую. Для лучшего восприятия структуры используем русские буквы в идентификаторах, в языке СИ этого делать нельзя.

Задание 1: Написать программу, проанализировать результат

struct УЧЕНИК { char Фамилия [15];

имя [15];

struct DATA ДАТА РОЖДЕНИЯ;

int класс, возраст;};

Определенный выше тип DATA включает три элемента: День, Месяц, Год, содержащие целые значения (int). Запись УЧЕНИК включает элементы: ФАМИЛИЯ [15]; ИМЯ[15]; ДАТА РОЖДЕНИЯ, КЛАСС, ВОЗРАСТ. ФАМИЛИЯ [15] и ИМЯ [15] – это символьные массивы из 15 компонент каждый. Переменная ДАТА РОЖДЕНИЯ представлена составным элементом (вложенной структурой) ДАТА. Каждой дате рождения соответствуют день месяца, месяц и год. Элементы КЛАСС и ВОЗРАСТ содержат значения целого типа (int). После введения типов ДАТА и УЧЕНИК можно объявить переменные, значения которых принадлежат этим типам.

Задание 2: Написать программу, проанализировать результат struct УЧЕНИК УЧЕНИКИ [50];

массив УЧЕНИКИ состоит из 50 элементов типа УЧЕНИК.

В языке СИ разрешено использовать массивы структуры; записи могут состоять из массивов и других записей.

Чтобы обратиться к отдельному компоненту структуры, необходимо указать ее имя, поставить точку и сразу за ней написать имя нужного элемента.

Задание 3: Написать программу, проанализировать результат Ученики [1]. КЛАСС = 3;

Ученики [1]. ДАТА РОЖДЕНИЯ. ДЕНЬ=5;

Ученики [1]. ДАТА РОЖДЕНИЯ. МЕСЯЦ=4;

Ученики [1]. ДАТА РОЖДЕНИЯ. ГОД=1979;

Первая строка указывает, что 1-й ученик учится в третьем классе, а последующие строки – его дату рождения: 5.04.79.

Каждый тип элемента структуры определяется соответствующей строкой объявления в фигурных скобках. Например, массив УЧЕНИКИ имеет тип УЧЕНИК, год является целым числом. Так как каждый элемент записи относится к определенному типу, его составное имя

может появляться везде, где разрешено использовать значение этого типа. Рассмотрим пример программы:

```
/* Демонстрация записи */
#include < stdio.h >
struct computer { int mem;
int sp;
char model [20]; };
/* Объявление записи типа computer, состоящей из трех элементов: mem, sp, model */
struct computer pibm =
{512, 1, “ПЭВМЕС 1840.05”}
/* Объявление и инициализация переменной pibm типа computer */
main ( )
{ printf (“ персональная ЭВМ % s\n\n “, pibm.model);
printf ( “объем оперативной памяти - % d К байт \n”, pibm.mem);
printf (“производительность - % d млн. операций в секунду \n”, pibm.sp);
/* вывод на экран значений элементов структуры */
}
```

В данной программе объявляется запись computer, которая состоит из трех элементов: mem (память ЭВМ), sp (быстродействие), model [20] (модель ПЭВМ). Переменная pibm имеет тип computer и является глобальной. Строки pibm.model, pibm.mem, pibm. sp в операторе printf вызывают обращение к соответствующим элементам записи pibm типа computer, которым ранее были присвоены определенные значения.

Результат работы программы имеет вид:

персональная ЭВМ ПЭВМ ЕС 1840.05

объем оперативной памяти – 512 К байт

производительность – 1 млн. операций в секунду

Рассмотрим использование в программе вложенных структур:

```
/* Демонстрация вложенных структур*/
# include < stdio.h >
struct date { int day;
int month;
int year; };
/* Объявление записи типа date*/
struct person { char fam [20];
char im [20];
char ot [20];
struct date f1;};
/* Объявление структуры типа person; одним из элементов записи person является запись
f1
типа date */
main ( )
{ struct person ind1;
/* обьявление переменной ind1 типа person */
printf ( “Укажите фамилию, имя, отчество, день, \n месяц”
“ и год рождения гражданина ind1\n”);
scanf (“ % S % S % S %d %d”, &ind1.fam, &ind1.im, &ind1.ot,
& ind1.f1.day, &ind1.f1.month, &ind1.f1.year );
/* Ввод сведений о гражданине ind1 */
printf (“ Фамилия, имя, отчество: % S % S % S \n”, ind1.fam, ind1.im, ind1.ot);
printf (“ Годрождения - % d \n”, ind1.f1.year);
printf (“ Месяцрождения - % d -й \n”, ind1.f1.month);
printf (“ День рождения - % d -й \n”, ind1.f1.day);
/* Вывод сведений о гражданине ind1 */
}
```

Структура типа date (дата) содержит три элемента: day (день), month (месяц), year (год). Структура типа person (человек) содержит четыре элемента: fam[20] (фамилия), im[20] (имя) , ot[20] (отчество), f1 (дата рождения). Последний из них (f1) – это вложенная запись типа date.

Результаты работы программы:

Укажите фамилию, имя, отчество, день, месяц и год рождения гражданина ind1

Алексеев

Сергей

Петрович

3

5

1978

Подчеркнутая информация вводится пользователем.

Сведения о гражданине ind1

Фамилия, имя, отчество: Алексеев Сергей Петрович

Год рождения – 1978

Месяц рождения – 5-й

День рождения – 3-й

В следующей программе рассмотрим использование структуры в виде элементов массива pibm. Каждый элемент состоит из следующих компонентов: mem (память), sp (объем винчестера), model [20] (модель ПЭВМ):

```
/* Массивы записей */
#include < stdio.h >
struct computer { int mem, sp;
char model [20];
pibm [10];};
/* объявлениезаписиитипа computer;
объявление массива pibm типа computer */
main ( )
{ int i, j, k, priz;
for ( i=0; i<10; i++)
{ printf (“Введите сведения о ПЭВМ %d и признак (0-конец;
\n другая цифра- продолжение)\n”, i);
printf (“ модельПЭВМ - ”);
scanf (“%S”, &pibm [i].model );
printf ( “объемоперативнойпамяти -”);
scanf (“%d”, &pibm[i].mem);
printf (“ объемвинчестера - ”);
scanf ( “%d , &pibm[i].sp ”);
printf (“признак - ”);
scanf (“ %d ”, &priz );
k=i;
if (!priz) break; }
/* Здесь !priz – операцияотрицания priz; break – выходизцикла for, если priz=0 */
for (i=0; i<10, i++);
{
printf ( “\n О какой ПЭВМ Вы хотите получить сведения?\n (Введите номер от 0 до 9)\n”
);
scanf ( “%d ”,&j );
if (j>k)
{ printf (“Нет сведений об этой ПЭВМ \n”);
continue; }
printf (“ персональнаяЭВМ %s\n ”, pibm[j].model);
printf (“объем оперативной памяти - % d Мб \n ”, pibm[j].mem);
printf (“объем винчестера - % d Мб \n ”, pibm[j].sp);
printf (“ признак – “ );
scanf ( “ %d ”, &priz);
```

if (!priz) break; }

/* Ввод сведений о ПЭВМ и занесение в массив pibm записей типа computer (первый цикл for); вывод на экран сведений о ПЭВМ (второй цикл for) */

}

Результаты работы программы:

Введите сведения о ПЭВМ и признак (0-конец; другая цифра – продолжение)

модель ПЭВМ – AT 486 SX

объем оперативной памяти – 32

объем винчестера – 4 Гбайта

признак – 1

Введите сведения о ПЭВМ и признак (0-конец; другая цифра – продолжение)

модель ПЭВМ – AT 386 DX

объем оперативной памяти – 64

объем винчестера – 14 Гбайт

признак – 0

О какой ПЭВМ Вы хотите получить сведения? (Введитн номер от 0 до 9)

1

модель ПЭВМ – AT 386 DX

объем оперативной памяти – 16 Мб

объем винчестера – 2,5 Гбайт

признак – 0

Практическая работа № 1.25. Использование поведенческих шаблонов

Цель работы: изучить механизм работы поведенческих шаблонов

Теоретический материал

Поведенческие шаблоны — шаблоны проектирования, определяющие алгоритмы и способы реализации взаимодействия различных объектов и классов.

Поведенческие шаблоны:

1. цепочка обязанностей (Chain of Responsibility);
2. команда (Command);
3. итератор (Iterator);
4. посредник (Mediator);
5. хранитель (Memento);
6. наблюдатель (Observer);
7. посетитель (Visitor);
8. стратегия (Strategy);
9. состояние (State);
10. шаблонный метод (Template Method).

Цепочка обязанностей — поведенческий шаблон проектирования предназначенный для организации в системе уровней ответственности.

Пример из жизни: например, у вас есть три платежных метода (A, B и C), настроенных на вашем банковском счёте. На каждом лежит разное количество денег. На A есть 100 долларов, на B есть 300 долларов и на C — 1000 долларов. Предпочтение отдается в следующем порядке: A, B и C. Вы пытаетесь заказать что-то, что стоит 210 долларов. Используя цепочку обязанностей, первым на возможность оплаты будет проверен метод A, и в случае успеха пройдет оплата и цепь разорвется. Если нет, то запрос перейдет к методу B для аналогичной проверки. Здесь A, B и C — это звенья цепи, а все явление — цепочка обязанностей.

Простыми словами: цепочка обязанностей помогает строить цепочки объектов. Запрос входит с одного конца и проходит через каждый объект, пока не найдет подходящий обработчик.

Ход работы

Обратимся к коду. Приведем пример с банковскими счетами. Изначально у нас есть базовый Account с логикой для соединения счетов цепью и некоторые счета:

abstract class Account

{

protected $successor;

```
        protected $balance;

        public function setNext(Account $account)
        {
            $this->successor = $account;
        }

        public function pay(float $amountToPay)
        {
            if ($this->canPay($amountToPay)) {
                echo sprintf('Оплата %s, используя %s' . PHP_EOL, $amountToPay,
get_called_class());
            } elseif ($this->successor) {
                echo sprintf('Нельзя заплатить, используя %s. Обработка ..' . PHP_EOL,
get_called_class());
                $this->successor->pay($amountToPay);
            } else {
                throw new Exception('Ни на одном из аккаунтов нет необходимого количества
денег');
            }
        }

        public function canPay($amount): bool
        {
            return $this->balance >= $amount;
        }
    }

    class Bank extends Account
    {
        protected $balance;

        public function __construct(float $balance)
        {
            $this->balance = $balance;
        }
    }

    class Paypal extends Account
    {
        protected $balance;

        public function __construct(float $balance)
        {
            $this->balance = $balance;
        }
    }

    class Bitcoin extends Account
    {
        protected $balance;

        public function __construct(float $balance)
        {
            $this->balance = $balance;
        }
```

```
}
```

Теперь приготовим цепь, используя объявленные выше звенья (например, Bank, Paypal, Bitcoin):

```
// Подготовим цепь
//     $bank->$paypal->$bitcoin
//
// Первый по приоритету банк
//     Если нельзя через банк, то Paypal
//     Если нельзя через Paypal, то Bitcoin

$bank = new Bank(100);          // Банк с балансом 100
$paypal = new Paypal(200);      // Paypal с балансом 200
$bitcoin = new Bitcoin(300);    // Bitcoin с балансом 300

$bank->setNext($paypal);
$paypal->setNext($bitcoin);

// Попробуем оплатить через банк
$bank->pay(259);

// Вывод
// ==============
// Нельзя заплатить, используя Банк. Обработка ..
// Нельзя заплатить, используя Paypal. Обработка ..:
// Оплата 259, используя Bitcoin!
```

Примеры на Java и Python.

Команда (Command)

Википедия гласит:

Команда — поведенческий шаблон проектирования, используемый при объектно-ориентированном программировании, представляющий действие. Объект команды заключает в себе само действие и его параметры.

Пример из жизни: Типичный пример: вы заказываете еду в ресторане. Вы (т.е. Client) просите официанта (например, Invoker) принести еду (то есть Command), а официант просто переправляет запрос шеф-повару (то есть Receiver), который знает, что и как готовить. Другим примером может быть то, что вы (Client) включаете (Command) телевизор (Receiver) с помощью пульта дистанционного управления (Invoker).

Простыми словами: Позволяет вам инкапсулировать действия в объекты. Основная идея, стоящая за шаблоном — это предоставление средств, для разделения клиента и получателя.

Обратимся к коду. Изначально у нас есть получатель Bulb, в котором есть реализация каждого действия, которое может быть выполнено:

```
// Получатель
class Bulb
{
    public function turnOn()
    {
        echo "Лампочка загорелась";
    }

    public function turnOff()
    {
        echo "Темнота!";
    }
}
```

Затем у нас есть интерфейс Command, который каждая команда должна реализовывать, и затем у нас будет набор команд:


```
interface Command
```

```
{
    public function execute();
    public function undo();
    public function redo();
}

// Команда
class TurnOn implements Command
{
    protected $bulb;

    public function __construct(Bulb $bulb)
    {
        $this->bulb = $bulb;
    }

    public function execute()
    {
        $this->bulb->turnOn();
    }

    public function undo()
    {
        $this->bulb->turnOff();
    }

    public function redo()
    {
        $this->execute();
    }
}

class TurnOff implements Command
{
    protected $bulb;

    public function __construct(Bulb $bulb)
    {
        $this->bulb = $bulb;
    }

    public function execute()
    {
        $this->bulb->turnOff();
    }

    public function undo()
    {
        $this->bulb->turnOn();
    }

    public function redo()
    {
        $this->execute();
    }
}
```

Затем у нас есть Invoker, с которым клиент будет взаимодействовать для обработки любых команд:

```
// Invoker
class RemoteControl
{
    public function submit(Command $command)
    {
        $command->execute();
    }
}
```

Наконец, мы можем увидеть, как использовать нашего клиента:

```
$bulb = new Bulb();

$turnOn = new TurnOn($bulb);
$turnOff = new TurnOff($bulb);

$remote = new RemoteControl();
$remote->submit($turnOn); // Лампочка загорелась!
$remote->submit($turnOff); // Темнота!
```

Шаблон команда может быть использован для реализации системы, основанной на транзакциях, где вы сохраняете историю команд, как только их выполняете. Если окончательная команда успешно выполнена, то все хорошо, иначе алгоритм просто перебирает историю и продолжает выполнять отмену для всех выполненных команд.

Примеры на Java и Python.

Итератор (Iterator)

Википедия гласит:

Итератор — поведенческий шаблон проектирования. Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из агрегированных объектов.

Пример из жизни: Старый радионабор будет хорошим предметом итератора, где пользователь может начать искать сигнал на каком-то канале и затем использовать кнопки переключения на следующий и предыдущий канал для перехода между соответствующими каналами. Или используем пример телевизора, где вы можете нажимать кнопки следующего или предыдущего канала для перехода через последовательные каналы, или, иными словами, они предоставляют интерфейс для итерирования между соответствующими каналами, песнями или радиостанциями.

Простыми словами: Представляет способ доступа к элементам объекта без показа базового представления.

Обратимся к примерам в коде. В PHP очень просто реализовать это, используя SPL (Standard PHP Library). Приводя наш пример с радиостанциями, изначально у нас есть Radiostation:

```
class RadioStation
{
    protected $frequency;

    public function __construct(float $frequency)
    {
        $this->frequency = $frequency;
    }

    public function getFrequency(): float
    {
        return $this->frequency;
    }
}
```


Затем у нас есть итератор:

```
use Countable;
use Iterator;

class StationList implements Countable, Iterator
{
    /** @var RadioStation[] $stations */
    protected $stations = [];

    /** @var int $counter */
    protected $counter;

    public function addStation(RadioStation $station)
    {
        $this->stations[] = $station;
    }

    public function removeStation(RadioStation $toRemove)
    {
        $toRemoveFrequency = $toRemove->getFrequency();
        $this->stations = array_filter($this->stations, function (RadioStation $station) use
($toRemoveFrequency) {
            return $station->getFrequency() !== $toRemoveFrequency;
        });
    }

    public function count(): int
    {
        return count($this->stations);
    }

    public function current(): RadioStation
    {
        return $this->stations[$this->counter];
    }

    public function key()
    {
        return $this->counter;
    }

    public function next()
    {
        $this->counter++;
    }

    public function rewind()
    {
        $this->counter = 0;
    }

    public function valid(): bool
    {
        return isset($this->stations[$this->counter]);
    }
}
```


Пример использования:

```
$stationList = new StationList();

// Добавление станций
$stationList->addStation(new RadioStation(89));
$stationList->addStation(new RadioStation(101));
$stationList->addStation(new RadioStation(102));
$stationList->addStation(new RadioStation(103.2));

foreach($stationList as $station) {
   echo $station->getFrequency() . PHP_EOL;
}

$stationList->removeStation(new RadioStation(89)); // Удалит 89 станцию
```

Примеры на Java и Python.

Посредник (Mediator)

Википедия гласит:

Посредник — поведенческий шаблон проектирования, обеспечивающий взаимодействие множества объектов, формируя при этом слабую связанность, и избавляя объекты, от необходимости явно ссылаться друг на друга.

Пример из жизни: Общим примером будет, когда вы говорите с кем-то по мобильнику, то между вами и собеседником находится мобильный оператор. То есть сигнал передаётся через него, а не напрямую. В данном примере оператор — посредник.

Простыми словами: Шаблон посредник подразумевает добавление стороннего объекта (посредника) для управления взаимодействием между двумя объектами (коллегами). Шаблон помогает уменьшить связанность (coupling) классов, общающихся друг с другом, ведь теперь они не должны знать о реализациях своих собеседников.

Разберем пример в коде. Простейший пример: чат (посредник), в котором пользователи (коллеги) отправляют друг другу сообщения.

Изначально у нас есть посредник ChatRoomMediator:

```
interface ChatRoomMediator
{
   public function showMessage(User $user, string $message);
}

// Посредник
class ChatRoom implements ChatRoomMediator
{
   public function showMessage(User $user, string $message)
   {
      $time = date('M d, y H:i');
      $sender = $user->getName();

      echo $time . '[' . $sender . ']:' . $message;
   }
}
```

Затем у нас есть наши User (коллеги):

```
class User {
   protected $name;
   protected $chatMediator;

   public function __construct(string $name, ChatRoomMediator $chatMediator) {
      $this->name = $name;
      $this->chatMediator = $chatMediator;
   }
```



```
   public function getName() {
```

```
        return $this->name;
    }

    public function send($message) {
        $this->chatMediator->showMessage($this, $message);
    }
}
```

Пример использования:

```
$mediator = new ChatRoom();

$john = new User('John Doe', $mediator);
$jane = new User('Jane Doe', $mediator);

$john->send('Привет!');
$jane->send('Привет!');

// Вывод
// Feb 14, 10:58 [John]: Привет!
// Feb 14, 10:58 [Jane]: Привет!
```

Хранитель — поведенческий шаблон проектирования, позволяющий, не нарушая инкапсуляцию, зафиксировать и сохранить внутреннее состояние объекта так, чтобы позднее восстановить его в этом состоянии.

Пример из жизни: В качестве примера можно привести калькулятор (создатель), у которого любая последняя выполненная операция сохраняется в памяти (хранитель), чтобы вы могли снова вызвать её с помощью каких-то кнопок (опекун).

Простыми словами: Шаблон хранитель фиксирует и хранит текущее состояние объекта, чтобы оно легко восстанавливалось.

Обратимся к коду. Возьмем наш пример текстового редактора, который время от времени сохраняет состояние, которое вы можете восстановить.

Изначально у нас есть наш объект EditorMemento, который может содержать состояние редактора:

```
class EditorMemento
{
    protected $content;

    public function __construct(string $content)
    {
        $this->content = $content;
    }

    public function getContent()
    {
        return $this->content;
    }
}
```

Затем у нас есть наш Editor (создатель), который будет использовать объект хранитель:

```
class Editor
{
    protected $content = '';

    public function type(string $words)
    {
        $this->content = $this->content . ' ' . $words;
    }
```



```
    public function getContent()
```

```
    {
        return $this->content;
    }

    public function save()
    {
        return new EditorMemento($this->content);
    }

    public function restore(EditorMemento $memento)
    {
        $this->content = $memento->getContent();
    }
}
```

Пример использования:

```
$editor = new Editor();

// Печатаем что-нибудь
$editor->type('Это первое предложение.');
$editor->type('Это второе.');

// Сохраняем состояние для восстановления : Это первое предложение. Это второе.
$saved = $editor->save();

// Печатаем ещё
$editor->type('И это третье.');

// Вывод: Данные до сохранения
echo $editor->getContent(); // Это первое предложение. Это второе. И это третье.

// Восстановление последнего сохранения
$editor->restore($saved);

$editor->getContent(); //..
```

:

Практическая работа № 1.26. Разработка приложения с использованием текстовых компонентов

Цель работы: изучить правила работы текстовыми компонентами

Теоретический материал

В языках C и C++ файл рассматривается как поток (stream), представляющий собой последовательность считываемых или записываемых байтов. При этом последовательность записи определяется самой программой.

C++ Builder позволяет работать с файлами тремя различными способами:

Работа с текстовыми компонентами

Каждый файл в программе на C++ должен быть связан с некоторым указателем. Этот указатель имеет тип FILE (определен в stdio.h) и используется во всех операциях с файлами. Синтаксис операции следующий:

```
# include < stdio.h >
FILE *fin, *fout;
```

Для работы с файлом его необходимо открыть функцией **fopen,** первый параметр которой содержит имя файла и путь к нему, а второй - режим открытия файла. Функция возвращает указатель на файл. Часто используемые режимы открытия файла:

r	Открывает файл для чтения

r+	Открывает существующий файл для чтения и записи
w	Создает файл для записи. Если файл уже существует, его содержимое уничтожается
w+	Создает файл для чтения и записи. Если файл уже существует, его содержимое уничтожается
a	Открывает файл для записи данных в конец файла. Если файл отсутствовал, он создается
a+	Открывает файл для чтения или записи данных в конец файла. Если файл отсутствовал, он создается

После режима может добавляться символ "t" - текстовый файл или "Ъ" -бинарный файл. Если символ не указан, то по умолчанию считается что файл текстовый. Например:

```
fin=fopen("a:\\dat.txt","r");
fout=fopen("a:\\out.txt","w");
```

При записи обмен происходит не непосредственно с файлом, а с некоторым буфером. Информация из буфера переписывается в файл только при его переполнении или при закрытии файла.

После окончания работы с файлом он обязательно должен быть закрыт функцией **fclose(FILE** *).

Если файл не удалось открыть, то возвращается нулевой указатель (NULL). Например:

```
# include < stdio.h >
FILE *lw;
if (lw=fopen("a.text","r")) == NULL){
Memo1->Lines->Add("Файл не открыт"); return; }
fclose(lw);
```

Работа с **текстовыми файлами**

Для записи в текстовый файл наиболее часто используется функция fprintf**:**

int *fprintf(FILE *stream, const char *format[]);

где параметр format определяет строку форматирования аргументов**,** заданных своими адресами. Обычно эта строка состоит из последовательности символов "%", после которых следует символ типа данных:

I или i	Десятичное, восьмеричное или шестнадцатеричное целое
D или d	Десятичное целое
U или u	Десятичное целое без знака
E или e	Действительное с плавающей точкой
s	Строка символов
c	Символ

Из отрытого текстового файла можно читать информацию как по строкам. так и посимвольно. Чтение строки осуществляется функцией fgets :

char *fgets(char *st, int n, FILE *stream);

где st - указатель на буфер, в который считается строка; п - число читаемых символов; stream - указатель на файл. Строка читается до тех пор, пока не будет прочитано п-1 символов, или до конца строки \n. В конце прочитанной строки ставится нулевой символ.

Для проверки достижения конца файла используется функция feof(F).

Чтение форматированных данных можно осуществлять с помощью функции fscanf:

int *fscanf(FILE *stream, const char *format[]);

Строка форматирования строится аналогично fprintf.

Следует обратить внимание на то, что при чтении данных всегда указываются адреса переменных (&), а не сами переменные.

Пример записи и чтения данных из файла:

```
# include < stdio.h >
Memo1->Clear(); FILE *lw;
// Запись данных в файл
if ((lw=fopen("a.text","wf)) == NULL) {
```

```
Memo1->Lines->Add("Файл не удалось создать");
return;   }
int num=10; char st[12] = "ИНФОРМАЦИЯ",sr[30];
fprintf(lw,"%s \n В группе %i человек",st, num); fclose(lw);
// Чтение данных из файла
if ((lw=fopen("a.text","rt")) == NULL)  {
Memo1->Lines->Add("Файл не удалось открыть "); return;
return; }
while (!feof(lw)) { fgets(sr,30,lw);
if (sr[strlen(sr)-1] =='\n') sr[strlen(sr)-1]=0; Memo1->Lines->Add(sr);
} fclose(lw);
```

В C++ определены три класса файлового ввода/вывода: **ifstream** - входные данные для чтения; **ofstream** - выходные файлы для записи; **fstream** - файлы для чтения и записи.

Очень удобно применять следующие операции: поместить в поток (<<) и извлечь из потока (>>). Пример программы:

```
#include <fstream.h>
Memo1->Clear();
// Ввод данных в файл
ofstream lw ("a.text");
if (!lw) {Memo1->Lines->Add("Файл не удалось создать "); return;}
int num=10;   double k=5.67;   char s[20] = "ИНФОРМАЦИЯ";
lw << num << " <<k << " << s << endl;
lw.close();
// Чтение данных из файла
ifstream lx ("a.text");
if (!lx){  Memo1->Lines->Add("Файл не удалось открыть ");return;}
Ix » num » k » s;
Memo1->Lines->Add(IlntToStr(num));
Memo1->Lines->Add(FloatToStr(k));
Memo1->Lines->Add(s); lx.close();
```

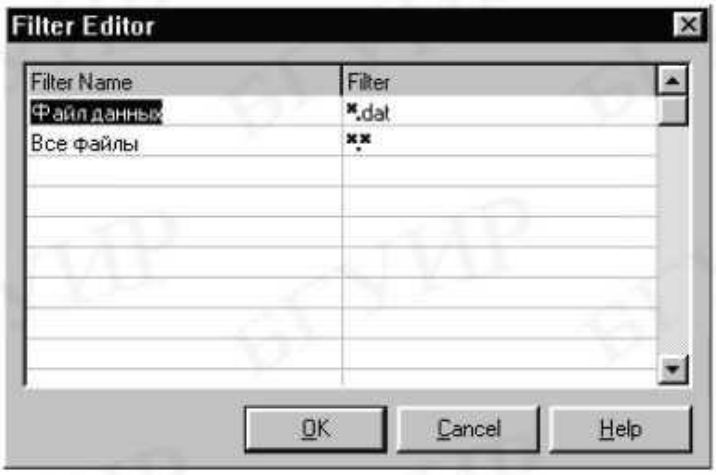
Помимо этих операции поместить в поток можно еще с помощью функций put и write. Возможности ввода/вывода можно существенно расширить, используя манипуляторы потока.

Компоненты TOpenDialog и TSaveDialog

Компоненты TOpenDialog и TSaveDialog находятся на странице DIALOGS. Все компоненты этой страницы являются невизуальными, т.е. не видны в момент работы программы. Поэтому их можно разместить в любом удобном месте формы. Оба рассматриваемых компонента имеют идентичные свойства и различаются только внешним видом. После вызова компонента появляется диалоговое окно, с помощью которого выбирается имя программы и путь к ней. В случае успешного завершения диалога имя выбранного файла и маршрут поиска содержатся в свойстве FileName. Для фильтрации файлов, отображаемых в окне просмотра, используется свойство Filter, а для задания расширения файла, в случае, если оно не задано пользователем, - свойство DefaultExt. Если необходимо изменить заголовок диалогового окна, используется свойство Title.

Для установки компонентов TOpenDialog и TSaveDialog на форму необходимо на странице Dialogs меню компонентов щелкнуть мышью по пиктограммаме и поставить её в любое свободное место формы.

Установка фильтра производится следующим образом. Выбрав соответствующий компонент, дважды щелкнуть по правой части свойства Filter инспектора объектов. Появится окно Filter Editor, в левой части которого записывается текст, характеризующий соответствующий фильтр, а в правой части - маска. Для OpenDialogl установим значения маски, как показано на рис. 7.1. Формат *.dat означает, что будут видны все файлы с расширением dat, а формат *.* - что будут видны все файлы (с любым именем и с любым расширением).

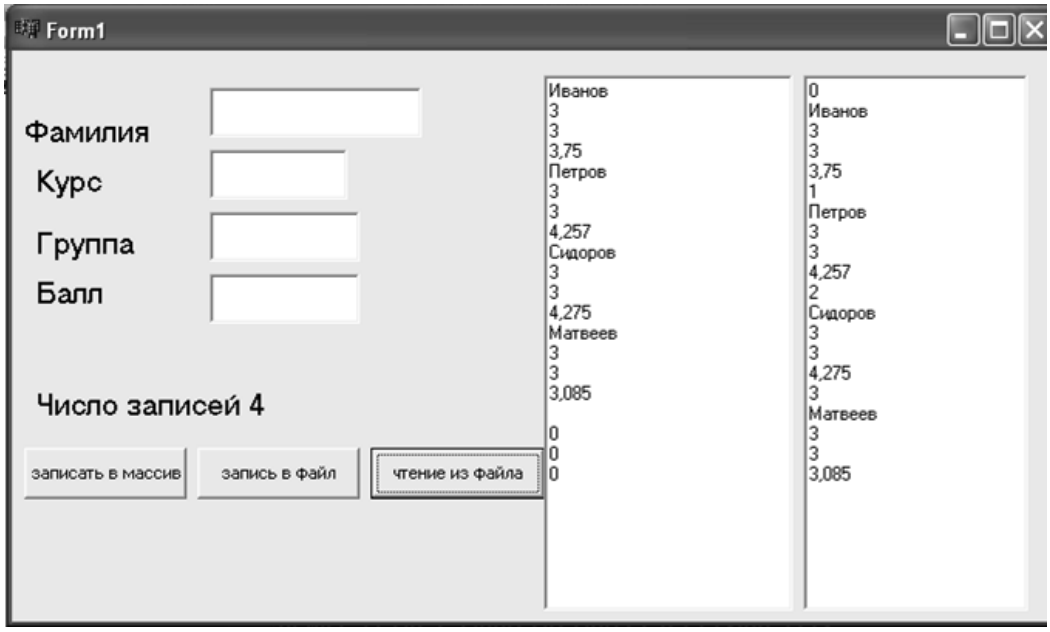


Для того чтобы файл автоматически записывался с расширением .dat, в свойстве DefaultExt запишем требуемое расширение - .dat.

Аналогичным образом настроим SaveDialogl для текстового файла (расширение .txt).

Ход работы

Составить программу, вводящую в файл или читающую из файла ведомость абитуриентов, сдавших вступительные экзамены. Каждая запись должна содержать фамилию, курс, группу и средний балл. Вывести список абитуриентов, записанный в файл и прочитанный из файла. Запись произвести в стиле C++ в текстовый файл.



Unit1.cpp

```
#define n 5                          //ограничим число студентов
struct stud {
char fam[15];
int kurs;
int grup;
float ball;};
stud st[n],sz[n];
int kol=0;
//---------------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender){
if(kol<n){
strcpy(st[kol].fam,Edit1->Text.c_str());
st[kol].kurs=StrToInt(Edit2->Text);
st[kol].grup=StrToInt(Edit2->Text);
st[kol].ball=StrToFloat(Edit4->Text);
Edit1->Text=""; Edit2->Text="";
Edit3->Text=""; Edit4->Text="";
Label5->Caption="Число записей "+IntToStr(kol+1);
kol++;  }
```

```
else Button1->Enabled=false;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{int i=0;   Memo1->Clear();
ifstream in("z1.txt");
if(!in) {ShowMessage("Не удается открыть файл."); }
while(!in.eof())
{
in>>sz[i].fam; in>>sz[i].kurs; in>>sz[i].grup; in>>sz[i].ball;
  if (in)  {
Memo1->Lines->Add(IntToStr(i));
Memo1->Lines->Add(sz[i].fam);
Memo1->Lines->Add(IntToStr(sz[i].kurs));
  Memo1->Lines->Add(IntToStr(sz[i].grup));
  Memo1->Lines->Add(FloatToStrF(sz[i].ball,ffGeneral,5,3));
i++; }
}
in.close();    }
//---------------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{int i;  AnsiString aaa;  Memo2->Clear();
//ofstream out("z1.txt");
ofstream out("z1.txt",ios::app);
for (i=0; i<kol; i++){
out<<st[i].fam<<" "<<st[i].kurs<<" "<<st[i].grup
          <<" "<<st[i].ball<<endl; }
for (i=0; i<5; i++){
Memo2->Lines->Add(st[i].fam);
 Memo2->Lines->Add(IntToStr(st[i].kurs));
  Memo2->Lines->Add(IntToStr(st[i].grup));
  Memo2->Lines->Add(FloatToStrF(st[i].ball,ffGeneral,5,3));
}
out.close();  }
//---------------------------------------------------------------------------
```

Практическая работа № 1.27. Разработка приложения с несколькими формами

Цель работы: изучить способ разработки приложений с несколькими формами

Ход работы

1. Создайте новый проект, выбрав в главном меню пункт **File / New Application**.

2. Измените значение свойства **Name** на **MainForm**, а свойства **Caption**— на **Multiple Forms Test Program**.

3. Сохраните проект. Дайте модулю имя **Main**, а проекту — имя **Multiple**.

4. Теперь поместите в форму кнопку (**Button**). Присвойте свойству **Name** значение **ShowForm2**, а свойству **Caption** — значение **Show Form 2**.

5. Выберите в главном меню пункт **File / New Form** (или щелкните на кнопке **New Form** панели инструментов), чтобы создать новую форму. После создания новая форма будет иметь имя **Form1** и разместится точно поверх главной формы. Надо, чтобы новая форма имела меньший размер и была более-менее отцентрирована относительно главной формы.

6. Измените размер и положение новой формы так, чтобы она была примерно в два раза меньше главной формы и располагалась в ее центре. Для перемещения формы используйте строку заголовка. Размер можно изменить перетаскиванием нижнего правого угла.

7. Измените значение свойства **Name** новой формы на **SecondForm**, а свойства **Caption** — на **A Second Form**.

8. Выберите в главном меню пункт **File / Save** (или щелкните на кнопке **Save File** панели инструментов) и сохраните файл под именем **Second**.

9. Выберите компонент **Label** и поместите его в новую форму. Измените текст свойства **Caption** на **This is the second form**. Измените размер и цвет текста. Отцентрируйте сообщение относительно формы. Теперь ваша форма должна выглядеть примерно как на рис. 3.1.

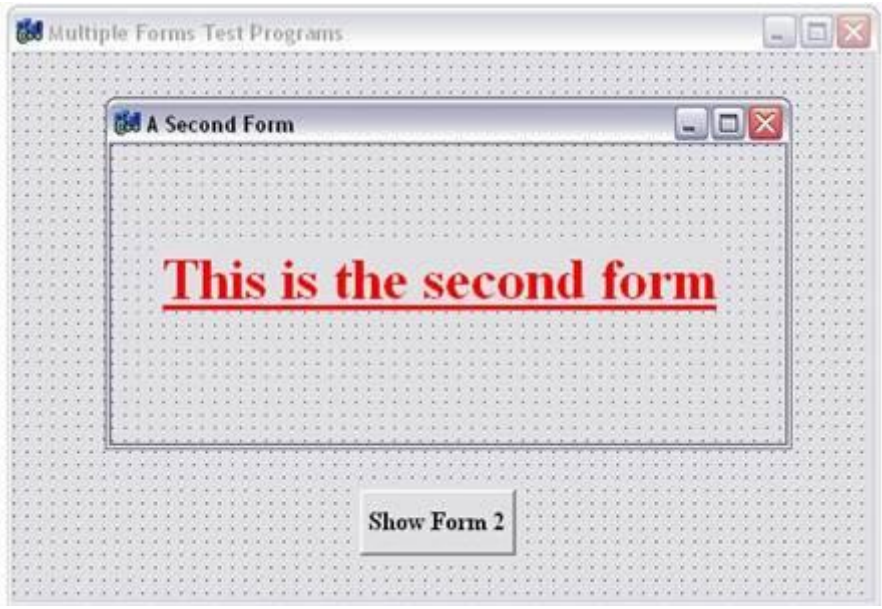


Рис. 3.1 Вид формы к этому моменту

10. Щелкните на главной форме. Обратите внимание, что вторая форма теперь закрыта главной формой. Дважды щелкните на кнопке **Show Form 2**. На экране появится окно **редактора кода** и курсор будет размещен как раз там, где вам нужно начинать ввод текста.

11. Введите текст, приведенный ниже (вам нужно набрать только одну строку в скобках):

```
//---------------------------------------------------------------
void __fastcall TMainForm::ShowForm2Click(TObject *Sender)
{
    SecondForm->ShowModal();
}
//---------------------------------------------------------------
```

12. Запустите программу

Вы получили сообщение об ошибке **Undefined symbol 'SecondForm'** (Не определено обозначение 'SecondForm').

