МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

С.А. Рыбалка, Г.А. Шкатова

ЯЗЫКИ И МЕТОДЫ ПРОГРАММИРОВАНИЯ

*Рекомендовано в качестве учебно-методического пособия
Редакционно-издательским советом
Томского политехнического университета*

Издательство
Томского политехнического университета
2014

УДК 004.43 (075.8)
ББК 32.973.2я73
Р93

Рыбалка С.А.

Р93 Языки и методы программирования: учебно-методическое пособие / С.А. Рыбалка, Г.И. Шкатова; Томский политехнический университет. – Томск: Изд-во Томского политехнического университета, 2014. – 72 с.

В пособии описываются практические приемы разработки приложений в стиле ООП в среде C++ Builder, начиная от постановки задачи и до разработки программного кода. Теоретические сведения сопровождаются практическими примерами с подробными комментариями.

Предназначено для студентов, обучающихся по направлению 010400 «Прикладная математика и информатика», для выполнения курсовой работы и подготовки пояснительной записки. Может быть полезно студентам других специальностей, занимающимся разработкой компьютерных приложений.

УДК 004.43 (075.8)
ББК 32.973.2я73

Рецензенты

Кандидат технических наук ведущий программист
Института сильноточной электроники СО РАН
О.С. Колобов

Кандидат технических наук доцент кафедры медицинской
и биологической кибернетики СибГМУ
О.В. Воробейчикова

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Данное методическое пособие предназначено для студентов, которые выполняют первую в процессе своего обучения курсовую работу. Целью курсовой работы является разработка и создание первого самостоятельного и достаточно большого программного продукта в среде RAD Studio (C++ Builder). Методическое пособие можно рассматривать как навигатор или путеводитель, назначение которого помочь студенту приобрести первый опыт не только в разработке законченной программы/проекта с использованием основных парадигм современного программирования, но и обеспечить его навыками в подготовке пояснительных записок в соответствии со стандартом ISO/IEC 14882 (1998).

Известно, что парадигма в программировании представляет собой совокупность идей и понятий, определяющая стиль написания программ. В общем случае различные языки программирования поддерживают различные парадигмы программирования (так называемые стили программирования). Отчасти, искусство программирования состоит в том, чтобы выбрать один из языков, наиболее полно подходящий для решения имеющейся задачи. Разные языки требуют от программиста различного уровня внимания к деталям при реализации алгоритма, результатом чего часто бывает компромисс между простотой реализации и производительностью, между временем затраченным программистом в ходе разработки и временем пользователя затрачиваемом при пользовании программой.

Особенно важно придерживаться определенного стиля и дисциплины при программировании на C++ [1, 5, 9]. Этот язык обладает настолько большой гибкостью и широкими возможностями, что, если не поставить себя в жесткие рамки с самого начала, программа быстро превратится в огромного неуправляемого монстра, не поддающегося отладке. Поэтому в перечень требований как обязательные, которые следует соблюдать при выполнении курсовой работы, выставляются следующие парадигмы программирования:

- нисходящее программирование;
- императивное программирование;
- модульное программирование;
- структурное программирование;
- ООП – программирование.

Эти требования касаются базового C++. А поскольку реализация программы осуществляется в среде визуального программирования

C++Builder, то естественным шагом является использование таких парадигм, как:

- событийное программирование;
- визуальное программирование.

Цель курсовой работы: получить навыки поэтапной разработки сложных приложений с использованием элементов объектно-ориентированного программирования (ООП) и с развитым пользовательским интерфейсом. В частности, при работе над курсовой работой студент должен:

1. получить практические навыки в разработке классов;
2. получить практические навыки в создании приложений, с использованием высокоуровневых методов программирования в среде C++ Builder;
3. приобрести навыки самостоятельной работы с литературными источниками;
4. получить навыки в изучении и анализе конкретных вопросов библиографического поиска;
5. получить навыки письменного оформления текста отчетов.

За время работы над курсовой работой студент должен пройти все основные этапы связанные с разработкой программных приложений.

1. Порядок выполнения работы

Рекомендуется придерживаться следующего порядка выполнения курсовой работы:

1. Получить у преподавателя тему курсовой работы.
2. Изучить методическое указание.
3. Познакомиться с заданием.
4. Познакомиться с графиком выполнения курсовой работы.
5. В папке для курсовой работы создать отдельный файл для хранения пояснительной записки к курсовой работе. В нем должны помещаться фрагменты, из которых в дальнейшем будет образован полный текст пояснительной записки. Первыми страницами в нем должны быть: титульный лист (приложение 3), лист задания (приложение 4). Рекомендуется сразу выставить по тексту названия глав и параграфов в нужном формате (приложение 1). В конце периода из совокупности созданных фрагментов должен сложиться весь отчет – пояснительная записка.
6. Предусмотреть ***постоянное копирование документа и проекта на другие носители*** для уменьшения потерь в случае повреждения и удаления папок.
7. Распечатать пояснительную записку и сдать ее для проверки.
8. Подготовить презентацию.
9. Защитить работу перед комиссией.

2. Требования и рекомендации

При работе над курсовой работой следует придерживаться следующих рекомендаций:

1. Исходные данные, если они есть, изначально следует разместить в файле данных.
2. Разрабатываемое приложение должно содержать не менее трех форм: главная форма; форма, на которой отражаются результаты решения задачи; форма, содержащая информацию об авторе, включая по возможности его фотоизображение.
3. В проекте должен быть предусмотрен *отдельный модуль без формы*, в котором должно быть размещено описание интерфейса класса и реализация его методов. То есть абстрактные типы данных (АТД) должны описываться в одном или нескольких отдельных модулях.
4. Исходный текст программного продукта должен быть написан в соответствии с требованиями структурного программирования.
5. Текст программы должен сопровождаться подробными комментариями.

6. В ролик, подготавливаемый к защите курсовой работы, следует включить следующие слайды: постановка задачи, концептуальная модель интерфейса, интерфейс класса, блок-схемы наиболее сложных алгоритмов, скриншоты окон с интерфейсами пользователя, скриншоты окон с результатами работы, выводы и др.

3. Структура пособия

В первой части пособия рассматривается пример анализа и исследования задачи с целью выделения объектов и построения информационных моделей. На базе построенных информационных моделей во второй главе будет производиться разработка классов – абстрактных типов данных (АТД), т.е. новых типов, необходимых для решения задачи. Последовательность действий, которая представлена на рис. 1. исходит из того, что в задаче уже выделены физические объекты. Именно физические объекты определяют интерфейс программы в целом, ее программную модель и в значительной степени влияют на логику решения задачи. Выделение объектов само по себе не является простой задачей. Для выделения объектов в части анализа и исследования задачи значительное место отводится разработке концептуальной модели пользовательского интерфейса.

Важность этапа разработки концептуальной модели пользовательского интерфейса сложно переоценить также из-за возросших требований к качеству программного продукта, как на отечественных, так и на западных рынках, т.к. в оценку программного продукта входит качество интерфейса, обеспечивающего взаимодействие пользователя с программой. Следует отметить четыре основных (все остальные – производные) критерия качества любого интерфейса [2]:

1. скорость работы пользователей;
2. количество человеческих ошибок;
3. скорость обучения;
4. субъективное удовлетворение[1].

Скорость выполнения работы с использованием программы является важным критерием эффективности интерфейса. Длительность выполнения работы пользователем состоит из следующих элементов:

- длительности восприятия исходной информации;
- длительности интеллектуальной работы (пользователь обдумывает, что он должен сделать);
- длительности физических действий пользователя;

[1] Подразумевается, что соответствие интерфейса задачам пользователя является неотъемлемым свойством интерфейса

- длительности реакции системы.

Как правило, длительность реакции системы является наименее значимым фактором. А вот значимость дизайна интерфейса нельзя недооценивать.

Процесс разработки дизайнов интерфейсов в настоящее время является настолько важным составляющим элементом при создании программного продукта, что он стал профессией отдельных людей [2]. Появились научные разработки в области проектирования пользовательского интерфейса, которые опираются на познавательные процессы человеческого сознания, т.е. знания из областей когнитивной и инженерной психологии, а также эргономики.

Согласно Дональду Норману, взаимодействие пользователя с системой (и не только с компьютерной) состоит из семи шагов [2]:

1. формирование цели действий;
2. определение общей направленности действий;
3. определение конкретных действий;
4. выполнение действий;
5. восприятие нового состояния системы;
6. интерпретация состояния системы;
7. оценка результата.

Из этого списка становится видно, что процесс размышления занимает почти все время, в течение которого пользователь работает с компьютером, во всяком случае, шесть из семи этапов полностью заняты умственной деятельностью. Соответственно, повышение скорости этих размышлений приводит к существенному улучшению скорости работы.

К сожалению, существенно повысить скорость собственно мышления пользователей невозможно. Тем не менее, уменьшить влияние факторов, замедляющих процесс мышления, вполне возможно. Очевидно, что значение этапа проектирования интерфейса приложения трудно переоценить. Иногда через него приходит понимание и закладываются основные концепции системы, ее структуры, а также выделяются объекты для проектирования классов.

Вторая часть пособия посвящена разработке классов на базе тех информационных моделей, которые построены для объектов задачи. Разработка классов ведется поэтапно в следующей последовательности: разрабатываются интерфейсы классов, при этом на начальном этапе в интерфейсе необходимые методы просто обозначаются. Прописывание прототипов производится только после того, как будут разработаны спецификации методов. Разработка классов завершается разработкой алгоритмов, реализующих методы класса, и написанием их кода. Мате-

риал этой части важен для освоения C++. В нем сочетатся теория программирования и личный опыт программистов [3, 6, 7, 8].

В третьей части изложена технология подготовки модуля для хранения разработанного класса в среде C++ Builder.

В четвертой части описывается технология использования методов класса. Значительное место в методическом пособии отводится описанию элементов обеспечения функциональности приложения. Здесь излагаются основы событийного программирования применительно к приложениям, изготовляемым в среде C++ Builder. И показываются некоторые приемы построения диалогов с пользователем.

В завершающей части представлены данные об управлении проектом и даны основные сведения о тестировании, отладке и сопровождении программного продукта. Сведения из этой главы могут оказать помощь при поиске ошибок и их локализации.

В приложениях к методическому пособию представлены требования к оформлению пояснительной записки; примеры оформления титульных листов, необходимых для оформления пояснительной записки. А также: пример подготовки аннотации; список вопросов, которые должны быть раскрыты во введении и примеры раскрытия этих вопросов; пример содержания пояснительной записки; пример заключения.

4. Нисходящее программирование

Программирование с использованием технологии «сверху вниз». Нисходящее программирование – методика разработки программ, при которой разработка начинается с определения целей решения проблемы/задачи, после чего идет последовательная детализация. А именно, чтобы решить задачу, она делится на подзадачи, подзадачи делятся на подзадачи и т.д. до тех пор, пока алгоритм каждой из подзадач не станет очевиден. Методы, на которые можно разделить первоначальную задачу, зависят непосредственно от методов, которыми можно склеивать/объединять между собой решения отдельных подзадач.

4.1. Императивное (процедурное) программирование

Использование процедур является естественным способом борьбы со сложностью любой задачи. В C++ задача может быть разделена на более простые и обозримые части с помощью функций, после чего программу можно рассматривать в более укрупненном виде – на уровне взаимодействия функций. Это важно, поскольку человек способен помнить ограниченное количество фактов.

Использование функций является первым шагом к повышению степени абстракции программы и ведет к упрощению ее структуры.

Разделение программы на функции позволяет также избежать избыточности кода, поскольку функцию записывают один раз, а вызывать ее на выполнение можно многократно из разных точек программы. Процесс отладки программы, содержащей функции, можно лучше структурировать. Часто используемые функции можно помещать в библиотеки. То есть формировать коллекции заранее отлаженных фрагментов программ. Таким образом, создаются более простые в отладке и сопровождении программы.

4.2. Модульное программирование

Модульное программирование предполагает формирование полного текста программы из нескольких связанных модулей. Развитие модульного программирования было обусловлено:

- возрастающими объемами программ;
- увеличивающейся внутренней сложностью;
- коллективным характером разработок.

В общем случае модуль представляет собой совокупность программных ресурсов, предназначенных для использования другими модулями и программами.

Интерфейсом модуля являются заголовки всех функций и описания, доступных извне типов, переменных и констант. Описания глобальных программных объектов во всех модулях программы должны быть согласованы.

Модульность в языке C++ поддерживается с помощью: директив препроцессора, пространств имен, классов памяти, исключений и раздельной компиляции (строго говоря, раздельная компиляция не является элементом языка, а относится к его реализации). Модульность ведет к повышению производительности при создании проекта. Прежде всего, маленькие модули могут быть закодированы быстро и легко. Во-вторых, универсальные модули могут многократно использоваться, что приводит к более быстрому построению последующих программ. В-третьих, модули программы могут быть проверены независимо, что помогает уменьшить время, потраченное на отладку.

4.3. Структурное программирование

Концепции/парадигмы структурного программирования предполагают создание удобочитаемых программ, характерными особенностями которых являются модульность, отказ от оператора безусловного перехода, ограниченное использование глобальных переменных. Основная идея структурного программирования соответствует принципу «Разделяй и властвуй», т.е. программа делится на ряд простых подзадач.

Структурное программирование – методология и технология разработки программных комплексов, основанная на принципах:

- нисходящего программирования или программирования "сверху вниз";
- модульного программирования.