Странно...

SecondFormдолжно быть правильным именем, ведь именно это имя мы дали второй форме...

Дайте подумать...

Ага!

Вспомните, что у нас есть два исходных файла с соответствующими заголовками. Но в модуле **MainForm** нет объявления переменной **SecondForm** (которая является указателем на экземпляр класса **TSecondForm**). Мы должны указать, где расположено объявление класса. Для этого нужно включить заголовочный файл для **SecondForm** в исходный файл **MainForm** с помощью директивы **#include**. Переключитесь в окно редактора кода и щелкните на вкладке **Main.cpp** для отображения модуля главной формы. Перейдите к началу файла. Первые несколько строк должны выглядеть следующим образом:

```
//---------------

#include <vcl.h>
#pragma hdrstop

#include "Main.h"
//---------------
```

Вы видите, что сюда включен заголовочный файл **Main.h**, но нет файла **Second.h**, потому что мы не давали указания **C++Builder**включить этот файл. Давайте сделаем это сейчас.

1. Выберите в главном меню пункт **File / Include Unit Hdr**. На экране появится диалоговое окно **Include Unit**, показанное на рис. 3.2.

Рис.3.2 Диалоговое окно **Include Unit**

2. Перед вами находится список доступных модулей. В данном случае единственным модулем в списке является **Second**. Щелкните на имени **Second**, а затем на кнопке ОК, чтобы закрыть диалоговое окно.

Диалоговое окно **Include Unit** отображает только те модули проекта, которые еще не включены в данный модуль. Включенные модули не содержатся в списке.

После щелчка на кнопке ОК C++Builder мгновенно добавит в файл директиву **#include** для **Second.h**:

```
//-----------------

#include <vcl.h>
#pragma hdrstop

#include "Main.h"
#include "Second.h"
//-----------------
```

Теперь в модуль **Main** включено объявление класса из модуля **Second**. Щелкните на кнопке **Run**, чтобы запустить программу. На этот раз компиляция пройдет без задержек и программа заработает. Когда вы щелкнете на кнопке **Show Form 2**, на экране появится вторая форма.

Как видите, C++Builder помогает вам управлять модулями. Вы должны использовать опцию **Include Unit Hdr**, чтобы обеспечить модулям доступ к объявлениям классов, использующихся в других модулях.

Поместите на форму компоненты **Label1, Edit, Label2, Button1 и Button2** со страницы **Standard** палитры компонент как показано на рис.4.

В метке **Label** имеется свойство **Word Wrap** — допустимость переноса слов длинной надписи, превышающей длину компонента, на новую строчку. Чтобы такой перенос мог осуществляться, надо установить свойство **WordWrap** в **true**, свойство **AutoSize** в **false** (чтобы размер компонента не определялся размером надписи) и сделать высоту компонента такой, чтобы в нем могло поместиться несколько строк. Если **WordWrap** не установлено в **true** при **AutoSize** равном **false**, то длинный текст, не помещающийся в рамке метки, просто обрезается.

Перейдите в обработчик события **OnClick**сделав двойной щелчок на компоненте **Buttonl**на форме.

При вводе из окна числовой информации надо использовать функции **StrToInt(s)** - преобразование строки **s** в целое значение и **StrToFloat(s)**– преобразование строки **s**в значение с плавающей запятой (описание смотрите в лекции). После десятичной точки должно быть 4 цифры.

Затем выполните перевод температур по формуле: C = (5 : 9) * (F— 32), где C - это температура по шкале Цельсия, а F -- по шкале Фаренгейта.

Потом вам надо занести в метку смешанную информацию, состоящую из строк символов и чисел. Для этого воспользуйтесь функциями **FloatToStrF(n)** и **IntToStr(k)**, переводящие соответственно число **n** с плавающей запятой и целое число **k** в строки. При вызове функции **FloatToStrF(n,f,k,m)**указывают: **f** — формат;**k** — точность; **m**— количество цифр после десятичной точки. Формат определяет способ изображения числа: **ffGeneral** — универсальный; **ffExponent** — научный; **ffFixed** — с фиксированной точкой;**ffNumber** — с разделителями групп разрядов; **ffCurrency** — финансовый. Точность — нужное общее количество цифр.

Для формирования текста, состоящего из нескольких фрагментов, можно использовать операцию ”+”, которая для строк означает их склеивание (конкатенацию).

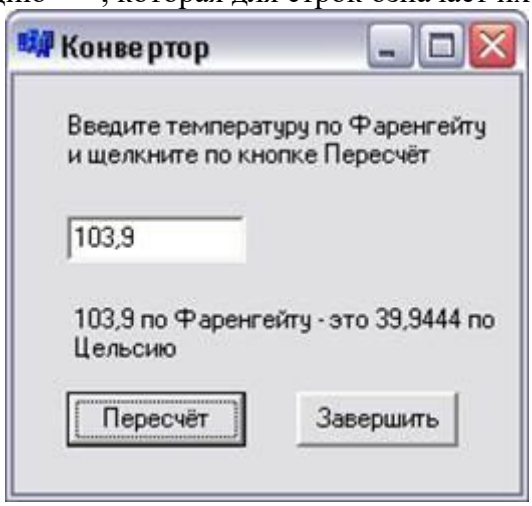


Рис.4 Вид формы конвертора

Сделайте, чтобы в TEdit можно было вводить только числа.

Для этого вам надо поместить следующий код в OnKeyPress любого TEdit'a:

```
// Key – код нажатой клавиши
// проверим является ли символ допустимым
if ((Key >= '0') && (Key <= '9'))// цифра
return;
else if ((Key == '.') || (Key == ','))
{
// DecimalSeparator - глобальная переменная - разделитель целой и дробной части, содержит символ, используемый в качестве разделителя при записи дробных чисел.
if ((Edit1->Text).Pos(DecimalSeparator) != 0)
Key = 0; // разделитель уже введен
```

Получить выполненную работу или консультацию специалиста по вашему учебному проекту


```
else// если ещё нет
Key = DecimalSeparator;
```

```
return;
}
if (Key == VK_BACK)// клавиша <Backspace>
return;
if (Key == VK_RETURN)// клавиша <Enter>
{
Edit1->SetFocus();//делаем компонент активным: в его поле можно набирать и выводить текст;
return;
}
// остальные клавиши запрещены
Key = 0;// код запрещенных символов заменим нулем, в результате символ в поле редактирования не отобразится
```

Для того, чтобы после нажатия на кнопку завершить форма закрылась используйте метод Close(), который закрывает форму.

Практическая работа № 1.28. Разработка приложения с не визуальными компонентами

Цель работы: изучить способ разработки приложения с не визиульными комплнентами.

Ход работы

Задание 1. Установите на форму компонент PopupMenu (находится на странице Standart). Присвойте свойству формы PopupMenu значение PopupMenu1. Дважды щелкните по компоненту PopupMenu1 и в открывшемся конструкторе меню создайте следующие пункты (рис.1).

Создайте обработчики для кнопок:

«Свернуть форму»: Form1.WindowState:=wsMinimized;

«Восстановить форму»: Form1.WindowState:=wsNormal;

Для кнопки «Закрыть» обработчик напишите самостоятельно.

Задание 2. На форме установите Image, OpenPictureDialog.

Создайте меню:

File: Open, (разделитель), Exit, (разделитель) - невидимый, Most Recent-невидимый

Options: Center, Stretch, Transparent

About

```
Var current:string;
procedure TForm1.Open1Click(Sender: TObject);
begin
  if OpenPictureDialog1.Execute then
    begin
     if Current<>'' then
        begin
          MostRecent1.Caption:=Current;
          N2.Visible:=true;
          MostRecent1.Visible:=true;
        end;
     Current:=OpenPictureDialog1.FileName;
     Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
    end;
end;

procedure TForm1.MostRecent1Click(Sender: TObject);
var S : string;
begin
  S:= MostRecent1.Caption;
  Image1.Picture.LoadFromFile(S);
  MostRecent1.Caption:=Current;
```

```
  Current:=S;
end;

procedure TForm1.Center1Click(Sender: TObject);
begin
  Center1.Checked := not Center1.Checked;
  Image1.Center := Center1.Checked;
end;
```

Задание 3. Измените меню File и соответствующие методы для отображения имён и повторного открытия не одной, а трёх последних картинок.

Добавьте в главное меню пункт Lines: Add, (разделитель) – невидимый.
Добавьте методы:

```
procedure TForm1.Add1Click(Sender: TObject);
var I : TMenuItem;
begin
I:=TMenuItem.Create(Lines1);
I.Caption:= Memo1.SelText;
I.OnClick:=Put;
Lines1.Add(I);
N3.Visible:=true;
end;

procedure TForm1.Put(Sender: TObject);
begin
Memo1.Text:=Memo1.Text + (Sender as TMenuItem).Caption;
end;
```

Задание 4. Остальные обработчики событий напишите самостоятельно. Добавьте в меню Text пункт "Alignment: Left", циклически изменяющий свойство Memo1.Alignment и своё свойство Caption для выравнивания по левому краю, по центру и по правому краю.

Практическая работа № 1.29. Разработка игрового приложения

Цель работы: ознакомиться с особенностями использования игровых ресурсов изучить способы разработки игрового приложения

Теоретический материал

Из чего состоит игра: игровые ресурсы

Любая *игра* состоит из *множества* частей. Но среди них можно выделить две основных. Первая – это игровые ресурсы, и вторая – это программный код.

Игровые ресурсы – это графические, музыкальные и иные ресурсы, которые используются для оформления игры. Рассмотрим различные форматы, в которых хранятся игровые ресурсы.

Графические файлы

Графические файлы используются для хранения изображений. Как правило, это – графические представления игровых объектов, которые применяются в двумерных или в псевдотрехмерных играх, когда трехмерный вид игровых объектов достигается лишь художественными средствами, без использования технологий трехмерной графики. Так же эти файлы могут быть использованы для создания элементов оформления игры и как текстуры для наложения на трехмерные модели.

JPEG

JPEG – это стандарт хранения файлов изображений, разработанный специально для хранения цифровых фотографий. *JPEG* расшифровывается как *Joint Photographic Experts*

Group – именно так называлась *рабочая группа*, которая разработала этот стандарт. Особенностью *JPEG* является возможность сжатия изображения с потерями качества. Как правило, в *JPEG* выделяют 10 уровней сжатия (от 1 до 10), однако эти изображения можно сжимать и с более точным указанием уровня сжатия. Чем сильнее сжатие – тем сильнее потери качества. На невысоких уровнях сжатия *JPEG*-файлы практически не содержат так называемых артефактов сжатия. В случае с *JPEG* это проявляется в заметном искажении изображения, особенно – содержащего четкие границы между цветами, четкие линии. Дело в том, что при сжатии по алгоритму *JPEG* изображение разбивается на фрагменты 8x8 пикселей. После этого цветовая информация о пикселях одного квадрата сжимается, яркостная же информация остаётся в более сохранном виде.

Как правило, в *JPEG* хранят различные двумерные изображения, которые можно использовать для организации элементов интерфейса пользователя, для создания различных экранов игры (экран приветствия, экраны с информацией о набранных очках и так далее). В *JPEG* можно хранить изображения (их называют *спрайтами*), которые используются в двумерных играх в качестве игровых объектов (преимущественно – прямоугольной формы), в качестве фоновых изображений.

Лучше всего использовать *JPEG* для хранения цветных изображений с плавными цветовыми переходами. В частности, это могут быть фотографии, нарисованные игровые объекты и так далее.

JPEG-файлы имеют расширение JPG или *JPEG*.

Для создания *JPEG*-файлов с успехом можно использовать популярные графические редакторы – такие, как Adobe Photoshop различных версий.

PNG

Формат *PNG* (*Portable* Network Graphics) отличается от формата JPG тем, что использует алгоритмы сжатия, сжимающие изображение без потери качества. В результате оказывается, что изображения, имеющие четкие цветовые переходы (такие, как схемы, графики, изображения для простых элементов управления) могут быть довольно сильно сжаты в *PNG*-файлах. Если изображение имеет плавные цветовые переходы – лучшим выбором для его сжатия будет JPG, хотя и *PNG* вполне можно использовать. Как и JPG, данный формат поддерживает полноцветные изображения, однако изображение можно создать с использованием нескольких фиксированных цветов, что идет на пользу размеру графического файла, и, естественно, при правильном подборе цветов, не сказывается на его качестве.

Формат *PNG* был разработан для замены устаревшего, но все еще популярного формата GIF. Основное назначение GIF-файлов – простая интернет-графика.

PNG-файлы имеют расширение *.PNG*. Для их создания можно использовать растровые графические редакторы, такие, как Adobe Photoshop.

Еще одна важная особенность формата *PNG* – поддержка прозрачности. То есть при создании файла можно указать, какие из пикселей изображения считать прозрачными. В результате *PNG* идеально подходит для хранения графического представления двумерных игровых объектов сложной формы.

TGA

TGA (Truevision *Graphic Adapter*), *TARGA* - Truevision Advanced *Raster Graphics Adapter* – это графический формат, который используется преимущественно для хранения игровых *текстур*. Этот формат был специально создан для хранения *текстур*, он используется во многих существущих играх именно для этих целей. Он поддерживает полноцветные изображения (8, 16, 24, 32-битные), поддерживают 8-ми битный *альфа-канал* для указания прозрачных участков изображения.

Текстура – это растровое (точечное) изображение, которое накладывается на поверхность полигона, являющегося частью трехмерной модели. Текстуры позволяют придавать трехмерным моделям цвета, создавать видимость рельефа, мелких деталей модели. При проектировании *3D*-модели создание мелких деталей обычно оказывается слишком ресурсоемким, использование же *текстур* позволяет получать привлекательно выглядящие модели, построенные с использованием сравнительно небольшого количества трехмерных элементов. Сегодня трехмерные модели создаются с высоким уровнем детализации – мощности современных компьютеров хватает для реалистичной обработки таких моделей. В играх прошлых лет модели обычно выглядят схематично (например, профиль колеса автомобиля в играх, использующих

низкополигональные модели, может выглядеть как многоугольник, но не как *окружность*), однако текстурирование позволяет даже таким моделям выглядеть достаточно реалистично.

Текстуры различаются глубиной цвета и разрешением – чем выше и то и другое – тем выше качество изображения. В применении к текстурам существует такое понятие, как тексель – число пикселей, приходящееся на минимальную единицу текстуры.

TGA-файлы имеют расширение *.TGA*

Для создания *TGA*-файлов можно применять всё тот же Adobe Photoshop и другие редакторы. Например, бесплатный редактор *Paint*.NET (http://www.getpaint.net/)

DDS

DDS (*Direct* Draw *Surface*) – это графический формат, разработанный специально для использования в *DirectX SDK* (http://msdn.microsoft.com/en-us/library/bb943990(VS.85).aspx). Он предназначен преимущественно для хранения *текстур*. Благодаря особенностям формата – поддержке хранения сжатых и несжатых *текстур*, аппаратной поддержке, этот формат идеален для хранения игровых *текстур*.

DDS-файлы имеют расширение *DDS*. Для их создания можно использовать *плагины* для обычных графических редакторов, для работы с такими файлами созданы специальные наборы утилит и *плагинов*.

BMP

BMP (*Bitmap*) – это графический формат, который хранит изображение в несжатом виде. Он поддерживает алгоритм сжатия, который, однако, используется достаточно редко. Создавать *BMP*-файлы можно практически во всех растровых редакторах. Особенность *BMP* – широкое распространение при создании графических интерфейсов пользователя в различных Windows-программах. *BMP*-файлы обычно имеют сравнительно большой объём.

Файлы трехмерных моделей

Трехмерные модели используются в трехмерных, объемных играх. Для придания им реалистичного вида, на такие модели обычно накладывают файлы *текстур*.

FBX, X

FBX (Autodesk FBX), – это универсальный формат для хранения трехмерных моделей и сопутствующей информации. Он используется во множестве трехмерных приложений, в частности, весьма популярен в компьютерных играх.

Существуют *плагины* для популярных программ создания трехмерной графики, позволяющие конвертировать созданные в них модели в этот формат. Найти ПО для работы с FBX-файлами можно на http://www.autodesk.ru/.

Формат X – это формат трехмерных файлов, который используется *DirectX*.

Файлы описания шрифтов, SPRITEFONT

SPRITEFONT-файл - это xml-файл с настройками шрифта. Он содержит инструкции, касающиеся визуализации шрифта в игровом окне XNA-проекта. Файлы этого типа имеют расширение .SPRITEFONT.

Звуковые файлы

MP3 – сжатый звуковой файл. Этот формат имеет огромную популярность за счёт возможности сильного сжатия звука с приемлемым качеством. *WAV* – обычно представляет собой несжатый звуковой файл, как правило, может использоваться для воспроизведения коротких звуковых фрагментов (звуковых эффектов) высокого качества.

Все вышеперечисленные форматы файлов – это игровые ресурсы, которые могут быть включены в игровой проект. Учитывая тенденцию к сближению технологий XNA и *Silverlight*, которая наблюдается сегодня, нельзя строго разделить перечисленные игровые ресурсы между двумя технологиями.

Игровая терминология

Игровая терминология складывается под сильным влиянием англоязычных названий. Часто в русском языке довольно сложно найти хорошую аналогию принятым английским терминам, поэтому в сфере игровой терминологии существует двойственность – довольно часто создатели игр пользуются англоязычными терминами в то время, как и русскоязычные названия так же находят применение. Рассмотрим некоторые термины, которые используются в игровой индустрии.

Sprite – этот термин часто заменяют русскоязычным неологизмом *спрайт* – словарь Lingvo 12 определяет понятие "*спрайт*" как "элемент динамического графического

отображения". В игровой индустрии синонимами слова *спрайт* являются такие слова, как "изображение", "картинка", иногда пользуются словом "*текстура*", однако обычно это понятие несет несколько иную смысловую нагрузку. Как правило *спрайт* – это двумерное изображение, причем, в узком смысле слова это – лишь изображение, а в широком – это игровой *объект*, который обладает гораздо более широким набором возможностей, нежели обычное изображение. Спрайты имеют прямоугольную форму, однако в компьютерных играх часто встречаются непрямоугольные объекты. Это достигается за счет задавания прозрачных областей при рисовании *спрайта*. Термин "анимированный *спрайт*" относится к *спрайту*, который выводится с использованием анимации, создающей иллюзию движения, перемещения каких-либо частей изображения – движение рук и ног персонажа при перемещении, движение колёс автомобиля, лопастей пропеллеров самолёта и т.д. *Анимация* обычно реализуется поочередной сменой нескольких статичных изображений, специально подготовленных для того, чтобы создать иллюзию движения.

Texture – текстура. Обычно *текстурами* называют двумерные изображения, которые "накладывают" на трехмерные модели. В терминологии XNA понятие текстуры и *спрайта* при разговоре о двумерных игровых объектах совпадает.

Background – фон. Так называется изображение, обычно – соответствующее размерам размерам игрового поля, которое является фоном для других изображений. Фон может быть неподвижным и подвижным. Подвижный фон (*scrolling background*) используется в играх, называемых скроллерами (scrollers). Скроллинг – это один из принятых игровых терминов. Он означает прокрутку, перемещение содержимого окна. Скроллинговые игры чрезвычайно распространены среди двумерных игр. Например, использование скроллингового фона позволяет создать иллюзию движения в двумерном гоночном *симуляторе*. Фон в двумерной игре может состоять из нескольких частей, движущихся с различной скоростью – это позволяет создать эффект трехмерности игрового мира.

2D-*game* – двумерная *игра – игра*, в которой использованы двумерные изображения.

3D-game – трёхмерная *игра* – использующая трехмерные модели и трехмерный игровой мир.

Tile – тайл – небольшое изображение, которое используется для конструирования уровней в играх. *Tile* можно перевести как "мозаика" или "черепица".

Polygone (полигон, многоугольник) – пространственный многоугольник, который используется для создания трехмерных объектов. Как правило, в компьютерной графике используются треугольники.

Pixel (пиксель) – наименьший элемент растрового изображения, точка, отображаемая на экране. Обычно в пикселях измеряют разрешение *текстур* (например – 1024x768), экранное разрешение монитора, размеры игровых окон. *Слово Pixel* – это аббревитура от *Picture*'s *ELement*.

Texel (тексель) – точка текстуры в трехмерном пространстве.*Слово Texel* – это сокращение от *Texture Element*.

Voxel (воксель) – точка трехмерного изображения. Это *слово* – аббревиатура от *Volumetric Pixel* – объемный пиксель.

Texture Filtering (фильтрация *текстур*) – уменьшение искажений при наложении *текстур* на трехмерный *объект*.

Camera (камера) – так называют точку в игровом пространстве, с которой игрок видит игровой мир. С точки зрения положения камеры игры можно подразделить на игры от первого лица (камера расположена так, что реализует вид как бы "из глаз" персонажа), игры с видом слева-сверху (*стратегии), игры* с видом сзади – камера располагается обычно позади игрока и немного выше его. Существуют и игры, где камера может принимать различные позиции, например, реализующие вид из глаз персонажа и вид сзади

Transparency (прозрачность) – прозрачными могут быть части двумерных изображений – это позволяет создавать изображения сложной формы, которые, фактически, ограничены прямоугольником. Прозрачными бывают и объекты в трехмерном игровом мире – например – это может быть прозрачная вода или стекло, определенным образом преломляющие свет.

Light Model (модель освещения) – способы моделирования освещения объектов.

Игровая документация

Если вы начинаете разрабатывать серьезный игровой проект – вам понадобится игровая документация. Эта лекция посвящена вопросам разработки такой документации. В частности, мы поговорим о концепт-документе, дизайн-документе и плане разработки игры.

В работе над этим материалом использованы образцы документов, рекомендованных компанией 1С для заполнения желающим сотрудничать с ней разработчикам. Актуальные версии образцов документов можно найти на (http://games.1c.ru/4_files/desdocpack.zip). *Автор* курса выражает признательность Сергею Герасеву – Менеджеру внешней разработки *игровых программ* 1С (gers@1c.ru) за содействие.

План разработки игры

План разработки игры – это рабочий документ, который направлен на детализацию некоторых положений, имеющихся в ранее составленных документов и на планирование производства игры. Он может иметь следующую структуру:

1. Анализ рынка
1.1. Целевая аудитория
1.2. Хиты и сравнение
2. Технический анализ
3. Ресурсы проекта
3.1. Персонал
3.2. Оборудование
3.3. Программное обеспечение
3.4. Финансовые ресурсы
Риски проекта
Календарный план

Документ gameproposaltemplate.doc содержит шаблон плана разработки от 1С, документ gameproposal.doc– пример плана разработки.

Практическая работа № 1.30. Разработка игрового приложения

Цель работы: создание игрового приложения

Ход работы

Создаем файл main.cpp

```
#include <Windows.h>

int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int)
{
	return 0;
}
```

Пока что он ничего не делает. Создаем каркас — класс Game
Game.h

```
#ifndef _GAME_H_
#define _GAME_H_

class Game
{
private:
	bool run;

public:
	Game();
	int Execute();
```



```
	void Exit();
```

```
};

#endif
```

Game.cpp

```
#include "Game.h"

Game::Game()
{
	run = true;
}

int Game::Execute()
{
	while(run);
	return 0;
}

void Game::Exit()
{
	run = false;
}
```

Создаем файл Project.h, он нам очень пригодится в будущем

```
#ifndef _PROJECT_H_
#define _PROJECT_H_

#include <Windows.h>

#include "Game.h"

#endif
```

Изменяем main.cpp

```
#include "Project.h"

int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int)
{
	Game game;
	return game.Execute();
}
```

1.2. Графика

Создаем 2 класса — Graphics для отрисовки графики и Image для отрисовки картинок

Graphics.h


```
#ifndef _GRAPHICS_H_
```

```
#define _GRAPHICS_H_

#include "Project.h"

#include "Image.h"
class Image;

class Graphics
{
private:
	SDL_Surface* Screen;

public:
	Graphics(int width, int height);

	Image* NewImage(char* file);
	Image* NewImage(char* file, int r, int g, int b);
	bool DrawImage(Image* img, int x, int y);
	bool DrawImage(Image* img, int x, int y, int startX, int startY, int endX, int endY);

	void Flip();
};

#endif
```

Image.h

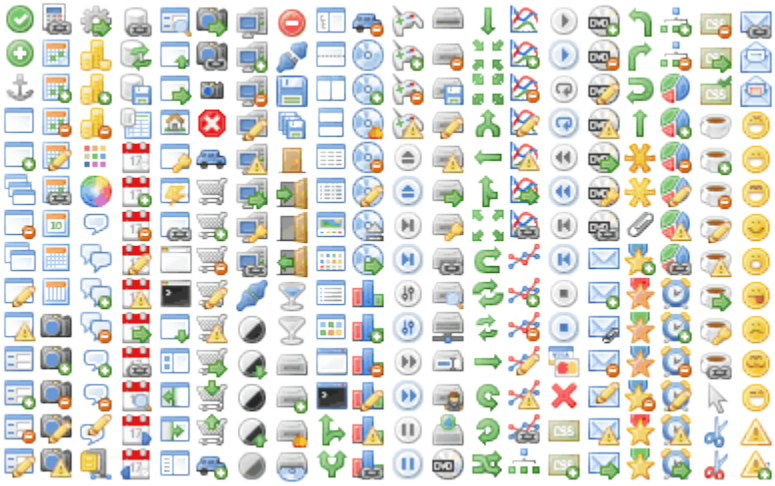
```
#ifndef _IMAGE_H
#define _IMAGE_H

#include "Project.h"

class Image
{
private:
	SDL_Surface* surf;
public:
	friend class Graphics;

	int GetWidth();
	int GetHeight();
};

#endif
```

Изменяем Project.h

```
#ifndef _PROJECT_H_
#define _PROJECT_H_

#pragma comment(lib,"SDL.lib")

#include <Windows.h>
#include <SDL.h>
```

```
#include "Game.h"
#include "Graphics.h"
#include "Image.h"

#endif
```

SDL_Surface — класс из SDL для хранения информации об картинке
Рассмотрим Graphics
NewImage — есть 2 варианта загрузки картинки. Первый вариант просто грузит картинку, а второй после этого еще и дает прозрачность картинке. Если у нас красный фон в картинке, то вводим r=255,g=0,b=0
DrawImage — тоже 2 варианта отрисовки картинки. Первый рисует всю картинку целиком, второй только часть картинки. startX, startY — координаты начала части картинки. endX, endY — конечные координаты части картинки. Этот метод рисования применяется, если используются атласы картинок. Вот пример атласа:



(изображение взято из веб-ресурса interesnoe.info)

Рассмотрим Image
Он просто держит свой сурфейс и дает право доступа к своим закрытым членам классу Graphics, а он изменяет сурфейс.
По сути, это обертка над SDL_Surface. Также он дает размер картинки

Graphics.cpp

```
#include "Graphics.h"

Graphics::Graphics(int width, int height)
{
	SDL_Init(SDL_INIT_EVERYTHING);
	Screen = SDL_SetVideoMode(width,height,32,SDL_HWSURFACE|SDL_DOUBLEBUF);
}

Image* Graphics::NewImage(char* file)
{
	Image* image = new Image();
	image->surf = SDL_DisplayFormat(SDL_LoadBMP(file));

	return image;
}
```

```
Image* Graphics::NewImage(char* file, int r, int g, int b)
{
	Image* image = new Image();
	image->surf = SDL_DisplayFormat(SDL_LoadBMP(file));

	SDL_SetColorKey(image->surf, SDL_SRCCOLORKEY | SDL_RLEACCEL,
		SDL_MapRGB(image->surf->format, r, g, b));

	return image;
}

bool Graphics::DrawImage(Image* img, int x, int y)
{
	if(Screen == NULL || img->surf == NULL)
	return false;

	SDL_Rect Area;
	Area.x = x;
	Area.y = y;

	SDL_BlitSurface(img->surf, NULL, Screen, &Area);

	return true;
}

bool Graphics::DrawImage(Image* img, int x, int y, int startX, int startY, int endX, int endY)
{
	if(Screen == NULL || img->surf == NULL)
	return false;

	SDL_Rect Area;
	Area.x = x;
	Area.y = y;

	SDL_Rect SrcArea;
	SrcArea.x = startX;
	SrcArea.y = startY;
	SrcArea.w = endX;
	SrcArea.h = endY;

	SDL_BlitSurface(img->surf, &SrcArea, Screen, &Area);

	return true;
}

void Graphics::Flip()
{
	SDL_Flip(Screen);
	SDL_FillRect(Screen,NULL, 0x000000);
}
```

В конструкторе инициализируется SDL и создается экран.
Функция Flip должна вызываться каждый раз после отрисовки картинок, она представляет получившееся на экран и чистит экран в черный цвет для дальнешней отрисовки.

Image.cpp

```
#include "Image.h"

int Image::GetWidth()
{
	return surf->w;
}

int Image::GetHeight()
{
	return surf->h;
}
```

Надо изменить Game.h, Game.cpp и main.cpp
Game.h

```
#ifndef _GAME_H_
#define _GAME_H_

#include "Project.h"
class Graphics;

class Game
{
private:
	bool run;

	Graphics* graphics;

public:
	Game();
	int Execute(int width, int height);

	void Exit();
};

#endif
```

Тут мы добавляем указатель на Graphics и в Execute добавляем размер экрана

Game.cpp

```
#include "Game.h"

Game::Game()
{
	run = true;
}

int Game::Execute(int width, int height)
{
	graphics = new Graphics(width,height);

	while(run);
```



```
	SDL_Quit();
```

```
	return 0;
}

void Game::Exit()
{
	run = false;
}
```

Ничего особенного, разве что не пропустите функцию SDL_Quit для очистки SDL

main.cpp

```
#include "Project.h"

int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int)
{
	Game game;
	return game.Execute(500,350);
}
```

Создаем экран размером 500 на 350.

1.3. Ввод

Создаем Input.h

```
#ifndef _INPUT_H_
#define _INPUT_H_

#include "Project.h"

class Input
{
private:
	SDL_Event evt;

public:
   void Update();

	bool IsMouseButtonDown(byte key);
	bool IsMouseButtonUp(byte key);
	POINT GetButtonDownCoords();

	bool IsKeyDown(byte key);
	bool IsKeyUp(byte key);
	byte GetPressedKey();

	bool IsExit();
};

#endif
```

SDL_Event — класс какого-нибудь события, его мы держим в Input'е для того, чтобы не создавать объект этого класса каждый цикл

Ниже расположены методы, не представляющие особого интереса. Примечание: методы с окончанием Down вызываются, когда клавиша была нажата, а с окончанием Up — когда

опущена.

Input.cpp

```
#include "Input.h"

void Input::Update()
{
	while(SDL_PollEvent(&evt));
}

bool Input::IsMouseButtonDown(byte key)
{
	if(evt.type == SDL_MOUSEBUTTONDOWN)
		if(evt.button.button == key)
			return true;
	return false;
}

bool Input::IsMouseButtonUp(byte key)
{
	if(evt.type == SDL_MOUSEBUTTONUP)
		if(evt.button.button == key)
			return true;
	return false;
}

POINT Input::GetButtonDownCoords()
{
	POINT point;
	point.x = evt.button.x;
	point.y = evt.button.y;

	return point;
}

bool Input::IsKeyDown(byte key)
{
	return (evt.type == SDL_KEYDOWN && evt.key.keysym.sym == key);
}

bool Input::IsKeyUp(byte key)
{
	return (evt.type == SDL_KEYUP && evt.key.keysym.sym == key);
}

byte Input::GetPressedKey()
{
	return evt.key.keysym.sym;
}

bool Input::IsExit()
{
	return (evt.type == SDL_QUIT);
}
```

Здесь мы обрабатываем наш объект событий в функции Update, а остальные функции просто проверяют тип события и его значения.

Изменяем теперь Game.h и Game.cpp

```
#ifndef _GAME_H_
#define _GAME_H_

#include "Project.h"

#include "Graphics.h"
class Graphics;
#include "Input.h"
class Input;

class Game
{
private:
	bool run;

	Graphics* graphics;
	Input* input;

public:
	Game();
	int Execute(int width, int height);

	Graphics* GetGraphics();
	Input* GetInput();

	void Exit();
};

#endif
```

Как видно, мы добавили указатель на Input и создали методы-возвращатели Graphics и Input

Game.cpp

```
#include "Game.h"

Game::Game()
{
	run = true;
}

int Game::Execute(int width, int height)
{
	graphics = new Graphics(width,height);
	input = new Input();

	while(run)
	{
		input->Update();
	}
```

```
	delete graphics;
	delete input;

	SDL_Quit();
	return 0;
}

Graphics* Game::GetGraphics()
{
	return graphics;
}

Input* Game::GetInput()
{
	return input;
}

void Game::Exit()
{
	run = false;
}
```

Практическая работа № 1.31. Разработка приложения с анимацией

Цель работы: рассмотреть способы разработки приложений с анимацией

Ход работы

Задание: Создать несложную анимацию, по дороге будут ездить на встречу друг другу автомобиль и мото-цикл.

Используемые компоненты:

TImage (Графический холст) находится на вкладке Additional.

TTimer (Таймер) находится на вкладке System.

В папке “Picture” расположены изображения fon.bmp, avto.bmp и moto.bmp. Это наши заготовки, которые мы будем использовать в программе.

Первый способ

1. Запускаем Delphi и создаем новое приложение: File->New->VCL Forms Application – Delphi.
2. Помещаем на форму 3 компонента TImage. Один в один из них загружаем фон, а в два других изображе-ние мотоцикла и автомобиля соответственно и в режиме таймера будем изменять положения компонетов TImage с изображением автомобиля и мотоцикла относительно фона.
3. В компонент Image1 в свойство Picture загружаем подготовленный нами файл fon.bmp, свойство Align устанавливаем alClient. А форму растянем до размеров фона.
4. В компонент Image2 в свойство Picture загружаем подготовленный нами файл avto.bmp и перемещаем его на дорогу.
5. В компонент Image3 в свойство Picture загружаем подготовленный нами файл moto.bmp и перемещаем его на дорогу.
6. Сравнить вашу форму с примером:

7. В коде программы перед разделом type добавим раздел const и объявим две константы

```
const
scr_width = 640; // ширина формы
scr_height = 480; // высота формы
```

В разделе var объявим переменные x,y,x1,y1 типа integer.

```
var
```

```
  Form1: TForm1;
  x,y,x1,y1:integer;
```

8. Создайте метод procedure TForm1.Timer1Timer и запишите его выполнение

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
x:=x+2;//текущая координата + шаг  для автомобиля
x1:=x1-2;//текущая координата + шаг  для мотоцикла
if x>scr_width+image2.Width then x:=-image2.Width;// ограничение справа
if x1<-image3.Width then x1:=scr_width;// ограничение слева
//рисуем
image2.Left:=x;
image3.Left:=x1;
end;
```

9. В свойствах таймера свойство интервал устанавливаем в пределах от 1 до 100, в зависимости от того какую скорость анимации вы хотите получить.

Второй способ

Второй способ заключается в том, что мы создаем для нашего случая 4 объекта типа TBitMap, в первый за-гружаем фон, во второй - автомобиль, в третий - мотоцикл, а четвертый будет буфером обмена в котором мы вначале будем формировать картинку, а потом выводить ее на экран. Это необходимо делать для того чтобы избежать мерцания картинки на экране во время движения в результате ее перерисовки. Если эффект мерцания для вас не существенен, то можно выводить изображение сразу на экран.

1. Помещаем на форму компонент TImage и компонент TTimer.
2. Для компонента Image1 свойство Align устанавливаем alClient.

А для формы свойство AutoSize устанавливаем True.

3. Объявите константы

```
const
scr_width = 640; // ширина экрана
scr_height = 480; // высота экрана
```

4. В описании переменных var объявляем четыре переменных для хранения графических картинок, тип tbitmap

```
var
fon:tbitmap;//Графический образ  Фона
avto:tbitmap;//Графический образ  автомобиля
moto:tbitmap;//Графический образ  мотоцикла
scr_buffer:tbitmap;//Графический образ  автомобиля
x,y,x1,y1:integer;//координаты  автомобиля и мотоцикла
```

5. Для того чтобы вывести на экран изображения нужно :

а) Активировать созданные переменные

```
//создаем объекты
fon:=TBitmap.Create; // фон
moto:=TBitmap.Create;//  мотоцикл
avto:=TBitmap.Create; // машина
scr_buffer:=TBitmap.Create; // буфер обмена
scr_buffer.Width:=scr_width; // ширина буфера
scr_buffer.Height:=scr_height; // высота буфера
```

б) Загрузить изображения в эти переменные

```
// загружаем объекты
moto.LoadFromFile('moto.bmp'); // мотоцикл
avto.LoadFromFile('avto.bmp'); // машина
fon.LoadFromFile('fon.bmp');// фон
x:=0; //начальные координаты машины
y:=430;
x1:=500;// начальные координаты мотоцикла
```

y1:=380;

в) Установить прозрачный фон вокруг машины и мотоцикла

avto.transparent:=true;//задаем прозрачность

moto.transparent:=true;

г) Вывод изображения на графический холст Image

//Определение ление координат

x:=x+2;//текущая координата + шаг для автомобиля

x1:=x1-2;//текущая координата + шаг для мотоцикла

if x>scr_width+avto.Width then x:=-avto.Width;// ограничение справа

if x1<-moto.Width then x1:=scr_width;// ограничение слева

//рисуем

scr_buffer.Canvas.Draw(0,0,fon);//возобновление фона

scr_buffer.Canvas.Draw(x1,y1,moto);//движение мотоцикла

scr_buffer.Canvas.Draw(x,y,avto);//движение машины

form1.Canvas.Draw(0,0,scr_buffer); //копируем содержимое буфера на экран

При этом первая цифра в скобках, это координата x, а вторая цифра соответственно координата y

д) Для того , чтобы изображения стали двигаться, необходимо динамически менять координаты изобра-жений на холсте, желательно стирая старое изображение.

Эту задачу выполняет компонент Timer.

Практическая работа № 1.32. Оптимизация кода

Цель работы: изучить технологию оптимизации программного кода

Ход работы

Задание:

Модифицируйте 2 программы, реализованные на C++:

1 программа

#include <iostream> #include <locale.h>

using namespace std;

int main()

{

int number; setlocale(LC_CTYPE,"Russian"); cout << "Введите число: "; cin >> number;

cin.ignore();

cout << "Вы ввели: "<< number <<"\n"; cin.get();

}

Описание: пользователю предлагаеться ввести цифру, но если он введет например: b6, то ему выдаст - "Вы ввели: 5 (Только номер). Необходимо добиться, чтобы программа различала отрицательные и положительные значения, цифры и буквы, а также автоматически выдавала значение введенного числа в квадрате.

2 программа

#include <stdio.h> /* Стандартный заголовочный файл ввода-вывода */ #include <iostream> /* Библиотека (стандарт) */

#include <locale.h> /* Русификатор */ #include <windows.h> /* Русификатор */ using namespace std;

+

int main(int argc, char* argv[])

{

setlocale(LC_CTYPE,"Russian");

double plus, minus, pow, div; // объявление переменных через запятую double a1; // отдельное объявление переменной a1

double a2; // отдельное объявление переменной a2 cout << "Введите первое число: ";

cin >> a1;

cout << "Введите второе число: "; cin >> a2;

plus = a1 + a2; // операция сложения minus = a1 - a2; // операция вычитания pow = a1 * a2; // операция умножения div = a1 / a2; // операция деления

cout << a1 << "+" << a2 << "=" << plus << endl; cout << a1 << "-" << a2 << "=" << minus << endl; cout << a1 << "*" << a2 << "=" << pow << endl; cout << a1 << "/" << a2 << "=" << div << endl; system("pause");

return 0;

}

Описание: простой калькулятор, который может: добавлять, вычитать, умножать и делить.

Необходимо реализовать функцию возведения числа в любую степень, а также умножение отрицательных чисел.

Задание:

−Решите задачу оптимизации кода (файл с кодом возьмите у преподавателя), используя только элементарные конструкции (последовательность, ветвления, циклы). Программа должна быть рабочей.

−Оптимизировать программу (можно использовать процедуры или функции).

−Оптимизированная программа должна содержать проверки всех переменных, которые вводятся с клавиатуры.

−Для созданных программ оценить метрические характеристики по Холстеду;

−Сравнить полученные результаты. Оформить результаты в таблицу. Сделать соответствующие выводы.

Расчет метрики Холстеда:

Метрика Холстеда относится к метрикам, вычисляемым на

основании анализа числа строк и синтаксических элементов исходного кода программы.

Метрика Холстеда позволяет оценить размер (в словах) и объем в битах программы на стадии анализа требований. Используя нормы выработки операторов в день можно оценить время на разработку.

Основу метрики Холстеда составляют четыре измеряемые характеристики программы: n1 — число уникальных операторов

+

Опреаторы	Число операторов	Операнды	Число операндов
::	7	a	2
=	8	b	2
==	1	c	2
.	6	mAparam	5
!=	1	mBparam	5
<	1	mCparam	3
-	3	result.x1	3
/	3	result.x2	3
*	2	result.status	2
- binary	3	det	3
return	1	4	3
{	12	2	2
}	12		
(	13		
)	13		

программы, включая символы-разделители, имена процедур и знаки операций (словарь операторов); n2 — число уникальных операндов программы (словарь операндов); N1 — общее число операторов в программе; N2 — общее число операндов в программе.

+, *, /, - это операторы

x, y, z, 999, -25, number1 - это операнды

На основании этих характеристик рассчитываются оценки:

Словарь программы (Halstead Program Vocabulary, HPVoc): n = n1 + n2;

Длина программы (Halstead Program Length, HPLen): N = N1 +
N2;

Объем программы (Halstead Program Volume, HPVol): V = N
log2 n;

Сложность программы (Halstead Difficulty, HDiff): D = (n1/2) × (N2 / n2);

На основе показателя HDiff предлагается оценивать усилия программиста при разработке при помощи показателя HEff (Halstead Effort): H = D × V.

Практическая работа № 1.33. Рефакторинг кода

Цель работы: изучить технологию рефакторинга программного кода

Ход работы

Задание: Задание 1:

В рамках лабораторной работы приведите примеры реализации рефакторинга в следующих системах:

−Visual Studio.
−Visual Assist X.
−Refactor.
−JustCode
−ReSharper
−CodeIt

Ход выполнения работы

1. Выполнить анализ программного кода разрабатываемого ПО и модульных тестов с целью выявления плохо организованного кода.
2. Используя шаблоны рефакторинга, выполнить реорганизацию программного кода разрабатываемого ПО.
3. Выполнить описание произведенных операций рефакторинга (было-стало-шаблон рефакторинга).
4. В случае необходимости скорректировать проектную документацию (диаграммы классов, последовательностей).
5. Сделать выводы по результатам выполнения работы.

Класс Main

Было:

```
package game;
import game.Characters.*;
import game.Characters.Character;
import game.Energetics.Energetic;
import game.Energetics.Lightning;
import game.Levels.Block;
import game.Levels.Level;
import game.Levels.Level_data;
import game.Weapon.Bullet;
import game.Weapon.Weapon;
import javafx.animation.AnimationTimer;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.image.Image;
```

```
import javafx.scene.input.KeyCode;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
public class Main extends Application {
public static ArrayList<Block> blocks = new ArrayList<>();
public static ArrayList<Bullet> bullets = new ArrayList<>();
public static ArrayList<Bullet> enemyBullets = new ArrayList<>();
public static ArrayList<EnemyBase> enemies = new ArrayList<>();
static HashMap<KeyCode, Boolean> keys = new HashMap<>();
public static Stage stage;
public static Scene scene;
public static Pane gameRoot = new Pane();
public static Pane appRoot = new Pane();
public static Menu menu;
public static Character booker;
public static HUD hud;
public static Weapon weapon;
public static Elizabeth elizabeth;
static VendingMachine vendingMachine;
static Tutorial tutorial;
private static CutScenes cutScene;
public static Energetic energetic;
public static Lightning lightning;
public static int levelNumber;
static Level level;
public static AnimationTimer timer = new AnimationTimer() {
@Override
public void handle(long now) {
update();
}
};
private static void update() {
for (EnemyBase enemy : enemies) {
enemy.update();
if (enemy.getDelete()) {
enemies.remove(enemy);
break;
}
}
Bullet.update();
Controller.update();
booker.update();
if (!energetic.getName().equals(""))
energetic.update();
if (levelNumber > 0)
elizabeth.update();
if (lightning != null) {
lightning.update();
if (lightning.getDelete())
lightning = null;
}
```

```
menu.update();
hud.update();
weapon.update();
if (booker.getTranslateX() > Level_data.BLOCK_SIZE * 295)
cutScene = new CutScenes(levelNumber);
}
@Override
public void start(Stage primaryStage) throws Exception {
stage = primaryStage;
scene = new Scene(appRoot, 1280, 720);
appRoot.getChildren().add(gameRoot);
level = new Level();
try (DataInputStream dataInputStream = new DataInputStream(new
FileInputStream("C:/DeadShock/saves/data.dat"))) {
levelNumber = dataInputStream.readInt();
level.createLevels(levelNumber);
level.changeImageView(levelNumber);
vendingMachine = new VendingMachine();
booker = new Character();
booker.setMoney(dataInputStream.readInt());
booker.setSalt(dataInputStream.readByte());
booker.setCountLives(2);
weapon = new Weapon();
weapon.setWeaponClip(dataInputStream.readInt());
weapon.setBullets(dataInputStream.readInt());
hud = new HUD();
elizabeth = new Elizabeth();
energetic = new Energetic();
} catch (IOException e) {
levelNumber = 0;
level.createLevels(levelNumber);
vendingMachine = new VendingMachine();
booker = new Character();
weapon = new Weapon();
hud = new HUD();
energetic = new Energetic();
tutorial = new Tutorial(levelNumber);
}
switch (levelNumber) {
case 0:
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 117, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 127, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 148, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 161, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 171, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 185, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 204, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 215, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 228, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 233, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 243, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 252, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 262, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 277, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 280, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 286, 200));
```

```
break;
case 1:
enemies.add(new EnemyRedEye(Level_data.BLOCK_SIZE * 57, 200));
enemies.add(new EnemyRedEye(Level_data.BLOCK_SIZE * 67, 200));
enemies.add(new EnemyRedEye(Level_data.BLOCK_SIZE * 74, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 87, 200));
enemies.add(new EnemyRedEye(Level_data.BLOCK_SIZE * 104, 150));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 117, 200));
enemies.add(new EnemyRedEye(Level_data.BLOCK_SIZE * 133, 200));
enemies.add(new EnemyRedEye(Level_data.BLOCK_SIZE * 156, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 177, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 193, 200));
enemies.add(new EnemyRedEye(Level_data.BLOCK_SIZE * 201, 200));
enemies.add(new EnemyRedEye(Level_data.BLOCK_SIZE * 216, 200));
enemies.add(new EnemyRedEye(Level_data.BLOCK_SIZE * 224, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 246, 200));
enemies.add(new EnemyRedEye(Level_data.BLOCK_SIZE * 260, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 277, 100));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 34,
Level_data.BLOCK_SIZE * 13));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 36,
Level_data.BLOCK_SIZE * 13));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 60,
Level_data.BLOCK_SIZE * 9));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 61,
Level_data.BLOCK_SIZE * 9));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 106,
Level_data.BLOCK_SIZE * 7));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 107,
Level_data.BLOCK_SIZE * 7));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 168,
Level_data.BLOCK_SIZE * 13));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 170,
Level_data.BLOCK_SIZE * 13));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 196,
Level_data.BLOCK_SIZE * 13));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 197,
Level_data.BLOCK_SIZE * 13));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 232,
Level_data.BLOCK_SIZE * 8));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 233,
Level_data.BLOCK_SIZE * 8));
break;
}
menu = new Menu();
appRoot.getChildren().add(menu.menuBox);
booker.translateXProperty().addListener( ((observable, oldValue, newValue) -> {
int offset = newValue.intValue();
if (offset > 600 && offset < gameRoot.getWidth() - 680) {
gameRoot.setLayoutX( - (offset - 600) );
level.getBackground().setLayoutX((offset - 600) / 1.5);
}
if (offset <= 100)
level.getBackground().setLayoutX(0);
}));
vendingMachine.createButtons();
```

```
stage.getIcons().add(new Image("file:/C:/DeadShock/images/icon.jpg"));
stage.setTitle("DeadShock");
stage.setResizable(false);
stage.setWidth(scene.getWidth());
stage.setHeight(scene.getHeight());
stage.setScene(scene);
stage.show();
timer.start();
}
public static void main(String[] args) {
launch(args);
}
}
```

Стало:

```
package game;
import game.Characters.*;
import game.Characters.Character;
import game.Energetics.Energetic;
import game.Energetics.Lightning;
import game.Levels.Block;
import game.Levels.Level;
import game.Levels.Level_data;
import game.Weapon.Bullet;
import game.Weapon.Weapon;
import javafx.animation.AnimationTimer;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.input.KeyCode;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
public class Main extends Application {
public static ArrayList<Block> blocks = new ArrayList<>();
public static ArrayList<Bullet> bullets = new ArrayList<>();
public static ArrayList<Bullet> enemyBullets = new ArrayList<>();
public static ArrayList<EnemyBase> enemies = new ArrayList<>();
static HashMap<KeyCode, Boolean> keys = new HashMap<>();
public static Stage stage;
public static Scene scene;
public static Pane gameRoot = new Pane();
public static Pane appRoot = new Pane();
public static Menu menu;
public static Character booker;
public static HUD hud;
public static Weapon weapon;
public static Elizabeth elizabeth;
static VendingMachine vendingMachine;
static Tutorial tutorial;
private static CutScenes cutScene;
public static Energetic energetic;
public static Lightning lightning;
```

```
public static int levelNumber;
static Level level;
public static AnimationTimer timer = new AnimationTimer() {
@Override
public void handle(long now) {
update();
}
};
private void initContent() {
appRoot.getChildren().add(gameRoot);
level = new Level();
try (DataInputStream dataInputStream = new DataInputStream(new FileInputStream("C:/DeadShock/saves/data.dat"))) {
levelNumber = dataInputStream.readInt();
level.createLevels(levelNumber);
level.changeImageView(levelNumber);
vendingMachine = new VendingMachine();
booker = new Character();
booker.setMoney(dataInputStream.readInt());
booker.setSalt(dataInputStream.readByte());
booker.setCountLives(2);
weapon = new Weapon();
weapon.setWeaponClip(dataInputStream.readInt());
weapon.setBullets(dataInputStream.readInt());
hud = new HUD();
elizabeth = new Elizabeth();
energetic = new Energetic();
} catch (IOException e) {
levelNumber = 0;
level.createLevels(levelNumber);
vendingMachine = new VendingMachine();
booker = new Character();
weapon = new Weapon();
hud = new HUD();
energetic = new Energetic();
tutorial = new Tutorial(levelNumber);
}
createEnemies();
menu = new Menu();
appRoot.getChildren().add(menu.menuBox);
booker.translateXProperty().addListener( ((observable, oldValue, newValue) -> {
int offset = newValue.intValue();
if (offset > 600 && offset < gameRoot.getWidth() - 680) {
gameRoot.setLayoutX( - (offset - 600) );
level.getBackground().setLayoutX((offset - 600) / 1.5);
}
if (offset <= 100)
level.getBackground().setLayoutX(0);
}));
vendingMachine.createButtons();
}
public static void createEnemies() {
switch (levelNumber) {
case 0:
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 117, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 127, 200));
```

```
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 148, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 161, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 171, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 185, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 204, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 215, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 228, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 233, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 243, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 252, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 262, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 277, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 280, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 286, 200));
break;
case 1:
enemies.add(new EnemyRedEye(Level_data.BLOCK_SIZE * 57, 200));
enemies.add(new EnemyRedEye(Level_data.BLOCK_SIZE * 67, 200));
enemies.add(new EnemyRedEye(Level_data.BLOCK_SIZE * 74, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 87, 200));
enemies.add(new EnemyRedEye(Level_data.BLOCK_SIZE * 104, 150));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 117, 200));
enemies.add(new EnemyRedEye(Level_data.BLOCK_SIZE * 133, 200));
enemies.add(new EnemyRedEye(Level_data.BLOCK_SIZE * 156, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 177, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 193, 200));
enemies.add(new EnemyRedEye(Level_data.BLOCK_SIZE * 201, 200));
enemies.add(new EnemyRedEye(Level_data.BLOCK_SIZE * 216, 200));
enemies.add(new EnemyRedEye(Level_data.BLOCK_SIZE * 224, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 246, 200));
enemies.add(new EnemyRedEye(Level_data.BLOCK_SIZE * 260, 200));
enemies.add(new EnemyComstock(Level_data.BLOCK_SIZE * 277, 100));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 34,
Level_data.BLOCK_SIZE * 13));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 36,
Level_data.BLOCK_SIZE * 13));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 60,
Level_data.BLOCK_SIZE * 9));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 61,
Level_data.BLOCK_SIZE * 9));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 106,
Level_data.BLOCK_SIZE * 7));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 107,
Level_data.BLOCK_SIZE * 7));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 168,
Level_data.BLOCK_SIZE * 13));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 170,
Level_data.BLOCK_SIZE * 13));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 196,
Level_data.BLOCK_SIZE * 13));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 197,
Level_data.BLOCK_SIZE * 13));
Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 232,
Level_data.BLOCK_SIZE * 8));
+Level_data.enemyBlocks.add(new Block("invisible", Level_data.BLOCK_SIZE * 233,
Level_data.BLOCK_SIZE * 8));
```