При этом логика алгоритма и программы должны использовать три основные структуры: последовательное выполнение, ветвление и повторение. Блоки в структурной программе не имеют несколько входов или выходов. Структурные программы лучше поддаются математической обработке, чем их неструктурные аналоги.

Структурное программирование обычно связывают с правилами, которыми следует придерживаться при написании текстов программ:

- сопровождение текста грамотными комментариями;
- подчеркиванием вложенности операторов: если оператор принадлежит другому оператору, то принадлежность следует показывать смещением[2] этого оператора относительно того оператора, которому он принадлежит;
- при написании программы на C++ следует фигурные скобки располагать на отдельных строчках и друг под другом.

4.4. Объектно-ориентированное программирование

Если приемы процедурного программирования концентрируются на алгоритмах, используемых для решения задачи, и не обращается внимание на структуры данных, то ООП концентрируется на сути задачи. Элементы программы разрабатываются в соответствии с объектами, присутствующими в описании задачи. При этом первичными считаются объекты (данные), которые могут активно взаимодействовать друг с другом с помощью механизма передачи сообщений (обычно называемого механизмом вызова методов). При разработке программы на основе принципов ООП функция программиста заключается в том, чтобы придумать и реализовать такие объекты, взаимодействие которых после старта программы приведет к достижению необходимого конечного результата.

ООП есть, по сути*, императивное программирование*, дополненное принципом инкапсуляции данных и методов в объект (принцип модульности) и наследованием (принцип повторного использования разработанного функционала). Общий подход ООП состоит в определении набора объектных типов – классов, которые интегрируют структуры

[2] Обычно сдвиг оператора производится на две позиции

данных и операции, необходимые для решения поставленной задачи, и включает в себя следующие концепции:

1. Наличие типов, определенных пользователем.
2. Скрытие деталей реализации (инкапсуляция).
3. Использование кода через наследование.
4. Разрешение интерпретации вызова функции во время выполнения программы (полиморфизм).

Как только эти классы определены, создаются переменные типов классов и вызываются операции для выполнения их обработки.

Одно из мощных преимуществ классов, как типов данных, заключается в том, что классам присуща структура, позволяющая моделировать реальные объекты. Отсюда и терминология – «Объектно-ориентированное программирование».

Известно, что реальные/физические объекты окружающего мира обладают тремя базовыми характеристиками:

1. Набор свойств, характеризующий объект.
2. Способность объекта изменять свойства.
3. Способность реагировать на события, как в окружающем мире, так и внутри самого себя.

Понятие класса тесно связано с понятием «Абстрактный тип данных».

4.5. Абстрактный тип данных

Хотя термины «тип данных» (или просто тип) и «абстрактный тип данных» звучат похоже, они имеют различный смысл. В языках программирования тип данных (переменной) обозначает множество значений, которые может принимать эта переменная. Абстрактный тип данных (АТД) определяется как *математическая модель с совокупностью операторов, определенных в рамках этой модели*. Таким образом, *классы* могут служить простым примером АТД. Также примером АТД могут служить и структуры. В модели АТД операторы могут иметь операндами не только данные, определенные АТД, но и данные других типов: базовые типы языка программирования или определенные в других АТД. Результат действия оператора также может иметь тип, отличный от АТД, определенного в данной модели. Но предполагается, что по крайней мене один операнд или результат любого оператора имеет тип данных, определенный в рассматриваемой модели АТД.

Моделирование объектов в программе также называется *абстракцией*. Речь идет об имитации реально существующих объектов, отражающей особенности их взаимодействия в окружающем мире. А концепции виртуальной реальности выводят принцип абстракции на совер-

шенно новый уровень, не связанный с физическими объектами. Абстракция необходима, потому что успешное использование ООП возможно лишь в том случае, если вы сможете выделить содержательные аспекты своей проблемы.

В определении нового типа всегда лежит идея – отделить (абстрагироваться) несущественные подробности реализации от тех качеств, которые существенны для его правильного использования. Для того чтобы создать абстрактный тип данных требуется разработать информационную и математическую модели реального или физического объекта.

5. Информационная модель

Информационная модель по определению представляет собой совокупность информации, характеризующей существенные свойства и состояние физического объекта, процесса, явления, а также взаимосвязь с внешним миром.

Информационные модели делятся на *описательные* модели и *формальные*.

Описательные информационные модели – это модели, созданные на естественном языке. Т.е. на любом языке общения между людьми: английском, русском, китайском, мальтийском и т.п. в устной или письменной форме.

Формальные информационные модели – это модели, созданные на формальном языке (т.е. научном, профессиональном или специализированном). Примеры формальных моделей: все виды массивов, таблицы, графы, схемы и т.д.

6. Математическая модель

Под математической моделью понимают систему математических соотношений – формул, уравнений, неравенств и т.д., отражающих существенные свойства объекта или явления.

Создать математическую модель – это значит записать математические соотношения, связывающие результаты с исходными данными. Очевидно, на начальном этапе следует определить, что считать исходными данными и результатами. Под результатами можно понимать, как получение новых свойств объекта, так и получение характеристик, определяющих качество или состояние объекта, например, для взаимодействия объекта с внешней средой.

7. Общая схема решения задачи

Из вышеизложенного следует, что для создания класса или нового абстрактного типа необходим предварительный анализ задачи. Этот анализ рекомендуется проводить по шагам последовательно в соответствии со схемой, предложенной на рис. 1.

В соответствии с этой схемой, на начальном этапе исследования задачи нужно *выделить физические объекты*. После чего нужно определить свойства, характеризующие каждый физический/реальный объект. Следует отметить, что из множества свойств объекта следует взять только те, которые необходимы для решения задачи. Чтобы избежать избыточности следует выяснить: какие из свойств являются базовыми; какие свойства можно получить на основании базовых свойств; какие свойства будут меняться; как воздействовать на объект с целью изменения его свойств; в каких процессах будет задействован объект и как это отразится на его свойствах.

Следующим шагом в исследовании задачи будет *построение информационной модели объекта*. Здесь требуется дать названия свойствам, установить диапазоны возможных значений свойств и определить их типы. Выяснить, что считать *исходными данными* и что считать *результатами*. Определить возможные места расположения данных. Если эти данные расположены или в результате решения будут располагаться на внешних носителях, например, в виде файлов, нужно описать структуру, в которой они представлены или будут представляться на этом носителе.

Построение математической модели сводится к записи математические соотношения, связывающие результаты или новые свойства с исходными данными. Эти соотношения, как правило, представлены в аналитическом виде формулами.

Информационная и математическая модели служат основой для *построения программной модели* объекта или класса. Построение класса начинается с разработки его интерфейса. Поля класса формируются из информационной модели. А в методы класса помимо стандартных методов (конструкторов и деструктора), включаются методы, связанные с изменением свойств объекта, получением новых свойств как воздействием на объект, так и на основании соотношений, заявленных в математической модели.

Пример 1. Физическая постановка задачи звучит так: требуется указать наилучший вариант консервной банки фиксированного объема V, имеющей форму прямого кругового цилиндра. При этом наилучшая банка должна иметь наименьшую длину швов (минимизировать объем

сварки). Требуется разработать интерфейс класса, выполняя шаги в соответствии со схемой, представленной на рис. 1.

Решение. Физический объект – «Консервная банка» (имя Cups). Отметим, что в общем случае можно предложить множество свойств для описания консервной банки: материал, цвет, вес, вид продукта для консервирования, стоимость и т.д. Но из всего набора возможных свойств должны быть выбраны только те, которые существенны для решения задачи данной задачи – это объем банки, высота банки, радиус окружности, представляющей основания банки, длину шва для сварки.

Рис. 1. Схема процесса создания программной модели (класса) в рамках ООП

Укажем модели объекта данной задачи:

а) Информационная модель объекта «Консервной банки». Свойства:
- объем V – вещественная переменная, принимающая положительные значения;
- высота h – вещественная переменная, принимающая положительные значения, не превышающие 20 см;
- радиус R – вещественная переменная, принимающая положительные значения, не превышающие 20 см;
- длина шва l – вещественная переменная, принимающая положительные значения. Длина шва определяется периметром двух донных окружностей и высотой банки.

б) Математическая модель представлена формулами:
$V = \pi \cdot R^2 \cdot h$; $l = (2 \cdot \pi \cdot R) \cdot 2 + h$; – определение объема и длины швов;
$h = \frac{V}{\pi \cdot R^2}$; $l(R) = 4 \cdot \pi \cdot R + \frac{V}{\pi \cdot R^2}$; – длина швов через радиус;
$l(R)' = 4 \cdot \pi - 2 \cdot \frac{V}{\pi \cdot R^3}$; $R = \sqrt[3]{\frac{V}{2 \cdot \pi^2}}$. – вычисление производной и поиск решения. То есть исходя из требуемого объема V можно вычислить радиус дна оптимальной R банки, а затем и ее высоту h.

Действия производимые с объектом: задать значение V, получить значение параметра R оптимальной банки и высоту h, напечатать значения параметров R и h.

в) Программная модель или класс может иметь вид, как это показано на рис. 2.

```
class Cups
{
   float V; //объем банки
   float h;  // высота
   float R; //радиус основания
   float l; //длина шва
    public:
      Cups();
     void Take_V();      //задать значение V
     void Print_Rhl();  //напечатать значения параметров R и
h
      void GetRhOpt();     //получить параметры R и h опти-
мальной банки
};
```

Рис. 2. Интерфейс класса «Консервная банка»

8. Анализ и исследование задачи

Задача данного этапа работы – перейти от физической постановки задачи к ее информационной модели. Траектория такого перехода сводится к последовательному выполнению следующих шагов: построение концептуальной схемы пользовательского интерфейса; описание сценариев взаимодействия пользователя с программой; разработка информационной модели; построение математической модели. Для того чтобы внести в исследование элемент конкретности и облегчить восприятие информации все пояснения будут проводиться в рамках решения конкретной задачи.

9. Постановка задачи

Требуется разработать класс «Фонотека», необходимый для решения следующих задач: получить информацию о доступных фонограммах, получить информацию об авторах и исполнителях музыкальных произведений, познакомиться с особенностями музыкального стиля и т.п.

9.1. Разработка концептуальной схемы пользовательского интерфейса

Проектирование интерфейса начинается с разработки концептуальной схемы. Концептуальная модель – это еще не пользовательский интерфейс. Она абстрактно (в терминах задач или экранной графики) описывает, что именно пользователь должен делать с системой, и какие концепты ему необходимо знать. Принятая структура концептуальной схемы интерфейса влияет абсолютно на все показатели качества системы. Допущенные ошибки в проектировании практически не могут быть обнаружены и решены на остальных этапах (для их обнаружения нужно слишком много везения, а для исправления – денег). Это значит, что чем больше внимания будет уделено проектированию, тем выше будет общее качество.

Собственно проектирование состоит из следующих этапов [2]:

1. определение необходимой функциональности системы;
2. создание пользовательских сценариев;
3. проектирование общей структуры;
4. получение ответа на вопрос: «Верно ли то, что методы, которые я собираюсь избрать, самые лучшие?»;
5. конструирование отдельных блоков;
6. создание глоссария;
7. сбор и начальная проверка полной схемы системы.

Каждый последующий этап в такой системе зависит от результатов предыдущих этапов. Соответственно, пропуск какого-либо этапа (за исключением, разве что, создания глоссария) негативно влияет на результаты всех последующих этапов.

9.1.1. Определение необходимой функциональности системы

Определение функциональности будущей системы – начальный этап работы в направлении по созданию приложения. Это исключительно важный этап, поскольку именно функциональность системы не только будет определять весь интерфейс в целом, но и позволит решить задачу определения реальных объектов – участников решения. Задача выделения объектов необходима для того, чтобы построить классы в соответствии со схемой, представленной на рис. 1.

Современная наука выдвинула два основных способа [2] определения *функциональности*, а именно, анализ целей пользователей и анализ действий пользователей. Эти способы фактически не конфликтуют друг с другом, более того, в процессе определения функциональности желательно использовать оба. После того как истинные цели пользователей установлены (и доказано, что таких пользователей должно быть достаточно много, чтобы оправдать создание системы), приходит время выбирать конкретный способ реализации функций, для чего используется второй метод.

Оказать помощь в решении задачи определения необходимой функциональности системы поможет *сбор информации о предполагаемой системе*. Этот этап можно реализовать в виде постановки вопросов и получения ответов на них. Причем, список вопросов должен быть тщательно продуман как со стороны изготовителя программного продукта, так и со стороны пользователя.

Содержание вопросов и ответов зависит от задачи. Ответы на вопросы позволяют определить: какая информация может понадобиться пользователю; какими знаниями владеет пользователь; как пользователь видит исходные данные и что он хочет получить в результате; цели действий пользователя; общей направленности действий; выделить конкретные действия; что он хочет получить на выходе; какие ситуации могут возникнуть и как, по мнению пользователя, система должна на них реагировать; в какое состояние может привести система в зависимости от действий пользователя; как пользователь воспринимает и интерпретирует новое состояние системы. В таблице 1 представлен возможный список вопросов и ответов для решения сформулированной выше задачи.

Анализ такой таблицы позволяет выделить объекты и обозначить функциональность системы.

9.1.2. Выделение объектов

С целью выделения объектов осуществляется поиск существительных и глаголов в списке вопросов и ответов, представленных в таблице. Существительные становятся объектами, которые в дальнейшем будут представлены классами, а глаголы станут методами класса.