```
break;
}
}
private static void update() {
for (EnemyBase enemy : enemies) {
enemy.update();
```

Практическая работа № 1.34. Пользовательская и программная модели интерфейса

Цель работы: закрепить теоретические знания по разработке пользовательского интерфейса; получить практические навыки по проектированию пользовательской и программной модели интерфейса.

Теоретический материал

Существуют три совершенно различные модели пользовательского интерфейса: модель рограммиста,
модель пользователя и программная модель. Программист, разрабатывая
пользовательский интерфейс, исходит из того, управление какими операциями ему необхо димо реализовать в пользовательском интерфейсе, и как это осуществить, не затрачивая ни существенных ресурсов компьютера, ни своих сил и времени. Его интересуют функциона льность,
эффективность, технологичность, внутренняя стройность и другие не связанные с удобств ом пользователя характеристики программного обеспечения. Именно поэтому большинство интерфейсов существующих программ вызывают серьезные нарекания пользователей.

С точки зрения здравого смысла хорошим следует считать интерфейс, при работе с которым
пользователь получает именно то, что он ожидал. Представление пользователя о функция х интерфейса можно описать в виде пользовательской модели интерфейса.

Пользовательская модель интерфейса -
это совокупность обобщенных представлений
конкретного пользователя или некоторой группы пользователей о процессах, происходящи х во
время работы программы или программной системы. Эта модель базируется на особеннос тях опыта конкретных пользователей, который характеризуется:

- уровнем подготовки в предметной области разрабатываемого программного обеспечения;
- интуитивными моделями выполнения операций в этой предметной области;
- уровнем подготовки в области владения компьютером; – устоявшимися стереотипами работы с компьютером.

Для построения пользовательской модели необходимо изучить перечисленные выше особенности опыта предполагаемых пользователей программного обеспечения. С этой цел ью
используют опросы, тесты и даже фиксируют последовательность действий, осуществляем ых в процессе выполнения некоторых операций, на пленку.

Приведение в соответствие моделей пользователя и программиста, а также построение на их базе программной модели (рис. 1.1)
интерфейса задача не тривиальная. Причем, чем сложнее автоматизируемая предметная область, тем сложнее оказывается построить программную модель
интерфейса, учитывающую особенности пользовательской модели и не требующую слишк ом больших затрат как в процессе разработки, так и во время работы. С этой точки зрения объектные интерфейсы кажутся наиболее перспективными, так как в их основе лежит именно отображение
объектов предметной области, которыми оперируют пользователи. Хотя на настоящий мо мент времени их реализация достаточно трудоемка.

При создании программной модели интерфейса также следует иметь в виду, что из менять пользовательскую модель непросто. Повышение профессионального уровня пользователей и их подготовки в области владения компьютером в компетенцию разработчиков программного обеспечения не входит, хотя часто грамотно построенный интерфейс, который адекватно отображает сущность происходящих процессов, способствует росту квалификации пользователей.

Интуитивные модели выполнения операций в предметной области должны стать основой для разработки интерфейса, а потому в большинстве случаев их необходимо не менять, а уточнять и совершенствовать. Именно нежелание или невозможность следования интуитивным моделям выполнения операций приводит к созданию искусственных надуманных интерфейсов, которые негативно воспринимаются пользователями.

Иногда кажется, что единственно доступный для изменения элемент - устоявшийся стереотип работы с компьютером. Однако ломка стереотипов - процедура болезненная. На это стоит решаться, если некоторое революционное изменение значительно расширяет возможности пользователя или облегчает его работу, например, переход к Windows-интерфейсам существенно упростил работу с компьютером огромному числу пользователей-непрофессионалов. Ломая же стереотипы по мелочам или неточно следуя принятой концепции, разработчик рискует оттолкнуть пользователей, которые просто не будут понимать, что происходит. В качестве примера можно вспомнить хотя бы путаницу с вызовом программ двойным щелчком правой клавиши мыши по пиктограмме рабочем столе или одинарным, если пиктограммы вынесена на панель Quick Launch (Быстрый Доступ) Windows.

Критерии оценки интерфейса пользователем. Многочисленные опросы и обследования, проводимые ведущими фирмами по разработке программного обеспечения, показали, что основными критериями оценки интерфейсов пользователем являются:

- простота освоения и запоминания операций системы - конкретно оценивают время освоения и продолжительность сохранения информации в памяти;
- скорость достижения результатов при использовании системы - определяется количеством вводимых или выбираемых мышью команд и настроек;
- субъективная удовлетворенность при эксплуатации системы удобство работы, утомляемость и т. д.).

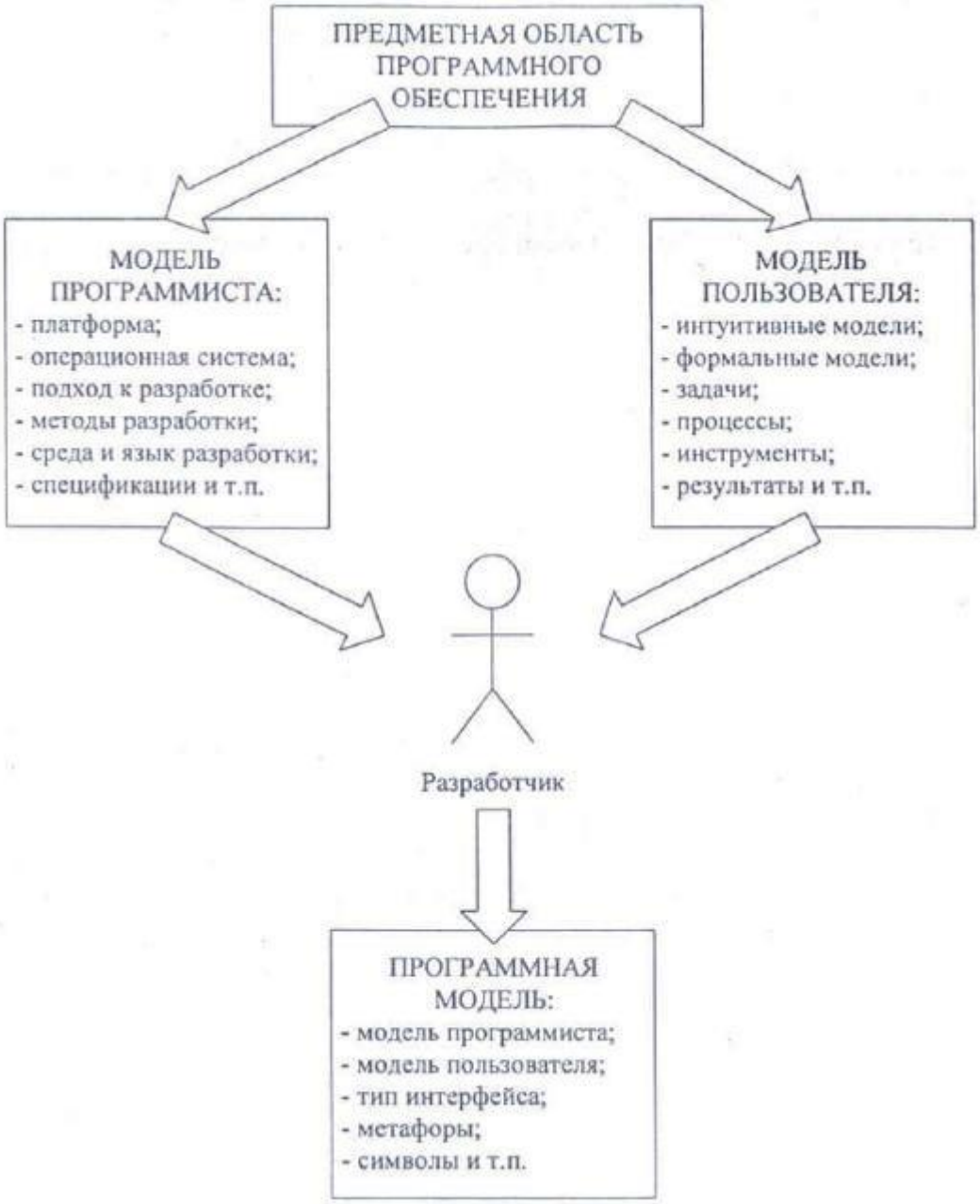


Рис.1.1. Процесс проектирования пользовательского интерфейса

Причем для пользователей - профессионалов, постоянно работающих с одним и тем же пакетом, на первое место достаточно быстро выходят второй и третий критерии, а для пользователей-непрофессионалов, работающих с программным обеспечением периодически и выполняющих сравнительно несложные задачи - первый и третий.

С этой точки зрения на сегодняшний день наилучшими характеристиками для пользователей-профессионалов обладают интерфейсы со свободной навигацией, а для пользователейнепрофессионалов - интерфейсы прямого манипулирования. Давно замечено, что при выполнении операции копирования файлов при прочих равных условиях большинство профессионалов используют оболочки типа Far, а непрофессионалы - «перетаскивание объектов» Windows.

Задания для выполнения №1

1. Разработать модель программиста, модель пользователя и программную модель для обучающе-контролирующей программы. Обучающе-контролирующая программа должна быть разработана в одной из сред ООП (Lazarus, Delphi, VB, C++ и т.д.) с испольованием всех, ранее изученных классов, компонентов и методов. Также программа должна включать теоретический, практический и контролирующий блоки. Модели представить в виде одного из вариантов: текстовое описание, схема, таблица.

2. Оформить работу в соответствии с требованиями ЕСПД (ГОСТ 19.101-77, ГОСТ 19.102-77, ГОСТ 19.103-77, ГОСТ 19.104-78, ГОСТ 19.10578, ГОСТ 19.106-78, ГОСТ 19.401-78, ГОСТ 19.604-78). При оформлении использовать MS Office или OpenOffice.org.
3. Сдать и защитить работу.

Варианты заданий
1. Обучающе-контролирующая программа по Lazarus
2. Обучающе-контролирующая программа по Pascal
3. Обучающе-контролирующая программа по Word
4. Обучающе-контролирующая программа по Excel
5. Обучающе-контролирующая программа по Access
6. Обучающе-контролирующая программа по Power Point
7. Обучающе-контролирующая программа по Ramus
8. Обучающе-контролирующая программа по PhotoShop
9. Обучающе-контролирующая программа по теме «Компьютер и его ПО»
10. Обучающе-контролирующая программа по теме «История ВТ»
11. Обучающе-контролирующая программа по HTML

Содержание и оформление отчета по лабораторной работе Отчѐт должен содержать:
1. Титульный лист.
2. Аннотацию.
3. Содержание.
4. Основную часть, оформленную в соответствии с требованиями ЕСПД (ГОСТ 19.101-77, ГОСТ 19.102-77, ГОСТ 19.103-77, ГОСТ 19.104-78, ГОСТ 19.105-78, ГОСТ 19.106-78, ГОСТ 19.401-78, ГОСТ 19.604-78). 5. Заключение (описание результатов работы).

Защита отчета по лабораторной работе заключается в предъявлении преподавателю полученных результатов (на экране монитора или у доски),

демонстрации полученных навыков и ответах на вопросы преподавателя

Практическая работа № 1.35. Разработка технического задания

Цель работы: изучение руководящих документов по составлению технического задания. Разработка проекта технического задания на программное обеспечение.

Теоретический материал

Общие сведения

Техническое задание (ТЗ, техзадание) - основной документ, содержащий требования заказчика к системе, в соответствии с которыми осуществляется создание и разработка конечного продукта.

Техническое задание на программу и программное обеспечение разрабатывается в соответствии с требованиями следующих документов:

- ГОСТ 19.201-78. Единая система программной документации. Техническое задание. Требования к содержанию и оформлению;
- ГОСТ 34.602-89. Информационная технология. Комплекс стандартов на автоматизированные системы. Техническое задание на создание автоматизированной системы.

Основанием для разработки ТЗ чаще всего является договор.

Техническое задание позволяет:

- исполнителю - понять суть задачи, показать заказчику «технический облик» будущего изделия, программного изделия или автоматизированной системы;
- заказчику - осознать, что именно ему нужно;
- обеим сторонам - представить готовый продукт;
- исполнителю - спланировать выполнение проекта и работать по намеченному плану;
- заказчику - требовать от исполнителя соответствия продукта всем условиям, оговоренным в ТЗ;
- исполнителю - отказаться от выполнения работ, не указанных в ТЗ;

- заказчику и исполнителю - выполнить полную попунктную проверку готового продукта;
- избежать ошибок, связанных с изменением требований (на всех стадиях и этапах создания, за исключением испытаний).

Если поставлены сжатые сроки подготовки ТЗ и заказчик не требует оформления документации в соответствии с государственным стандартом, то можно использовать шаблон технического задания по стандарту IEEE Std 830, который предполагает, что детальные требования могут быть обширными и не существует оптимальной структуры для всех систем. По этой причине стандартом рекомендуется обеспечивать такое структурирование детальных требований, которое делает их оптимальными для понимания. Стандартом рекомендуются различные способы структурирования детальных требований для различных классов систем.

Существует и третья альтернатива для выбора шаблона технического задания, когда заказчик предлагает использовать принятый в компании корпоративный шаблон для описания требований к информационным системам.

Для внесения изменений или дополнений в техническое задание на последующих стадиях разработки программы или программного изделия выпускают дополнение к нему. Согласование и утверждение дополнения к техническому заданию проводят в том же порядке, который установлен для технического задания.

ГОСТ 19.201-78. Единая система программной документации. Техническое задание. Требования к содержанию и оформлению

Данный стандарт устанавливает порядок построения и оформления технического задания на разработку программы или программного изделия для вычислительных машин, комплексов и систем независимо от их назначения и области применения.

Согласно стандарту техническое задание должно содержать следующие разделы:

- введение;
- основания для разработки;
- назначение разработки;
- требования к программе или программному изделию;
- требования к программной документации;
- технико-экономические показатели;
- стадии и этапы разработки;
- порядок контроля и приемки;
- в техническое задание допускается включать приложения.

В зависимости от особенностей программы или программного изделия, допускается уточнять содержание разделов, вводить новые разделы или объединять отдельные из них.

В разделе «Введение» указывают наименование, краткую характеристику области применения программы или программного изделия и объекта, в котором используют программу или программное изделие.

В разделе «Основания для разработки» должны быть указаны:

- документ (документы), на основании которых ведется разработка;
- организация, утвердившая этот документ, и дата его утверждения;
- наименование и (или) условное обозначение темы разработки.

Применительно к специфике учебного процесса основанием может служить задание на курсовое проектирование, приказ по институту от . . за №., договор . . за N., и т.п.

Наименование: «Разработка лабораторной работы № 1».

Шифр: «ЛАБА-001».

В разделе «Назначение разработки» должно быть указано функциональное и эксплуатационное назначение программы или программного изделия.

Раздел «Требования к программе или программному изделию» должен содержать следующие подразделы:

- требования к функциональным характеристикам;
- требования к надежности;

- условия эксплуатации;
- требования к составу и параметрам технических средств;
- требования к информационной и программной совместимости;
- требования к маркировке и упаковке;
- требования к транспортированию и хранению;
- специальные требования.

1. В подразделе «Требования к функциональным характеристикам» должны быть указаны требования к составу выполняемых функций, организации входных и выходных данных, временным характеристикам и т.п.

Например: Программа должна позволять ... вычислять ... строить... создавать ...

Исходные данные: текстовый файл с заданной ...

Входные и выходные данные : графическая и текстовая информация - результаты анализа системы.; текстовые файлы - отчеты о ... диагностика состояния системы и сообщения о всех возникших ошибках.

2. В подразделе «Требования к надежности» должны быть указаны требования к обеспечению надежного функционирования (обеспечение устойчивого функционирования, контроль входной и выходной информации, время восстановления после отказа и т.п.).

Для программного обеспечения надежность зависит не столько от исполнителя, сколько от надежности технических средств и операционной системы, поэтому в разделе обязательно следует написать такую фразу: «Требования к обеспечению надежного (устойчивого) функционирования программы не предъявляются»

Время восстановления после отказа, вызванного сбоем электропитания технических средств (иными внешними факторами), не фатальным сбоем (не крахом) операционной системы, не должно превышать нескольких минут при условии соблюдения условий эксплуатации технических и программных средств. Время восстановления после отказа, вызванного неисправностью технических средств, фатальным сбоем (крахом) операционной системы, не должно превышать времени, требуемого на устранение неисправностей технических средств и переустановки программных средств.

Перечень аварийных ситуаций также составляет заказчик и согласовывает с исполнителем, т.е. это время на перезагрузку операционной системы, если отказ не фатален и не вызван крахом операционной системы или выходом из строя технических средств.

Отказы программы возможны вследствие некорректных действий оператора (пользователя) при взаимодействии с операционной системой. Во избежание возникновения отказов программы по указанной выше причине следует обеспечить работу пользователя без предоставления ему административных привилегий.

3. В подразделе «Условия эксплуатации» должны быть указаны условия эксплуатации (температура окружающего воздуха, относительная влажность и т.п. для выбранных типов носителей данных), при которых должны обеспечиваться заданные характеристики, а также вид обслуживания, необходимое количество и квалификация персонала.

Здесь можно ограничиться следующими фразами: «Условия эксплуатации программы совпадают с условиями эксплуатации ПЭВМ IBM PC и совместимых с ними ПК», «Программа должная быть рассчитана на непрофессионального пользователя» и т.п.

4. В подразделе «Требования к составу и параметрам технических средств» указывают необходимый состав технических средств с указанием их основных технических характеристик.

Как правило, тут указывают минимальные требования к технике, на которой будет функционировать разработанное программное обеспечение. Например, IBM-совместимый персональный компьютер (ПЭВМ), включающий в себя:

- процессор Pentium-1000 с тактовой частотой, ГГц - 10, не менее;
- оперативную память объемом, Гб - 2, не менее;
- и так далее.

5. В подразделе «Требования к информационной и программной совместимости» должны быть указаны требования к информационным структурам на входе и выходе и методам решения, исходным кодам, языкам программирования и программным средствам, используемым программой.

При необходимости должна обеспечиваться защита информации и программ.

Например: Программа должна работать автономно под управлением ОС MS Windows версии не ниже 7. Базовый язык программирования — C++.

6. В подразделе «Требования к маркировке и упаковке» в общем случае указывают требования к маркировке программного изделия, варианты и способы упаковки.

Например, программное изделие должно иметь маркировку с обозначением товарного знака компании-разработчика, типа (наименования), номера версии, порядкового номера, даты изготовления и номера сертификата соответствия Госстандарта России (если таковой имеется). Маркировка должна быть нанесена на программное изделие в виде наклейки, выполненной полиграфическим способом с учетом требований ГОСТ 9181-74.

Упаковка программного изделия должна осуществляться в упаковочную тару предприятия-изготовителя (поставщика).

Упаковка программного изделия должна проводиться в закрытых вентилируемых помещениях при температуре от +15 до +40 °С и относительной влажности не более 80 % при отсутствии агрессивных примесей в окружающей среде.

7. В подразделе «Требования к транспортированию и хранению» должны быть указаны для программного изделия условия транспортирования, места хранения, условия хранения, условия складирования, сроки хранения в различных условиях.

Допускается транспортирование программного изделия в транспортной таре всеми видами транспорта (в том числе в отапливаемых герметизированных отсеках самолетов без ограничения расстояний). При перевозке в железнодорожных вагонах вид отправки - мелкий малотоннажный.

При транспортировании и хранении программного изделия должна быть предусмотрена защита от попадания пыли и атмосферных осадков. Не допускается кантование программного изделия. Климатические условия транспортирования приведены ниже:

- температура окружающего воздуха, °С - от ... до ...;
- атмосферное давление, кПа - ..;
- относительная влажность воздуха при 25 °С - ... .

В разделе «Требования к программной документации» должен быть указан предварительный состав программной документации и, при необходимости, специальные требования к ней.

В состав программной документации должны входить:

- техническое задание;
- программа и методика испытаний;
- руководство системного программиста;
- руководство оператора;
- ведомость эксплуатационных документов.

Программа и методика испытаний потребуется, чтобы показать заказчику, что разработанная исполнителем программа соответствует требованиям согласованного и утвержденного технического задания. После проведения совместных (приемо-сдаточных) испытаний заказчик и исполнитель подпишут акт завершения работы. Таким образом, работа будет закрыта, условия договора выполнены.

В разделе «Технико-экономические показатели» должны быть указаны: ориентировочная экономическая эффективность, предполагаемая годовая потребность, экономические преимущества разработки по сравнению с лучшими отечественными и зарубежными образцами или аналогами.

В разделе «Стадии и этапы разработки» устанавливают необходимые стадии и сроки разработки, этапы и содержание работ (перечень программных документов, которые должны быть разработаны, согласованы и утверждены), а также, как правило, определяют исполнителей.

Главное - грамотно определиться со сроками. По возможности, старайтесь равномерно распределить этапы по срокам (и суммам). Помните, что не все проекты доживают до последней стадии. А отчеты должны быть по каждому этапу. Помните также, что больше всего времени займет рабочий проект. Если вы не успеете сделать в срок документацию, то Заказчик имеет полное право вообще не принять работу со всеми вытекающими последствиями.

Основными и непременными стадиями и этапами являются само техническое задание, эскизный проект, технический и рабочий проекты.

Эскизный проект. На этой стадии детально разрабатываются структуры входных и выходных данных, определяется форма их представления. Разрабатываются общее описание алгоритма, сам алгоритм, структура программы, план мероприятий по разработке и внедрению программы.

Технический проект. Содержит разработанный алгоритм решения задачи, а также методы контроля исходной информации. Здесь же разрабатываются средства обработки ошибок и выдачи диагностических сообщений, определяются формы представления исходных данных и конфигурация технических средств.

Рабочий проект. На этой стадии осуществляются программирование и отладка программы, разработка программных документов, программы и методики испытаний. Подготавливаются контрольно-отладочные примеры. Окончательно оформляются документация и графический материал.

В разделе «Порядок контроля и приемки» должны быть указаны виды испытаний и общие требования к приемке работы.

Например, контроль и приемка разработки осуществляются на основе испытаний контрольно-отладочных примеров. При этом проверяется выполнение всех функций программы.

В приложениях к техническому заданию при необходимости приводят:

- перечень научно-исследовательских и других работ, обосновывающих разработку;
- схемы алгоритмов, таблицы, описания, обоснования, расчеты и другие документы, которые могут быть использованы при разработке;
- другие источники разработки.

Задание на лабораторную работу

Разработать проект технического задания на программное обеспечение в соответствии с ГОСТ 19.201-78. Программное обеспечение должно включать в свой состав:

- серверную часть;
- клиента, функционирующего на ПЭВМ;
- мобильного клиента, функционирующего на смартфоне;
- веб-клиента, функционирующего в браузере.

По желанию состав программного обеспечения может быть расширен.

Требования к функциональным характеристикам необходимо указать для каждого клиента.

Вопросы для самопроверки

1. Зачем разрабатывают ТЗ?
2. Какими стандартами регулируется содержимое технического задания?
3. Какие существуют стадии и этапы разработки?

4. Раскройте понятие «время восстановления после отказа».
5. Что должен содержать подраздел «Требования к функциональным характеристикам» раздела «Требования к программе или программному изделию»?
6. Обязательно ли присваивать условное обозначение темы разработки?

Практическая работа № 36. Разработка интерфейса пользователя. Практическая работа № 1.37. Проектирование пользовательского интерфейса десктопного приложения . Практическая работа № 1.38. Проектирование пользовательского интерфейса десктопного приложения

Цель работы: изучение принципов проектирования пользовательского интерфейса. Разработка пользовательского интерфейса десктопного приложения.

Теоретический материал

Общие сведения о десктопных приложениях

Десктопные приложения - это программы, требующие наличия оператора (человека, работающего с программой), содержащие в себе всю полную функциональность и способные работать отдельно на любой машине изолированно от других приложений. Microsoft Word, Excel, Блокнот, однопользовательские игры - все это примеры десктопных приложений. Для их работы необходимы лишь достаточные аппаратные ресурсы компьютера, само приложение и набор библиотек, содержащих функции для работы с приложением.

Десктопные приложения могут быть также и многопользовательскими. Например, редактор файлов, который в зависимости от логина и пароля, введенных при запуске, будет давать доступ к различным файлам. И программа, и файлы находятся на одном компьютере, просто производится локальное разграничение доступа для разных пользователей.

Пользовательский интерфейс

Пользовательский интерфейс (UI) - это способ, которым вы выполняете какую-либо задачу с помощью какого-либо продукта, т.е. совершаемые вами действия и то, что вы получаете в ответ.

Программный интерфейс не только решает нашу проблему взаимодействия с приложением, но и делает это взаимодействие максимально комфортным. Нам важно наличие интерфейса, позволяющего при меньшем количестве усилий ознакомиться с возможностями приложения и понять принципы работы в нем.

Чтобы не возникло проблем при использовании какого-либо приложения, можно визуализировать его функциональные возможности в виде понятных элементов, и за этой визуализацией кроется целая кухня UX/UI-дизайна.

Грань между UX (**User Experience**) и UI (**User Interface**) очень тонка, но если разобраться, то становится ясно, что UX помогает понять пользователя. В UX-дизайне больше психологического аспекта, нежели технологического. UX изучает пользователя: как пользователь живет, что он думает, как и что делает, что его окружает. Перед дизайнером ставится задача - помочь обычному человеку легко разобраться с вашим программным продуктом и получить при этом удовлетворение от работы с ним.

А понять пользователя очень важно. Никому не захочется заполнить двадцать полей формы для регистрации на сайте или перещелкать штук пятнадцать вкладок, прежде чем добраться до нужной функции. «Пользователя не следует заставлять взаимодействовать с программой дольше, чем абсолютно необходимо для решения той или иной задачи» (из книги Алана Купера «Психбольница в руках пациентов»).

Этапы разработки пользовательского интерфейса

Полный цикл разработки интерфейса представлен на рис. 2.1.

Рис. 2.1. Этапы разработки пользовательского интерфейса

Для сокращения общего времени разработки определение стилистики начинается после пользовательских сценариев.

Исследование. На этапе исследования проводится сбор информации о продукте, клиенте, его конкурентах или близких аналогах, сбор статистики использования текущего интерфейса (например, сайта или мобильного приложения), анализ устройств предполагаемой целевой аудитории.

Если уже известно, кто будет воплощать интерфейс в жизнь (разработчики), то знакомимся с ними и выясняем их возможности и ограничения.

Этот этап помогает понять, для кого разрабатывается интерфейс, с какими ограничениями следует его делать (размеры экранов, интерактивность), как не стоит делать (например, быть непохожими на конкурентов).

Пользовательские сценарии. На основе предоставленного описания работы интерфейса создается список задач (пользовательских сценариев), которые может выполнять пользователь в рамках интерфейса. Например, обновить аватарку в профиле.

Все задачи расписываются по шагам, которые необходимо предпринять для решения задачи. Например:

- зайти на сайт;
- авторизоваться;
- перейти в профиль;
- нажать на аватарку;
- выбрать файл;
- подтвердить или изменить кадрирование изображения;
- сохранить.

Составленные списки шагов для каждой задачи помогают понять, где путь для решения слишком долог относительно остальных задач. Этап пользовательских сценариев больше всего подходит для сокращения пути решения задач пользователей в рамках интерфейса.

Пример выше можно сократить на несколько шагов. Например, сделать сохранение автоматическим, а обрезание изображения - опциональным.

Структура интерфейса. Полученный список шагов на предыдущем этапе ложится в основу структуры интерфейса. Становится известно количество экранов, их краткое содержание и положение в общей структуре. На данном этапе строится карта экранов ***(User Flow Diagram)***. Пример карты экранов приведен на рис. 2.2.

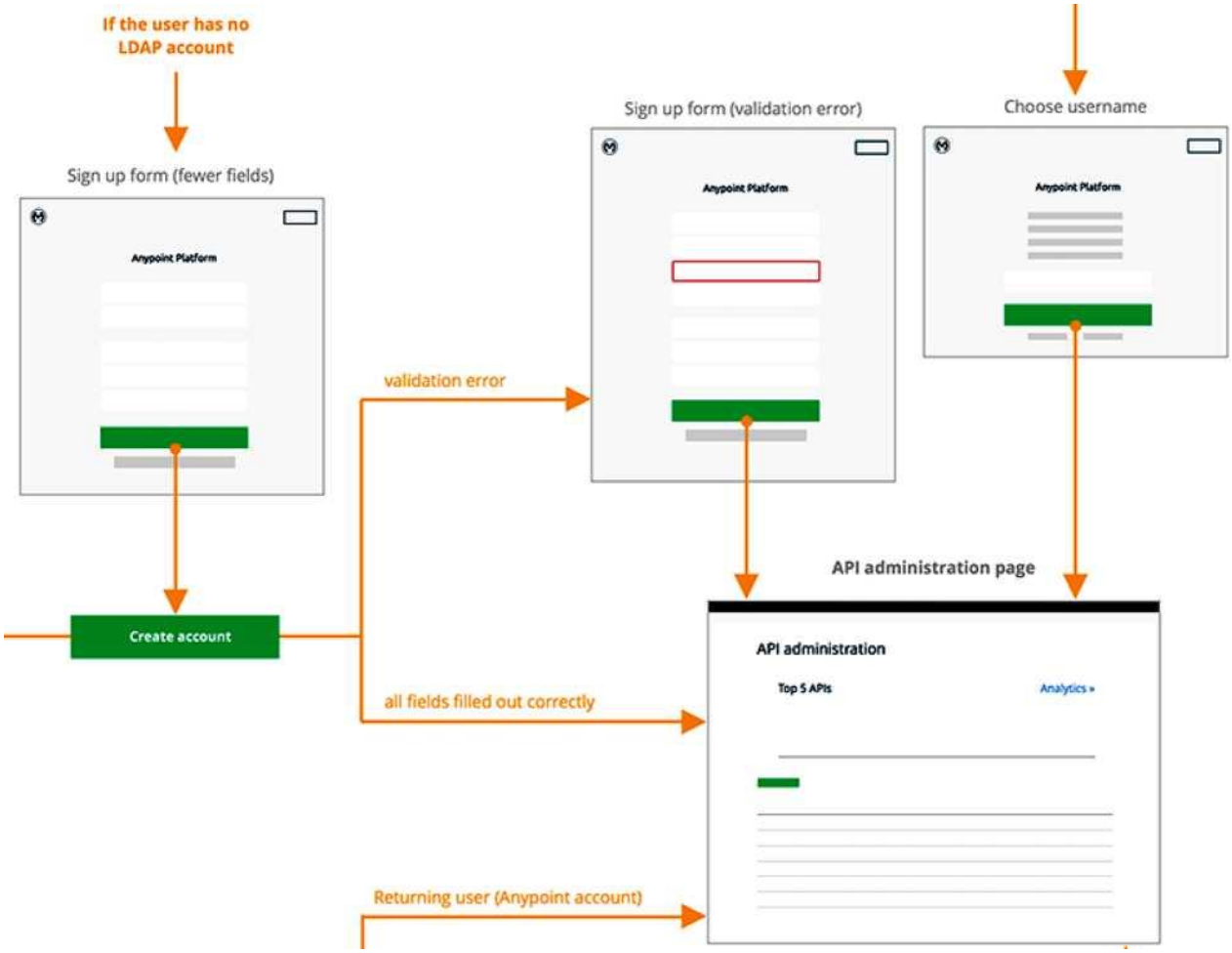


Рис. 2.2. Фрагмент карты экранов *(User Flow Diagram)*

Прототипирование интерфейса. В большинстве случаев реализуется два схематичных прототипа: черновой и финальный. Исключения составляют небольшие интерфейсы: простенькие мобильные приложения или маленькие сайты.

Черновой прототип представляет собой схематичные изображения экранов, связанные между собой. При черновом варианте на схемах изображены зоны и описания этих зон. Например, список новостей или шапка сайта. Все без деталей.

Черновой прототип помогает более наглядно понять, насколько объемным будет приложение, как много информации будет на каждом экране, как много нужно кликать, чтобы добраться до нужной страницы.

Следующим шагом идет финальный прототип, в котором схемы страниц все еще остаются связанными между собой, но на страницах уже видны все кнопки, тексты, чекбоксы, формы и прочие элементы.

Пример чернового прототипа интернет-магазина приведен на рис. 2.3.

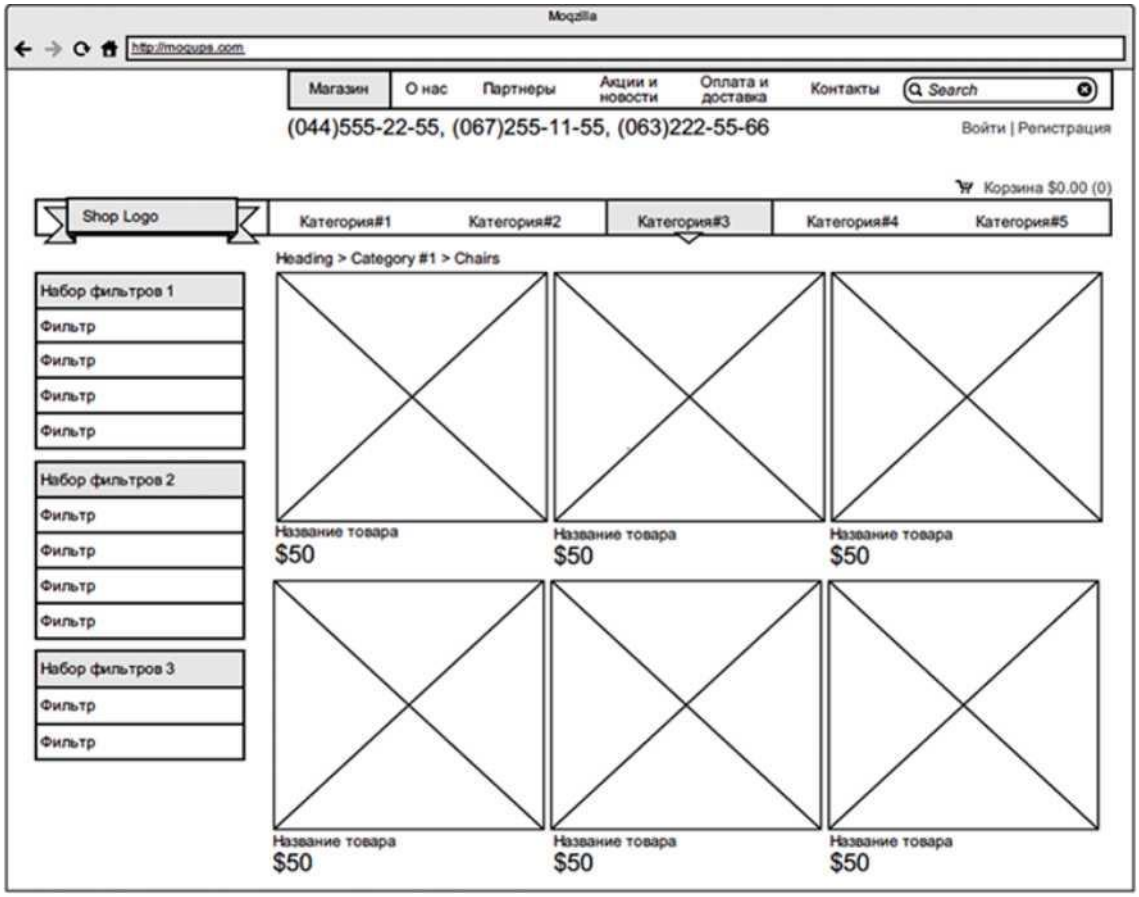


Рис. 2.3. Пример чернового прототипа интернет-магазина

В прототипах планируется функционал, расположение элементов страниц относительно друг друга, но никак не оформление. Цвета, изображения, иконки - это все этап оформления. На этапе проектирования невозможно сказать, как они будут взаимодействовать между собой, как будут смотреться вместе, будут ли перекрикивать друг друга.

Определение стилистики. После этапа исследования и параллельно с этапами проектирования идет определение будущей стилистики интерфейса.

Для выбора стилистики готовятся несколько наборов изображений ***(moodboards)***. Эти наборы представлены страничками сайтов, иллюстрациями, кнопками, шрифтовыми композициями, связанными между собой стилистически.

Существует множество различных концепций, например: ***material design***, ***metro***, ***skeuomorphism*** и т.д. При выборе стиля интерфейса следует учесть текущие тенденции в дизайне, адаптивность, время на разработку и внедрение дизайна, и много других не менее важных моментов.

UI*-кит* - набор готовых решений пользовательского интерфейса. Это могут быть кнопки, поля ввода, «хлебные крошки», меню, переключатели, формы - все те элементы, что помогают пользователям взаимодействовать с сайтом или приложением.

Пример набора элементов из ***Flat UI-кита*** приведен на рис. 2.4.

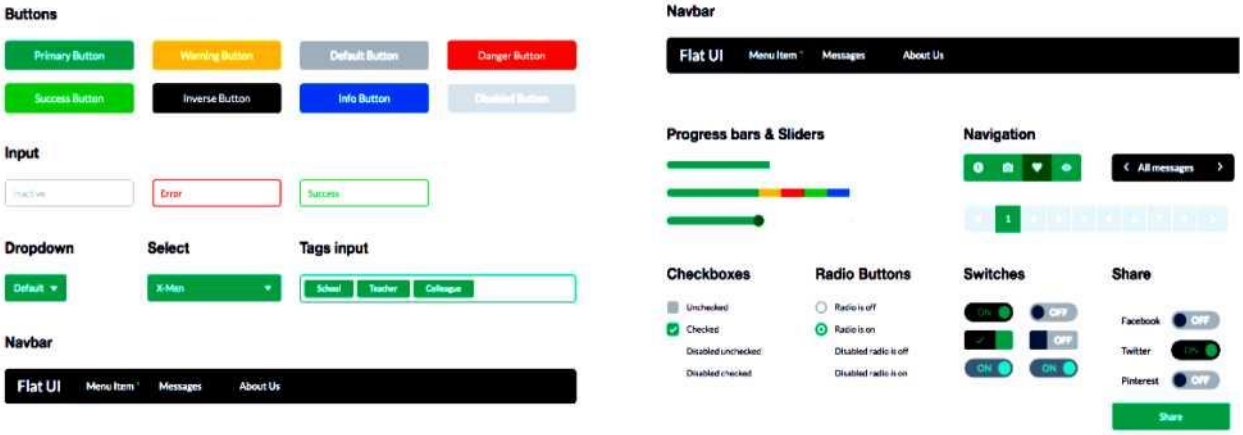


Рис. 2.4. Пример набора элементов из ***Flat UI***-кита

Обычно готовый кит представляет собой графику в слоях для работы в ***Photoshop*** или ***Sketch***. В документе хранятся элементы для дизайна интерфейсов, которые можно использовать в неизменном виде или редактировать их и подстраивать под стиль проекта.

Дизайн-концепция. Она призвана показать оформление приложения и будущий вид всего приложения. Если предыдущий этап определения стилистики только дал направление, то дизайн-концепция призвана скрестить выбранное направление с имеющимся содержанием интерфейса.

Дизайн-концепция может быть представлена любым объемом, но стараются его минимизировать для экономии времени. Обычно концепция представлена 1-3 экранами интерфейса. Если речь идет о сайте, то стараются показать вид одной и той же страницы для нескольких устройств.

Для разработки дизайн-концепции используются online-инструменты:

- https://www.axure.com
- http://mockupbuilder.com
- https://www.fluidui.com
- и т.д.

Оформление всех экранов. После утверждения дизайн концепции настает время оформления всех остальных экранов интерфейса. Дизайн- концепция - это предположение о том, как может выглядеть весь интерфейс. Когда же очередь доходит до оформления всех экранов, тогда и происходит финализация внешнего вида: становится ясно, правильно ли подобраны кегль или интерлиньяж, хорошо ли сочетается толщина линий иконок с текстом, не конфликтует ли оформление форм (кнопок, полей ввода) с другими элементами экрана и многое другое.

Планом для оформления всех экранов являются структура и схематичный прототип интерфейса. Однако случаются отхождения от этого плана. Так при оформлении может выясниться, что всплывающее окно будет намного нагляднее и эффективнее, чем разъезжающийся блок информации посреди экрана.

Все оформленные экраны собираются в интерактивный прототип, который создаст максимально приближенный опыт использования интерфейса без прибегания к услугам разработчиков.

Анимация интерфейса. Часто этот этап начинается еще с момента дизайн-концепции и продолжается на протяжении оформления всех экранов.

Стараются показать только какие-либо нестандартные случаи анимации интерфейса, которые не предусмотрены операционной системой. Например, нету никакой надобности показывать, с какой скоростью будет выезжать следующий экран в интерфейсе приложения. Однако это тоже можно считать анимацией интерфейса.

Для ***Material design*** есть гайдлайны, которые наглядно объясняют, как надо анимировать и как не надо.

Эти гайдлайны подходят для анимации интерфейсов любой платформы.

Подготовка материалов для разработчиков. На данном этапе уже присутствуют макеты интерфейса во всех состояниях, прототип, связывающий весь интерфейс воедино и видеоролики, показывающие анимацию. Чтобы помочь разработчикам в реализации интерфейса, дизайнеры готовят все необходимые для этого материалы:

- спрайты;
- шрифт со всеми иконками;
- ***UI****-Kit* с повторяющимися элементами интерфейса и их состояниями.

Для иконок и прочей графики из интерфейса, для всех расстояний, отступов, размеров используют специальное программное обеспечение, например, Zeplin, которое самостоятельно готовит иконки и код.

Задание на лабораторную работу

Разработать пользовательский интерфейс для десктопного клиента, описанного в лабораторной работе № 1. Разработку пользовательского интерфейса производить согласно п. 2. настоящего пособия.

1. Произвести анализ аналогов, выявить их достоинства и недостатки, результаты анализа отразить в отчете.

2. Разработать пользовательские сценарии и привести их часть в отчете.

3. Разработать карту экранов в части описанных пользовательских сценариев.
4. Разработать черновой прототип экранов.
5. Подобрать подходящие стилистики для приложения не менее двух.
6. На основании одной из стилистик разработать дизайн концепцию приложения.

Вопросы для самопроверки

1. В чем отличие понятий UI и UX?
2. Какие этапы включает разработка пользовательского интерфейса?
3. Что такое UI-кит?
4. Для чего разрабатывают дизайн-концепцию?
5. В чем отличие чернового прототипа интерфейса от финального?

Практическая работа № 1.39. Проектирование пользовательского интерфейса мобильного приложения. Практическая работа № 1.40. Проектирование пользовательского интерфейса мобильного приложения

Цель работы: изучение принципов проектирования пользовательского интерфейса мобильного приложения.

Теоретический материал

Общие сведения

Мобильное приложение ***(Mobile app)*** - программное обеспечение, предназначенное для работы на смартфонах, планшетах и других мобильных устройствах.

Первоначально мобильные приложения использовались для быстрой проверки электронной почты, но их высокий спрос привел к расширению их назначений и в других областях, таких как игры для мобильных телефонов и GPS, общение, просмотр видео и пользование интернетом.

Поведенческие шаблоны

Использование мобильных гаджетов вращается вокруг множества нюансов, которые нельзя не принимать во внимание, например, расположение большого пальца.

По этой причине навигационные кнопки, как правило, находятся в нижней части экрана. Пример расположения навигационных кнопок показан на рис. 3.1.

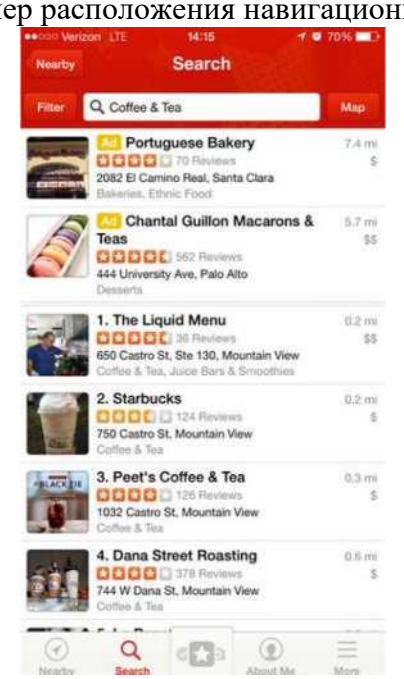


Рис. 3.1. Пример расположения навигационных кнопок

Предполагается два вида взаимодействия: жесты и анимация.

Пользователи уже привыкли к возможности использовать разные жесты для различных ситуаций. Типовые жесты приведены на рис. 3.2.

Рис. 3.2. Типовые жесты для взаимодействия с мобильным приложением

И распространенные типы анимации также вызывают ряд ожиданий, основанных на предыдущих опытах взаимодействия с приложениями.

Учет движений

Анатомический фактор - очень важный элемент проектирования. Учитывайте строение тела человека и статистику использования мобильных устройств при проектировании. Левый верхний угол подходит для размещения важной информации, в то время как нижняя граница экрана - для навигации.

Схемы наиболее удобных для человека жестов представлены на рис. 3.3.

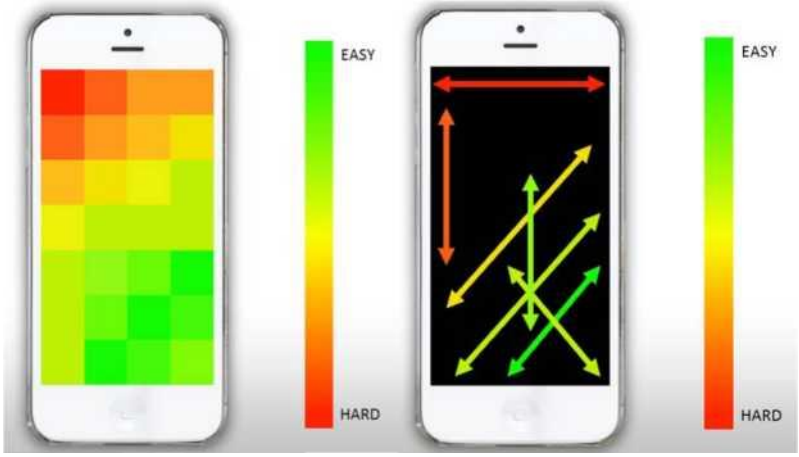


Рис. 3.3. Схемы наиболее удобных для человека жестов

Глобальный график с распределением ориентации смартфона при работе пользователей приведен на рис. 3.4.

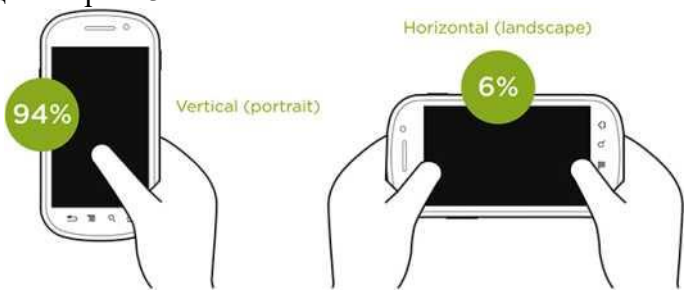


Рис. 3.4. Ориентация смартфона при работе

Почти половину времени пользователи проводят, держа устройство правой рукой и используя для работы с экраном только большой палец. Распределение положения рук при работе со смартфоном приведено на рис. 3.5.

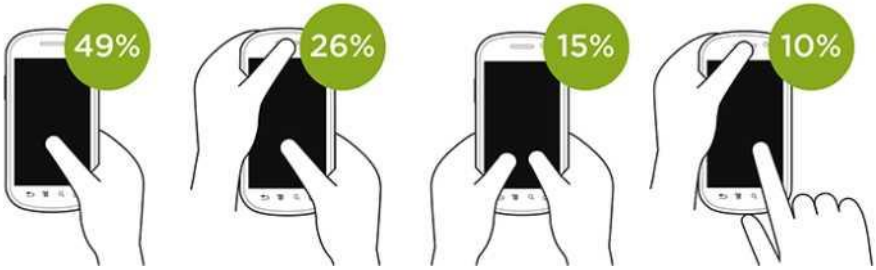


Рис. 3.5. Глобальное распределение положения рук при работе со смартфоном

На рис. 3.6 приведено глобальное распределение смартфонов по размеру диагонали.

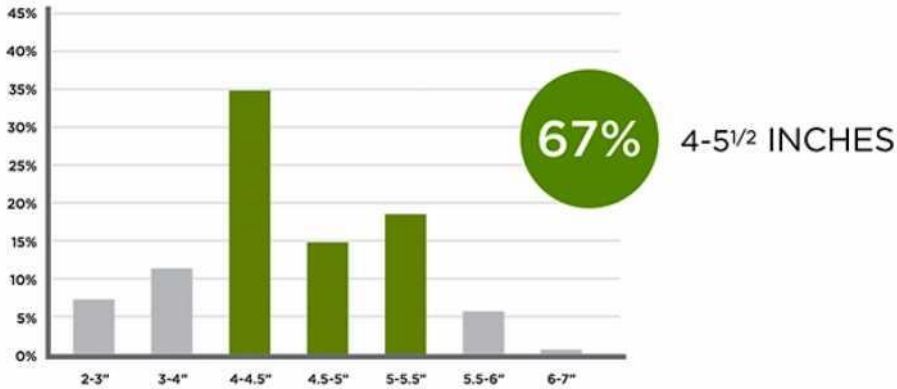


Рис. 3.6. Глобальное распределение смартфонов по размеру диагонали

Из рис. 3.6 видно, что большинство пользователей используют смартфоны с диагональю экрана в пределах 4-5,5 дюймов.

Задание на лабораторную работу

Разработать пользовательский интерфейс для мобильного клиента, описанного в лабораторной работе № 1. Разработку пользовательского интерфейса производить с учетом п. 3 согласно п. 2 настоящего пособия.

1. Произвести анализ аналогов, выявить их достоинства и недостатки, результаты анализа отразить в отчете.
2. Разработать пользовательские сценарии и привести их часть в отчете.
3. Разработать карту экранов в части описанных пользовательских сценариев.
4. Разработать черновой прототип экранов.
5. Подобрать подходящие стилистики для приложения не менее двух.
6. На основании одной из стилистик разработать дизайн-концепцию приложения для экранов диагональю 4 дюйма и 6 дюймов.

Вопросы для самопроверки

1. Опишите основные шаги при проектировании интерфейса мобильного приложения.
2. В чем отличие material design для ОС Android от Apple Human Interface Guidelines для iOS?
3. Для какой диагонали смартфонов проектированием интерфейса можно пренебречь?
4. На какое место на экране обычно размещают панель навигации?
5. Назовите типовые жесты для взаимодействия с приложением.

Практическая работа № 1.41. Адаптивный веб-дизайн. Практическая работа № 1.42. Адаптивный веб-дизайн

Цель работы: изучить принципы проектирования пользовательских интерфейсов сайта, обеспечивающих его правильное отображение на различных устройствах, подключенных к интернету и динамически подстраивающихся под заданные размеры окна браузера.

Теоретический материал

Общие сведения

Целью адаптивного веб-дизайна *(responsive web design, RWD)* является универсальность отображения содержимого веб-сайта для устройств, для того чтобы веб-сайт был удобно просматриваемым с устройств различных форматов и с экранами различных разрешений. По технологии адаптивного веб-дизайна не нужно создавать отдельные версии веб-сайта для отдельных видов устройств. Один сайт может работать на смартфоне, планшете, ноутбуке и телевизоре с выходом в Интернет, т.е. на всем спектре устройств. Пример применения адаптивного дизайна приведен на рис. 4.1

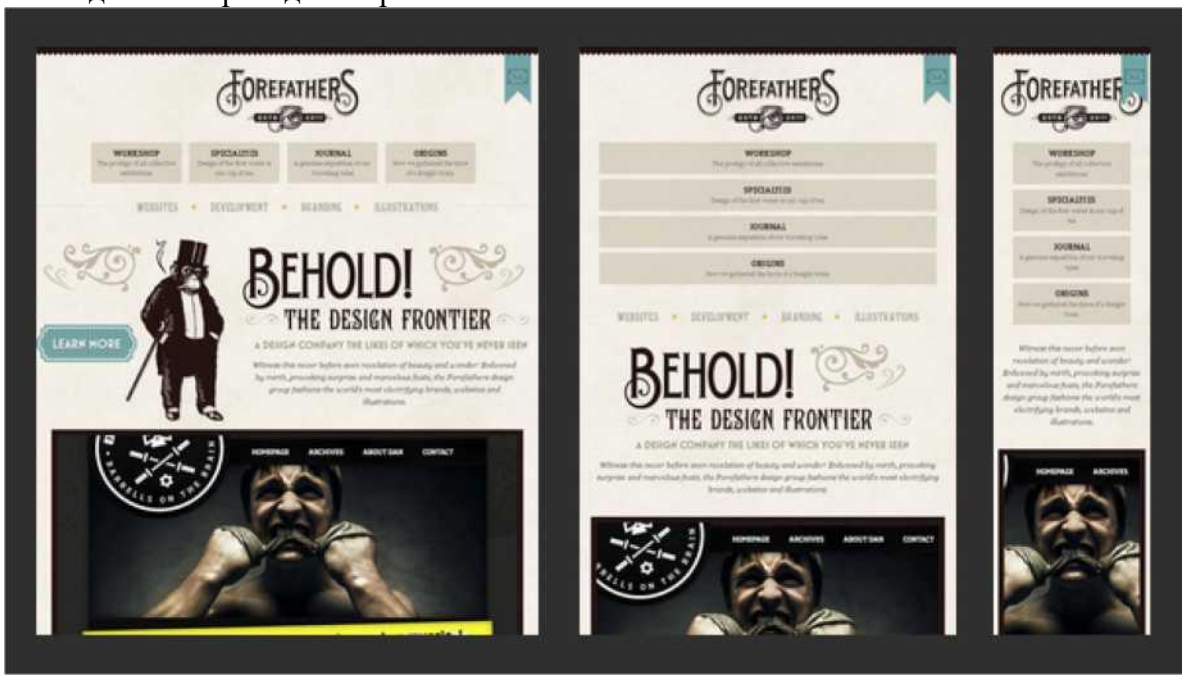


Рис. 4.1. Пример адаптивного веб-дизайна

Применение адаптивного веб-дизайна обусловлено следующими аспектами:

большое разнообразие устройств, с которых можно выходить в Интернет. В настоящее время существует множество устройств, которыми люди пользуются, в том числе и для того, чтобы выходить в Интернет. Все они различаются размером экрана, разрешением и, соответственно, тем, как может отображаться на них веб-сайт. Поэтому важно, чтобы ваш сайт хорошо смотрелся и правильно отображался у любого из пользователей, независимо от того, какое устройство он использует;

популярность мобильных устройств с выходом в Интернет и увеличение мобильного интернет-трафика. С ростом популярности мобильных устройств количества пользователей, которые заходят с них на сайты, заметно увеличилось, поэтому просто игнорировать их уже нельзя - это не один-два человека в полгода, это значительная часть вашей аудитории, и им должно быть удобно пользоваться вашим сайтом (иначе они не будут этого делать);

срочная информация. Если ваш ресурс содержит новостную/ срочную информацию, высока вероятность, что пользователю может понадобится прочитать эту информацию именно с телефона (потому что других устройств у него под рукой нет) в данный момент времени, нужно позаботиться о том, чтобы у него была возможность это сделать.

Отличие адаптивного сайта от мобильной версии (приложения) сайта

Мобильные версии сайтов и мобильные приложения, специально разработанные для различных мобильных устройств, также решают проблему с удобством просмотра сайта, но имеют некоторые недостатки:

под каждый тип операционной системы нужно свое приложе- ние/версия сайта. Это требует дополнительных ресурсов как временных, так и денежных;

необходимость загрузки приложения. Для того, чтобы пользоваться вашим приложением, пользователям необходимо его загрузить. Это требует каких-то дополнительных усилий от пользователей, и многие не будут этого делать, если точно не уверены, что приложение им очень нужно, и они планируют регулярно его использовать;

разделение трафика. С точки зрения продвижения сайта мобильное приложение не удобно тем, что разделяет весь трафик ресурса на трафик сайта и трафик приложения, что будет выглядеть, как меньшая посещаемость сайта;

необходимость интеграции материалов сайта. В случае с мобильным приложением необходимо либо синхронизировать сайт с приложением (дополнительные ресурсы), либо делать двойную работу по наполнению сайта и приложения материалами.

В отличие от мобильных приложений, адаптивный дизайн - это один адрес сайта, один дизайн, одна система управления и содержание сайта.

Недостатки адаптивного дизайна

Адаптивный дизайн может дать многочисленные преимущества, но абсурдно предположение, что он подходит для всех веб-проектов. Можно выделить следующие недостатки:

зачастую адаптивный дизайн приводит к неоправданному увеличению времени загрузки сайта на мобильных устройствах. Концепция RWD подразумевает, что пользователи всех платформ получают одинаковый контент, оптимизированный под конкретные разрешения экранов.

В то же время на сайте может быть масса информации, показывать которую мобильным пользователям нет никакой необходимости. Однако она загружается автоматически, как только они заходят на сайт, и зачастую это происходит в местности со слабым сигналом сотовой связи;

пользователи не смогут посмотреть полную версию. На большинстве неадаптивных мобильных сайтов внизу страницы присутствует опция «Переключиться на полную версию» или другая аналогичная кнопка. Она позволяет посетителям просматривать сайт для настольных компьютеров, минуя таблицы стилей для мобильной версии. Пример для сайта kinopoisk.ru приведен на рис. 4.2.

ft https://www.kinopoisk.ru/'>fo O

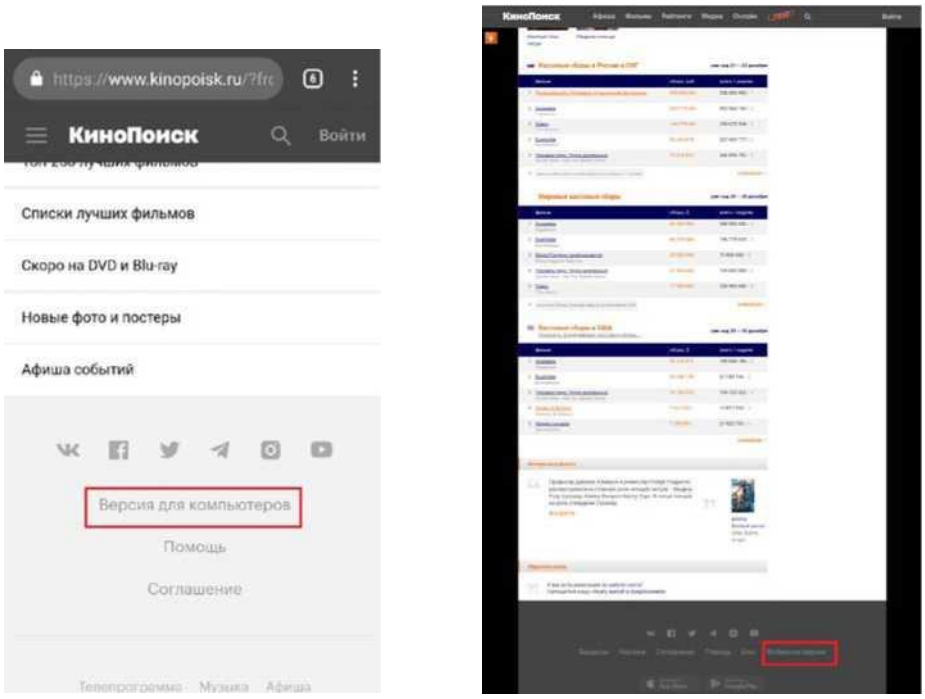


Рис. 4.2. Пример перехода к разным версиям сайта

Принципы адаптивности

Основными принципами при проектировании адаптивного вебдизайна являются:

1) *проектирование для мобильных устройств с самых ранних этапов ("mobile first""}.* Проектирование начинается с адаптивной версии веб-сайта для мобильных устройств. На этом этапе дизайнеры стремятся правильно передать смысл и основные идеи с использованием небольшого экрана и всего одной колонки. Содержимое при необходимости сокращают, удаляя второстепенные информационные блоки и оставляя самое важное;

2) *работа с медиазапросами (media queries').* Запросы определяют:

- тип устройств: проекторы, смартфоны, мониторы, телевизоры и пр.;
- условия.

На соответствующий запрос и ответ будут применяться соответствующие устройству параметры отображения из файла стилей css;

3) *применение гибкого макета на основе сетки (flexible, grid-based layout) и использование гибких изображений (flexible images).*

Рассмотрим основные виды адаптивных макетов, существующие в настоящее время:

а) *резиновый.* Простой в реализации и очевидный для пользователя тип представления сайта. Основные блоки сжимаются до ширины экрана мобильного устройства, где такое невозможно - перестраиваются в одну длинную ленту;

б) *перенос блоков.* Очевидный способ для многоколоночного сайта: при уменьшении ширины экрана дополнительные блоки (сайдбары) переносятся в нижнюю часть макета (рис. 4.3);

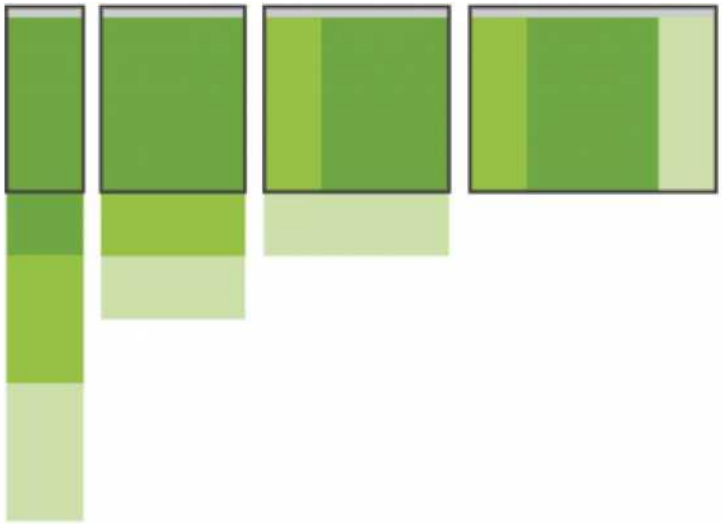


Рис. 4.3. Способ «перенос блоков»

в) *переключение макетов.* Этот способ наиболее удобен при чтении сайта с различных устройств: под каждое разрешение экрана разрабатывается отдельный макет. Способ трудоемок, поэтому менее популярен, чем предыдущие два (рис. 4.4);

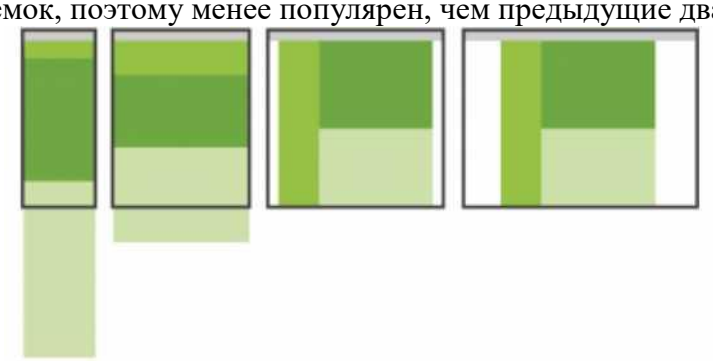


Рис. 4.4. Способ «переключение макетов»

г) *адаптивность «малой кровью».* Очень простой способ, который подходит для несложных сайтов. Достигается элементарным масштабированием изображений и типографики. Не очень популярен, так как не обладает гибкостью (рис. 4.5);

UULM

Рис. 4.5. Способ «адаптивность «малой кровью»

д) *панели.* Способ, пришедший из мобильных приложений, где дополнительное меню может появляться при горизонтальном или вертикальном этапе. Главный недостаток - неочевидность действий для пользователя: очень непривычно видеть мобильную навигацию на веб-сайте (рис. 4.6).

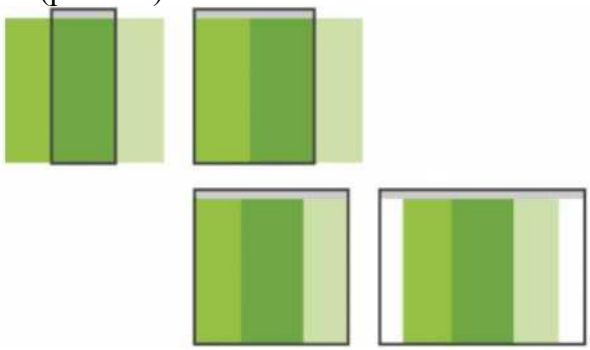


Рис. 4.6. Способ «панели»

Нужно помнить, что представленные выше макеты не являются универсальными решениями - для каждого проекта необходимо выбирать наиболее подходящий под нужды и возможности способ.

Задание на лабораторную работу

Разработать пользовательский интерфейс для веб-клиента, описанного в лабораторной работе № 1. Подготовить макеты для отображения не менее трех страниц для ширины экрана следующего разрешения:

- для смартфонов 320 px, 480 px;
- планшетов 768 px;
- нетбуков и некоторых планшетов 1024 px;
- персональных компьютеров 1280 px и более.

Вопросы для самопроверки

1. Раскройте понятие «адаптивный веб-дизайн».
2. Зачем нужен адаптивный веб-дизайн?
3. В чем отличие адаптивного сайта от мобильной версии (приложения) сайта?
4. Каковы недостатки адаптивного веб-дизайна?
5. Перечислите основные виды адаптивных макетов.

Практическая работа № 1.43. Разработка протокола взаимодействия веб-сервисов

Цель работы: изучить принципы проектирования протокола взаимодействия веб-сервисов с использованием протокола SOAP.

Теоретический материал

Веб-сервисы

Веб-служба, веб-сервис (англ. *web service)* - идентифицируемая уникальным веб-адресом *(URL-адресом)* программная система со стандартизированными интерфейсами, а также *HTML-документ* сайта, отображаемый браузером пользователя.

Веб-службы могут взаимодействовать друг с другом и со сторонними приложениями посредством сообщений, основанных на определенных протоколах *(SOAP, XML-RPC* и т.д.) и соглашениях *(REST)*. Вебслужба является единицей модульности при использовании сервис-ориентированной архитектуры приложения.

На сегодняшний день наибольшее распространение получили следующие протоколы реализации веб-сервисов:

- *SOAP (Simple Object Access Protocol)* — по сути это тройка стандартов *SOAP/WSDL/UDDI;*
- *REST (Representational State Transfer);*
- *XML-RPC (XML Remote Procedure Call).*

На самом деле *SOAP* произошел от *XML-RPC* и является следующей ступенью его развития. В то время как *REST* - это концепция, в основе которой лежит скорее архитектурный стиль, нежели новая технология, основанный на теории манипуляции объектами *CRUD (Create Read Update Delete*) в контексте концепций *WWW*.

Концепция построения веб-сервисов с использованием SOAP

Веб-сервис идентифицируется строкой *URI*. Веб-сервис имеет программный интерфейс, представленный в машинно-обрабатываемом формате *WSDL.* Другие системы взаимодействуют с этим веб-сервисом путем обмена сообщениями протокола *SOAP.* В качестве транспорта для сообщений используется протокол *HTTP.* Описание веб-сервисов и их *API* могут быть найдены средствами *UDDI.* Концептуальная схема технологии приведена на рис. 5.1, где:

- *SOAP (Simple Object Access Protocol)* - протокол обмена сообщениями между потребителем и поставщиком веб-сервиса;
- *WSDL (Web Services Description Language)* - язык описания внешних интерфейсов веб-службы;
- *UDDI (Universal Discovery, Description and Integration)* - универсальный интерфейс распознавания, описания и интеграции, используемый для формирования каталога веб-сервисов и доступа к нему.

Рис. 5.1. Концепция веб-сервиса

Связь между протоколами приведена на рис. 5.2.

UDDI

Публикация и поиск сервисов

WSDL

x Описание интерфейсов

SOAP

Обмен сообщениями

HTTP, SMTP, FTP, HOP ...

Транспортная инфраструктура

Рис. 5.2. Протоколы веб-сервисов

Все спецификации, используемые в технологии, основаны на *XML* и, соответственно, наследуют его преимущества (структурированность, гибкость и т.д.) и недостатки (громоздкость, медлительность).

SOAP

SOAP - простой протокол доступа к объектам (компонентам распределенной вычислительной системы), основанный на обмене структурированными сообщениями. Как любой текстовый протокол, *SOAP* может использоваться с любым протоколом прикладного уровня: *SMTP*, *FTP*, *HTTPS* и др., но чаще всего *SOAP* используется поверх *HTTP*.

Все сообщения *SOAP* оформляются в виде структуры, называемой конвертом *(envelop)*, включающей следующие элементы:

- идентификатор сообщения (локальное имя);
- опциональный элемент Header (заголовок);
- ноль или более ссылок на используемые пространства имен;
- ноль или более свойств, доступных в этом пространстве имен;
- обязательный элемент Body (тело сообщения);
- ноль или более ссылок на используемые пространства имен;
- дочерние элементы тела сообщения.

Пример SOAP-запроса на сервер интернет-магазина:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<getProductDetails xmlns="http://warehouse.example.com/ws">
<productID>12345</productID>
</getProductDetails>
</soap:Body>
</soap:Envelope>
```

Пример ответа:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
```

```
<getProductDetailsResponse xmlns="http://warehouse.example.com/ws">
<getProductDetailsResult>
<productID>12345</productID>
<productName>СТаКаН гранёный</productName>
<description>СТаКаН граненый. 250 М.л.</description>
<price>9.95</price>
<currency>
<code>840</code>
<alpha3>USD</alpha3>
<sign>$</sign>
<name>US dollar</name>
<accuracy>2</accuracy>
</currency>
<inStock>true</inStock>
</getProductDetailsResult>
</getProductDetailsResponse>
</soap:Body>
</soap:Envelope>
```


WSDL

Язык описания веб-сервисов *(Web services Description Language,* WSDL) предназначен для унифицированного представления внешних интерфейсов веб-службы. Текущая версия протокола (на момент написания этой лекции) *WSDL 2.0* и она имеет некоторые отличия от предыдущих версий приведенные в табл. 5.1 и на рис. 5.3.

Таблица 5.1

Основные элементы протокола WSDL

Элемент WSDL 1.1	Элемент WSDL 2.0	Краткое описание
PortType	Interface	Представляет описание интерфейса веб-сервиса (список операций и их параметров)
Service	Service	Список системных функций
Binding	Binding	Специфицирует интерфейсы и задает параметры связывания с протоколом SOAP: стиль связывания (RPC/Document) и транспорт (SOAP). Эта секция доступна и для каждой из операций
Operation	Operation	Определяет операцию, представляемую веб-сервером. WSDL-операция — это аналог традиционным функциям и процедурам
Message	—	Сообщение, связанное с определенной операцией. Содержит информацию, необходимую для выполнения данной операции. Каждое сообщение может состоять из нескольких логических частей, описывающих типы данных и имена атрибутов. В версии 2.0 было исключено, так как была внедрена поддержка XML Schema для всех элементов
Types	Types	Описание данных в соответствии с XML Schema

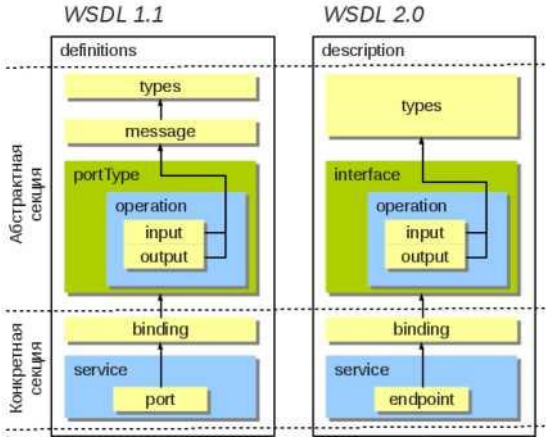


Рис. 5.3. Структура протокола WSDL

В спецификации WSDL 1.1 было определено четыре шаблона обмена сообщениями (типы операций):

- однонаправленные операции (One-way): операция может принимать сообщение, но не будет возвращать ответ;
- запрос-ответ (Request-response): операция может принять запрос и должна вернуть ответ;
- вопрос-ответ (Solicit-response): операция может послать запрос и будет ждать ответ на него;
- извещение (Notification): операция может послать сообщение, но не будет ожидать ответ.

В версии WSDL 2.0 эти шаблоны изменены и расширены в сторону поддержки сообщений об ошибках (например, шаблон Robust-in-only), но для обратной совместимости поддерживаются типы WSDL 1.1.

Пример описания веб-сервиса на языке WSDL (версия 2.0):

```
<?xml version="1.0"?>
<wsdl:description xmlns:wsdl="http://www.w3.org/ns/wsdl"
xmlns:wsoap= "http://www.w3.org/ns/wsdl/soap"
xmlns:hy="http://www.herongyang.com/Service/"
targetNamespace="http://www.herongyang.com/Service/">
<wsdl:documentation>
Hello_WSDL_20_SOAP.wsdl
Copyright (c) 2009 HerongYang.com, All Rights Reserved.
</wsdl:documentation>
<wsdl:types>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.herongyang.com/Service/">
<xsd:element name="Hello" type="xsd:string"/>
<xsd:element name="HelloResponse" type="xsd:string"/>
</xsd:schema>
</wsdl:types>
<wsdl:interface name="helloInterface" >
<wsdl:operation name="Hello"
pattern="http://www.w3.org/ns/wsdl/in-out"
style="http://www.w3.org/ns/wsdl/style/iri">
<wsdl:input messageLabel="In"
element="hy:Hello" />
<wsdl:output messageLabel="Out" element="hy:HelloResponse" />
</wsdl:operation>
</wsdl:interface>
<wsdl:binding name="helloBinding"
interface="hy:helloInterface"
type="http://www.w3.org/ns/wsdl/soap"
wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/">
<wsdl:operation ref="hy:Hello"
```

```
wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"/> </wsdl:binding>
<wsdl:service name="helloService"
interface="hy:helloInterface">
<wsdl:endpoint name="helloEndpoint"
binding="hy:helloBinding"
address="http://www.herongyang.com/Service/Hello_SOAP_12.php"/> </wsdl:service>
</wsdl:description>
```

В данном примере:

- веб-сервис helloService определен с эндпойнтом helloEndpoint, доступным по адресу http://www.herongyang.com/Service/Hello_ SOAP_12.php;
- эндпойнт helloEndpoint ссылается на связывание helloBinding;
- связывание helloBinding определено с использованием протокола SOAP 1.2 поверх HTTP;
- связывание helloBinding ссылается на интерфейс hellointerface;
- интерфейс helloInterface определеляет операцию Hello, которая требует элементы входящего и исходящего сообщений;
- каждый элемент Hello/HelloResponse определяет в XML-схеме секции types .

Пример описания сложных структур и списков:

```
<types>
<xs:schema xmlns:tns="http://schemas.xmlsoap.org/wsdl/"
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:xs="http://www.w3.org/2001/XMLSchema" element-
FormDefault="qualified" targetNamespace="http://localhost/">
<complexType name="Message">
<sequence>
<element name="phone" type="string" minOccurs="1" maxOccurs="1"
/>
<element name="text" type="string" minOccurs="1" maxOccurs="1" />
<element name="date" type="dateTime" minOccurs="1" maxOccurs="1"
/>
<element name="type" type="decimal" minOccurs="1" maxOccurs="1"
/>
</sequence>
</complexType>
<complexType name="MessageList">
<sequence>
<element minOccurs="1" maxOccurs="unbounded" name="message"
type="Message"/>
</sequence>
</complexType>
<element name="Request">
<element name="messageList" type="MessageList" />
</element>
<element name="Response">
<complexType>
<sequence>
<element name="status" type="boolean" />
</sequence>
</complexType>
</element>
</xs:schema>
</types>
```

В данном примере представлено:

- описание структуры Message, состоящей из полей phone, text, date, type;
- описание списка сообщений MessageList, состоящего из неограниченного количества элементов типа Message;

- описание запроса Request со списком сообщений;
- описание ответа Response с булевской переменной, содержащей статус выполнения запроса.

UDDI

Universal Description, Discovery and Integration (UDDI - универсальный интерфейс распознавания, описания и интеграции) - открытый стандарт, утвержденный OASIS, определяющий методы публикации и обнаружения сетевых программных компонентов сервис-ориентированной архитектуры (SOA). В практической реализации UDDI представляет собой сетевой реестр (службу каталогов), представляющий данные и метаданные о веб-сервисах и доступный по адресу: URL: http:// uddi.xml.org/services.

Задание на лабораторную работу

Разработать протокол взаимодействия сервера и клиентов, описанных в лабораторной работе № 1:

1. Подготовить WSDL-описание команд сервера.
2. Подготовить примеры SOAP запросов и ответов в соответствии с WSDL-описанием.

Вопросы для самопроверки

1. Что такое веб-сервис?
2. Что такое сервис-ориентированная архитектура?

3. Какие протоколы реализации веб-сервисов получили наибольшее распространение?
4. Опишите структуру SOAP сообщения.
5. В чем отличие спецификаций WSDL 1.1 от WSDL 2.0?

Практическая работа № 1.44. Разработка REST API. Практическая работа № 1.45. Разработка REST API

Цель работы: изучить принципы проектирования протокола взаимодействия веб-сервисов согласно архитектурному стилю взаимодействия компонентов распределенного приложения в сети REST.

Теоретический материал

REST

REST *(Representational state transfer)* - это стиль архитектуры программного обеспечения для распределенных систем, таких как World Wide Web, который, как правило, используется для построения вебслужб. Термин REST был введен в 2000 г. Роем Филдингом, одним из авторов HTTP-протокола. Системы, поддерживающие REST, называются RESTful-системами.

В общем случае REST является очень простым интерфейсом управления информацией без использования каких-то дополнительных внутренних прослоек. Каждая единица информации однозначно определяется глобальным идентификатором, таким как URL. Каждая URL в свою очередь имеет строго заданный формат.

Отсутствие дополнительных внутренних прослоек означает передачу данных в том же виде, что и сами данные, т.е. данные не заворачиваются в XML, как это делает SOAP и XML-RPC.

Каждая единица информации однозначно определяется URL - это значит, что URL, по сути, является первичным ключом для единицы данных, т.е. например третья книга с книжной полки будет иметь вид /book/3, а 35-я страница в этой книге - /book/3/page/35. Отсюда и получается строго заданный формат. Причем совершенно не имеет значения, в каком формате находятся данные по адресу /book/3/page/35 - это может быть и HTML, и отсканированная копия в виде jpeg-файла, и документ Microsoft Word.

Наиболее широко распространенные инструменты для описания RESTful API:

- www.mashape.com
- www.swagger.io
- www.apiary.io

HTTP-методы

Как происходит управление информацией сервиса - это целиком и полностью основывается на протоколе передачи данных. Наиболее распространенный протокол - HTTP. Для HTTP действие над данными задается с помощью методов.

Проектирование операций на HTTP-методы становится легче, когда вы знаете характеристики всех методов HTTP. Ниже представлены две характеристики, которые должны быть определены перед использованием HTTP-метода:

- **безопасность**: HTTP-метод считается безопасным, когда вызов этого метода не изменяет состояние данных. Например, когда вы извлекаете данные с помощью метода GET - это безопасно, потому что этот метод не обновляет данные на стороне сервера;
- **идемпотентность**: когда вы получаете один и тот же ответ, сколько раз вы вызываете один и тот же ресурс, он известен как идемпотентный. Например, когда вы пытаетесь обновить одни и те же данные на сервере, ответ будет таким же для каждого запроса, сделанного с одинаковыми данными.

Не все методы являются безопасными и идемпотентными. В табл. 6.1 представлен список методов, которые используются в REST-приложениях и показаны их свойства.

Таблица 6.1.

REST-методы

HTTP-метод	Безопасный	Идемпотентный
GET	Да	Да
POST	Нет	Нет
PUT	Нет	Да
DELETE	Нет	Да
OPTIONS	Да	Да
HEAD	Да	Да

Ниже приведен краткий обзор каждого метода и рекомендации по их использованию:

- **GET**: метод является безопасным и идемпотентным. Обычно используется для извлечения информации и не имеет побочных эффектов;
- **POST**: метод не является ни безопасным, ни идемпотентным. Этот метод наиболее широко используется для создания ресурсов;
- **PUT**: метод является идемпотентным. Вот почему лучше использовать этот метод вместо POST для обновления ресурсов. Избегайте использования POST для обновления ресурсов;
- **DELETE**: как следует из названия, метод используется для удаления ресурсов. Но этот метод не является идемпотентным для всех запросов;
- **OPTIONS**: метод не используется для каких-либо манипуляций с ресурсами. Но он полезен, когда клиент не знает других методов, поддерживаемых для ресурса, и используя этот метод, клиент может получить различное представление ресурса;
- **HEAD**: этот метод используется для запроса ресурса с сервера. Он очень похож на метод GET, но HEAD должен отправлять запрос и получать ответ только в заголовке. Согласно спецификации HTTP, этот метод не должен использовать тело для запроса и ответа;
- **HTTP**: определяет различные коды ответов для указания клиенту различной информации об операциях. Далее представлен список кодов ответов HTTP:
- 200 OK - это ответ на успешные GET, PUT, PATCH или DELETE. Данный код также используется для POST, который не приводит к созданию;
- 201 Created - данный код состояния является ответом на POST, который приводит к созданию;
- 204 Нет содержимого - это ответ на успешный запрос, который не будет возвращать тело (например, запрос DELETE);
- 304 Not Modified - используйте этот код состояния, когда заголовки HTTP-кеширования находятся в работе;
- 400 Bad Request - данный код состояния указывает, что запрос искажен, например, если тело не может быть проанализировано;
- 401 Unauthorized - данный код возвращается, если не указаны или недействительны данные аутентификации. Также полезно активировать всплывающее окно auth, если приложение используется из браузера;
- 403 Forbidden - данный код возвращается, когда аутентификация прошла успешно, но аутентифицированный пользователь не имеет доступа к ресурсу;
- 404 Not found - данный код возвращается, если запрашивается несуществующий ресурс;
- 405 Method Not Allowed - данный код возвращается, когда запрашивается HTTP-метод, который не разрешен для аутентифицированного пользователя;
- 410 Gone - данный код состояния указывает, что ресурс в этой конечной точке больше не доступен. Полезно в качестве защитного ответа для старых версий API;
- 415 Unsupported Media Type - данный код возвращается, если в качестве части запроса был указан неправильный тип содержимого;

- 429 Too Many Requests - данный код возвращается, когда запрос отклоняется из-за ограничения скорости;
- 500 - внутренняя ошибка сервера.

Маршрут отправки запроса

Маршрут - это адрес, по которому отправляется ваш запрос. Его структура примерно следующая:

root-endpoint/path?

- root-endpoint - это точка приема запроса на стороне сервера (API). К примеру, конечная точка GitHub - https://api.github.com;
- path - это путь, определяющий запрашиваемый ресурс, это что-то вроде автоответчика, который просит вас нажать 1 для одного сервиса, 2 - для другого и т.д.

Для понимания того, какие именно пути вам доступны, вам следует просмотреть документацию. К примеру, предположим, вы хотите получить список репозиториев для конкретного пользователя на Git. Согласно документации, вы можете использовать следующий путь для этого:

/users/:username/repos

Вам следует подставить под пропуск имя пользователя. Например, чтобы найти список репозиториев пользователя torvalds, вы можете использовать маршрут:

https://api.github.com/users/torvalds/repos

Последняя часть маршрута - это параметры запроса. Технически запросы не являются частью REST-архитектуры, но на практике сейчас все строится на них. Так что давайте поговорим о них более детально. Параметры запроса позволяют использовать в запросе наборы пар «ключ-значение». Они всегда начинаются знаком вопроса. Каждая пара параметров разделяется амперсантом:

?query1=value1&query2=value2

Для описанного ранее метода доступны параметры, описанные в табл. 6.2.

Таблица 6.2

Параметры метода

Наименование	Тип	Возможные значения	Значение по умолчанию
type	string	all, owner, member	owner
sort	string	created, updated, pushed, full name	full_name
	string	asc, desc	При использовании sort=full_name asc, в противном случае desc

Например, чтобы получить список недавно запущеных репозиториев пользователя torvalds, вам следует ввести следующее:

https://api.github.com/users/torvalds/repos?sort=pushed

JSON

JSON - JavaScript Object Notation - общий формат для отправки и приема данных посредством REST API. Ответ, отправляемый Github, также содержится в формате JSON.

JSON-текст представляет собой (в закодированном виде) одну из двух структур:

- набор пар ключ: значение. В различных языках это реализовано как объект, запись, структура, словарь, хэш-таблица, список с ключом или ассоциативный массив. Ключом может быть только строка (регистрозависимая: имена с буквами в разных регистрах считаются разными), значением - любая форма;
- упорядоченный набор значений. Во многих языках это реализовано как массив, вектор, список или последовательность.

В качестве значений в JSON могут быть использованы:

- **объект** - это неупорядоченное множество пар ключ: значение, заключенное в фигурные скобки «{ }». Ключ описывается строкой, между ним и значением стоит символ «:». Пары ключ-значение отделяются друг от друга запятыми;

- **массив (одномерный)** - это упорядоченное множество значений. Массив заключается в квадратные скобки «[ ]». Значения разделяются запятыми;
- **число**;
- **литералы** true, false и null.

Строка - это упорядоченное множество из нуля или более символов юникода, заключенное в двойные кавычки. Символы могут быть указаны с использованием escape-последовательностей, начинающихся с обратной косой черты «\»(поддерживаются варианты \', \", \\, \/, \t, \n, \r, \f и \b), или записаны шестнадцатеричным кодом в кодировке Unicode в виде \uFFFF.

Строка очень похожа на одноименный тип данных в языках C и Java. Число тоже очень похоже на C- или Java-число, за исключением того, что используется только десятичный формат. Пробелы могут быть вставлены между любыми двумя синтаксическими элементами.

Следующий пример показывает JSON-представление объекта, описывающего человека. В объекте есть строковые поля имени и фамилии, объект, описывающий адрес, и массив, содержащий список телефонов.

Как видно из примера, значение может представлять собой вложенную структуру.

```
{
"firstName": "Иван",
"lastName": "Иванов",
"address": {
"streetAddress": "Московское ш., 101, кв.101",
"city": "Ленинград",
"postalCode": "101101"
},
"phoneNumbers": [
"812 123-1234",
"916 123-4567"
]
}
```

Обратите внимание на пару "postalCode": "101101". В качестве значений в JSON могут быть использованы как число, так и строка. Поэтому запись "postalCode": "101101" содержит строку, а "postalCode": 101101 - уже числовое значение. Учитывая неопределенность типа переменных в JS (определены только типы значений), в дальнейшем, как правило, не возникает проблем с приведением типа. Но если данные в формате JSON обрабатываются в другой среде, отличной от JS, то нужно быть внимательным.

На языке XML подобная структура выглядела бы примерно так:

```
<person>
<firstName>Иван</firstName>
<lastName>Иванов</lastName>
<address>
<streetAddress>Московское ш., 101, кв.101</streetAddress>
<city>Ленинград</city>
<postalCode>101101</postalCode>
</address>
<phoneNumbers>
<phoneNumber>812 123-1234</phoneNumber>
<phoneNumber>916 123-4567</phoneNumber>
</phoneNumbers>
</person>
```

или так:

```
<person firstName="Иван" lastName="Иванов">
<address streetAddress="Московское ш., 101, кв.101" city="Ленинград" postalCode="101101" />
<phoneNumbers>
```

<phoneNumber>812 123-1234</phoneNumber>
<phoneNumber>916 123-4567</phoneNumber>
</phoneNumbers>
</person>

Задание на лабораторную работу

Разработать протокол взаимодействия сервера и клиентов, описанных в лабораторной работе № 1. Разработать следующие методы:

- GET без параметров и с параметрами;
- POST;
- PUT;
- DELETE.

Привести все коды возврата и описание возвращаемых данных в случае их наличия. Все данные в теле команд должны быть представлены в формате JSON.

Вопросы для самопроверки

1. В чем особенности архитектурного стиля взаимодействия компонентов распределенного приложения в сети REST?
2. Назовите наиболее широко распространенные инструменты для описания RESTful API.
3. Поясните следующие характеристики методов: «безопасность» и «идемпотентность».
4. Перечислите методы, которые используются в REST. Дайте их краткую характеристику.
5. Из каких составных частей состоит маршрут отправки запроса?

Практическая работа № 1.46.Теоретические основы Технологии ado.Net

Цель работы: изучить основы технологии ado.Net

Теоретическая часть

Технология ADO.NET

Microsoft ADO.NET (ActiveX Daia Objects) — объектная модель доступа к данным; набор средств, позволяющих приложению управлять и взаимодействовать со своим файловым или серверным хранилищем данных. Библиотеки ADO.NET включают классы, которые служат для подсоединения к источнику данных, выполнения запросов и обработки их результатов. Объектная модель ADO Net представлена на рисунке 1.

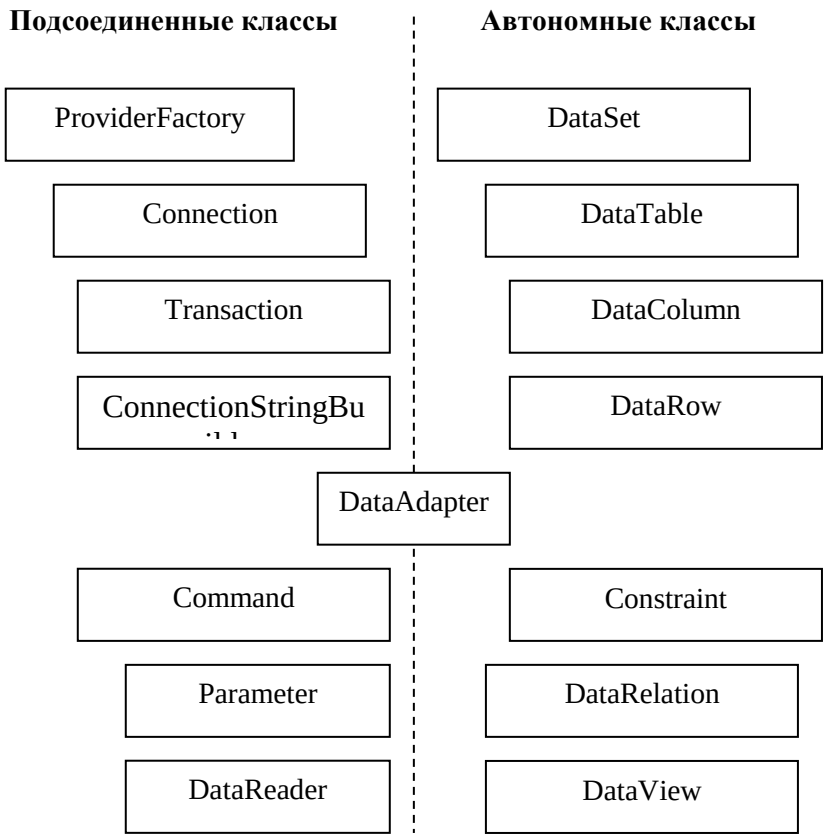


Рисунок 1 – Объектная модель ADO .Net

Подсоединенные объекты используются для управления соединением, транзакциями, для выборки данных и передачи изменений они взаимодействуют непосредственно с БД. Большинство подключенных объектов реализовано в рамках того, что называется поставщиками данных. Поставщики данных Net (NET data provider) - это набор классов, предназначенных для взаимодействия с хранилищем данных определенного типа (рисунок 2).

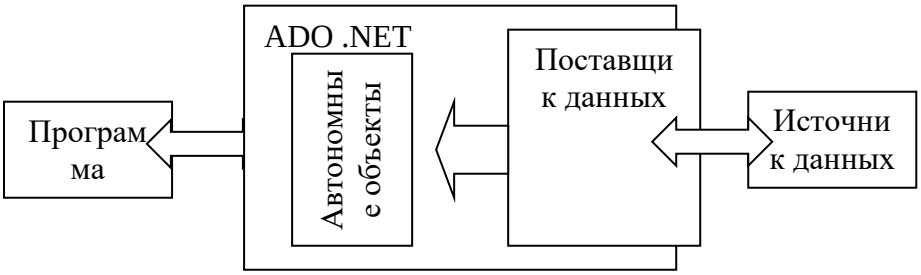


Рисунок 2 – Поставщики данных.Net

Каждый поставщик данных .NET реализует одинаковые базовые классы — ProviderFactory, Connection, ConnectionStringBuilder, Command, DataReader, Parameter и Transaction, имена которых зависят от поставщика.

Например, у поставщика SQL Client .NET Data Provider есть объект SqlConnection, а у поставщика ODBC .NET Data Provider это объект QdbcConnection. Но независимо от типа поставщика, объект Connection реализует одинаковые базовые функции посредством одних и тех же базовых интерфейсов.

Базовые классы подсоединенных объектов

ProviderFactory — новый класс в ADO.NET версии 2.0, действующий как фабрика объектов, которая дает возможность создавать образцы других классов для поставщика данных .NET. Каждый класс ProviderFactory предлагает метод Create, в котором создаются объекты Connection, ConnectionStringBuilder, Command, Parameter, DataAdapter и CommandBuilder.

Класс Connection - применяется для соединения с БД и отсоединения от нее. С помощью свойств этого объекта можно задать тип источника, его расположение прочее. Объект Connection

выступает в качестве канала, по которому другие классы, например DataAdapter и Command, взаимодействуют с БД для передачи изменении и выборки их результатов.

ConnectionStringBuilder - новый класс в ADO.NET версии 2.0. Этот класс делает проще процесс построения строк подключения для поставщика данных .NET. Каждый класс ConnectionStringBuilder предоставляет свойства, которые соответствуют опциям, доступным в той самой строке подключения поставщика данных. Создав строку подключения с помощью класса ConnectionStringBuilder, можно получить доступ к строке подключения средствами свойства ConnectionString класса ConnectionStringBuilder.

Transaction используется для выполнения группы команд вместе как неделимую операцию. В классе Connection есть метод BeginTransaction, позволяющий создавать объекты Transaction. С помощью объекта Transaction удается подтвердить или отменить все коррективы, сделанные в ходе транзакции.

Класс Command - может осуществлять запрос к БД, вызов хранимой процедуры или прямой запрос на возврат содержимого конкретной таблицы. Команда может возвратить или нет какой-либо результат (в зависимости от этого выполнение объекта Command запускается различными методами).

Parameter - используется для создания параметризованного объекта Command. Для использования параметров создаются классы Parameter, соответствующие всем параметрам запроса, затем они добавляются в класс Parameters объекта Command. Класс Parameter ADO.NET предоставляет свойства и методы, позволяющие определить тип данных и значение параметров.

Класс DataReader предназначен для максимально быстрой выборки и просмотра возвращаемых запросом записей. Он позволяет просматривать результаты запроса по одной записи за раз. При переходе к следующей записи содержимое предыдущей отбрасывается. Объект DataReader не поддерживает обновление, и возвращаемые им данные доступны только для чтения. Поскольку класс DataReader реализует лишь ограниченный набор функций, он очень прост и имеет высокую производительность.

Класс DataAdapter воплощает новую концепцию моделей доступа к данным Microsoft. Это своеобразный шлюз между БД и отсоединенными объектами модели ADO.NET. Он устанавливает подключение или, если подключение уже установлено, содержит достаточно информации, чтобы воспринимать данные автономных объектов и взаимодействовать с базой данных предписанным образом.

Базовые классы автономных объектов

Объекты, составляющие автономную часть модели ADO.NET, не взаимодействуют напрямую с подсоединенными объектами, для этого используется DataAdapter. Автономные объекты не должны знать о базе данных, поэтому они могут совместно использоваться различными базами.

При автономной работе с данными живое соединение с БД не понадобится, однако вы не увидите изменений, внесенных другими пользователями после выполнения вами исходного запроса.

При работе в автономном режиме ADO.NET ведет пул реальных физических подключений для различных запросов, таким образом многократно повышая свою производительность.

Класс DataSet содержит набор данных. Данные в нем отсоединены от БД. Все изменения данных кэшируются в объектах DataRow. Кроме того, класс DataSet предоставляет функции чтения и записи в файл и область памяти. Можно сохранить только содержимое объекта DataSet, только его структуру или и то и другое. ADO.NET хранит эти данные в виде XML-документа.

Класс Data Table похож на таблицу базы данных. Он состоит из объектов DataColumn, DataRow и различных налагаемых па них ограничений. Он хранит данные в формате строк и столбцов.

Класс DataColumn соответствует столбцу таблицы. В действительности же DataColumn содержит не данные, хранящиеся в объекте DataTable, а информацию о структуре столбца. Такая разновидность информации называется метаданными (metadata).

Класс DataTable содержит свойство — **Constraint** — типа ConstrsantsCollection. Оно позволяет создавать объекты ForeignKeyConstraint или UniqueConstraint и ассоциировать различные столбцы с определенными условиями, которым должны соответствовать данные из DataTable

*О*бъект DataTable предоставляет через набор Rows содержимое всех записей данных. Когда вы изменяете содержимое записи, **DataRow** кэширует эти изменения, чтобы позже передать их в БД. Таким образом, при изменении значения поля записи объект DataRow хранит оригинальное и текущее значения поля, что обеспечивает успешное обновление содержимого БД.

DataRelation позволяет задать отношения между различными таблицами, с помощью которых можно проверять соответствие данных из различных таблиц, а также просматривать родительские и дочерние строки из различных объектов DataTable. Кроме того, объекты DataRelation предоставляют свойства, позволяющие обеспечить ссылочную целостность.

Выбрав результаты запроса в объект DataTable, его содержимое можно просматривать разными способами посредством объекта **DataView**. Просматривать содержимое одного объекта DataTable можно одновременно посредством нескольких объектов DataView.

DataSet со строгим контролем типов — это класс, наследованный от класса DataSet и включающий свойства и методы, основанные на указанной вами схеме. Кроме того, этот класс содержит другие классы для объектов DataTable и DataRow, они позволяют создавать более эффективный код доступа к данным

Практическая работа № 1.47. Создание базы данных в среде MssqlServerManagement.

Практическая работа № 1.48. Создание базы данных в среде MssqlServerManagement

Цель работы: ознакомиться с основными конструкциями SQL, технологиями среды MS SQL Server Management, объектами SMO (среды MS Visual Studio) для резервного копирования и восстановления БД.

Ход работы

Задание: Необходимо в среде SQL Server Management Studio:

Создать новую базу данных (далее БД); 2) создать в этой БД таблицы и связи для реализации всех функций предметной области МММ.

Для работы с базами данных MS SQL Server 2005 существует много возможностей - одна из них это инструмент SQL Server Management Studio (далее SSMS).

1. Запустите среду SQL Server Management Studio и подключитесь к локальному серверу MS SQL Server, используя технологию 1.
2. Если соединение прошло успешно, то откроется основное окно среды Management Studio. Рассмотрите подробно все компоненты среды.
3. Среда Management Studio представляет данные в виде окон, выделенных для отдельных типов данных. Сведения о базе данных отображаются в обозревателе объектов и окнах документов. Найдите в открытом окне среды раздел «Обозреватель объектов», вид которого представлен на рисунке 3.

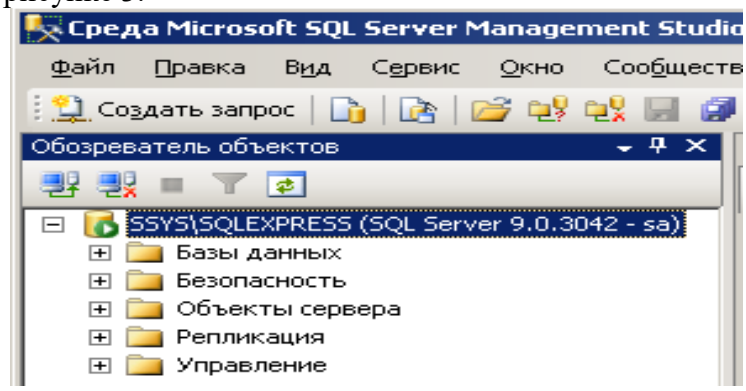


Рисунок 3 – Обозреватель объектов в среде SSMS

Обозреватель объектов является представлением в виде дерева, в котором отображаются все объекты базы данных на сервере. Обозреватель объектов включает сведения по всем серверам, к которым он подключен.

4. Далее найдите в открытом окне среды раздел «Окно документов». Окно документов представляет собой наиболее крупную часть среды Management Studio. В окнах документов могут размещаться редакторы запросов и окна обзора. По умолчанию отображается страница «Подробности обозревателя объектов», подключенная к экземпляру компонента Database Engine на данном компьютере.
5. Найдите на панели инструментов среды кнопку «Создать запрос» и нажмите ее. Откроется окно создания запроса. С помощью данного окна можно создавать и выполнять запросы к БД. Наберите в основной области окна запроса select * from information_schema.tables и выполните запрос, нажав на кнопку «Выполнить». В нижней части окна запроса появится таблица с результатами, найдите ее. (Выполненный вами запрос извлекает информацию о существующих таблицах подключенной сейчас БД – это по умолчанию база данных Master и таблицы в ней системные).
6. Найдите в открытом окне среды раздел «Окно базы данных», вид которого представлен на рисунке 4.

Рисунок 4 – Окно базы данных в среде SSMS

Именно это поле со списком позволяет выбрать применяемую в данный момент базу данных для выполнения запросов. По умолчанию в данном окне выбрана БД для учетной записи в которой зарегистрировался пользователь (для пользователя Sa это по умолчанию БД master)

Далее приступим к созданию своей бд, для этого:

7. В своей папке с проектом Лабы_ИСРКЭС создайте новую папку БД_МММ, в которой будет храниться БД.
8. Самый простой способ создать базу данных — воспользоваться графическим интерфейсом SQL Server Management Studio. Создайте новую базу данных на Вашем локальном сервере MS SQL Server, используя технологию 2.
9. Убедитесь в обозревателе объектов в появившейся только что базе данных МММ.
10. Ознакомитесь с описанием предметной области в приложении 1.
11. Далее приступим к созданию таблиц. Существует несколько методов, рассмотрим один из них. В обозревателе объектов разверните узел БД МММ_ВашеФИО и выберите в списке ветвь Таблицы → вызовите контекстное меню и выберите «Создать».
12. Создадим сначала таблицу Модель. Определим сначала все столбцы и типы данных в окне создания таблицы. Вводите имена столбцов без пробелов, кириллицей, в нижнем регистре. Конструктор таблицы «Модель» изображен на рисунке 5.

Модель *

Имя столбца	Тип данных	Разрешить значения null
код_модели	int	☐
название_модели	varchar(30)	☐
себестоимость	money	☐
время_изготовления	smalldatetime	☑
продажная_цена	money	☑
описание	nchar(350)	☑
фото	image	☑

Рисунок 5 – Структура таблицы «Модель»

13. Сделайте столбец код_модели первичным ключом, используя технологию 3.
14. Закройте окно создания таблицы и сохраните, если еще этого не сделали, таблицу под именем модель.
15. Сделайте столбец код_модели счетчиком, используя технологию 4.
16. Создайте аналогично остальные таблицы для раздела Заказ и Реализация продукции предметной области (таблицы Магазин, Заказ, Состав_заказа, Готовая_продукция). Внимательно определяйте типы данных.
17. Создайте связи между таблицами предметной области, используя технологию 5.

18. Заполните таблицу «Модель» данными, используя информацию из файла каталог (см. Сетевую папку. Местонахождение папки спросить у преподавателя.)

Практическая работа № 1.49. Копирование и восстановление базы данных

Цель работы: ознакомиться с основными конструкциями SQL, технологиями среды MS SQL Server Management, объектами SMO (среды MS Visual Studio) для резервного копирования и восстановления БД.

Задание №1. необходимо создать резервные копии базы данных «MMM» с использованием полного резервного копирования, разностного резервного копирования и резервного копирования журнала транзакций.

Ход работы:

1. Запустите SQL Server Management Studio (SSMS), подключитесь к своему экземпляру SQL Server, используя технологию 1.
2. Создайте папку с именем c:\Student\ВашаПапка\test.
3. Откройте окно нового запроса. Измените контекст на базу данных master, используя технологию 6. Наберите и исполните следующую команду, чтобы создать полную резервную копию базы данных:

   BACKUP DATABASE MMM TO DISK = 'C:\.....TEST\AW.BAK’

   Ознакомьтесь с результатами запроса – какая информация обработана, сколько страниц, сколько файлов.
4. Внесите изменение в таблицу «Модель» базы данных MMM. Добавьте одну запись (придумайте сами)/
5. Откройте окно нового запроса наберите и исполните следующую команду, чтобы создать резервную копию журнала транзакций и сохранить только что внесенное изменение:

   BACKUP LOG MMM TO DISK = ‘C:\.....TEST\AW1.TRN'

   Ознакомьтесь с результатами запроса – какая информация обработана, сколько страниц, сколько файлов.
6. Внесите еще одно изменение в таблицу«Модель».
7. Откройте окно нового запроса наберите и исполните следующую команду, чтобы создать разностную резервную копию базы данных:

   BACKUP DATABASE MMM TO DISK = ‘C:\.....\TEST\AWDIFF1.BAK' WITH DIFFERENTIAL

   Ознакомьтесь с результатами запроса – какая информация обработана, сколько страниц, сколько файлов.
8. Внесите еще одно изменение в таблицу«Модель».
9. Откройте окно нового запроса наберите и исполните следующую команду, чтобы создать полную резервную копию базы данных в указанном месте на диске:

   BACKUP LOG MMM TO DISK = ‘C:\....TEST\AW2.TRN'

   Ознакомьтесь с результатами запроса – какая информация обработана, сколько страниц, сколько файлов.

Задание №2. необходимо провести восстановление базы данных «MMM» из сделанных в задании №1 резервных копий.

Ход работы:

1. Если необходимо, запустите SSMS, подключитесь к своему экземпляру SQL Server, используя технологию 1.
2. Выполните восстановление БД из первой полной резервной копии (C:\...TEST\AW.BAK) средствами оболочки SSMS. Для этого выполните:
   - В обозревателе объектов вызовите контекстное меню на вашей БД и выберите задачу восстановления базы данных (см. рисунок 6).

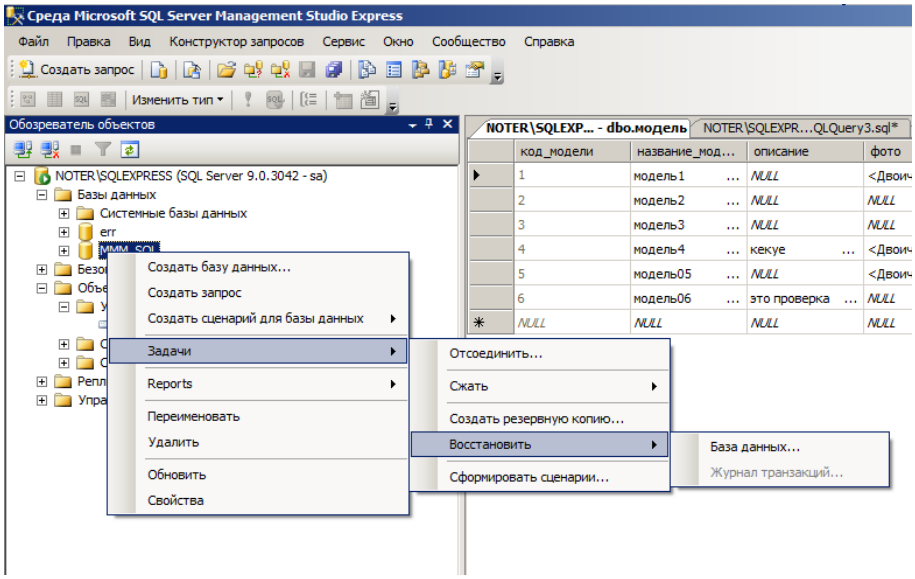


Рисунок 6 – Восстановление БД

- В открывшемся окне необходимо задать следующие параметры восстановления

На закладке «Общие» необходимо выбрать:

a. Базу данных для восстановления (вашу MMM)
b. Выбрать источник набора данных для восстановления с устройства → файл C:\...TEST\AW.BAK
c. После определения файла-источника данных необходимо флажком выбрать базу данных для восстановления (рисунок 7).

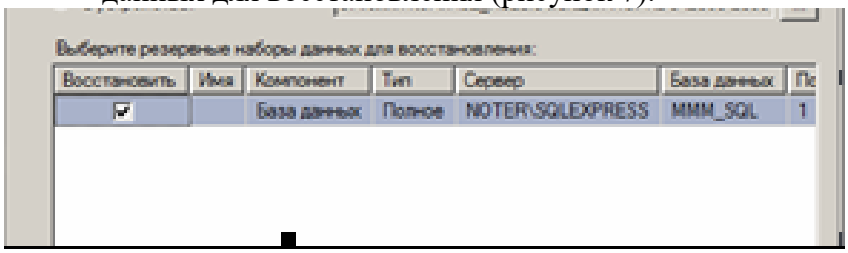


Рисунок 7- Выбор БД для восстановления

На закладке «Параметры»

a. необходимо включить опцию «Перезаписать БД» и «оставить БД готовой к использованию», (рисунок 8).

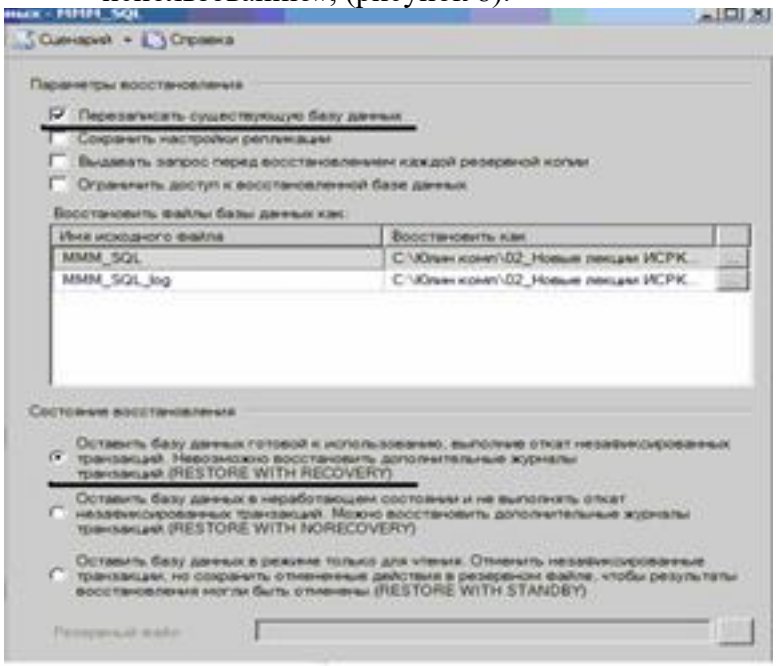


Рисунок 8 – Задание параметров восстановления

3. Нажмите ОК

4. После восстановления БД, откройте таблицу «Модель» и убедитесь, что она не содержит всех добавлений, вносимых вами в процессе выполнения упражнения, так как восстановление происходило из первой резервной копии (без изменений).

Задание №3. необходимо организовывать со стороны клиентского приложения, созданного в Visual Studio удаленное администрирование БД (резервное копирование).

Ход работы:

В Visual Studio

1. Создайте новый проект Windows Application и сохраните его в своей папке под именем Лабы_МММ_2 семестр.
2. В главную форму добавьте меню, изображенное на рисунке 9:

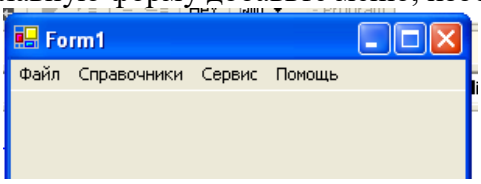


Рисунок 9 – Главное меню проекта

Файл (Открыть, Закрыть, Выход)
Справочники (Модель, Магазин, Дерево моделей)
Заказы (Работа с заказами)
Отчеты (Прайс-лист, Бланк заказов)
Администрирование БД (Резервное копирование, Безопасность)
Сервис (Калькулятор)
Помощь (Справка, О программе)

3. Добавьте новую форму в проект
4. Добавьте на только что созданную форму компоненты в соответствии с рисунком 10.

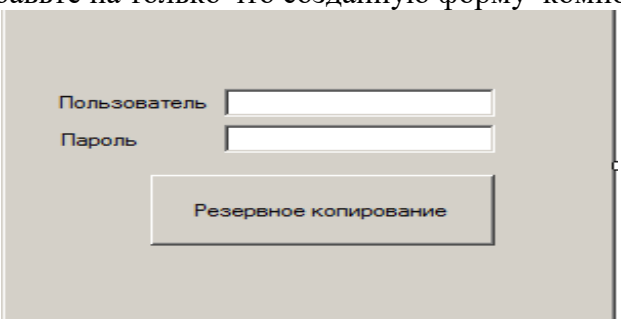


Рисунок 10 – Форма для подключения к серверу

5. Обеспечьте функциональную работу формы (напишите обработчик кнопки «Резервное копирование» с использованием объектов SMO. Описание объектов SMO, их свойств и методов см. в лекционном материале.)
6. Добавьте возможность открытия данной формы при выборе в главной форме пункта меню Администрироване БД → Резервное копирование
7. Запустите проект, проверьте работу формы.
8. Закройте проект
9. Убедитесь в появлении файла резервной копии на диске (файл, который указан в тексте программы).
10. Откройте SSMS. Добавьте в таблицу «Модель» новую строку данных (самостоятельно).
11. Средствами оболочки SSMS, выполните восстановление БД из резервной копии, созданной вашей программой
12. Убедитесь, что после восстановления добавленных строк в таблице «Модель» нет.

Задание №4. Ответьте на вопросы теста и представьте результаты преподавателю.

1. Вы выполняете разностное резервное копирование базы длимых AdveniurtWorks каждые четыре часа, начиная с 04:00. полная резервная копия создается в полночь. Какие данные будут содержаться в разностной резервной копии сделанной в полдень?
    a. A Страницы данных, измененные после полуночи.
    b. B. Экстенты, измененные после полуночи.

c. C. Страницы данных, измененные после 08:00
d. D. Экстенты. измененные после 08:00.

2. Вы выполняете полное резервное копирование базы данных Adventure Wortks,, которое завершается в полночь. Разностное резервное копирование выполняется по расписанию каждые четыре часа, начиная с 04:00. Резервное копирование журнала транзакций происходит по расписанию каждые пять минут. Какую информацию будет содержать резервная копия журнала транзакций, созданная в 09:15?
   a. A. Все транзакции, начатые после 09:10.
   b. B. Транзакции, завершенные после 09:10.
   c. C. Страницы, измененные после 09:10.
   d. D. Экстенты, измененные после 09:10.

Практическая работа № 1.50. Разработка формы для работы с данными в среде VsualStudio без кода

Цель работы: ознакомиться с возможностью мастеров среды MS Visual Studio для создания формы для работы с данными.

Задание: Необходимо создать простое (без написания кода) Windows-приложение, которое позволит взаимодействовать с данными в БД – просмотр, удаление, добавление данных в таблицу «Модель». Пример формы изображен на рисунке 11.

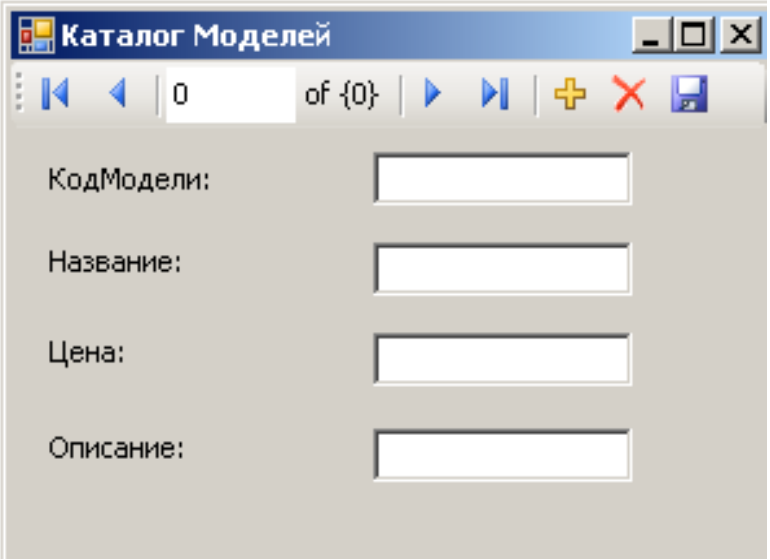


Рисунок 11 – Форма для работы с данными из таблицы «Модель»

Ход работы:

1. Добавим новую форму в проект

Откройте в Visual Studio проект, созданный в лабораторной работе № 2. Добавьте в проект новую форму. Добавьте обработчик для открытия только что добавленной формы при выборе в главном меню проекта пункта Справочники→Каталог моделей

2. Добавьте в проект новый «источник данных» с помощью мастера, включающий в себя все столбцы таблицы «Модель», используя технологию 19.
3. Рассмотрим, созданный мастером код. Для этого:

- В окне Solution Explorer среды Microsoft Visual Studio найдите добавленный элемент – класс DataSet со строгим контролем типов, откройте (Дважды кликнув мышью) и рассмотрите его содержимое.
- Выберите команду главного меню View→Server Explorer - и на экране появится одноименное окно. В иерархической структуре элементов окна Server Explorer должен появиться элемент, соответствующий тому соединению, которое использовано в мастере Data Source Configuration.
- В главном меню выберите команду Data→Show Data Sources. Отобразится источник данных, который был создан с помощью мастера. Найдите узлы, соответствующие выбранной таблице «Модель», убедитесь что выделенные столбцы будут расположены в виде вложений под таблицами, в которые они входят.

Теперь рассмотрим, как же можно использовать новый источник данных.

4. Добавление элементов в форму с помощью окна Data Source

Найдите в окне Data Source узел для таблицы «Модель» и разверните его (нажав на +). Выберите и перетащите его на форму все узлы-столбцы таблицы кроме столбца с Фото. Обратите внимание, что вместе с добавленными вами элементами на форме появился “Элемент управления навигацией”. Visual Studio добавила навигационный элемент управления и разместила его в верхней части формы.

5. Запустите проект. С помощью элемента BindingNavigator переместитесь к последней модели, после чего измените значение в одном из столбцов — ИмяМодели. Чтобы сохранить изменения, щелкните кнопку, на которой изображена дискета. После щелчка кнопки вы не сможете сразу увидеть на экране результаты завершения процедуры сохранения изменений. Чтобы убедиться, что внесенные изменения успешно сохранены в БД, закройте форму и перезапустите ваш проект. Внесенные изменения должны быть отображены в форме.
6. Изучение кода, сгенерированного в Visual Studio

Отобразите созданную в данной работе форму и откройте ее код события Load.

Из кода, сгенерированых в Visual Studio видно, что событие Load содержит код, который для таблицы моделей вызывает метод Fill в объектах TableAdapter. Метод Fill выполняет запрос, содержащийся в объекте TableAdapter, и сохраняет результаты в классе DataSet. При этом на класс DataSet ссылается таблица Моделей. Благодаря этому коду информация о моделях доступна в классе DataSet со строгим контролем типов в момент появления формы — ее же можно отобразить в элементе управления TextBox.

Событие Click для Saveltem

Код события Click объекта Saveltem элемента управления BindingNavigator, относящегося к информации о модели, состоит из трех строк. В первой строке вызывается **метод Validate** для формы, чтобы элементы управления могли проверить достоверность данных, введенных пользователем. Во второй строке кода вызывается метод EndEdit объекта BindingSource, который относится к информации о моделях. Он передает привязанным и объекту BindingSource элементам управления команду записи ожидающих изменений а источник данных. Без ЭТОЙ строки изменения так и останутся в кэше элементов управления и не будут отправлены в БД посредством следующей строки. В заключительной строке кода вызывается метод Update объекта TableAdapter для получения информации о модели; он передает ожидающие изменения в класс DataSet со строгим контролем типов.

Практическая работа № 1.51. СозданиеSql- запросов. Практическая работа № 1.52. СозданиеSql- запросов в среде sql Server Management Studio

Цель работы: ознакомиться с основными конструкциями языка SQL для манипулирования данными.

Задание: необходимо создать резервные копии базы данных «МММ» с использованием полного резервного копирования, разностного резервного копирования и резервного копирования журнала транзакций.

Указание: При выполнении работы используйте «Справочные материалы по SQL», которые расположены в дополнительном файле. (расположение файла спросить у преподавателя)

Ход работы:

1. Если необходимо, запустите SSMS, подключитесь к своему экземпляру SQL Server, используя технологию 1.
2. Откройте окно нового запроса. Измените контекст на базу данных МММ_вашеФИО, используя технологию 6.
3. Заполните таблицы Магазин, Заказ, Состав_заказа, Готовый продукт своими данными (не менее 5 строк в каждой таблице)
4. Наберите, исполните и сохраните тексты запросов для выполнения следующих функций в вашей БД (запросы создавайте с использованием языка SQL).
    a. Извлечь все данные из таблицы Модель (запрос SELECT)

b. С помощью запроса добавить в таблицу Готовый_продукт одну запись с данными (запрос INSERT)

Серийный номер	Код модели	Дата производства
0076AA-Key	1	01.03.2009

c. С помощью запроса удалить из таблицы Модель запись о модели с кодом =2. (запрос DELETE)
d. Извлечь из таблицы Модель те названия моделей, чья цена >100 (запрос SELECT)
e. Посчитать с помощью запроса среднюю цену всех моделей (запрос SELECT)
f. Извлечь из таблиц Модель, Заказы, Магазины следующие данные – Заказанные названия моделей, количество моделей, названия магазинов (запрос SELECT – для объединения таблиц)

Задания для самостоятельной работы:

g. Вывести названия магазинов, начинающихся с буквы 'M'(запрос SELECT, условие LIKE)
h. Подсчитать количество готовых продуктов для каждой модели (запрос SELECT с Group by, агрегатная функция COUNT)
i. Для каждого магазина посчитать среднюю стоимость всех заказов за все время сотрудничества. (запрос SELECT с Group by, агрегатная функция AVG)

Практическая работа № 1.53. Программирование с помощью встроенного языка transact sql в Microsoft Sql Server

Цель работы: ознакомиться с основными принципами программирования в MS SQL Server средствами встроенного языка Transact SQL.

Указание*: Для получения более подробной информации о работе тех или иных операторов или функций можно запустить утилиту Books Online из состава MS SQL Server и в разделе «Указатель» набрать искомый ключевой элемент.*

Так же при выполнении работы можно использовать материалы презентации к лекции (см. файл Презентация_Обзор SQL Server. О расположении файла спросить у преподавателя).

Задание№1: Необходимо ознакомится с основами языка Transact-SQL.

Ход работы:

1. Ознакомьтесь с правилами обозначения синтаксиса команд в справочной системе MS SQL Server (утилита Books Online).

2. Изучите правила написания программ на Transact SQL.

3. Изучите правила построения идентификаторов, правила объявления переменных и их типов.

4. Изучите правила работы с циклами и ветвлениями.

5. Если необходимо, запустите SSMS, подключитесь к своему экземпляру SQL Server, используя технологию 1.
6. Откройте окно нового запроса. Измените контекст на базу данных МММ_вашеФИО, используя технологию 6.
7. Создайте, наберите, исполните и сохраните тексты запросов для выполнения следующих заданий по темам:

Объявление переменных

Объявить переменную Perem1 типа денежный, а переменную Perem2 типа число с целой частью равной 8 и дробной частью равной 2.

Присвоение значений переменным и вывод значений на экран

Определить минимальную цену модели в каталоге моделей, результат поместить в переменную, вывести значение переменной на экран.

Условная конструкция IF

Подсчитать количество магазинов в таблице Магазины. Если их в таблице меньше 3, то ничего не сообщать, в противном случае вывести сообщение вида "В таблице ... магазинов " (вместо многоточия поставить точное количество поставщиков).

Цикл WHILE

Определить количество записей в таблице Магазины. Пока записей меньше 10, делать в цикле добавление записи в таблицу с автоматическим наращиванием значения ключевого поля, а вместо названия магазина ставить значение 'не известен'.

Задание№2: Необходимо научиться создавать и использовать хранимые процедуры на сервере БД.

Ход работы:

1. Если необходимо, запустите SSMS, подключитесь к своему экземпляру SQL Server, используя технологию 1.
2. Откройте окно нового запроса. Измените контекст на базу данных МММ_вашеФИО, используя технологию 6.
3. Создайте, наберите, исполните и сохраните текст запроса для создания хранимой процедуры с помощью оператора Create procedure, причем самостоятельно определить имя процедуры в вашей БД: Хранимая процедура должна «Вывести информацию о моделях со стоимостью больше указанного числа, отсортированных по названию моделей».
4. При создании хранимой процедуры воспользуйтесь описанием синтаксиса команды и примерами из файла Презентация_Обзор SQL Server.
5. В SQL Server Management Studio в разделе БД МММ→ Программирование →хранимые процедуры проверить наличие вашей процедуры.
6. Выполнить (вызвать) вашу процедуру с входящей ценой = 50, проверить результат выполнения.

Задание№3: Необходимо научиться создавать и использовать триггеры на сервере БД.

Ход работы:

1. Если необходимо, запустите SSMS, подключитесь к своему экземпляру SQL Server, используя технологию 1.
2. Откройте окно нового запроса. Измените контекст на базу данных МММ_вашеФИО, используя технологию 6.
3. В вашей БД МММ создать таблицу модель_безопасность, состоящую из 3-ех столбцов (название_модели (тип – символьный (50)), имя_пользователя (тип – символьный (50)), дата_добавления (тип – дата-время)) – без ключевого поля и без связей с другими таблицами.
4. Создать триггер «вставка_модель» для таблицы Модель с помощью оператора Create Trigger. Данный триггер должен при добавлении записи в таблицу Модели, добавлять запись безопасности в таблицу модель_безопасность. Подробное описание работы триггера приведено далее.

В МММ любые добавления в таблицу моделей (каталог_моделей) происходят редко и выполняются под строгим контролем. При этом отслеживается название добавленной модели, дата внесения добавлений и имя пользователя, который внес это изменение.

Поэтому необходимо реализовать триггер на таблице Модель (каталог_моделей) запускающийся при выполнении операции INSERT и протоколирующий информацию о добавленных моделях в таблицу модель_безопасность. То есть необходимо создать триггер, который при добавлении новой модели в каталог создавал бы в таблице модель_безопасность строку, в которой бы фиксировалось значение названия, дата внесения этого названия в таблицу и имя пользователя, который произвел добавление.

5. В SQL Server Management Studio в разделе БД МММ→Таблица Модели (Каталог моделей) →триггеры проверить наличие вашего триггера.
6. Проверить действие триггера.

Практическая работа № 1.54. Разработка формы работы с магазинами с использованием объекта Command. Практическая работа № 1.55. Разработка формы работы с магазинами с использованием объекта Command

Цель работы: ознакомиться с основными свойствами и методами подключенных объектов ADO .Net (Connection, Command, Parameters) с основными конструкциями языка SQL для манипулирования данными.

Задание: В среде MS Visual Studio необходимо создать Windows-приложение, которое позволит добавлять данные в таблицу Магазины с использованием объекта Command, форма изображена на рисунке 12.

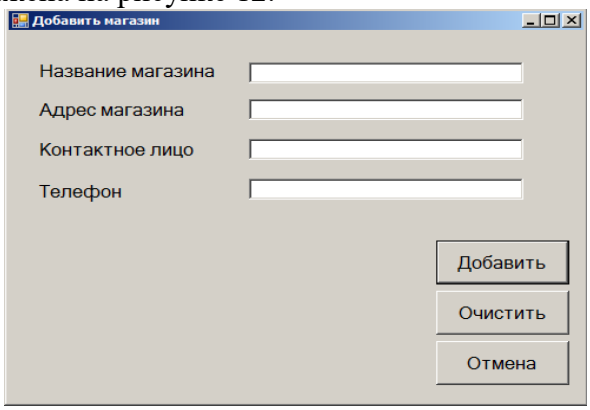


Рисунок 12 – Форма для добавления данных в таблицу «Магазин»

Ход работы:

Указание: При выполнении данного задания используйте лекционный материал (Лекция - Работа с подключенными объектами ADO).

1. Добавьте в Ваш проект новую форму, поместите на форму попарно компоненты Label и TextBox для каждого поля таблицы «Магазины» (Название_магазина, адрес, и т.д. – см. свою базу данных в среде MS SQLServer Management Studio (Далее SSMS)).
2. Добавьте обработчик для открытия только что добавленной формы при выборе в главном меню проекта пункта Справочники→Магазины → Добавить
3. Добавьте в код формы ссылку на пространства имен для работы с объектами ADO:
   Imports System.Data
   Imports System.Data.SqlClient
4. Добавьте конфигурационный файл приложения, используя технологию 7 и добавьте в него описание строки подключения к вашей базе данных МММ, используя технологию 8. Перед добавлением конфигурационного файла проверьте менеджер проектов «Solution Explorer» на наличие уже существующего файла конфигурации (app.config). Если он уже есть, то откройте его (двойным щелчком) и добавляйте описание строки подключения в уже существующий.
5. В коде формы «Добавить магазин», в процедуре-обработчике кнопки «Добавить» опишите переменную и создайте экземпляр объекта Connection, с использованием строки подключения, описанной вами ранее в конфигурационном файле, используя технологии 9,10
6. Опишите переменную и создайте экземпляр объекта Command, используя технологию 11. Создаваемый вами объект Command должен быть подключен к ранее (в пятом пункте) созданному объекту Connection.
7. Задайте для только что созданного объекта Command текст запроса на вставку данных (INSERT) с использованием информации из текстовых полей на форме, используя ОДНУ из технологий 12,13 или 14.
8. Откройте подключение к БД, используя технологию 15.
9. Выполните одним из методов объект Command, используя ОДНУ из технологий 16,17 или 18
10. Запустите проект, добавьте новую запись в таблицу Магазины.
11. Откройте базу данных МММ в среде MS SSMS. Убедитесь в появлении новой строки в таблице «Магазины»

Практическая работа № 1.56. Создание, удаление и редактирование данных в отсоединенной среде

Цель работы: Познакомиться со свойствами и методами автономных объектов: DataSet, DataAdapter, DataTable, DataView.

Задание №1: В среде MS Visual Studio необходимо создать Windows-приложение, которое позволит просматривать данные из таблицы Магазины с использованием объекта DataSet, DataAdapter. Форма представлена на рисунке 13.

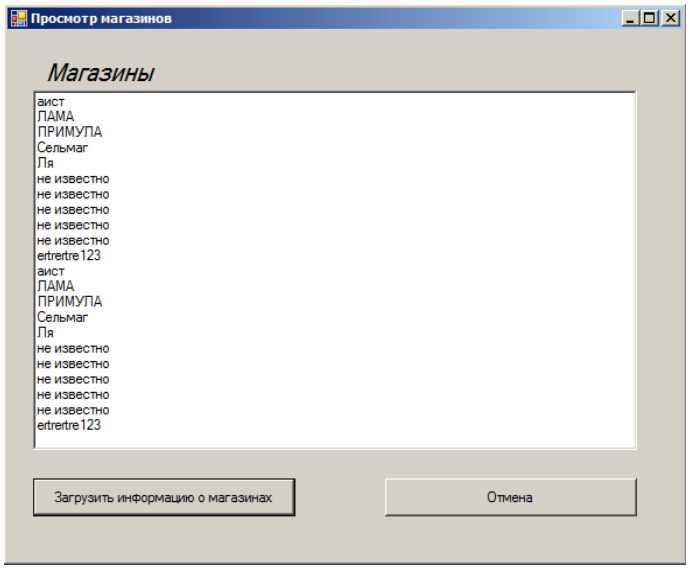


Рисунок 13 – Форма для просмотра таблицы «Магазины»

Ход работы:

Указание: При выполнении данного задания используйте лекционный материал (Лекция - Работа с автономными объектами ADO).

1. Добавьте в проект новую форму и добавьте на форму следующие компоненты: Label, ListBox, 2 Button в соответствии с рисунком 13
2. Добавьте обработчик для открытия только что добавленной формы при выборе в главном меню проекта пункта Справочники→Магазины → Просмотр
3. Добавьте в код формы ссылку на пространства имен для работы с объектами ADO:
   Imports System.Data
   Imports System.Data.SqlClient
4. Добавьте в код формы ссылку на пространства имен для работы с файлами конфигурации Imports System.Configuration
5. Необходимо написать обработчик на кнопку «Загрузить информацию о магазинах» для возможности просмотра информации о магазинах (таблица Магазины БД МММ). Для этого в обработчике:
   - опишите и создайте переменную-экземпляр класса DataSet, используя технологию 20;
   - опишите и создайте переменную-экземпляр класса DataAdapter, используя технологию 21;
   - вызовите метод Fill объекта DataAdapter для заполнения объекта DataSet, используя технологию 22;
   - загрузите данные из объекта DataSet в объект ListBox, используя код, объединяющий технологии 23,24
6. Запустите проект и проверьте работу только что созданной формы.

Задание №2: В среде MS Visual Studio необходимо создать Windows-приложение, которое позволит просматривать, добавлять и удалять данные из таблицы Магазины с использованием автономных объектов. Форма представлена на рисунке 14.

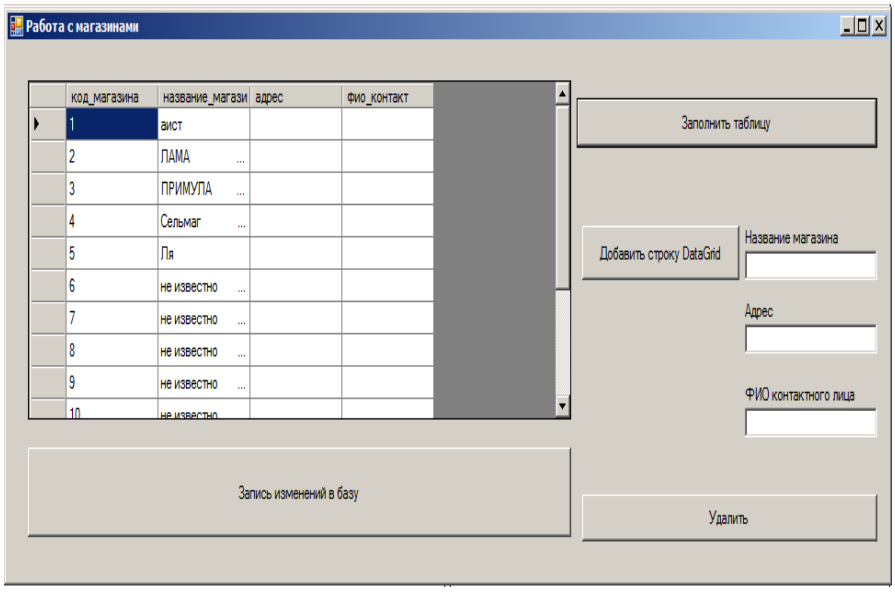


Рисунок 14 – Форма для общей работы с таблицей «Магазины»

Ход работы:

Указание: При выполнении данного задания используйте лекционный материал (Лекция - Работа с автономными объектами ADO).

1. Добавьте в проект новую форму, добавьте на форму компоненты: DataGridView, button («Заполнить таблицу»)
2. Добавьте обработчик для открытия только что добавленной формы при выборе в главном меню проекта пункта Справочники→Магазины → Общая работа с магазинами
3. Добавьте в код формы ссылку на пространства имен для работы с объектами ADO:
   Imports System.Data
   Imports System.Data.SqlClient
4. Добавьте в код формы ссылку на пространства имен для работы с файлами конфигурации Imports System.Configuration
5. Опишите и создайте ГЛОБАЛЬНУЮ переменную-экземпляр класса DataSet, используя технологию 20
6. Опишите и создайте ГЛОБАЛЬНУЮ переменную-экземпляр класса DataAdapter: Dim ЗадайтеИмяОбъектаDataAdapter As New SqlDataAdapter
7. Далее необходимо написать обработчик на кнопку «Заполнить таблицу» для возможности просмотра полной! информации о магазинах (таблица Магазины БД МММ). Для этого в обработчике:
   - задайте необходимые свойства объекта DataAdapter, используя технологию 25;
   - вызовите метод Fill объекта DataAdapter для заполнения объекта DataSet, используя технологию 22;
   - загрузите данные из объекта DataSet в объект DataGridView, используя технологию 26.
8. Вернитесь на форму и с помощью окна свойств (Properties) задайте для DataGridView1 следующие свойства:
   - MultiSelect = False (возможно выделение только одного элемента)
   - EditMode = EditProgrammatically (невозможность редактирования DataGrid прямо из формы)
9. Добавьте на форму еще одну кнопку («Добавить запись»). Добавьте попарно несколько ЭУ TextBox и Label в соответствии с количеством столбцов в таблице «Магазины» для добавления записей в эту таблицу (рисунок 14). Создадим обработчик на кнопку «Добавить запись», который добавляет запись в DataGridView1 из текстовых полей. Для этого в обработчике:
   - создайте новый экземпляр строки Магазины, используя технологию 27;
   - присвойте значения каждому столбцу строки из соответствующего текстового поля, используя технологию 28;
   - добавьте новую строку к коллекции Rows таблицы Магазины, используя технологию 29;

10. Добавьте на форму еще одну кнопку («Удалить запись»). Напишем обработчик на кнопку «Удалить запись», который удаляет запись из DataGridView1. Для этого в обработчике:
    - для удаления необходимой строки ее надо найти, вызвав метод Find. Для того, чтобы этот метод работал необходимо создать в объекте DataTable первичный ключ. Создайте первичный ключ, используя технологию 30;
    - присвойте значения каждому столбцу строки из соответствующего текстового поля, используя технологию 31;
    - удалите найденную строку из коллекции Rows таблицы Магазины, используя технологию 32;
11. Запустите созданную форму. Проверьте работу кнопок. Убедитесь, что изменения, произведенные вами в DataGridView не записываются в базу данных.
12. Запись обновлений в базу данных**.** Добавьте на форму еще одну кнопку («Запись изменений в базу»). Напишем обработчик на эту кнопку, используя метод Update объекта DataAdapter. Для того, чтобы метод Update корректно работал, необходимо задать команды INSERT, UPDATE, DELETE для объекта DataAdapter. Для этого можно воспользоваться SqlCommandBuilder (мы так и сделаем) или создавать команды вручную. Для того, чтобы можно было воспользоваться объектом SqlCommandBuilder необходимо, чтобы в именах объектов БД не было пробелов, иначе автоматической генерации команд вы не добьетесь. Поэтому запустите SQL Server Management Studio, убедитесь, что в таблице Магазин нет пробелов в названиях таблицы и названиях столбцов. Если эти пробелы есть, то переименуйте названия столбцов.

Далее в MS Visual Studio необходимо выполнить следующие действия в обработчике «Запись изменений в базу»:

- создайте новый экземпляр объекта SqlCommandBuilder, используя технологию 33;
- вызовите метод Update объекта DataAdapter и проверьте произошли ли обновления, если они произошли, то сообщите об этом пользователю. Используйте технологию 34;
- обновите информацию на форме

ИмяОбъектаDataSet.AcceptChanges()
ИмяОбъектаDataSet.Clear()
ИмяОбъектаDataAdapter .Fill(*ИмяОбъектаDataSet*)
ИмяDataGridView.DataSource = *ИмяОбъектаDataSet*.Tables(*НомерТаблицыВНаборе*)

Задание №3: В среде MS Visual Studio необходимо создать Windows-приложение, которое позволит производить просмотр, сортировку, фильтрацию, добавление и удаление данных из таблицы «Магазины» с помощью объекта DataView. Форма представлена на рисунке 15.

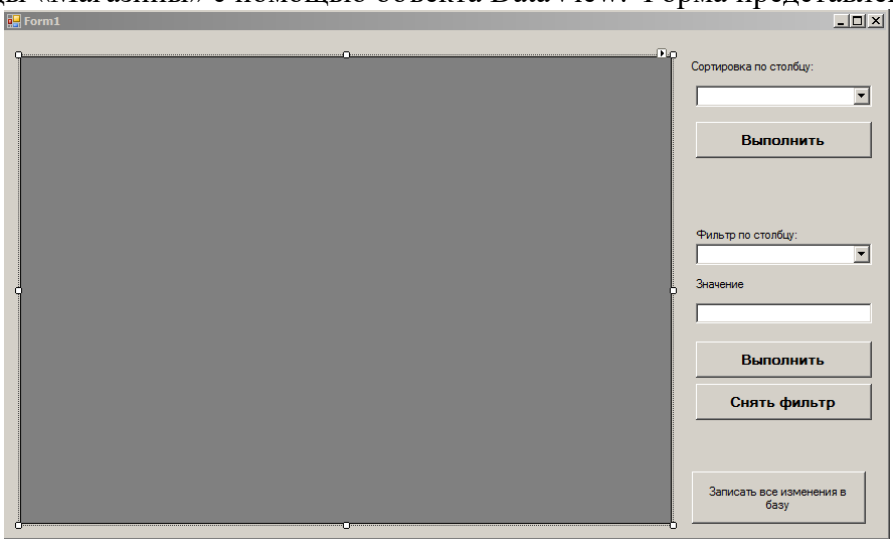


Рисунок 15 – Форма для сортировки и фильтрации Магазинов

Ход работы:

Указание: При выполнении данного задания используйте лекционный материал (Лекция - Работа с автономными объектами ADO).

1. Добавьте в Ваш проект новую форму. Добавьте в форму следующие элементы управления: DataGridView, Label+ComboBox, Button, Label+ComboBox, Label+TextBox, 3 Button. Настройте подписи добавленных элементов в соответствии с рисунком 15.
2. Добавьте обработчик для открытия только что добавленной формы при выборе в главном меню проекта пункта Справочники→Магазины → Сортировка и фильтрация
3. Добавьте в код формы описание ГЛОБАЛЬНОЙ для проекта переменной типа DataView:

   Dim ЗадайтеИмяПеременной DataView As DataView

   При выполнении данного задания объекты DataSet и DataAdapter мы будем создавать с помощью мастера (а не программно, как в предыдущих заданиях).
4. Создайте новый источник данных, включающий в себя всю информацию из таблицы Магазины, используя технологию 19.
5. Постройте проект. Для этого в главном меню выберите Build → Build проект. После построения проекта в панели инструментов ToolBox должны появиться новые объекты DataSet и TableAdapter(аналог DataAdapter для одной таблицы). Найдите МагазинTableAdapter и перетащите его на форму. Найдите MMM_вашаБД_DataSet и перетащите его на форму.
6. Добавьте обработчик на событие Form_Load. В этом обработчике необходимо заполнить созданный мастером DataSet, вызвать конструктор DataView и указать DataView в качестве источника данных для DataGridView на форме:

   ИмяTableAdapterКоторыйСоздалМастериВыПеретащилиНаФорму.**Fill**(*ИмяDataSetКоторыйСоздалМастериВыПеретащилиНаФорму*.*Имя таблицы с магазинами в вашей БД*)

   ИмяПеременной DataView = New DataView (*ИмяDataSet КоторыйСоздал Мастер иВы Перетащили НаФорму*.*Имя таблицы с магазинами в вашей БД*)

   ИмяDataGridView1.DataSource = *ИмяПеременной DataView*
7. Добавьте в Combobox, который отвечает за фильтрацию (см. рисунок 15) элементы = названиям столбцов из таблицы «Магазины» вашей БД. Для этого откройте БД на SQL Server и внимательно просмотрите названия столбцов, чтобы не ошибиться. Затем в VisualStudio для Combobox2 с помощью окна Properties добавьте элементы Items.
8. Напишите обработчик на кнопку «*Выполнить фильтр*». Для фильтрации данных в DataView используется свойство RowFilter этого объекта, которое устанавливается равным строке фильтрации. Синтаксис строки фильтрации таков: ИмяСтолбцаВ таблице = 'ЗначениеДляОтбора', например Город='Омск'. Таким образом, в нашем обработчике мы формируем строку фильтра и записываем ее в свойство RowFilter объекта DataView.

   Dim filterSTR As String

   filterSTR = ComboBox2.Text & "='" & TextBox1.Text & "'"

   ИмяПеременной DataView.RowFilter = filterSTR
9. Напишите обработчик на кнопку «*Снять фильтр*»

   ИмяПеременной DataView.RowFilter = ""
10. Напишите обработчик на кнопку «*Записать все изменения в базу*». Для записи всех изменений в БД используется, как и в предыдущем задании метод Update объекта DataAdapter. В этом задании объект DataAdapter для одной таблицы нам создал мастер сразу со всеми командами (Insert, Delete, Update, Select). Экземпляр этого объекта вы создали, перетащив на форму TableAdapter. Таким образом, обработчик будет таким:

    ‘ исправьте имена объектов на ваши

    МагазинTableAdapter1.Update(MmM_SQLDataSet1.магазин)
11. Запустите форму. Проверьте работу фильтра. Добавьте, измените, удалите строки с данными в объекте DataGridView с клавиатуры. Проверьте работу кнопки «Записать изменения в базу»
12. САМОСТОЯТЕЛЬНО напишите обработчик на кнопку «Выполнить сортировку», пользуясь материалом из лекции «Объект DataView»

Практическая работа № 1.57. Быстрое создание пользовательского интерфейса посредством связывания с данными

Цель работы: Познакомиться с технологией создания связанных с данными элементов управления, ознакомиться со свойствами и методами объекта BindingSource.

Задание: Выполнение данной лабораторной работы расписано по шагам. В результате ее выполнения вы получите приложение для оформления заказа на продукцию компании МММ детские машины следующего вида. Приложение выполнено с использованием технологии связывания с данными. Обратите внимание на рисунок 16 – в поле код_магазина столбец подстановки: при выборе в списке Названия_магазина, в таблицу Заказов записывается код_магазина. Столбец Итого – вычисляемый.

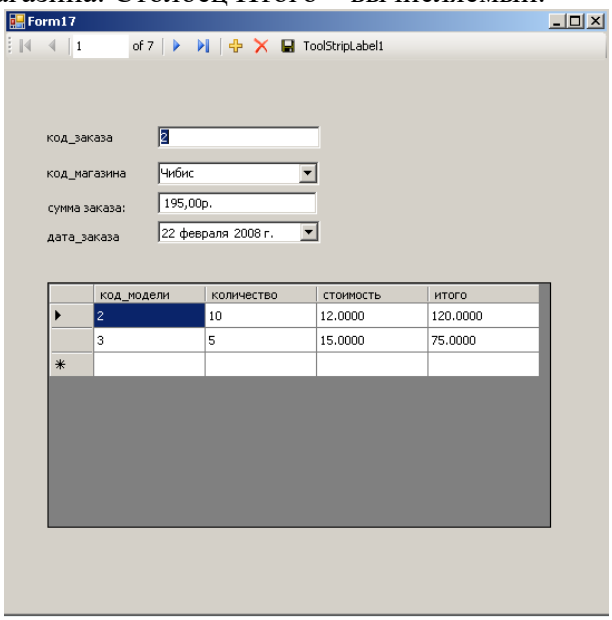


Рисунок 16 – Форма для работы с Заказами

Ход работы:

1. Проверим и изменим при необходимости схему данных. Для этого откройте среду Microsoft SQL Server Management Studio и выполните следующее:
   - измените при необходимости таблицу заказ так, чтобы она имела следующую структуру:

(ключ) номер_заказа	Int, **не СЧЕТЧИК !**
код_магазина	int
дата_заказа	datetime

   - измените при необходимости таблицу состав_заказа, чтобы она имела следующую структуру:

(ключ) код_заказа	int
(ключ) код_модели	int
количество	int
стоимость	money

   - добавьте по 5 записей в таблицы «Заказ» и «Состав заказа»
   - Схема данных для используемых в данной лабораторной таблиц должна выглядеть как на рисунке 17. **Остальные таблицы БД оставьте без изменений.**

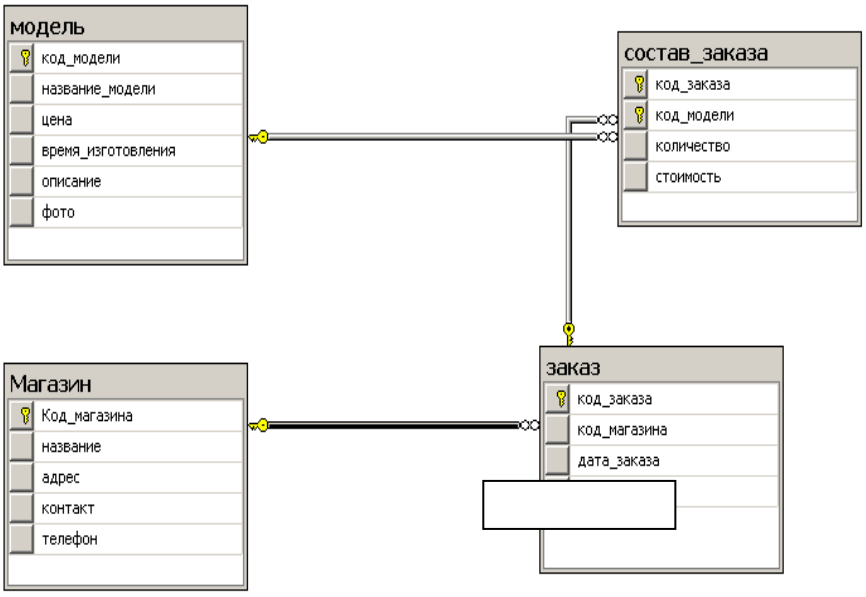


Рисунок 17 – Схема данных для таблиц предметной области Заказ

В пакете Windows Forms из состава Microsoft .NET Framework реализована поддержка связывания с данными. Благодаря связыванию появляется функциональная возможность вывода содержимого объекта *DataSet* в различных элементах управления, причем пользователь может изменять это содержимое. Коротко говоря, связывание с данными упрощает и ускоряет создание приложений для работы с данными, поскольку уменьшает размер кода, необходимого для создания пользовательского интерфейса.

Этап 1. Создание объекта DataSet со строгим контролем типов

1. Добавьте в ваш проект новую форму.
2. Добавьте в главное меню вашего приложения пункт Заказ→Оформить, и обработчик, который открывает только что добавленную форму при выборе данного пункта.
3. Создадим для задействованных в этой лабораторной связанных таблиц БД объект DataSet со строгим контролем типов, содержащий объекты DataTable.
   - Выберите в главном меню Data → Add New Data.
   - Затем укажите “построить новый источник данных на основе **DataBase**”
   - Выберите строку подключения, с помощью которой вы подключитесь к БД МММ.
   - Выберите для добавления в *DataSet* в таблице Заказ (все столбцы), в таблице состав_заказа (все столбцы), в таблице Модели (кодмодели, название модели), в таблице магазины (код_магазина, название_магазиа).
   - Нажмите кнопку Finish. Теперь объект *DataSet* со строгим контролем типов можно использовать.
   - Постройте проект (в главном меню выберите Build → Build вашПроект)
   - Убедитесь, что в панели инструментов ToolBox (ВВЕРХУ) мастер создал отдельные объекты *TableAdapter*, который инкапсулирует *SqlDataAdapter* автоматически для каждой таблицы, выбранной вами в мастере.
4. Начнем работу с таблицы ЗАКАЗЫ.

Этап 2. Добавление простых связанных элементов управления в форму

1. Создадим простые связанные элементы для отображения (ввода) столбцов из таблицы Заказы. Для этого добавьте на форму 3 компонента Label для подписей столбцов, 2 компонента TextBox, и 1 DateTimePicker для отображения столбцов кодЗаказа, код_Магазина, дата_заказа. Измените свойство Text компонент Label на соответствующие названия столбцов - кодЗаказа, код_Магазина, дата_заказа.
2. Для связи элементов управления с полями БД в дальнейшем будет использоваться объект BindingSource. Класс *BindingSource* доступен как элемент закладки Data в окне Toolbox. Добавьте его на форму, перетащив из окна Toolbox. Чтобы связать класс *BindingSource* с объектом *DatdTable* из состава *DataSet* измените в окне Properties*:*

- свойство *DataSource* = DataSetOrder (имя DataSet который вы создали на 1-ом этапе данной лабораторной работы).
- свойство *DataMember= таблица (Заказ).*

Рассмотрим далее как простые ЭУ (например *TextBox*), можно связывать с классом *BindingSource.*

3. Сначала выберите элемент управления, который хотите связать (TextBox1), а затем в окне свойств откройте раздел DataBindings (он находится вверху окна свойств). Далее необходимо выполнить одно из действий:
   - Для большинства элементов управления свойство, которое необходимо связать, должно быть указано в этом разделе. Если это так - Выберите свойство Text, затем выберите соответствующий столбец в ниспадающем списке (BindingSource → Код_заказа)

     ИЛИ
   - Если нужного вам свойства нет в коротком списке под разделом DataBindings указанного окна, нажмите кнопку с многоточием (...) в элементе Advanced под разделом DataBindings, а затем выберите требуемое свойство в появившемся диалоговом окне (свойство Text), а затем как в первом пункте.

     Аналогично свяжите TextBox2 с код_магазина, DateTimePicker с датой заказа.

Этап 3. Получение данных

1. Откройте код события Load формы, и проверьте, сгенерировался ли там автоматически вызов метода Fill. (может быть с разницей в именах). Если нет такого кода запишите (исправив если это необходимо имена объектов). Найдите в лекционном материале для сего вызывается метод Fill объекта DataAdapter

   ЗаказTableAdapter.Fill(Mmm_sqlDataSet3.заказ)

Этап 4. Перемещение по результатам

2. Добавим на форму возможность навигации по записям. Перетащите элемент управления BindingNavigator с закладки Data Toolbox на свою форму. Выбрав элемент управления BindingNavigator, нажмите свойство BindingSource в окне Properties и выберите в ниспадающем списке объект BindingSource (созданный на 2-ом этапе).

Этап 5. Добавление и удаление элементов в BindingNavigator

Элемент управления *BindingNavigator* по умолчанию включает в себя кнопки, позволяющие пользователю добавлять и удалять элементы.

Для демонстрации возможностей изменим действия связанные с кнопкой «Удалить» - чтобы на экран выводилось диалоговое окно с предложением пользователю подтвердить удаление текущего элемента. Существует следующая проблема - пока код выполняется, элемент управления *BindingNavigator* уже неявно вызывает метод *RemoveCurrent* соответствующего объекта *BindingSource.* Элемент управления *BindingNavigator* имеет свойство *Deleteitem.* Как только вы нажмете связанный с ним тип *ToolStripitem,* этот элемент вызовет метод *RemoveCurrent* соответствующего объекта *BindingSource.* Сбрасывание свойства *Deleteitem* элемента управления *BindingNavigator* не позволит последнему неявно вызвать указанный метод.

1. В окне Properties для элемента управления BindingNavigator выберите свойство DeleteItem. Затем выберите вверху списка вариант none.
2. Теперь дважды щелкните пункт Delete ToolStripButton на элементе управления BindingNavigator — и добавьте следующий код.