В результате анализа таблицы 1 можно выделить следующие объекты и действия.

Объекты: фонотека; музыкальные стили; исполнители; авторы.

Программа должна предоставлять пользователям следующие возможности: решать задачи, заявленные в условии; задавать исходные данные; выдавать результаты в зависимости от критериев; генерировать значения по какому-либо правилу; формировать файл с целью повторного использования; извлекать данные из файла данных и т.д.

Таблица 1

Пример списка вопросов и ответов для определения функциональности системы и выделения объектов

Вопрос	Ответ
Для кого создается программный продукт?	Любитель музыки. Имеет гуманитарное образование.
Какими знаниями обладает пользователь?	Не имеет специального образования в области компьютерных технологий. Имеет простейшие навыки работы с ПК
С какой целью проектируется программный продукт?	Предоставить пользователю возможность: получить информацию о доступных фонограммах, получить информацию об исполнителя, об авторах музыкальных произведений; познакомиться с особенностями музыкального стиля произведения.
Как пользователь видит исходные данные?	В виде таблицы, в которой содержится список сведений об имеющихся фонограммах. Фотоизображения авторов.
Какие проблемы есть сейчас у пользователей?	Информация не организована. Трудности, связанные с фильтрацией и обработкой данных.
Что он хочет получить в результате?	Выборочные сведения, характеризующие авторов, исполнителей, стилей и т.д.
Как представляет пользователь свое взаимодействие с программой?	В виде диалогов, где ему предлагается выбрать нужное действие с системой подсказок и возможностью как выполнить действие, так и отказаться от выполнения.
Какие ситуации могут возникнуть и как, по мнению пользователя, система должна на них реагировать?	Могут отсутствовать данные по запросу пользователя. В этом случае нужно иметь сообщение об отсутствии, а также возможности смены критерия выбора, смены цели и действия, и предоставление возможности завершить задачу.

Окончание табл. 1

В какое состояние может привести система в зависимости от действий пользователя?	Открыть диалог для задания данных, если требуется участие пользователя с системой подсказок. Выполнить одну из команд, предложенных в диалоге.
Дополнительные события и действия, предусмотренные сценарием?	Помощь в выборе целей и последовательность действий, необходимых для достижения выбранной цели, а также в случае ошибочных действий. Помощь программы в интерпретации нового состояния системы после очередного перехода.
Какая информация может понадобиться участникам сценария для достижения их целей?	Например, правила заполнения диалоговых окон, которые предоставляет программа.
Что пользователь хочет получить на выходе (решение задачи). В каком виде.	Таблицы данных. Список строк, в которых содержатся тексты с затребованной информацией. Файлы с информацией для повторного использования данных.

Действия:

- Предоставить пользователю список стилей музыкальных произведений, представленных в фонотеке.
- Предоставить пользователю фонограммы заданного стиля.
- Представить пользователю характеристику стиля с заданным названием стиля.
- Предоставить пользователю список авторов, создавших произведения, выполненных в заданном стиле.
- Предоставить пользователю список авторов-исполнителей произведений, выполненных в заданном стиле.
- Предоставить пользователю названия произведений, выполненных в заданном стиле.
- Предоставить пользователю информацию об авторе-исполнителе
- Предоставить пользователю список всех исполнителей
- Предоставить пользователю список произведений определенного исполнителя
- Предоставить пользователю сведения об авторе музыкального произведения
- Предоставить пользователю всю фонотеку.

Отметим, что эти действия связаны с разными объектами, которые перечислены в списке объектов.

Выделенные объекты и представленные действия – это список всех видимых пользователю объектов приложения и действий, которые

пользователь может совершать над каждым объектом. В реализации системы могут присутствовать и другие объекты (по усмотрению разработчика и скорее всего они обязательно будут), но предполагается, что они будут невидимыми для пользователя и будут являться прерогативой самих разработчиков программы.

После того как информация о будущей системе собрана, можно приступить к созданию общей структуры системы («вид с высоты птичьего полета»). Проектирование общей структуры состоит из двух параллельно проходящих процессов: выделения независимых блоков и определения связи между ними [2]. Под отдельным функциональным блоком понимается функция/группа функций, связанных по назначению или области применения.

В приведенном примере функциональные блоки можно сформировать в соответствии со списком действий, выделенных при анализе таблицы 1. Кроме этого, программа должна представлять пользователю необходимые справки: о программе, об изготовителе программы.

Существует три основных вида связи между блоками [2]. Это *логическая связь, связь по представлению пользователей и процессуальная связь.* Логическая связь определяет взаимодействие между фрагментами системы с точки зрения разработчика. Пользователи имеют свое мнение о системе, и это мнение тоже является важным видом связи. Все три типа взаимосвязи должны быть заранее предусмотрены при конструировании системы.

Разберем это подробнее.

Логическая связь. С установлением логической связи между модулями обычно проблем не возникает. Важно только помнить, что полученные связи очень существенно влияют на навигацию в пределах системы (особенно, когда система многооконная). Поэтому, чтобы не перегружать интерфейс, стоит избегать как слишком уж отдельных блоков (их трудно найти), так и блоков, связанных с большим количеством других. Считается для одного блока оптимальным числом связей – три.

Связь по представлению пользователей. В информационных системах (читай – в интернете), когда необходимо гарантировать, что пользователь найдет всю нужную ему информацию, необходимо устанавливать связи между блоками, основываясь не только на точке зрения разработчика, но основываясь на представлениях пользователей. Дело в том, что чуть ли не единственный распространенный способ поиска, а именно поиск по классификации признаков, работает только в том случае, когда пользователи согласны с принципами этой классификации. Большинство же понятий однозначно классифицированы быть не могут из-за наличия слишком большого количества значимых признаков.

Процессуальная связь. Установление качественной процессуальной связи обычно довольно трудная задача, поскольку единственным источником информации является наблюдение за пользователями. В то же время установление такой связи дело исключительно полезное. Зачем, например, рисовать на экране сложную систему навигации, если точно известно, к какому блоку пользователь перейдет дальше? В этом смысле зачастую оправдано навязывать пользователю какую-либо процессуальную связь, жертвуя удобством, зато выигрывая в скорости обучения (поскольку пользователю приходится думать меньше). Жестко заданная связь позволяет также уменьшить количество ошибок, поскольку от пользователя при ней не требуется спрашивать себя «не забыл ли я чего?». Хорошим примером жестко заданной процессуальной связи является устройство мастеров, при котором пользователя заставляют нажимать кнопку **«Далее»**.

Следует отметить, что объекты концептуальной модели приложения могут образовывать структурную иерархию, в которой дочерние блоки будут перенимать действия родительских блоков. В зависимости от приложения объекты могут также образовывать иерархию включения, в которой некоторые объекты включают в себя другие. Использование двух этих типов иерархии в концептуальной модели значительно облегчает проектирование и разработку связного и понятного пользовательского интерфейса.

Подобный анализ объектов и действий помогает управлять реализацией системы, поскольку он указывает наиболее удобный вид иерархии объектов, а также методы работы, которые предусматривает каждый вид. Он также облегчает структуру команд приложения, т.к. позволяет разработчику увидеть, какие действия применимы к разным объектам и могут быть спроектированы как обобщенные. В свою очередь, это делает структуру команд более легкой для изучения пользователем: вместо того, чтобы осваивать большое количество объектно-ориентированных команд достаточно изучить несколько обобщенных, применяемых к разным объектам.

Можно представить схему в несколько этапов: сначала в укрупненных блоках, а потом отдельные блоки прописать более подробно. Тогда схема будет представлена на нескольких рисунках.

На рис. 3. приведена концептуальная схема функционирования программы, построенная в укрупненных блоках на базе некоторого абстрактного примера. Для большинства задач различие будет только в блоках:

Рис. 3. Абстрактный пример построения концептуальной схемы функционирования программы, представленной в укрупненных блоках

- «Просмотреть диалог» – здесь должны быть представлены альтернативы целей пользователя (просмотреть фонотеку; получить информацию о музыкальном стиле; получить информацию об исполнителе; и т.д.).
- «Создать сообщение в диалог» – здесь должна быть конкретность по задаче. Например, если пользователь выбрал цель: «Получить информацию о музыкальном стиле», система должна предоставить ему диалоговое окно, в котором имеется поле для выбора стиля из предусмотренного в системе списка стилей с возможностью подтвердить свой выбор (передать сообщение в диалог) или отказаться от него. А также возможность перейти к блоку «Получить решение» с указанием, в каком виде нужно получить решение (передать сообщение в диалог) и куда его направить.
- «Изменить данные». Такой блок должен обеспечить пользователю возможность изменить цель (вернуться к блоку «Просмотреть диалог») или отредактировать «Сообщение в диалог» в уже выбранной цели.

На рис. 4 представлен один из вариантов детализации фрагмента концептуальной схемы для задачи о фонотеке при условии выбора действия «Найти фонограмму». Здесь: значок  означает – предоставить пользователю информацию в соответствии с выбранным видом;  означает – выбрать критерий поиска;  – закрыть задачу;  означает – определить новую цель.

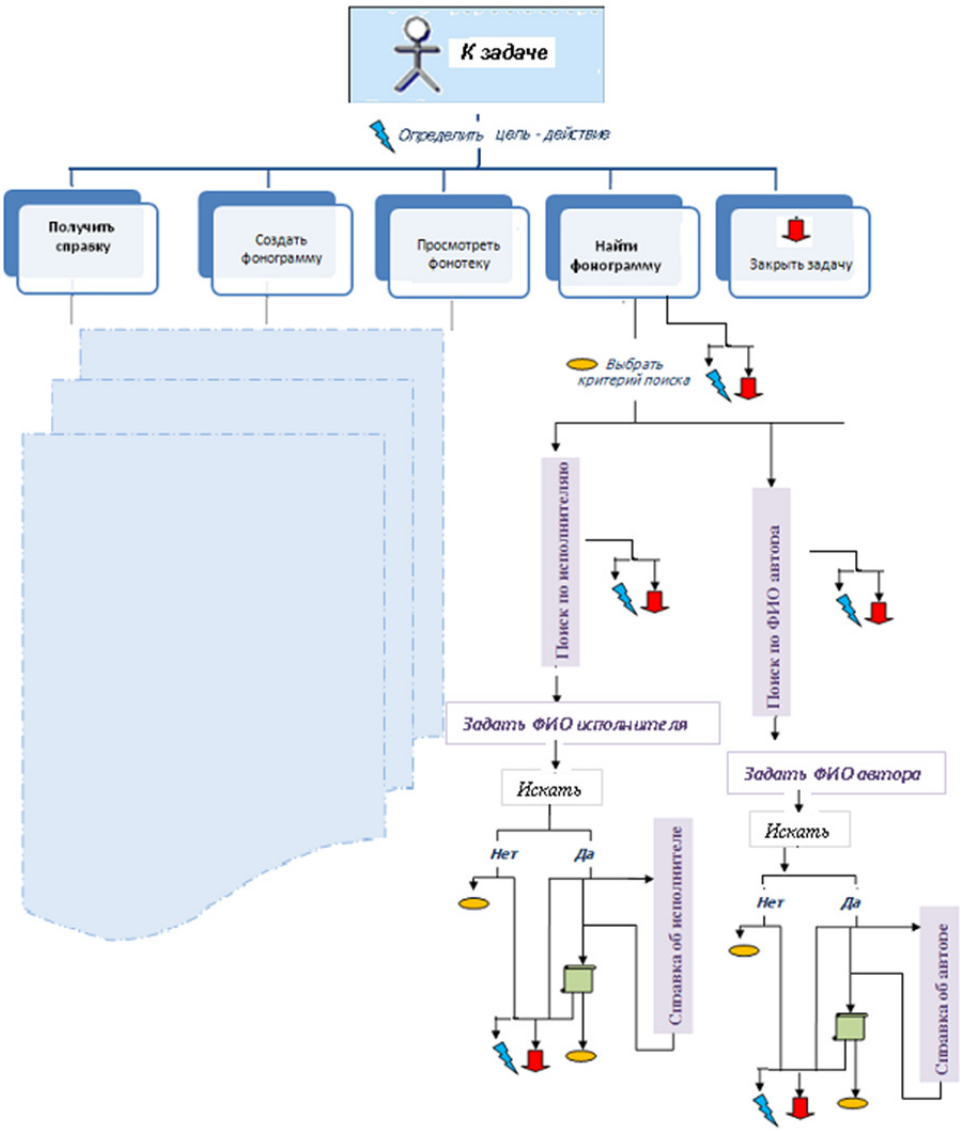


Рис. 4. Фрагмент концептуальной схемы
с детализацией цели «Найти фонограмму»

Концептуальная схема/модель представляет собой начальный этап разработки терминологии программного продукта, то есть словаря терминов, которые будут использоваться для идентификации каждого объекта и действия, реализованного в программе. Это способствует сохранению устойчивости терминологии не только в самой программе, но и в сопутствующей документации.

Словарь терминов, характеризующих предметную область: фонотека, фонограмма, стиль музыкального произведения, автор музыкального произведения, исполнитель музыкального произведения, критерий поиска.

При разработке концептуальной схемы высоко оценивается индивидуальность. Если объект/действие не требует участия пользователя –

не рекомендуется показывать его пользователю, пользователь не должен об этом знать.

9.2. Создание пользовательских сценариев

После разработки концептуальной схемы функционирования программы/приложения необходимо представить словесный сценарий, описывающий работу пользователя с приложением и детали управления – логику пользователя при решении задачи. При этом нужно использовать лишь терминологию из области задач. Здесь не следует конкретизировать, как именно проходит взаимодействие, но уделяя возможно большее внимание всем целям пользователей. Количество сценариев может быть произвольным, главное, что они должны включать все типы задач, стоящих перед системой, и быть сколько-нибудь реалистичными. Сценарии очень удобно различать по именам участвующих в них вымышленных персонажей. Пользователь должен знать:

- что он будет иметь в результате перехода в новое состояние (после выполнения очередного действия) и как интерпретировать эти результаты;
- как взаимодействовать с диалоговыми окнами, которые ему предоставляет программа;
- какие сообщения ему может послать программа в процессе своей работы, и как ему реагировать на эти сообщения;
- как определять пригодность объектов для их использования;
- как управляться с объектами.

Список, как видим, довольно внушительный. Помочь разобраться в том, что делать пользователю в той или иной ситуации должен интерфейс, со встроенной системой подсказок действий. Следовательно, должен быть продуман механизм управления программой через элементы интерфейса.

Описание сценария взаимодействия пользователя с интерфейсом можно, например, начать словами:

В соответствии с концептуальной схемой, представленной на рис. 4 (конечно, схема д. б. настроена под задачу), после запуска программы пользователю предоставляется возможность:

- Получить информацию об авторе.
- Получить информацию о программе.
- Приступить к решению задачи.
- Закрыть задачу.

Затем описать/сказать, что будет, если выбрать **каждое** из направлений. Например, сказать:

«Если пользователь выбирает направление «Приступить к решению задачи», ему предоставляется список целей/задач, которые он может достигнуть при выборе данного блока:

1. Получить справку.
2. Создать новую фонограмму.
3. Просмотреть фонотеку.
4. Найти фонограмму.
5. Закрыть задачу».

А затем все подробности, связанные с очередным выбором. Например, описание задачи «Найти фонограмму» может быть следующим:

«Если пользователь выбирает направление «Найти фонограмму», ему предоставляется диалоговое окно с предложением выбора критерия для поиска фонограммы. В диалоговом окне пользователю для выбора предоставляется список возможных критериев поиска фонограммы и несколько кнопок для выполнения действий:

- Сменить цель.
- Завершить задачу.
- Произвести поиск.

Причем, действие «Произвести поиск» пользователь может выполнить, только в том случае, если будет выбран критерий в списке критериев.

Если пользователь выберет критерий «Поиск по исполнителю» и выполнит команду «Произвести поиск», программа предоставит окно, в которое пользователю будет предложено вписать ФИО исполнителя. После заполнения полей данных он может выбрать действия: «Искать». Результаты поиска могут быть «Да» или «Нет». «Да» говорит о том, что найдены фонограммы в соответствии с выбранным критерием. Направление «Нет» говорит о том, что поиск не привел к успеху, т.е. нет фонограмм с исполнителем ФИО. В случае «Нет» пользователь может выбрать действия:

- Сменить цель.
- Завершить задачу.
- Выбрать новый критерий для поиска фонограммы.

В случае «Да» можно выполнить действия:

1. Выбрать вид отображаемой информации и выполнить команду для того, чтобы задать вид отображаемой информации и отобразить информацию в соответствии с выбранным видом. После чего перейти к пунктам 2 или 3.
2. Сменить цель.
3. Завершить задачу.

Следует отметить, что этот сценарий ссылается только *на объекты и действия области задач*, а не на свойства пользовательского интерфейса.

9.3. Разработка информационной модели объектов

В этом разделе следует собрать информацию о тех объектах, которые были выделены на этапе разработки концептуальной модели функционирования интерфейса приложения, и представить информационные модели этих объектов.

На этапе разработки концептуальной модели интерфейса приложения были выделены следующие объекты:

- фонограмма;
- фонотека;
- музыкальные стили;
- авторы-исполнители;
- авторы произведений.

9.3.1. Объект «Фонограмма»

Объект «Фонограмма» можно описать следующими свойствами:

- Название произведения.
- Автор произведения.
- Автор-исполнитель.
- Музыкальный стиль.
- Год создания.

Анализ свойств объекта «Фонограмма» показывает, что этот объект нельзя рассматривать независимо от других объектов задачи. Во-первых, объект «Фонограмма» является элементом объекта «Фонотека». Во-вторых, объекты: «Музыкальные стили», «Авторы произведения», «Авторы-исполнители» связаны с объектом «Фонограмма» в некоторую структурную иерархию, как это показано на рис. 5. Из схемы видно, что объект «Фонограмма» можно рассматривать как объект, который будет пользоваться информацией и действиями от других объектов: «Музыкальные стили», «Авторы произведения», «Авторы-исполнители».

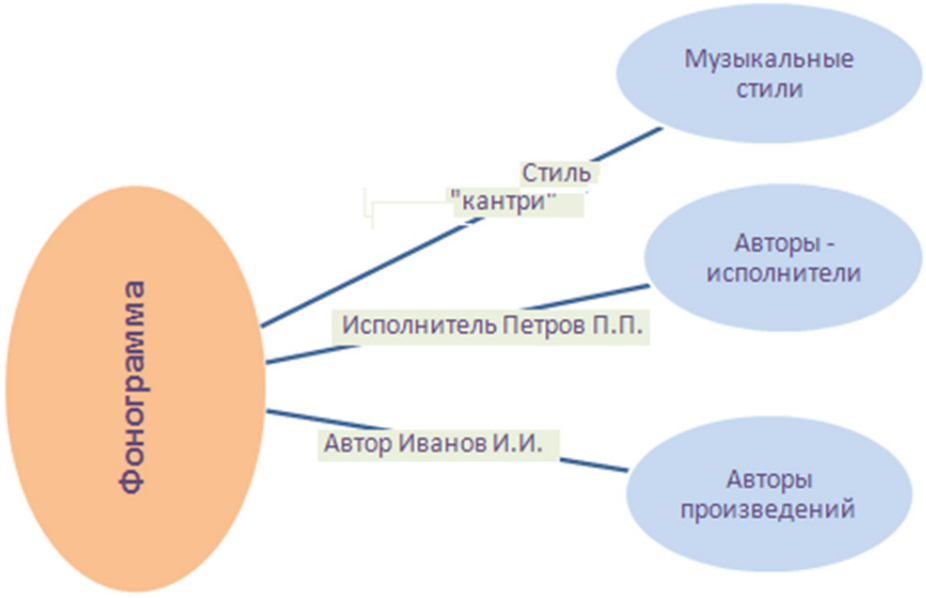


Рис. 5. Схема связности объектов:
«Стили произведения», «Авторы произведения»,
«Авторы-исполнители» с объектом «Фонограмма»

Прежде чем приступить к разработке информационной модели для выделенных объектов *следует определиться с входными и выходными данными задачи*. В первую очередь нужно определить, какие данные следует считать исходными и где будут расположены эти исходные данные.

Задача определения способа хранения, порядка и правил организации данных, является очень важным элементом при разработке приложения. Четкое структурирование данных, выполняемое в процессе разработки, способствует уменьшению сложности системы и снижает вероятность ошибок из-за их неправильного использования.

Очевидно, что в качестве места хранения данных здесь целесообразно использовать файл. Это может быть один файл. Файлов может быть несколько. На начальном этапе проектирования необходимо определить количество файлов и описать структуру этих файлов.

Для реализации данного примера предлагается следующий набор текстовых файлов:

- файл для хранения списка фонограмм (имя файла – «Фонотека.txt»);
- файл, в котором будет храниться информация, представляющая музыкальные стили (имя файла – «Музыкальные стили.txt»);
- файл, в котором будет содержаться информация об авторах-исполнителях (имя файла – «Исполнители.txt»).

Кроме этого для каждого автора музыкального произведения будет создано три файла. Наименования этих файлов для удобства работы будут совпадать с ФИО автора, а различие только в расширении. Предположим, что условное ФИО автора – «Автор1». Тогда в файле с именем «Автор1.bgr» будут храниться биографические данные, в файле с именем «Автор1.lst» – будет храниться список его произведений, а в файле с именем «Автор1.bmp» – его графическое изображение. Будем предполагать, что перечисленные файла могут располагаться на любом диске и в любом месте диска, но обязательно соблюдается условие – все файлы одной фонотеки *сосредоточены в одной папке*. Структура файлов может быть такой, как это представлена на рис. 6.

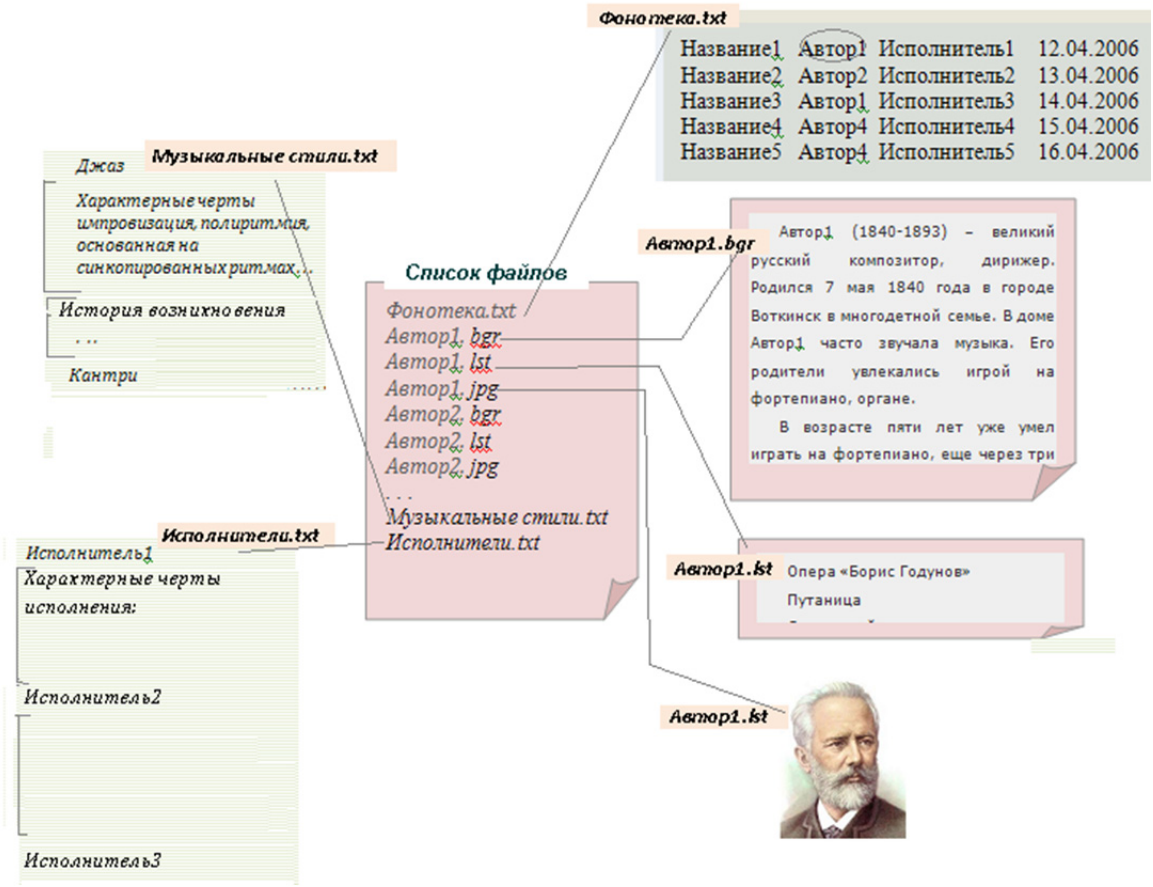


Рис. 6. Структура файлов для программы «Фонотека»

Следует отметить, что в файле «Фонотека.txt» данные располагаются по строчкам, в каждой строке записаны отдельные сведения о фонограмме, которые разделяются пробелами. В файле «Музыкальные стили.txt» в отдельной строке расположено наименование стиля, а затем описание стиля, например в 5 строках, а затем в 3-ех строках история зарождения. Это одинаково для всех стилей. Под стиль отводится в файле 9 строк. Например, если стилей 12, то количество строк в файле

получается из расчета 9×12. В файле «Исполнители.txt» в отдельной строке ФИО исполнителя, затем, для примера, в 8-х строках характеристика исполнителя.

Описание информационной модели объекта «фонограмма» (FonoGramma) представлено в таблице 2.

Таблица 2

Пример описания информационной модели объекта «фонограмма»

Имя объекта		TFonoGramma	
Набор свойств и характеристика свойств			
Название свойства	Название в программе	Тип и диапазон	Описание типа
Название произведения	NameOpus	Массив символов. Количество символов от 1 до 36	char NameOpus [37]
Автор	FIOAuthor	Массив символов. Количество символов от 1 до 24	char FIOAuthor [25]
Исполнитель	FIOCast	Массив символов. Количество символов от 1 до 24	char FIOCast [25]
Музыкальный стиль	MusicStyle	Массив символов. Количество символов от 1 до 20	char MusicStyle [21]
Год создания	DateFromCreate	Дата	TDate DateFromCreate

Очевидно, что информацию об объекте «фонограмма» целесообразно инкапсулировать в структуру данных, т.к. именно структура может иметь набор характеристик, представленных разными типами. Для описания шаблонов структур в C++ предусмотрен тип `struct` – структурный тип.