```
Dim result As DialogResult
result = MsgBox("Вы действительно хотите удалить заказ?", MsgBoxStyle.YesNo)
    If result = Windows.Forms.DialogResult.Yes Then
      BindingSource1.RemoveCurrent()
    End If
```

Этап 6. Передача изменений

Добавим на навигатор кнопку “Сохранить” для передачи изменений в БД

1. Для добавления нового элемента выберите элемент управления BindingNavigator. На нем справа от последнего элемента появится маленькая направленная вниз стрелочка. Нажмите эту стрелку и вы увидите список всех доступных элементов (Button, Label и других). Добавьте новый элемент Button, задайте ему Свойство DisplayStyle=Text, Свойство text = Сохранить.

2. Добавьте в событие «Щелчок по кнопке Сохранить» соответствующий обработчик. (если необходимо измените имен ЭУ так как они называются у вас)

   Me.Validate

   Me.BindingSource1.EndEdit()

   Me. ЗаказTableAdapter.Update(me.Mmm_sqlDataSet)

   1-ая строка = После вызова метода *Validate* форма проверяет достоверность данных.

   2-ая строка = Благодаря объекту *BindingSource* можно соответственно записывать или пропускать изменения с помощью методов *EndEdit* или *CancelEdit.* При переходе к другой записи будет неявно вызван метод *EndEdit,* если изменения были внесены через связанные элементы управления.

   3-ая строка = метод *Update* передает отложенные изменения, хранимые в объекте *DataTable.*

Этап 7. Просмотр дочерних данных

На данном этапе приложение позволяет просматривать и изменять данные таблицы Заказы. Доработаем форму так, чтобы можно было работать с составом заказов.

1. Откройте в конструкторе класс DataSet со строгим контролем типов (созданный на 1-ом этапе данной лабораторной работе) – это можно сделать, например, дважды щелкнув по объекту DataSet в окне SolutionExplorer.
2. Конструктор класса DataSet со строгим контролем типов автоматически добавляет объект DataRelation между двумя таблицами DataSet, но не связывает новый объект DataRelation с объектом ForeignKeyConstraint. Дважды щелкните объект DataRelation между таблицами Заказ и Состав_заказа в конструкторе, выберите опцию Both Relation And Foreign Key Constraint и задайте свойствам Update Rule и Delete Rule значения Cascade.
3. Закройте конструктор окна DataSet с сохранением
4. Теперь посмотрите содержимое объекта *DataSet* со строгим контролем типов в окне *Data Sources*. Вы увидите два отдельных узла состав_заказа, как на рисунке 18: один из узлов — брат, а второй — сын узла «Заказ».

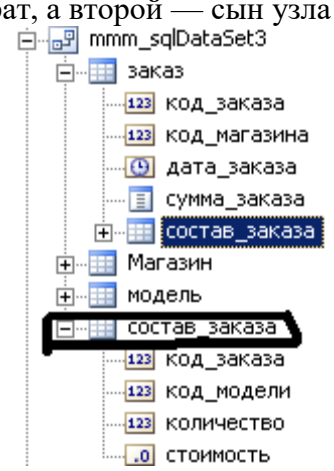


Рисунок 18 – Источник данных

Если пытаться перетащить один из узлов на форму, то произойдет создание объектов *DataGndView, TableAdapter, BindingSource* для работы с составом_заказов. Разница между показанными на рисунке двумя объектами состоит в том, что при перетаскивании узла состав_заказа БРАТ элемент управления *DataGridView* выводит на экран все записи таблицы состав_заказа. Если перетащить узел состав_заказа, который приходится узлу заказ сыном, то в *DataGridView* выводится на экран будут только записи таблицы состав_заказа для нужного нам заказа.

ПОЭТОМУ перетащите на вашу форму объект-сын состав_заказов.

5. Для объекта DataSet необходимо, чтобы записи в таблице состав_заказа соответствовали выбранному в данный момент в верхней части формы значению таблицы Заказ. Попробуйте запустить форму. Запустив форму в таком виде, как мы сейчас ее сделали, вы получите сообщение об исключительной ситуации. Чтобы указанного сообщения не было, необходимо изменить код таким образом, чтобы код для получения информации о составе заказа появлялся после запроса информации о заказе. Откройте код события Load формы, и исправьте код следующим образом (возможно имена объектов у вас будут другими). Порядок

вызова метода Fill должен быть именно таким (сначала для родительской Заказ, потом для дочерней состав_заказов)

Me.ЗаказTableAdapter.Fill(Me.Mmm_sqlDataSet3.заказ)

Me.Состав_заказаTableAdapter.Fill(Me.Mmm_sqlDataSet3.состав_заказа)

Этап 8. Совершенствование пользовательского интерфейса

Сделаем так, чтобы работать с приложением стало удобнее. Основные усовершенствования формы это:

1. элемент управления CotboBox для таблицы Магазины,

Создадим аналог МастераПодстановки в Access для поля Код_магазина в оформлении таблиц Заказы. Для этого:

- Убедитесь, что информация из таблицы Магазины (код, название) ранее была добавлена вами в объект *DataSet* со строгим контролем типов, например, открыв его в окне DataSolution. (если информации из таблицы Магазины нет, добавьте ее, например, с помощью контекстного меню на DataSet → Configure DataSet with Wizard)
- Удалите элемент управления *TextBox,* который отображал столбец код_магазина. Теперь перетащите элемент управления *ComboBox* из окна Toolbox на форму. Нажмите небольшую стрелку прямо над этим элементом управления и укажите, что вы хотите использовать элементы, которые можно связывать с данными «Use data Bound Items», как изображено на рисунке 19.

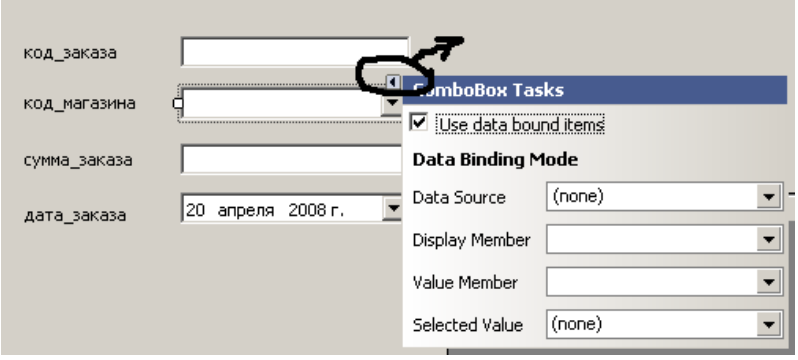


Рисунок 19 – Смарт-теги элемента управления ComboBox

- Диалоговое окно под названием ComboBox Tasks расширится. В ниспадающем списке Data Source выберите таблицу Магазины (под элементами Other Data Sources → Project Data Sources → ВашDataSet →Магазины).
- Теперь задайте свойству *DisplayMember* имя столбца название_магазина, а свойству *ValueMember* имя столбца код_магазина.
- Наконец, задайте свойству *SelectedValue* имя столбца код_магазина ИЗ объекта *BindingSource*, созданного вами для таблицы Заказы.

2. улучшенный формат для числовых значений и столбцы на основе выражения (название_модели и суммаИтого), предназначенные для отображения на экране полезных расчетных значений.

Добавим в наше приложение возможность узнать общую стоимость каждого вида товара, а также общую сумму всего товара в заказе. Для создания столбцов, значения которых вычисляются на основе выражений можно использовать свойство *Expression* объекта *DataColumn.*

- Таблица **состав_заказа** содержит столбцы количество и стоимость. Откройте в конструкторе объект DataSet (2 раза левой кнопкой мыши по dataset в окне SolutionExplorer). С помощью контекстного меню добавьте в таблицу Состав_заказа новый столбец (Add → column) с названием итого. В окне свойств задайте свойству *Expression* объекта *DataColumn* значение количество*стоимость.
- Аналогично в открытом окне конструктора DataSet в таблицу Заказы добавьте столбец сумма_заказа, который отображает общую сумму заказа, задав свойству *Expression* объекта *DataColumn* значение Sum(Child(FK_состав_заказа_заказ).итого)
- Свойству *DataType* всех только что добавленных объектов *DataColumn* нужно присвоить значение *System.Decimal.*

- После этого все новые столбцы можно добавить на форму. Для создания нового столбца сумма_заказа перетащите его из окна Data Sources на форму. Чтобы поместить столбец итого на элемент управления *DataGridView,* щелкните правой кнопкой мыши структуру, выберите сначала пункт Add Column, а затем — столбец итого.
- Тип данных нового столбца сумма_заказа — *System.Decimal.* Поэтому элемент управления *TextBox* на форме с информацией отображает содержимое этого столбца в стандартном числовом формате. Так, стоимость единицы товара, равная 491,20 руб, в элементе *TextBox* будет отображаться как 491,2000. Изменим формат этого столбца, чтобы данные отображались в более привычном виде. Для этого выберите этот элемент на форме в среде Visual Studio. Затем перейдите в окно Properties, откройте свойство DataBindings, расположенное в верхней части окна, выберите вариант Advanced и нажмите кнопку с многоточием (...). Появится диалоговое окно Formatting and Advanced Binding. Задайте свойству *FormatType* значение *Currency.* Теперь элемент управления *TextBox* будет отображать содержимое столбца в формате денежных единиц ($491.20, €491.20 и т. п.).
- Элемент управления *DataGridView* также позволяет указывать формат строки для столбцов в структуре. Выберите на форме элемент управления DataGridView, щелкните его правой кнопкой мыши и выберите в контекстном меню пункт Edit Columns. В появившемся диалоговом окне выберите столбец, формат которого необходимо изменить (столбец Итого), выберите свойство DefauItCellStyle и нажмите кнопку с многоточием (...).В появившемся диалоговом окне CellStyle Builder выберите свойство Format и нажмите кнопку с многоточием (...). Наконец, в появившемся диалоговом окне Format String укажите нужный формат.

Задания для самостоятельной работы.

1. Откройте форму, созданную в лабораторной работе №3 - **Разработка формы для работы с данными в среде Vsual Studio без кода,** и убедитесь, что при использовании вами мастера он автоматически добавил объект *ИмяТаблицы*BindingSource в форму.
2. Добавьте на форму следующие элементы управления: 1 Label (название модели), 1 элемент TextBox и 2 кнопки («Поиск по названию», «Фильтр по названию») и напишите обработчики на данные кнопки, выполняющие поиск и фильтр соотвественно, с использованием свойств и методов объекта BindingSource.

Практическая работа № 1.58. Безопасность вMssqlServer

Цель работы: Познакомиться с политикой безопасности MS SQL Server, с возможностями Transact – SQL по созданию схем, логинов, пользователей и определения прав пользователей. Научиться организовывать со стороны клиентского приложения удаленное управление правами доступа к данным БД.

Задание №1: Создание логинов, пользователей и предоставление прав пользователям средствами transact-sql.

Указание: Перед выполнением работы ознакомьтесь с теоретическим материалом в презентации к лекции Введение в SQL Server, тема Безопасность.

Ход работы:

1. Запустите MS SQL Server Management Studio, подключитесь к серверу, используя технологию 1
2. Выберите контекстом свою базу данных свою БД, используя технологию 6
3. Найдите на панели инструментов среды кнопку «Создать запрос» и нажмите ее.
4. С помощью команд Transact – SQL создадим новый логин для вашей базы данных с именем «qwerty» и паролем «123456». Для создания логинов используется запрос CREATE LOGIN. Ознакомьтесь с синтаксисом запроса, представленным на рисунке 20.

Синтаксис:

```
CREATE LOGIN login { WITH <option_list1> | FROM <sources> }
<sources> ::=
  WINDOWS [ WITH <windows_options> [ ,...n ] ]
  | CERTIFICATE certificate_name
  | ASYMMETRIC KEY asym_key_name
<option_list1> ::=
  PASSWORD = 'password' [ HASHED ] [ MUST_CHANGE ]
  [ , <option_list2> [ ,...n ] ]
<option_list2> ::=
  SID = sid
  | DEFAULT_DATABASE = database_name

  | DEFAULT_LANGUAGE = language_id
  | CHECK_EXPIRATION = { ON | OFF }
  | CHECK_POLICY = { ON | OFF }
  [ CREDENTIAL = credential_name ]
<windows_options> ::=
  DEFAULT_DATABASE = database_name
  | DEFAULT_LANGUAGE = language_id
```

Рисунок 20 – Синтаксис запроса Create Login

5. Для создания нашего логина необходимо набрать и выполнить следующий запрос:
   CREATE LOGIN qwerty WITH PASSWORD='123456'
6. После создания логина, можно приступить к созданию пользователя для этого логина. Создадим одноименного пользователя «qwerty». Для создания пользователей используется запрос CREATE USER. Ознакомьтесь с синтаксисом запроса, представленным на рисунке 21.

```
CREATE USER user_name
  [ { FOR | FROM }
    { LOGIN login
      | CERTIFICATE certificate_name
      | ASYMMETRIC KEY asym_key_name
  }
]
[ WITH DEFAULT_SCHEMA = schema_name ]
```

Рисунок 21 – Синтаксис запроса Create User

7. Для создания нашего пользователя необходимо набрать и выполнить следующий запрос:
   CREATE USER qwerty FOR LOGIN qwerty
8. После добавления логина и пользователя отключитесь от сервера, нажав в обозревателе объектов кнопку «Отключить», изображение которой представлено на рисунке 22.

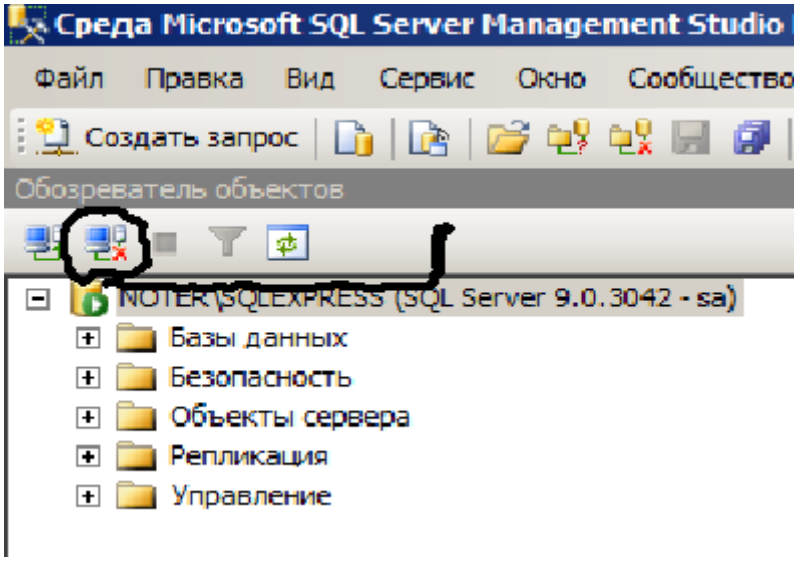


Рисунок 22 – Отключение от сервера

9. Подключитесь к серверу с помощью нажатия на кнопку «Подключение» под именем qwerty с паролем 123456
10. Разверните в обозревателе объектов свою БД, разверните узел «Таблицы» и убедитесь, что данный пользователь не имеет доступа ни к каким таблицам БД.
11. Отключитесь и подключитесь заново к серверу с правами администратора (то есть пользователь sa).
12. Добавим для пользователя qwerty возможность просмотра таблицы Модель и добавления записей в таблицу Модель. Для передачи прав пользователю используется SQL запрос GRANT. Ознакомьтесь с синтаксисом запроса, представленным на рисунке 23.

```
GRANT
  { ALL [ PRIVILEGES ] }
  | permission_name [ ( column_name [ ,...n ] ) ] [ ,...n ]
    [ ON [ class:: ] securable ]
  TO principal [ ,...n ] [ WITH GRANT OPTION ]
  [ AS principal ]
```

Рисунок 23 – Синтаксис запроса GRANT

13. Для передачи прав пользователю qwerty на просмотр и добавления записей в таблице Модель необходимо набрать и выполнить следующие два запроса:
    Grant Select On модель to qwerty;
    Grant Insert On модель to qwerty;
14. После успешного выполнения предыдущих запросов, отключитесь и подключитесь заново к серверу как пользователь qwerty.
15. Разверните в обозревателе объектов свою БД, разверните узел «Таблицы»→ таблица Модель, просмотрите записи, добавьте новую и убедитесь, что данный пользователь имеет права просмотра и добавления для таблицы.

Задание №2: В среде MS Visual Studio необходимо создать Windows-приложение, которое позволит добавлять новых пользователей для вашей БД вида, представленного на рисунке 24.

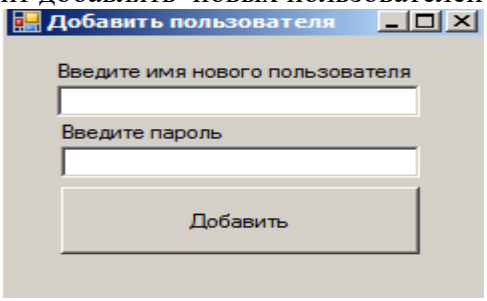


Рисунок 24 – Форма для добавления пользователя

Ход работы:

1. Добавьте в проект новую форму. Добавьте на форму следующие компоненты: 2 Label, 2 textBox, 1 Button и измените их свойства в соответствии с рисунком 24.
2. Добавьте обработчик для открытия только что добавленной формы при выборе в главном меню проекта пункта Администрирование→Добавить пользователя.
3. Добавьте в код формы ссылку на пространства имен для работы с объектами ADO:
   Imports System.Data
   Imports System.Data.SqlClient
4. В коде формы «Добавить пользователя», в процедуре-обработчике кнопки «Добавить» опишите переменную и создайте экземпляр объекта Connection и задайте ей параметры подключения, используя технологии 9,10.
5. Опишите переменную и создайте экземпляр объекта Command, используя технологию 11. Создаваемый вами объект Command должен быть подключен к ранее (в четвертом пункте) созданному объекту Connection.
6. Задайте для только что созданного объекта Command текст запроса на добавление логина (CREATE LOGIN) с использованием информации из текстовых полей на форме, используя технологию 12 ИЛИ технологию 13.
7. Откройте подключение к БД, используя технологию 15.
8. Выполните одним из методов объект Command, используя ОДНУ из следующих технологий 16,17,18.
9. Закройте подключение к БД, используя технологию 15.
10. Запустите проект, добавьте нового пользователя к вашей БД.
11. Подключитесь к базе данных МММ в среде MS SSMS, используя технологию 1 под именем только что добавленного пользователя. Убедитесь что это возможно.

Задание №3. В среде MS Visual Studio необходимо создать Windows-приложение, которое позволит добавлять новых пользователей для вашей БД вида, изображенного на рисунке 25.

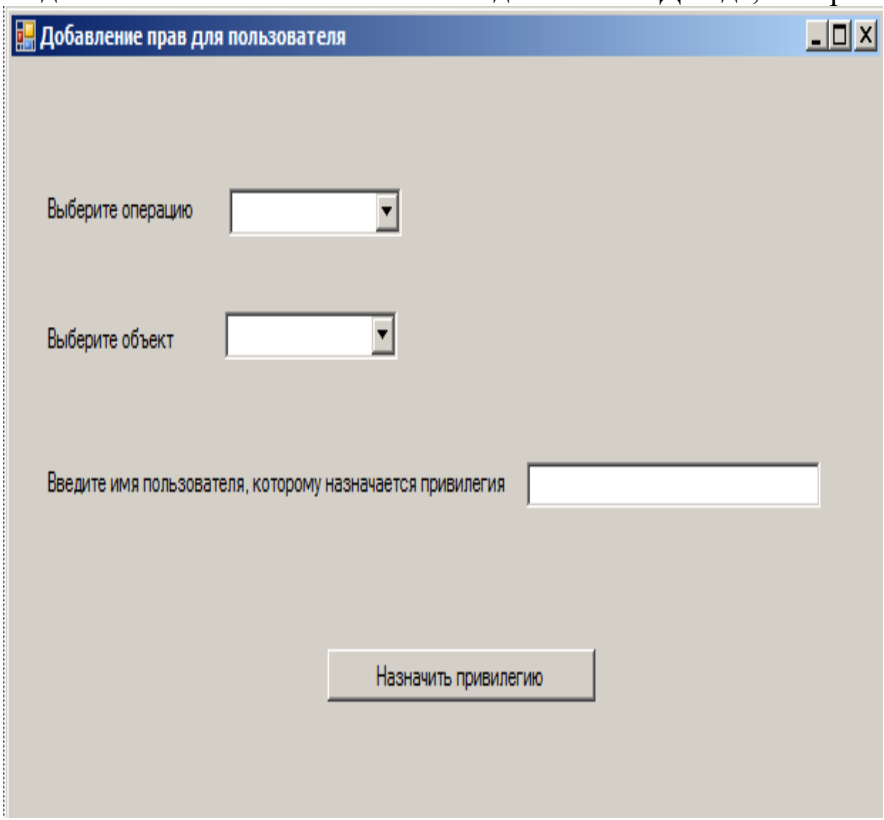


Рисунок 25 – Форма для добавления привилегий пользователя

Ход работы:

1. Добавьте в проект новую форму. Добавьте на форму следующие компоненты: 3 Label, 2 ComboBox, 1 textBox, 1 Button и измените их свойства в соответствии с рисунком 25. В список ComboBox1 занести перечень значений:
   - INSERT;
   - UPDATE;

- DELETE.

В список ComboBox2 занести перечень значений = название таблиц Вашей БД, например:

- Модель;
- Готовый_продукт;
- Заказ;
- Состав_заказа;
- Магазин.

2. Добавьте обработчик для открытия только что добавленной формы при выборе в главном меню проекта пункта Администрирование→Добавить права для пользователя
3. Добавьте в код формы ссылку на пространства имен для работы с объектами ADO:
   Imports System.Data
   Imports System.Data.SqlClient
4. В коде формы «Добавить права для пользователя», в процедуре-обработчике кнопки «Назначить привилегию» опишите переменную и создайте экземпляр объекта Connection и задайте ей параметры подключения, используя технологии 9,10.
5. Опишите переменную и создайте экземпляр объекта Command, используя технологию 11. Создаваемый вами объект Command должен быть подключен к ранее созданному объекту Connection.
6. Задайте для только что созданного объекта Command текст запроса на добавление Прав пользователю (GRANT) с использованием информации из полей со списком и текстовых полей на форме, используя технологию 12 ИЛИ технологию 13.
7. Откройте подключение к БД, используя технологию 15.
8. Выполните одним из методов объект Command, используя ОДНУ из следующих технологий 16,17,18.
9. Закройте подключение к БД, используя технологию 15.
10. Запустите проект, добавьте для какого-либо пользователя права на просмотр таблицы Модели вашей БД.
11. Подключитесь к базе данных МММ в среде MS SSMS под именем только что добавленного пользователя. Убедитесь что возможно просматривать записи из таблицы Модели.

Практическая работа № 1.59. Создание отчетных форм для баз данных средствами MsVisualStudio

Цель работы: Познакомиться с возможностями по созданию отчетов, предоставляемыми MS Visual Studio.

Задание №1. В среде MS Visual Studio необходимо создать Windows-приложение, которое позволит выводить и печатать прайс-лист организации, представленного на рисунке 26. Отчет необходимо создать средствами MS Report Viewer.

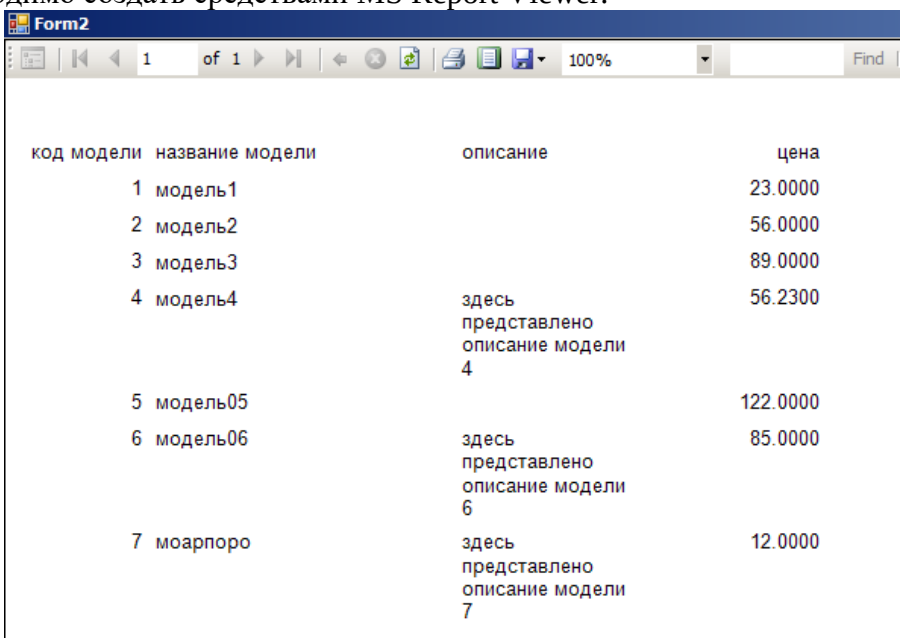


Рисунок 26 – Отчет, созданный средствами MS Report

Ход работы:

1. Добавьте в Ваш проект новый источник данных, содержащий все сведения из таблицы «Модель», используя технологию 19. Обратите внимание, что в окне SolutionExplorer появился новый элемент МММ_ВашеФИО_DataSet.xsd
2. Добавьте в проект конструктор отчета, для этого:
   - В окне SolutionExplorer вызовите контекстное меню решения и выберите Add→ New item → закладка Reporting → Report→ Кнопка Add
   - Обратите внимание, что в окне SolutionExplorer появился новый элемент Report1.rdlc
3. Добавьте в конструктор таблицу, для этого:
   - В панели инструментов конструктора отчета щелкните элемент «Таблица», а затем щелкните в области конструктора отчета.
   - В конструкторе отчетов появится таблица с тремя столбцами, занимающими всю ширину отчета.
   - В первом столбце щелкните маркер правой кнопкой мыши и выберите команду «Вставить столбец слева».
4. Добавьте данные в таблицу конструктора отчетов, для этого:
   - В окне «Data Source», найдите источник данных, созданный в пункте 2, разверните узел таблицы Модель, чтобы показать поля данных. Затем выполните следующие шаги:
     - Перетащите поле **Код модели** из окна **DataSource** во вторую строку (строку детализации) первого столбца в таблице. При этом выполняются два действия. Во-первых, в ячейку сведений помещается текст **=Fields!КодМодели.Value**. Во-вторых, заголовок столбца автоматически помещается в первую строку, непосредственно над выражением поля.
     - Аналогично перетащите поля название модели, описание модели, цена из окна «Источники данных» в строку детализации остальных столбцов в таблице.
5. Добавьте в Ваш проект новую форму. Добавьте обработчик для открытия только что добавленной формы при выборе в главном меню проекта пункта Отчеты→Прайс-лист
6. Добавьте элемент управления ReportViewer на форму, для этого:
   - Перетащите элемент управления ReportViewer из панели инструментов на форму.
   - Откройте панель смарт-тегов в элементе управления ReportViewer, щелкнув треугольник в правом верхнем углу. Щелкните раскрывающийся список «Выбор отчета» и выберите файл Report1.rdlc
   - На панели смарт-тегов щелкните элемент «Закрепление в родительском контейнере (Dock in Parent Container)».
7. Запустите проект и просмотрите отчет на форме.

Задание №2: В среде MS Visual Studio необходимо создать Windows-приложение, которое позволит формировать и передавать в MS Excel отчет по заказам вида, изображенного на рисунке 27. Отчет должен быть создан с использованием технологии COM.

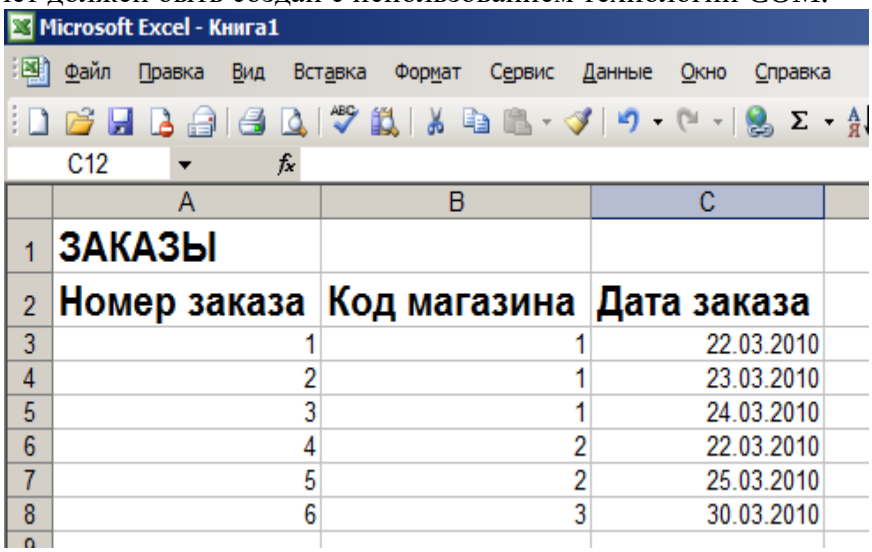


Рисунок 27 –Отчет по заказам в MS Excel

Ход работы:

Указание: При выполнении данного задания используйте лекционный материал (Лекция – Отчеты в Visual Studio).

1. Добавьте в главное меню Вашего проекта пункт Отчеты → Заказы
2. Добавьте обработчик для открытия программы MS Excel при выборе только что добавленного пункта меню. Для этого
   - Подключите библиотеку для работы с MS Excel в меню Project (Проект) → команду Add Reference (Добавить ссылку)→ В открывшемся диалоговом окне Add Reference (Добавить ссылку) перейдите на вкладку COM и выберите нужную библиотеку объектов. Например, для взаимодействия с Microsoft Excel требуется подключить библиотеку Microsoft Excel 11.0 Objecl Library. Цифра 11 обозначает установленную на компьютере версию Microsoft Excel (11 соответствует версии Microsoft Office 2003) → Нажмите кнопку ОК.
   - Добавьте в код ГЛАВНОЙ ФОРМЫ проекта ссылку на пространство имен, содержащие объекты продуктов Microsoft Office:

   Imports Microsoft.Office.Interop
   - Откройте обработчик пункта меню Отчеты → Заказы и напишите код, запускающий программу MS Excel:

   Dim *ЗадайтеИмяПеременнойExcel* As New Excel.Application
   ИмяПеременнойExcel.Visible = True
3. Запустите проект и убедитесь, что при выборе в главном меню Отчеты → Заказы открывается окно программы MS Excel.
4. Допишите обработчик на пункт меню Отчеты → Заказы для добавления рабочей книги с рабочими листами в MS Excel. Для этого в обработчике:
   - Опишите и создайте переменную-экземпляр рабочей книги
     Dim *ЗадайтеИмяПеременнойРабочейКниги* As Excel.Workbook
     ИмяПеременнойРабочейКниги = ИмяПеременнойExcel.Workbooks.Add
   - Опишите переменную – экземпляр рабочего листа, присвойте ему значение первого листа рабочей книги, активизируйте его.
     Dim *ЗадайтеИмяПеременнойРабочегоЛиста* As New Excel.Worksheet
     ИмяПеременнойРабочегоЛиста = ИмяПеременнойРабочейКниги.Worksheets(1)
     ИмяПеременнойРабочегоЛиста.Activate()
   - Выведите на рабочий лист заголовок отчета «заказы» и заголовки столбцов таблицы Заказы из вашей БД. (Номер заказа, Код магазина, Дата заказа). Зададим жирный крупный шрифт для заголовков. Для этого добавьте в код:
     ИмяПеременнойРабочегоЛиста.Cells(1, 1) = "Заказы"
     ИмяПеременнойРабочегоЛиста.Range("a2").Value = "Номер заказа"
     ИмяПеременнойРабочегоЛиста.Range("b2").Value = "Код магазина"
     ИмяПеременнойРабочегоЛиста.Range("c2").Value = "Дата заказа"
     ИмяПеременнойРабочегоЛиста.Range("a1:c2").Font.Bold = True
     ИмяПеременнойРабочегоЛиста.Range("a1:c2").Font.Size = 16
   - Запустите проект и убедитесь, что при выборе в главном меню Отчеты → Заказы открывается окно программы MS Excel с рабочей книгой и активным первым рабочим листом.
5. Допишите обработчик на пункт меню Отчеты → Заказы для выгрузки информации из таблицы Заказы в MS Excel. Для этого в обработчике:
   - Создайте и опишите переменную – экземпляр класса Command или DataView, которая извлекает данные из таблицы Заказ. Для этого, например, для Command:
     - Добавьте в код ссылку на пространства имен для работы с объектами ADO:
       Imports System.Data

     Imports System.Data.SqlClient

- В коде формы, в продолжение процедуры-обработчика меню Отчеты→Заказы, опишите переменную и создайте экземпляр объекта Connection и задайте ей параметры подключения, используя технологии 9,10.
- Опишите переменную и создайте экземпляр объекта Command, используя технологию 11. Создаваемый вами объект Command должен быть подключен к ранее созданному объекту Connection.
- Задайте для только что созданного объекта Command текст запроса на выборку всех данных (SELECT *) из таблицы Заказы, используя технологию 12.
- Откройте подключение к БД, используя технологию 15.
- Выполните запрос объекта Command, используя технологию 18.

- Выведите данные из переменной-экземпляра класса Command в ячейки рабочего листа. Для этого:
  - Откройте цикл, который перебирает все записи из объекта DataRead и записывает их в нужные ячейки Excel, начиная с третьей строки (потому что первая строка – заголовок отчета, вторая строка – заголовок таблицы уже заполнены ранее)

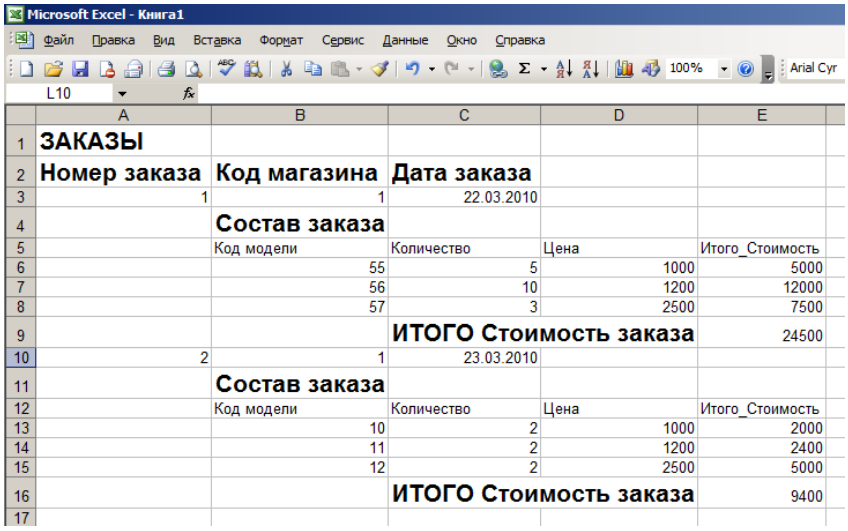
```
Dim I As Integer
I=3
While ИмяОбъектаDataReader.Read
 ‘ Выводим номер заказа в ячейку из строки i, столбца 1
ws.Cells(i, 1) = ИмяОбъектаDataReader (“ЗадайтеИмяСтолбцаИзНабораДанных,КоторыйВамНеобходимоВывести")
‘ Выводим код магазина в ячейку из строки i, столбца 2
ws.Cells(i, 2) = ИмяОбъектаDataReader (“ЗадайтеИмяСтолбцаИзНабораДанных,КоторыйВамНеобходимоВывести")
‘ Выводим дату заказа в ячейку из строки i, столбца 3
ws.Cells(i, 3) = ИмяОбъектаDataReader (“ЗадайтеИмяСтолбцаИзНабораДанных,КоторыйВамНеобходимоВывести")
i=i+1
End While
ИмяОбъектаDataReader.Close()
```

- Добавьте формулы для расчета итоговой суммы заказа на рабочий лист Excel. Для этого необходимо в ячейку из строки с номером i+1, столбца 2 добавить такую формулу Excel: =Сумм(диапазон, по которому считается сумма). В нашей таблице Заказы не предусмотрен столбец суммы, поэтому, просто для примера!! Посчитаем сумму по коду магазинов. Код магазина у нас начинается выводиться в ячейке B3, заканчивает в ячейке столбца B, строки I-2. Для этого добавьте в обработчик такой код:

```
ws.Cells(i+1, 1) = "Сумма кода магазинов"
    ws.Cells(i+1, 2) = "=Сумм(B3:B" & Cstr(I-2) & “)”
```

- Запустите проект и убедитесь, что при выборе в главном меню Отчеты → Заказы открывается в программе MS Excel отчет по данным из таблицы Заказы.

Задание №3: Самостоятельно измените созданное вами ранее приложение по отчету Заказов в Excel, добавив в него возможность просматривать информацию по Составу_заказов таким образом, чтобы приложение выглядело как изображение на рисунке 28.



	A	B	C	D	E
1	**ЗАКАЗЫ**				
2	**Номер заказа**	**Код магазина**	**Дата заказа**		
3	1	1	22.03.2010		
4		**Состав заказа**			
5		Код модели	Количество	Цена	Итого_Стоимость
6		55	5	1000	5000
7		56	10	1200	12000
8		57	3	2500	7500
9			**ИТОГО Стоимость заказа**		24500
10	2	1	23.03.2010		
11		**Состав заказа**			
12		Код модели	Количество	Цена	Итого_Стоимость
13		10	2	1000	2000
14		11	2	1200	2400
15		12	2	2500	5000
16			**ИТОГО Стоимость заказа**		9400
17					

Рисунок 28 –Отчет по составу заказов в MS Excel

Указание:

- Создайте и опишите переменную – экземпляр класса Command или DataSet, которая извлекает данные из таблицы Состав_Заказа.
- Выведите данные из переменной-экземпляра класса Command или DataSet в ячейки рабочего листа (Данные по Составу Заказа должны выводиться сразу после информации о заказе. Внимательно работайте с циклами)
- Добавьте формулы для расчета в столбцы “Итого стоимость” (=цена+количество)

Практическая работа № 1.60. Самостоятельная работа по автономным и подключенным объектам. Практическая работа № 1.61. Самостоятельная работа по автономным и подключенным объектам

Цель работы: Закрепить знания о подключенных и автономных объектах, полученные в процессе выполнения предыдущих лабораторных работ.

Задание №1: В среде MS Visual Studio необходимо с помощью теоретического материала из лекций, теории по объекту TreeView (см. лабораторные работы 1 семестра по дереву) необходимо создать форму для работы с Готовой продукцией, изображенную на рисунке 29.

1. Форма должна открываться из основного меню проекта (В главное меню добавьте пункт Склад → Готовая продукция)
2. Форма работает с данными из 2 таблиц вашей БД – таблица Модель и таблица Готовая продукция
3. Верхняя часть формы представляет собой компонент Дерево, узлы которого – это наименование существующих моделей, а св-во Tag узлов –это код модели.

Указание – Для заполнения дерева можно, например, сделать следующее:

- Создать и выполнить объект SQLCommand с командой SELECT, извлекающей данные из таблицы Модель
- В цикле While перебрать все строки результата выполнения команды (см. пример из лекции) и здесь же в цикле выполнить метод Add для формирования дерева.
- И весь этот код поместить в событие Load формы.

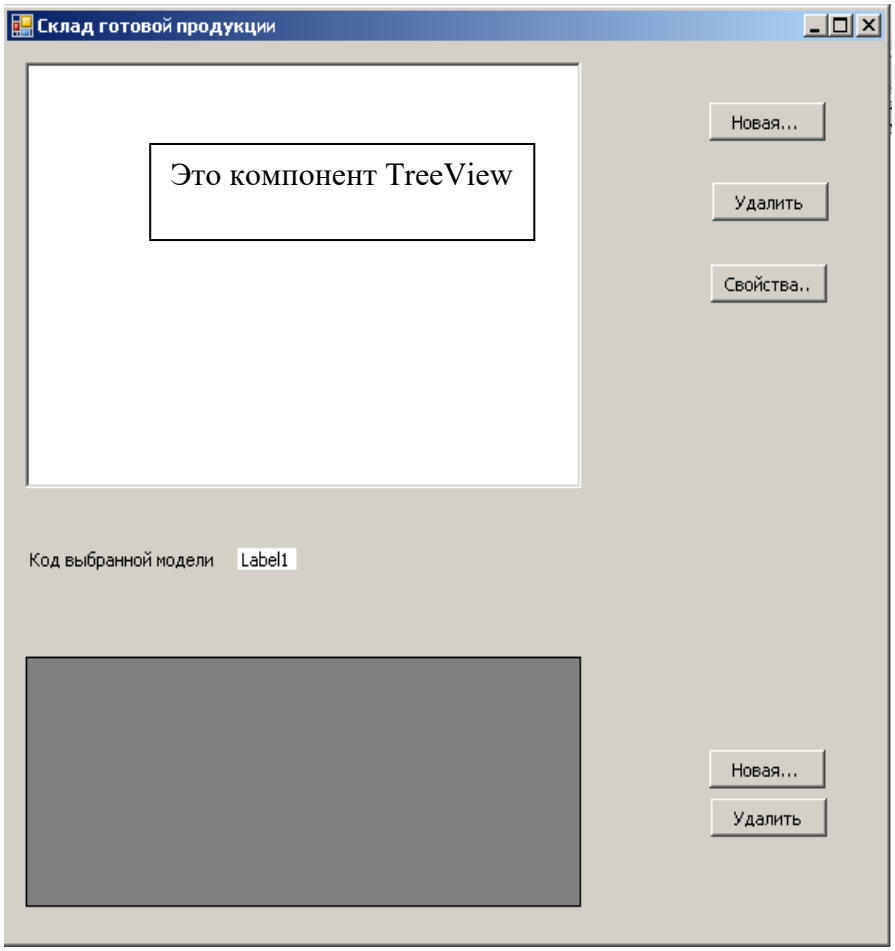


Рисунок 29 –Форма для работы с готовой продукцией

4. При нажатии на кнопку «Новая» в верхней части формы открывается новая форма, изображенная на рисунке 30, с помощью которой вы можете добавить записи в таблицу Модель.

   Указание – работайте с объектом Command и запросом Insert.

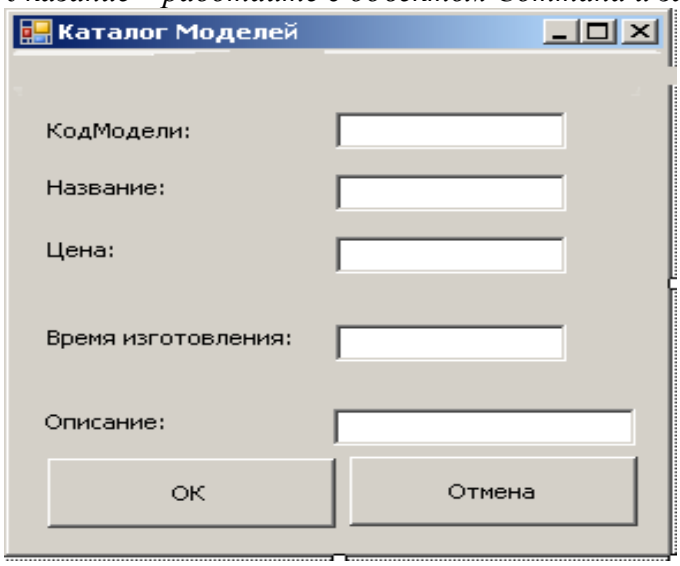


Рисунок 30 –Форма для добавления моделей

5. При нажатии на кнопку «Удалить» удаляется выделенный узел из дерева и строка из таблицы «Модель».

   Указание – работайте объектом Command и запросом Delete.

6. При нажатии на кнопку «Свойства» открывается форма, в которой отображаются все сведения о выбранной в дереве модели. Форма представлена на рисунке 31.

   Указание – работайте с объектом Command и запросом Select.

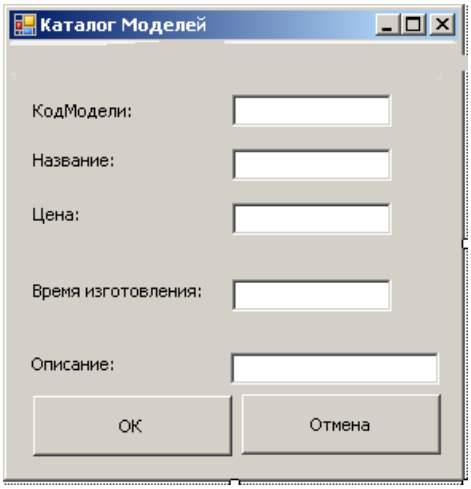
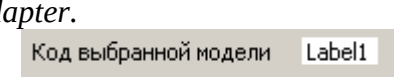


Рисунок 31 – Форма для просмотра свойств моделей

7. Нижняя часть формы представляет собой компонент DataGridView, в котором отображаются сведения из таблицы Готовая продукция. При выборе в верхней части формы (в дереве) какого-либо узла=модели, в нижней части формы отображается вся готовая продукция данной модели.

   Указание – напишите обработчик на событие Select объекта дерево, Работайте с объектом DataSet и DataAdapter.

8. Данная часть формы 

-это компонент Label, в котором отображается Код выбранной в дереве модели

   Указание – напишите обработчик на событие Select объекта дерево, Работайте со свойством Text объекта label.

Список литературы
Основные источники

1. Гагарина, Л. Г. Технология разработки программного обеспечения: учебное пособие / Л.Г. Гагарина, Е.В. Кокорева, Б.Д. Сидорова-Виснадул; под ред. Л.Г. Гагариной. — Москва: ФОРУМ : ИНФРА-М, 2021. — 400 с. — (Среднее профессиональное образование). - ISBN 978-5-8199-0812-9. - Текст: электронный. - URL: https://znanium.com/catalog/product/1189951 (дата обращения: 27.05.2021).
2. Маркин, А. В. Программирование на SQL: учебное пособие для среднего профессионального образования / А. В. Маркин. — Москва: Издательство Юрайт, 2021. — 435 с. — (Профессиональное образование). — ISBN 978-5-534-11093-7. — Текст: электронный // Образовательная платформа Юрайт [сайт]. — URL: https://urait.ru/bcode/476040 (дата обращения: 27.05.2021).
3. Немцова, Т. И. Программирование на языке высокого уровня. Программирование на языке Object Pascal: учеб. пособие / Т.И. Немцова, С.Ю. Голова, И.В. Абрамова; под ред. Л.Г. Гагариной. — Москва: ИД «ФОРУМ»: ИНФРА-М, 2018. — 496 с. + Доп. материалы [Электронный ресурс; Режим доступа: https://new.znanium.com]. — (Профессиональное образование). - ISBN 978-5-8199-0753-5. - Текст: электронный. - URL: https://znanium.com/catalog/product/944326 (дата обращения: 27.05.2021).
4. Огнева, М. В. Программирование на языке С++: практический курс: учебное пособие для среднего профессионального образования / М. В. Огнева, Е. В. Кудрина. — Москва: Издательство Юрайт, 2021. — 335 с. — (Профессиональное образование). — ISBN 978-5-534-05780-5. — Текст: электронный // Образовательная платформа Юрайт [сайт]. — URL: https://urait.ru/bcode/473118(дата обращения: 27.05.2021).
5. Соколова, В. В. Разработка мобильных приложений: учебное пособие для среднего профессионального образования / В. В. Соколова. — Москва: Издательство Юрайт, 2021. — 175 с. — (Профессиональное образование). — ISBN 978-5-534-10680-0. — Текст: электронный // Образовательная платформа Юрайт [сайт]. — URL: https://urait.ru/bcode/475892 (дата обращения: 27.05.2021).

Дополнительные источники

1. Голицына, О. Л. Языки программирования: учебное пособие / О.Л. Голицына, Т.Л. Партыка, И.И. Попов. — 3-е изд., перераб. и доп. — Москва: ФОРУМ: ИНФРА-М, 2021. - 399 с. - (Среднее профессиональное образование). - ISBN 978-5-00091-613-1. - Текст: электронный. - URL: https://znanium.com/catalog/product/1209231 (дата обращения: 27.05.2021).
2. Гуров, В. В. Микропроцессорные системы: учебник / В.В. Гуров. — Москва: ИНФРА-М, 2021. — 336 с. + Доп. материалы [Электронный ресурс]. — (Среднее профессиональное образование). - ISBN 978-5-16-015323-0. - Текст: электронный. - URL: https://znanium.com/catalog/product/1514901 (дата обращения: 27.05.2021).
3. Дорогов, В. Г. Основы программирования на языке С: учебное пособие / В.Г. Дорогов, Е.Г. Дорогова; под ред. Л.Г. Гагариной. — Москва: ФОРУМ: ИНФРА-М, 2020. — 224 с. — (Среднее профессиональное образование). - ISBN 978-5-8199-0809-9. - Текст: электронный. - URL: https://znanium.com/catalog/product/1082440 (дата обращения: 27.05.2021).
4. Хорев, П. Б. Объектно-ориентированное программирование с примерами на C#: учебное пособие / П.Б. Хорев. — Москва: ФОРУМ: ИНФРА-М, 2021. — 200 с. — (Среднее профессиональное образование). - ISBN 978-5-00091-713-8. - Текст: электронный. - URL: https://znanium.com/catalog/product/1195623 (дата обращения: 27.05.2021).
5. Чернышев, С. А. Основы программирования на Python: учебное пособие для среднего профессионального образования / С. А. Чернышев. — Москва: Издательство Юрайт, 2021. — 286 с. — (Профессиональное образование). — ISBN 978-5-534-15160-2. — Текст: электронный // Образовательная платформа Юрайт [сайт]. — URL: https://urait.ru/bcode/487638 (дата обращения: 27.05.2021).

Интернет-ресурсы

1. Электронная библиотечная система Znanium: сайт.- URL: https://znanium.com/ – Текст: электронный.
2. Электронная библиотечная система Юрайт: сайт. - URL: https://urait.ru/ -Текс: электронный.