В соответствии с обозначениями, представленными в таблице 2 интерфейс структуры, описывающей отдельную фонограмму, может иметь вид, как это показано в листинге 1.

Листинг 1

Интерфейс структуры,
описывающей элемент объекта «Автор-исполнитель»

```
struct TFonoGramma          //фонограмма
{
   char NameOpus [37];     // Название произведения
   char  FIOAuthor [25];   // Автор
   char  FIOCast [25];     // Исполнитель
   char  MusicStyle [21];     // Музыкальный стиль
   TDate DateFromCreate;   // Год создания
};
```

9.3.2. Объект «Фонотека» – TFonoteka

«Фонотека» включает в себя множество объектов – фонограмм, где каждая фонограмма описывается структурным типом `TFonoGramma`. Число фонограмм заранее неизвестно и может изменяться в процессе решения задачи. Поэтому «Фонотеку» следует представить, как динамический массив структур типа `TFonoGramma`, память под который будет отводиться во время выполнения программы. Тогда описание динамического массива можно представить двумя характеристиками/переменными:

1. Указатель на элемент массива – фонограмму.
2. Количество фонограмм.

Информационная модель объекта «Фонотека» представлена в таблице 3.

Таблица 3

Пример описания информационной модели объекта «фонотека»

Имя объекта		TFonoteka	
Набор свойств и характеристика свойств			
Название свойства	Название в программе	Тип и диапазон	Описание типа
Указатель на элемент массива – фонограмму	ListFono	Указатель на структуру, описывающую отдельную фонограмму	`TFonoGramma` *ListFono
Количество фонограмм	FCount	Целое положительное. Верхний диапазон ограничен 32767	int FCount
Действия с объектом			
Предоставить пользователю список стилей музыкальных произведений, представленных в фонотеке. Предоставить пользователю фонограммы заданного стиля. Представить пользователю характеристику стиля с заданным названием стиля. Предоставить пользователю список авторов, создавших произведения, выполненных в заданном стиле. Предоставить пользователю список авторов-исполнителей произведений, выполненных в заданном стиле. Предоставить пользователю названия произведений, выполненных в заданном стиле_____ Предоставить пользователю информацию об авторе-исполнителе Предоставить пользователю список всех исполнителей Предоставить пользователю список произведений определенного исполнителя _____ **Предоставить пользователю сведения об авторе музыкального произведения** **Предоставить пользователю всю фонотеку**			

Если проанализировать действия, перечисленные в списке действий с объектом «Фонотека», можно увидеть, что часть этих действий (до первой черты) в прерогативе объекта-предка «Музыкальные стили», а часть (до второй черты) – в прерогативе объекта «Авторы-исполнители». Соответственно, эти действия должны быть перенесены в список действий этих объектов. А в собственном списке действий остаются только те, которые выделены жирным шрифтом.

9.3.3. Объект «Музыкальные стили»

Каждый элемент объекта «Музыкальные стили» (TMusicListStyles) характеризуется свойствами:

- Название стиля.
- Характеристика стиля.
- История стиля.

Информационная модель отдельного элемента этого объекта может быть такой, как это показано в таблице 4.

Таблица 4

Пример описания информационной модели отдельного элемента «Музыкальный стиль»

Имя элемента		TMusicStyle	
Набор свойств и характеристика свойств			
	Название в программе	Тип и диапазон	Описание типа
Название стиля	NameStyle	Массив символов. Количество символов от 1 до 20	char NameStyle [20]
Характеристика стиля	TemperStyle	Текстовая информация, представленная набором строк. Число строк ограничено возможностями типа.	TStringList* lstTemper
История стиля	HistoryStyle	Текстовая информация, представленная набором строк. Число строк ограничено возможностями типа.	TStringList* lstHistory

Интерфейс структуры, описывающей отдельный стиль, может иметь вид, как это показано в листинге 2.

Листинг 2

Интерфейс структуры,
описывающей элемент объекта «Музыкальный стиль»

```
struct TMusicStyle        //музыкальный стиль
{
   char  NameStyle [20];     // Название стиля
   TStringList* lstTemper;   // Характеристика стиля
   TStringList* lstHistory;     // История стиля
};
```

Поэтому множество музыкальных стилей, объединенных в объекте «Музыкальные стили» (TMusicListStyles) следует представить, как динамический массив структур типа TMusicStyle, который можно представить двумя характеристиками/переменными:

1. Указатель на элемент массива – музыкальный стиль.
2. Количество музыкальных стилей.

Информационная модель объекта «Музыкальные стили» представлена в таблице 5.

9.3.4. Объект «Авторы-исполнители»

На листинге 3 представлен интерфейс структуры, описывающий отдельный элемент объекта «Авторы-исполнители», а в таблице 6 информационная модель этого объекта.

Листинг 3

Интерфейс структуры,
описывающей элемент объекта «Автор-исполнитель»

```
struct TInfoCast  //  Об исполнителе
{
   char FIOCast [24]; //  ФИО исполнителя
   TStringList*   AboutCost; // данные об исполнителе
};
```

Таблица 5

Пример описания информационной модели объекта «Музыкальные стили»

Имя объекта		TMusicListStyles	
Набор свойств и характеристика свойств			
	Название в программе	Тип и диапазон	Описание типа
Указатель на элемент массива – музыкальный стиль	ListStyle	Указатель на структуру, описывающую отдельный музыкальный стиль	TMusicStyle *ListStyle

Окончание табл. 5

Количество музыкальных стилей	FCount	Целое положительное. Верхний диапазон ограничен 32767	int FCount
Действия с объектом			
Предоставить пользователю список стилей музыкальных произведений, представленных в фонотеке. Предоставить пользователю фонограммы заданного стиля. Представить пользователю характеристику стиля с заданным названием стиля. Предоставить пользователю список авторов, создавших произведения, выполненных в заданном стиле. Предоставить пользователю список авторов-исполнителей произведений, выполненных в заданном стиле. Предоставить пользователю названия произведений, выполненных в заданном стиле.			

Таблица 6

Пример информационной модели объекта «Авторы-исполнители»

Имя объекта		CastsOpus	
Набор свойств и характеристика свойств			
	Название в программе	Тип и диапазон	Описание типа
Указатель на элемент массива – автора исполнителя	ListCast	Указатель на структуру, описывающую отдельного автора-исполнителя	`TInfoCast` *ListCast
Количество авторов-исполнителей	FCount	Целое положительное. Верхний диапазон ограничен 32767	int FCount
Действия с объектом			
Предоставить пользователю информацию об авторе-исполнителе Предоставить пользователю список всех исполнителей Предоставить пользователю список произведений определенного исполнителя			

Принятая структура расположения исходных данных в файлах, представленная на рис. 6, *не требует отдельного описания структуры данных об авторе музыкальных произведений*, поскольку информацию можно найти с помощью чтения файлов на основании поля `FIOAuthor`, представленного в описании фонограммы.

10. Разработка классов

Задачу разработки классов можно решать по-разному. Здесь каждый программист может проявлять свою индивидуальность, иметь свои предпочтения. Для поставленной задачи предлагается к рассмотрению три варианта решения:

1. Использование класса-наследника. Разрабатываются три класса «Фонотека», «Авторы-исполнители», «Музыкальные стили». Связываются эти классы таким образом, чтобы класс «Фонотека» являлся потомком от классов «Авторы-исполнители» и «Музыкальные стили».
2. Использование классовых переменных в качестве полей класса. Разрабатываются три класса «Фонотека», «Авторы-исполнители», «Музыкальные стили». Связываются эти классы таким образом, чтобы экземпляры классов «Авторы-исполнители» и «Музыкальные стили» были включены в поля класса «Фонотека».
3. Использование списка строк с привязанными структурами в качестве полей класса. Разрабатывается один класс «Фонотека». Информационные модели объектов «Авторы-исполнители», «Музыкальные стили» представляются в виде списка строк (исполнителей и стилей), где к каждой строке привязана своя структура данных. И этими списками дополняется характеристика объекта «Фонотека».

Следует отметить, что в перечне объектов отдельно не фигурирует объект «Фонограмма», т.к. он является составной частью объекта «Фонотека».

Создание классов начинают с разработки их интерфейсов. После чего приступают к разработке алгоритмов методов класса и представлению алгоритмов в виде программных кодов.

10.1. Использование класса-наследника

В разработке будут участвовать три класса. Предположим, что связь между классами организована по схеме, как это показано на рис. 7. А шаблон интерфейса класса представлен на рис. 8. Как видно из схемы на рис. 8, в список полей класса помещают набор характеристик объекта, взятый из информационной модели объекта, а в список методов – помимо стандартных (конструкторов и деструктора) помещают действия, выделенные на этапах разработки концептуальной схемы интерфейса.

Рис. 7. Схема наследования классов

10.2. Разработка интерфейсов классов

Напомним, что процесс объединения переменных и методов, в результате которого и получается класс, называется *инкапсуляцией*. При разработке классов имеет значения порядок, в котором создаются классы. В первую очередь следует создавать классы-предки, а затем уже классы-наследники. По этой рекомендации разработку будем вести в последовательности: класс для представления объекта «Музыкальные стили», класс для объекта «Авторы-исполнители», класс для объекта «Фонотека».

Описание классов разделено на две части – *интерфейсную* («заголовочную») и *описательную*. В интерфейсной части располагается описание заголовка класса, в котором указывается название класса, по которому будут создаваться его экземпляры, описания свойств и заголовков (прототипов) методов. Описание самих методов располагается обычно в файле реализаций.

Здесь квадратные скобки «**[**…**]**» следует воспринимать как обозначение, что данный фрагмент кода программы может отсутствовать (а не как часть кода).

Разработку интерфейсов классов рекомендуется проводить в два этапа. На начальном этапе разработки интерфейсов классов, функции – методы класса (действия объектов) можно просто обозначить (в комментариях), а спецификации и разработку методов произвести на последующих шагах детализации.

```
class <Имя класса>
{
    [
    private:
      // закрытые поля данных
      // закрытые конструкторы
      // закрытый деструктор
      // закрытые функции – методы
    ]
    [
    public:
      //открытые поля данных
      // открытые конструкторы
      // открытый деструктор
      // открытые функции – методы
    ]
    [
    protected:
      // защищенные поля данных
      // защищенные конструкторы
      // защищенный деструктор
      // защищенные функции-методы
    ]
};
```

Рис. 8. Шаблон интерфейса класса

Для класса, описывающего объект «Авторы-исполнители», введем наименование `CastsOpus`. Поля класса `CastsOpus` формируются на базе информационной модели объекта, которая представлена в таблице 6. Поскольку класс является предком и ему предстоит передавать данные своим потомкам (классу, описывающему объект «Фонотека»), описание полей класса следует пометить спецификатором `protected`. Такое решение также позволяет выявить ошибки типа нарушения прав доступа к данным еще на этапе компиляции программы. Интерфейс класса может иметь вид, как это показано на листинге 4. Как рекомендовано – здесь методы просто обозначены в комментариях.

Листинг 4

Интерфейс класса объекта «Авторы-исполнители»

```
class TCastsOpus // «Авторы-исполнители»
{
  protected:
   InfoCast *ListCast;
   int FCount;
   public:
       // конструктор
        // деструктор
        // получить информацию об авторе-исполнителе
// получить список всех исполнителей
// получить список произведений определенного исполнителя
};
```

Аналогично создается класс, назовем его `TMusicListStyles`, для объекта «Музыкальные стили» на базе таблицы 5. Интерфейс класса может иметь вид, как это показано на листинге 5.

Листинг 5

Интерфейс класса, описывающего объекта «Музыкальные стили»

```
class TMusicListStyles  //  «Музыкальные стили»
{
  protected:
   TMusicStyle *ListStyle;
   int FCount;
   public:
       // конструктор
        // деструктор
     //Предоставить список стилей, представленных в фонотеке.
     //Предоставить фонограммы заданного стиля.
     //Представить характеристику стиля по названию.
     //Предоставить список авторов с произведениями в заданном стиле.
```

```
     //Предоставить список авторов-исполнителей в заданном
стиле.
     //Предоставить названия произведений заданного стиля.
     // получить сведения об истории зарождения заданного
стиля
};
```

После того как разработаны интерфейсы классов-предков, можно приступить к разработке интерфейса класса-потомка.

При разработке класса «Фонотека» следует помнить, что класс-наследник приобретает все свойства и методы родительского класса (в данном случае их два). Он имеет доступ к любому члену родительского класса (полю и методу), за исключением тех, которые описаны с областью видимости `private` (в нашем случае их нет).

Разработка класса «Фонотека» ведется на базе информационной модели из таблицы 3. Класс формируется аналогично классу-предку, за исключением того факта, что в заголовке класса следует перечислить имена классов-предков, указав для них спецификатор `public`, как это показано на листинге 6.

Листинг 6

Интерфейс структуры,
описывающей элемент объекта «Музыкальные стили»

```
class Fonoteka: public CastsOpus, public MusicListStyles //
«Фонотека»
{
   TFonoGramma *ListFono;
   int FCount;
   public:
     // конструктор
      // деструктор
     //получить сведения об авторе музыкального произведения
      //распечатать всю фонотеку
};
```

Завершающим этапом в разработке интерфейсов классов является разработка спецификаций и подготовка прототипов методов для тех действий, которые обозначены в строках-комментариях на листингах 4–6.

10.3. Разработка спецификаций методов классов

Спецификация программы – точная и полная формулировка задачи, содержащая информацию, необходимую для построения алгоритма (программы) решения этой задачи

На этом этапе проектировать методы класса следует по принципу «черного ящика»: объект реагирует на входные параметры, результаты

являются выходными параметрами, а фактическая реализация остается пока неизвестной. Причем, в соответствии со спецификой класса, *входные и выходные параметры метода класса являются либо полями класса, либо параметрами метода.*

Методы можно разделить на две группы:

- стандартные методы – конструкторы и деструкторы;
- другие методы – те, которые представлены в информационной модели, а также те, которые обеспечивают логику задачи, в том числе методы, связанные с обработкой объекта, и методы для выполнения вспомогательных операций, выявленные на этапе анализа или необходимые для реализации отдельных методов.

10.3.1. Разработка спецификаций конструкторов

Очевидно, что начинать надо с разработки конструкторов, т.к. они играют важнейшую роль при создании экземпляров[3] классов, от имени которых будут производиться все действия с объектом.

Конструктор класса – это специализированный метод класса, который автоматически вызываются при описании объекта. Основное назначение конструктора класса заключается в инициализации данных класса, а также выполнении подготовительных операций, необходимых для работы с экземпляром класса.

Конструкторы отличаются от остальных методов класса следующими свойствами:

1. Имя конструктора должно совпадать с именем класса. Благодаря возможности перегрузки функций, у класса может быть несколько конструкторов с разными списками параметров.
2. Конструктор не должен возвращать никакого значения. В заголовке конструктора не должно присутствовать даже ключевое слово `void` (Опознаватель конструктора – нет возвращаемого типа).
3. Конструктор должен быть объявлен в разделе `public`, иначе мы не получим доступ к нему извне.
4. Из конструктора можно вызывать любые методы класса, нужно только помнить, что на момент вызова конструктора данные класса не инициализированы.
5. Если в классе не объявлен конструктор, то компилятор C++ создает его по умолчанию для этого класса.
6. Конструктор, который имеет один параметр с типом ссылки на этот же класс, называется конструктором копирования.
7. Конструктор можно наследовать.

[3] Переменная, **описанная** с типом класса, называется экземпляром класса

Конструктор призван решать две основные задачи:

1. Выделять память под структуры данных, необходимые для поддержания жизнедеятельности экземпляра класса, то есть выполнить инициализацию экземпляра; заполнить поля класса нулевыми или иными значениями; установить нулевые ссылки (значение NULL) для свойств-указателей; а также устанавливает нулевые длины строкам, если они прописаны в полях класса.
2. Вернуть ссылку на экземпляр класса, которую можно сохранить в переменной для доступа к свойствам и методам объекта, а также для его последующего разрушения.

Для того чтобы создать конструкторы для описанных классов, нужно иметь представление о том, откуда и как может поступать информация в поля классов для создания экземпляров.

В общем случае функционирование любого приложения (программы) можно рассматривать, как обработку некоторого входного потока данных в выходной набор данных. Данные поступают от входа, преобразуются по правилам решения задачи (в том числе передаются полям экземпляра класса), и в преобразованном виде направляются к выходу внешним пользователям.

Движение информации для поставленной задачи показано на рис. 9.

Из схемы видно, что для создания экземпляра класса `TMusicListStyles` требуются данные из файла «Музыкальные стили.txt». На основании этого файла конструктор должен сформировать поля класса в соответствии с возложенной на него задачей. Тогда прототип конструктора можно представить в виде:

```
TMusicListStyles(char* NamFile);
```

Здесь параметр `NamFile` представляет собой имя файла, содержимое которого необходимо для построения экземпляра класса `TMusicListStyles`. Аналогично можно написать прототип конструктора для класса `TCastsOpus`:

```
TCastsOpus (char* NamF);
```

Здесь параметр `NamF` представляет собой имя файла, содержимое которого необходимо для построения экземпляра класса `TCastsOpus`.

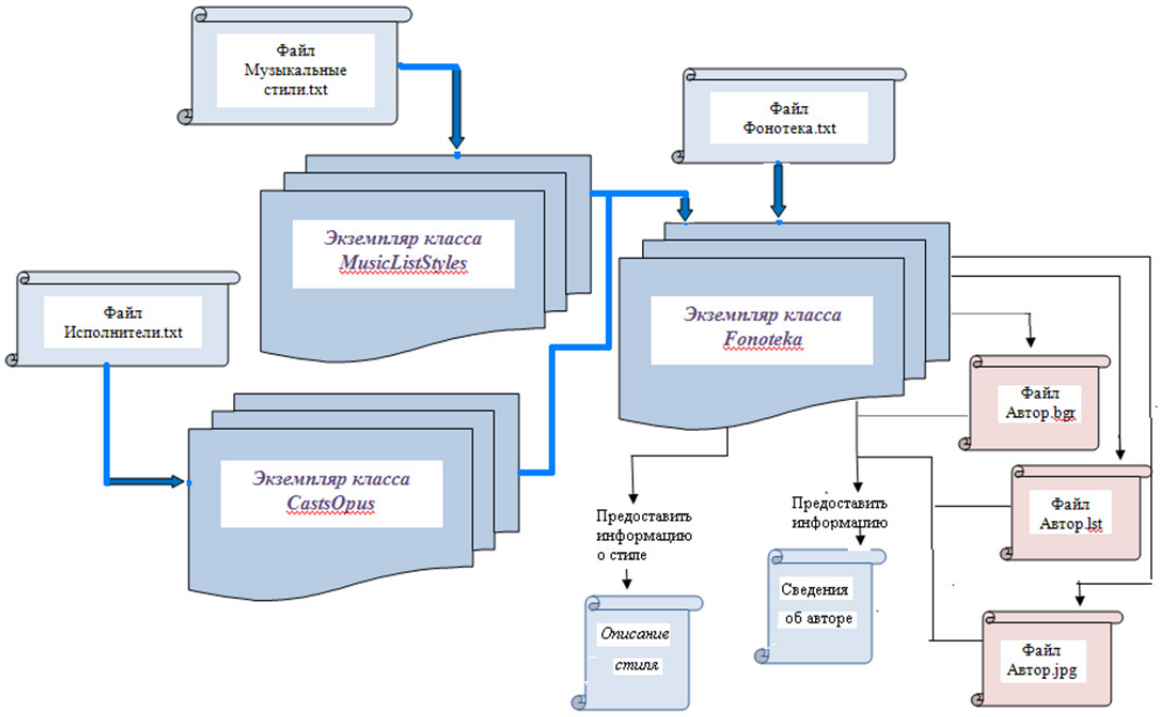


Рис. 9. Фрагмент схемы движения информации для иерархии классов

Сложнее дело обстоит с разработкой конструктора класса `TFonoteka`. Этот класс является потомком двух классов `TMusicListStyles` и `TCastsOpus`, и для создания его экземпляра необходимо подготовить экземпляры классов-предков `TMusicListStyles` и `TCastsOpus`, а для этого им потребуются передать файлы. А также для построения собственных полей ему необходимы данные из файла «Фонотека.txt». Т.е. в прототипе конструктора должны присутствовать три параметра:

```
Fonoteka (char* NamFile1, char* NamFile2, char* NamFile3);
```

Здесь параметр `NamFile1` представляет собой имя файла, содержимое которого передается экземпляру класса `TMusicListStyles`. Параметр `NamFile2` представляет собой имя файла, содержимое которого передается экземпляру класса `TCastsOpus`, а параметр `NamFile3` представляет собой имя файла, содержимое которого необходимо для построения собственных полей класса.

10.3.2. Разработка спецификаций деструкторов

Деструктор класса – это специализированный метод класса, который автоматически вызываются при выходе из блока, в котором был описан объект с типом класса. Деструктор является противоположно-

стью конструктора. Его задача – выполнить операции, необходимые при завершении работ с экземпляром класса.

Деструкторы должны подчиняться следующим правилам:

1. Имя деструктора должно совпадать с именем класса, с добавлением символа «~» перед именем.
2. Деструктор должен быть объявлен в разделе public.
3. Деструктор не должен иметь параметров.
4. У класса может быть только один деструктор.
5. Деструктор нельзя наследовать.

Экземпляр класса естественным образом занимает место в памяти, которая необходима для хранения, как свойств этого экземпляра, так и дополнительной информации, например, таблицы виртуальных методов. В функцию деструктора входит задача *разрушения объекта* и *освобождения памяти*, которая выделена для него в конструкторе, а также *разрушения связей* с другими элементами программы, если они были предусмотрены.

В связи с вышеизложенным прототипы конструкторов в интерфейсах классов должны быть представлены в виде:

`~TMusicListStyles();` – для класса `TMusicListStyles`

`~TCastsOpus ();` – для класса `TCastsOpus`

`~ TFonoteka ();` – для класса `TFonoteka`

10.3.3. Разработка спецификаций методов класса

Как разрабатывать спецификации методов класса, покажем на примере методов класса `TMusicListStyles`[4].

На рис. 9 имеется блок «Описание стиля», которому сопутствуют слова «Предоставить информацию о стиле». Задача этого блока связана с классом `TMusicListStyles` и должна обеспечиваться одним из методов этого класса. В списке задач, связанных с объектом «Музыкальные стили», представлены следующие задачи:

1. Предоставить пользователю список стилей музыкальных произведений, представленных в фонотеке.
2. Предоставить пользователю фонограммы заданного стиля.
3. Представить характеристику стиля с заданным названием стиля.
4. Предоставить пользователю список авторов, создавших произведения, выполненных в заданном стиле.
5. Предоставить пользователю список авторов-исполнителей произведений, выполненных в заданном стиле.

[4] Методы классов CastsOpus и Fonoteka формируются по образу и подобию

6. Предоставить пользователю названия произведений, выполненных в заданном стиле.
7. Предоставить пользователю сведения об истории зарождения заданного стиля.

Для обеспечения каждого из этих действий можно предусмотреть отдельный метод. При разработке спецификаций к методам (как и при разработке описаний самих методов) всегда исходят из того факта, что в соответствии со спецификой класса, *входными параметрами любого метода класса являются поля этого класса* (они формируются на этапе создания экземпляра класса) и *параметры метода*[5].

Например, прототип метода «Представить характеристику стиля с заданным названием стиля» может иметь вид:

```
void AboutStyle (char* NameStyle, TMemo* aMem);
```

Здесь параметр `NameStyle` используется для задания стиля музыкального произведения, а параметр aMem – определяет многострочный редактор, куда направляется выходная информация.

Очевидно, что названия стилей и информация о них формируются на этапе создания экземпляра класса и являются входными данными для всех методов класса. Недостающая информация передается через параметры метода. В данном случае через `NameStyle` и `aMem`.

В принципе для первых 6 действий можно разработать один метод, в котором предусмотреть три параметра:

- параметр, выполняющий роль критерия для выбора из фонотеки типа необходимой информации;
- параметр, задающий характеристику требуемой информации;
- параметр, сообщающий, куда направляется выходная информация.

В данном случае в качестве значений критерия для выбора типа информации можно использовать: список стилей; заданный стиль. В качестве значений параметра, задающего характеристику требуемой информации можно использовать: список фонограмм; список авторов произведений; список авторов исполнителей; список названий произведений. Список значений для параметра, сообщающего, куда направляется выходная информация: в файл; в сетку; в `Memo`, в `ListBox`. Но этот метод достаточно сложен для первого знакомства с программированием. Потому рекомендация – отдельный метод для каждого действия.

[5] Входными могут быть глобальные переменные. В данном случае они не рассматриваются.

На листинге 7 представлен возможный вариант частично откорректированного интерфейса класса `MusicListStyles` (листинг 5) с учетом добавления некоторых прототипов методов класса.

Листинг 7

Интерфейс класса, описывающего объект «Музыкальные стили»

```
class MusicListStyles  //«Музыкальные стили»
{
  protected:
   MusicStyle *ListStyle;
   int FCount;
   public:
       MusicListStyles(char* NamFile);   // конструктор
        ~MusicListStyles(); // деструктор
        //Предоставить список стилей, представленных в фоно-
теке.
        // aLst – имя списка для отображения выходной инфор-
мации
      void GetListStyle (TListBox* aLst);

     //Предоставить фонограммы заданного стиля.
     // aMem – имя редактора для отображения выходной инфор-
мации
     // NameStyle – наименование стиля для отбора фонограмм

          //вернуть порядковый номер заданного стиля в спис-
ке стилей
        int GetNumberStyle(char*);

        //поместить в компонент aMem характеристику стиля по
его номеру
       void PutMemoTempere(int Num,TMemo* aMem);

     //Представить характеристику стиля по названию.
     //Предоставить список авторов с произведениями в задан-
ном стиле.
     //Предоставить список авторов-исполнителей в заданном
стиле.
     //Предоставить названия произведений заданного стиля.
     // Получить сведения об истории зарождения заданного
стиля
};
```

10.3.4. Разработка алгоритмов методов класса

Разработку алгоритмов методов класса рекомендуется представлять в следующей последовательности:

1. разработка алгоритмов конструкторов;

2. разработка алгоритмов деструкторов;
3. разработка алгоритмов методов класса.

Описание отдельных алгоритмов предлагается представлять по схеме: прототип; спецификация; описание локальных переменных; словесное описание сути алгоритма; пошаговое представление алгоритма или представление в виде блок-схемы.

Ниже приведен пример описания конструктора класса `TMusicListStyles`. Конструктор класса `MusicListStyles` с одним параметром – «имя файла», а прототип имеет вид:

```
MusicListStyles(UnicodeString NamF);//прототип
```

Конструктор формирует экземпляр класса на основании той информации, которая содержится внутри файла, имя которого передается через параметр конструктора `NamF`. Следовательно, прежде чем воспользоваться услугами такого конструктора, следует создать файл и подготовить его соответствующим образом. В данном случае – это текстовый файл с именем «Музыкальные стили.txt», расположенный в той же папке, что и все файлы фонотеки. На рис. 9 представлена структура этого файла, и в соответствии с предложенным описанием, для каждого музыкального стиля отводится 9 строк:

- название стиля – 1 строка
- характеристика стиля – 5 строк
- история стиля – 3 строки.

Для решения задачи используются следующие вспомогательные/локальные переменные:

Way – строковая переменная для хранения полного имени файла;

aLst – вспомогательный список для временного хранения содержимого файла;

Temp – количество строк в файле;

k – текущий номер читаемой структуры;

i – текущий индекс строки списка;

Buf – вспомогательная переменная для хранения строки символов;

Ptr – вспомогательная переменная для хранения строки типа AnsiString;

и т.д.

Суть алгоритма: для простоты содержимое файла «Музыкальные стили.txt» переносится в список строк. Задаются поля класса, и отводится память под структуры класса. Затем строки из списка строк копируются в поля структур (1+5+3), а именно, в поле наименования стиля, в поле – характеристика стиля и в поле – история стиля.

Алгоритм конструктора с параметром «имя файла» можно описать шагами:

а) в переменную Way копируется путь к файлу «Фонотека.txt»;
б) путь складывается с именем «Музыкальные стили.txt» (полное имя файла);
в) объявляется список aLst;
г) создается список aLst;
д) содержимое файла «Музыкальные стили.txt» загружается в список aLst;
е) определяется число строк в списке и заносится в переменную Temp;
ж) определяется количество стилей, характеризуемых с помощью информации из файла – (число строк)/9 и инициализируется поле класса `FCount;`
з) отводится память под массив структур, описанный в полях класса;
и) устанавливается начальное значение k;
к) выполняется цикл по условию – пока в списке aLst есть строки. В цикле:
   a. очередная строка из aLst переносится в Buf, а затем копируется в поле `NameStyle` структуры `ListStyle[k];`
   b. следующие 5 строк (в цикле) списка aLst копируются в поле `lstTemper` структуры `ListStyle[k];`
   c. следующие 3 строки (в цикле) списка aLst копируются в поле `lstHistory` структуры `ListStyle[k];`
   d. изменяется номер структуры на 1;
л) уничтожается список aLst.

Описание конструктора представлено на листинге 8.

Листинг 8

Описание конструктора класса `TMusicListStyles`

```
TMusicListStyles::TMusicListStyles(UnicodeString NamF)
{
   AnsiString Way = ExtractFileDir (NamF);// выделяется путь
   Way = Way+"\\Музыкальные стили.txt";// полное имя файла
   TStringList* aLst;// объявляется список aLst;
   aLst = new TStringList;// создается список
   aLst->LoadFromFile(Way); //содержимое файла копируется в
список
   int Temp = aLst->Count; //число строк
   FCount = Temp/9; // число структур и инициализация поля
класса FCount
   ListStyle =  new MusicStyle[FCount];//память под структу-
ры
```

```
   int k=0;//начальное значения счетчика структур
   for(int i=0; i<Temp&&k<FCount; i=i+9)//пока есть строки
      { char Buf [24];//вспомогательная
        AnsiString Ptr = aLst->Strings[i];//прочитать строку
типа AnsiString
        strcpy(Buf, Ptr.c_str());//скопировать в Buf
        strcpy(ListStyle[k].NameStyle,   Buf);//сформировать
поле NameStyle
        for(int   j=i+1;   j<i+5;   j++)//сформировать   поле
lstTemper
            {ListStyle[k].lstTemper->Add(aLst->Strings[j]);
            }
        for(int  j=i+5;  j<i+9;  j++)//  сформировать  поле
lstHistory
            {ListStyle[k].lstHistory->Add(aLst->Strings[j]);
            }
        k++; // изменить номер структуры
      }
   delete aLst;//удалить
}
```

Описание конструктора класса `CastsOpus` можно построить по образу и подобию.

Конструктор класса `TFonoteka` с одним параметром – «имя файла»

Схема алгоритма конструктора класса `TFonoteka` – наследника классов `TMusicListStyles` и `TCastsOpus` может быть построена аналогичным образом за исключением тех особенностей, которые связаны с написанием заголовка. В заголовке нужно указать, какие данные из списка параметров следует передать конструкторам классов предков. Описание конструктора представлено на листинге 9.

Листинг 9

Описание конструктора класса `TFonoteka`

```
TFonoteka::TFonoteka(UnicodeString Nam-
File):CastsOpus(NamFile), MusicListStyles(NamFile)
     {    TStringList* aLst;//вспомогательный список
          aLst = new TStringList; // создать
          aLst->LoadFromFile(NamFile);//загрузить  из  файла
строки в список
          int Temp = aLst->Count;//количество строк
          FCount = Temp;//инициализировать поле класса
          ListFono = new FonoGramma [Temp];// отвести память
          for(int i=0; i<Temp; i++)//цикл по числу строк
            {
                 char St[100];
                 AnsiString T = aLst->Strings[i];
```

```
                strcpy(St, T.c_str());
                St>>ListFono[i];//выполнить чтение из стро-
ки в структуру
            }
          delete aLst;
     }
```

Следует обратить внимание, что для чтения из строки в структуру используется операция ‘>>’, которая включена в интерфейс структуры `TFonoGramma`, как ее «друг» в виде, представленном на листинге 10.

Листинг 10

Описание дружественной оператор-функции в структуре TFonoGramma

```
friend void operator >>(char Ptr[100], TFonoGramma& Wrk)
 {
  char Pt[12];
  sscanf(Ptr,"%s  %s  %s  %s",  &Wrk.NameOpus,&Wrk.FIOAuthor,
&Wrk.FIOCast, &Pt);
  AnsiString PP = Pt;
  Wrk.DateFromCreate = StrToDate(PP);
};
```

Метод класса `GetNumberStyle`

Метод класса `GetNumberStyle`, предназначенный для поиска номера стиля в списке музыкальных стилей имеет прототип:

```
int GetNumberStyle(char* NameForTest); //прототип
```

Здесь `NameForTest` – наименование стиля, номер которого возвращается методом в качестве результата. Если такой стиль отсутствует в списке стилей функция возвращает ‘–1’.

Предлагается два алгоритма для решения задачи.

Вариант 1.

Алгоритм решения заключается в следующем: создается вспомогательный список из наименований стилей. С помощью функции/метода списка IndexOf находится порядковый номер стиля.

Пошаговая детализация:

1. Объявить переменную типа «список».
2. Создать список.
3. Выполнить цикл «по числу стилей». В цикле из структур ListStyle[i]брать поле NameStyle и добавлять к списку.
4. Вызвать функцию IndexOf.
5. Удалить вспомогательный список.
6. Вернуть результат.

Описание метода представлено на листинге 11.

Листинг 11

Описание метода `GetNumberStyle` *класса* `MusicListStyles`

```
int MusicListStyles::GetNumberStyle(char* NameForTest)
{  TStringList* iLst;
   iLst = new TStringList;
   for(int i=0; i<FCount; i++)
   {
        iLst->Add(ListStyle[i].NameStyle);
   }
   AnsiString Ptr = NameForTest;
   int Temp = iLst->IndexOf(Ptr);
   delete iLst;
   return Temp;
}
```

Для поиска номера строки с заданным значением используется функция
`IndexOf`, имеющая прототип:

```
int IndexOf(const AnsiString S); //прототип
```

Которая возвращает индекс указанной строки S. Если такой строки нет в списке, возвращается ‘–1’.

Вариант 2.

Алгоритм сводится к перебору структур списка структур `TListStyle`. При переборе поле структур `NameStyle` сравнивается со значением искомого стиля `NameForTest`, указанного в списке параметров. Как только встречается заданный стиль, просмотр прерывается и найденный номер возвращается в качестве результата. Если не будет найден – возвращает ‘–1’. Описание метода представлено на листинге 12.

Листинг 12

Описание метода `GetNumberStyle` *класса* `TMusicListStyles`

```
int TMusicListStyles::GetNumberStyle1(char* NameForTest)
{
   int Temp = -1;
   for(int i=0; i<FCount ; i++)
   {
        if(strcmp(NameForTest,ListStyle[i].NameStyle)==0)
            return i;
   }
   return Temp;
}
```

Метод класса PutMemoTempere

Метод класса PutMemoTempere, предназначенный для поиска информации по порядковому номеру стиля, характеризующей музыкальный стиль имеет прототип:

```
void PutMemoTempere(int Num,TMemo* aMem);
                    //прототип
```

Здесь Num – номер стиля в списке; aMem – определяет компонент для отображения информации. Описание метода представлено на листинге 13.

Листинг 13

Описание метода PutMemoTempere класса MusicListStyles

```
void MusicListStyles::PutMemoTempere(int Num,TMemo* aMem)
 {
     aMem->Lines->Assign(ListStyle[Num].lstTemper);
 }
```

Метод класса PutToGrid

Метод класса PutToGrid, предназначенный для отображения фонотеки в сетке имеет прототип:

```
void TFonoteka::PutToGrid(TStringGrid* aGrid);
                    //прототип
```

Метод имеет один параметр aGrid, через который передается имя сетки. Алгоритм можно описать следующими шагами:

1. устанавливаются размеры сетки aGrid;
2. в цикле, по номеру элемента структуры, в ячейки строки (в колонки) сетки заносятся последовательно поля структуры.

Описание метода представлено на листинге 14.

Листинг 14

Описание метода PutToGrid класса Fonoteka

```
void Fonoteka::PutToGrid(TStringGrid* aGrid)
{
     aGrid->RowCount = FCount+1;
     aGrid->ColCount = 5;
     for(int i=0; i<FCount; i++)
       {
            aGrid->Cells[1][i+1] = ListFono[i].NameOpus;
            aGrid->Cells[2][i+1] = ListFono[i].FIOAuthor;
            aGrid->Cells[3][i+1] = ListFono[i].FIOCast;
            AnsiString          T          =          Date-
ToStr(ListFono[i].DateFromCreate);
```

```
            aGrid->Cells[4][i+1] = T;
        }
}
```

Метод класса PutPut

Метод класса PutPut, предназначенный для отображения информации о стилях и исполнителях имеет прототип:

```
void PutPut(TMemo* aMem, TMemo* bMem);//прототип
```

Здесь параметр aMem – определяет визуальный компонент, в который заносится характеристика и история всех стилей; bMem – определяет визуальный компонент, в который заносится информация о всех авторах исполнителях.

Алгоритм сводится к выполнению циклов, в которых поочередно считываются строки и заносятся в компоненты aMem и bMem. Описание метода представлено на листинге 15.

Листинг 15

Описание метода PutPut класса TFonoteka

```
void TFonoteka::PutPut(TMemo* aMem, TMemo* bMem)
{
     int Temp = MusicListStyles:: FCount;
     for(int i=0; i<Temp; i++)
     {
       aMem                                              ->Lines-
>AddStrings(MusicListStyles::ListStyle[i].lstTemper);
       aMem                                              ->Lines-
>AddStrings(MusicListStyles::ListStyle[i].lstHistory);
     }
     Temp = CastsOpus:: FCount;
     for(int i=0; i<Temp; i++)
     {
       bMem                                              ->Lines-
>AddStrings(CastsOpus::ListCast[i].AboutCast);
     }
}
```

10.4. Использование классовых переменных в качестве полей класса

В разработке также как и в предыдущем случае будут участвовать три класса. Но связь между классами организована в виде, как это показано на рис. 10. Здесь объект «Фонотека» берет информацию о музыкальных стилях и авторах-исполнителях «на месте» (из своих полей), пользуясь экземплярами классов TMusicListStyles и TCastsOpus.

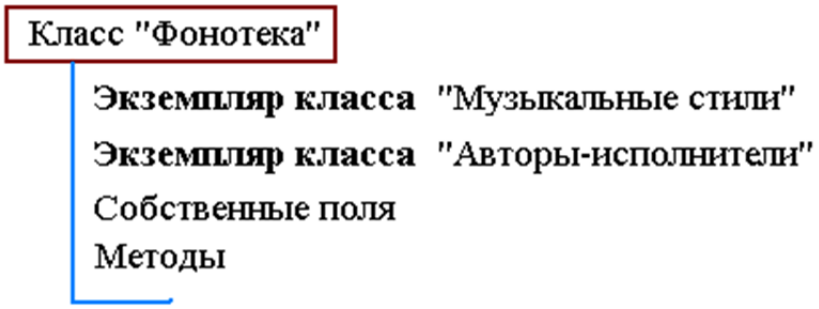


Рис. 10. Схема включения классовых переменных в поля класса

В соответствии со схемой, представленной на рис. 10, интерфейс класса (назовем его `TFonoIn`) может быть таким, как это показано на листинге 16. Конечно же, в этом случае все действия, связанные с объектами, должны быть включены в список действий класса `TFonoIn`. На листинге они помечены комментариями.

Листинг 16

Представление интерфейса класса `TFonoIn`

```
class TFonoIn
{
   FonoGramma *ListFono; //указатель на фонограмму
   int FCount; // количество фонограмм
   MusicListStyles      *MLStyle;//      экземпляр      класса
MusicListStyles
   CastsOpus *COpus; // экземпляр класса COpus
        public:
           FonoIn(UnicodeString NamFile);//конструктор
           void  PutToGrid(TStringGrid*  aGrid);//напечатать
фонотеку
              //Выдать характеристику стиля iStyle в aMem
            void PutNumberStyle(char* iStyle, TMemo* aMem);

           // Здесь прототипы остальных методов класса
};
```

Конструктор класса `TFonoIn` должен выполнять все действия, связанные с созданием полей `ListFono`, но в его задачу также входит создание экземпляров классов `TMLStyle` и `TCOpus`. Код конструктора представлен на листинге 17.

Листинг 17

Представление интерфейса класса `TFonoIn`

```
TFonoIn::TFonoIn(UnicodeString NamFile)
     {
          TStringList* aLst;
          aLst = new TStringList;
```

```
        aLst->LoadFromFile(NamFile);
        int Temp = aLst->Count;
        FCount = Temp;
        ListFono = new FonoGramma [Temp];
        for(int i=0; i<Temp; i++)
          {
               char St[100];
               AnsiString T = aLst->Strings[i];
               strcpy(St, T.c_str());
               St>>ListFono[i];
          }
        delete aLst;
        MLStyle = new MusicListStyles (NamFile);
        COpus = new CastsOpus (NamFile);
    }
```

Метод класса `PutNumberStyle`

Метод класса `PutNumberStyle`, предназначенный для отображения информации о заданном музыкальном стиле, имеет прототип:

```
void PutNumberStyle(char* iStyle, TMemo*
                    aMem);//прототип
```

Здесь `iStyle` – наименование стиля; `aMem` – визуальный компонент для отображения информации.

Алгоритм сводится к вызовам методов класса `TMusicListStyles` через экземпляр `MLStyle`. Описание метода представлено на листинге 18.

Листинг 18

Описание метода `PutNumberStyle` класса `TFonoIn`

```
void FonoIn::PutNumberStyle(char* iStyle, TMemo* aMem)
{    int Temp = MLStyle->GetNumberStyle1(iStyle);//найти но-
мер в списке
     MLStyle->PutMemoTempere(Temp,aMem);//взять информацию
}
```

10.5. Использование в качестве полей класса список строк с привязанными структурами

В таком случае в разработке только один класс в виде, как это показано на рис. 11. Здесь объект «Фонотека» берет информацию о музыкальных стилях и авторах-исполнителях «на месте» (из своих полей), пользуясь списками структур, хранящих информацию о стилях и исполнителях.

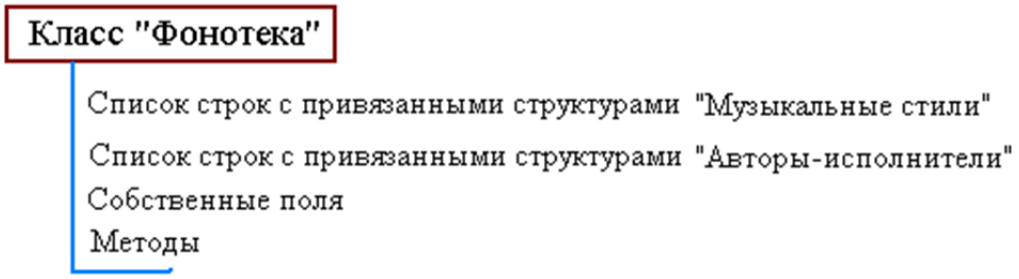


Рис. 11. Схема интерфейса класса «Фонотека» с «привязанными» структурами

Для реализации схемы показанной на рис. 11 предлагается изменить информационные модели объектов «Музыкальные стили» и «Авторы-исполнители». А именно, информационную модель объекта «Музыкальные стили», представленную в табл. 5, представить с использованием списка строк, как это показано в табл. 7. А информационную модель объекта «Авторы-исполнители», представленную в табл. 6, представить с использованием списка строк, как это показано в табл. 8.

Таблица 7

Пример описания информационной модели объекта «Музыкальные стили» с использованием списка строк

Имя объекта		MusicListStyles	
Набор свойств и характеристика свойств			
	Название в программе	Тип и диапазон	Описание типа
Указатель на список строк, описывающих музыкальные стили	MLStyle	В каждой строке списка хранится название музыкального стиля, и к ней привязывается структура типа MusicStyle с информацией, описывающей музыкальный стиль заданного названия	TStringList *MLStyle
Действия с объектом			

Таблица 8

Пример информационной модели объекта «Авторы-исполнители» с использованием списка строк

Имя объекта		CastsOpus	
Набор свойств и характеристика свойств			
	Название в программе	Тип и диапазон	Описание типа
Указатель на список строк, описывающих авторов исполнителей	COpus	В каждой строке списка хранится ФИО исполнителя, и к ней привязывается структура типа InfoCast с информацией, описывающей автора-исполнителя	TStringList *COpus
Действия с объектом			

Тогда интерфейс класса (назовем его FonoZZ) можно представить, как это показано на листинге 19.

Листинг 19

Интерфейс класса «Фонотека» по схеме, представленной на рис. 11

```
class TFonoZZ
{
   FonoGramma *ListFono; // массив фонограмм
   int FCount;  // число фонограмм
   TStringList *MLStyle;// список «Музыкальные стили»
   TStringList *COpus; // список «Авторы-исполнители»
     public:
      TFonoZZ(UnicodeString NamFile); //конструктор
      void PutToGrid(TStringGrid* aGrid); //поместить данные
в сетку

        //создать список «Авторы-исполнители»
      void DoCastsOpus (UnicodeString NamF);

        //создать список «Музыкальные стили»
      void DoMusicListStyles(UnicodeString NamF);

         // получить информацию о стиле по его наименованию
      void GetStyle(AnsiString NSt,TMemo* aMem);

     // другие методы
};
```

Для реализации конструктора создадим два отдельных метода: метод Do1MusicListStyles – предназначенный для формирования списка «Музыкальные стили»; метод DoCastsOpus – для формирования списка «Исполнители». Эти методы можно построить по образу и подобию. Потому детально разберем только один из них.

Метод класса Do1MusicListStyles,

Метод класса Do1MusicListStyles, предназначенный для формирования списка строк о музыкальных стилей с привязанными структурами, имеет прототип:

```
void Do1MusicListStyles(UnicodeString NamF); //прототип
```

Здесь NamF – имя файла с информацией о музыкальных стилях.

Для реализации метода «поправим» структуру файла «Музыкальные стили.txt». А именно, в начале этого файла будет записано целое

число, предопределяющее количество музыкальных стилей, как это показано на рис. 12.

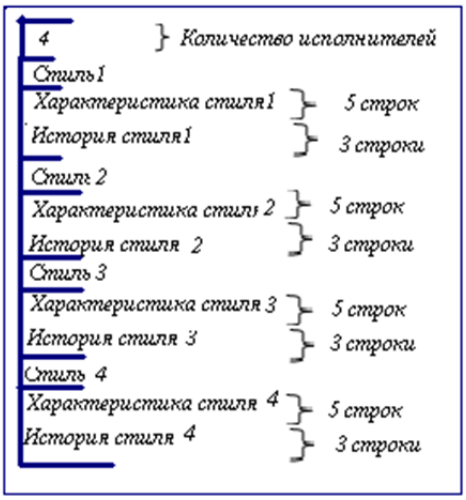


Рис. 12. Структура файла «Исполнители.txt»

Алгоритм сводится к последовательному чтению строк файла. Из строк – наименований музыкальных стилей создается список строк (это поле класса), и к каждой такой строке привязывается структура с данными о стиле.

Используются следующие локальные переменные:

`Way` – строковая переменная для хранения полного имени файла;

`Work` – динамическая структура для хранения данных о стиле;

`Fin` – файловая переменная;

`Buf` – вспомогательная строка для хранения строки из файла;

`Buf1` – для хранения названия стиля;

`aLst` – вспомогательный список для временного хранения содержимого файла;

`Count` – количество стилей;

`k` – текущий номер читаемой структуры;

`i` – текущий индекс строки из файла;

`StrName` – вспомогательная переменная для хранения строки символов.

Пошаговый алгоритм:

а) Отделяется маршрут в *Way*.
б) Формируется имя файла.
в) Преобразуется имя файла в строку.
г) Открывается файл для чтения.
д) Проверка, если файл не существует, то – выход.

е) Проверка, если файл пуст, то – выход.
ж) Читается первая строка файла.
з) Преобразуется в целое число – это кол-во стилей.
и) Создается список поля класса – MLStyle.
к) Выполняется цикл по числу стилей. В цикле:
   a. Отводится память под структуру и определяется ее адрес.
   b. Читается строка из файла и копируется в поле структуры NameStyle.
   c. Запоминается название стиля.
   d. Выполняется цикл по числу строк – характеристик. В цикле читаются строки и добавляются к полю структуры lstTemper
   e. Выполняется цикл по числу строк – историй стиля. В цикле читаются строки и добавляются к полю структуры lstHistory
   f. к строке списка с названием стиля привязывается структура с описаниями стиля
л) закрывается файл.

Описание метода представлено на листинге 20.

Листинг 20

Описание метода `Do1MusicListStyles` *из класса* `TFonoZZ`

```
void TFonoZZ::Do1MusicListStyles(UnicodeString NamF)
{
   AnsiString Way = ExtractFileDir (NamF);  //отделить марш-
рут к файлу
   Way = Way+"\\Музыкальные стили.txt";     // сформировать
имя файла
   MusicStyle* Work;    //Динамическая структура
   char StrName[100];   // для хранения имени файла
   ifstream Fin;   //файловая переменная
   AnsiString Ptr = Way;   //промежуточное преобразование
   strcpy(StrName,Ptr.c_str());  //имя файла в строку
   Fin.open(StrName,ios::in);   //открыть файл для чтения
   if( Fin==NULL)return;    //если ошибка при открытии – вы-
ход
   if(Fin.eof())return;   //если файл пустой – выход
   char Buf[100];    //вспомогательная строка для хранения
строки из файла
   Fin.getline(Buf,100);  // прочитать 1-ю строку из файла
   int Count = atoi(Buf);   //  преобразовать строку в целое
число

   for(int i=0; !Fin.eof(); i=i+9)   //цикл по числу муз-ых
стилей
      {
        Work = new  MusicStyle;    //память под структуру
```

```
        Fin.getline(Buf,100);  //прочитать из файла строку

           //скопировать в поле структуры NameStyle
        strcpy(Work->NameStyle, Buf);

        AnsiString Buf1=Buf;    //сохранить название стиля
        for(int j=i+1; j<i+6; j++)   //цикл по числу строк –
характеристик
            {
                  // читать из файла строку – характеристику
               Fin.getline(Buf,100);

                   // добавить к полю структуры lstTemper
               (Work->lstTemper)->Add(Buf);
            }
        for(int j=i+6; j<i+9; j++)     //цикл по числу строк
– история стиля
            {
                // читать из файла строку – историю
                Fin.getline(Buf,100);

                   // добавить к полю структуры lstHistory
               (Work->lstHistory)->Add(Buf);
            }

                // к строке списка добавить строку с привя-
занной
                // структурой с описаниями стиля
          MLStyle->AddObject(Buf1,(TObject*)Work);
      }
   Fin.close();
}
```

Для привязки к строке списка определенной структуры используется метод, объявленный в классе `TStrings`, потомком которого является класс `TStringList`:

```
int AddObject(const AnsiString S, System::TObject* AObject);
```

Этот метод добавляет в список строку и связанный с ней объект. Возвращает индекс добавленной строки и объекта.

Описание конструктора класса `TFonoZZ` реализуется подобно конструктору класса `TFonoteka`, только в конец его дописываются операторы создания списков и операторы вызова методов `DoCastsOpus` и `Do1MusicListStyles`. Описание конструктора представлено на листинге 21.

Листинг 21

Описание конструктора класса `TFonoZZ`

```
TFonoZZ::TFonoZZ(UnicodeString NamFile)
     {
          TStringList* aLst;
          aLst = new TStringList;
          aLst->LoadFromFile(NamFile);
          int Temp = aLst->Count;
          FCount = Temp;
          ListFono = new FonoGramma [Temp];
          for(int i=0; i<Temp; i++)
            {
                 char St[100];
                 AnsiString T = aLst->Strings[i];
                 strcpy(St, T.c_str());
                 St>>ListFono[i];
            }
          delete aLst;

          MLStyle = new TStringList;
          COpus = new TStringList;
          DoCastsOpus (NamFile);
          Do1MusicListStyles(NamFile);
     }
```

На листинге 22 представлено описание метода `GetStyle`.

Листинг 22

Описание метода `GetStyle` *класса* `TFonoZZ`

```
void TFonoZZ::GetStyle(AnsiString NSt,TMemo* aMem)
{ int Num;
  Num =  MLStyle-> IndexOf(NSt);    //
  aMem->Lines->AddStrings(((MusicStyle*            )MLStyle-
>Objects[Num])->lstTemper);
}
```

Здесь используется метод:

```
void AddStrings(TStrings* Strings);
```

для добавления в список строк `aMem` группу строк из другого объекта `MLStyle->Objects[Num])->lstTemper`.

Класс `TStringList` наследует класс `TStrings`, реализуя многие его абстрактные свойства и методы и вводя некоторые новые возможности:

- сортировку строк в списке;
- запрещение.

11. Подготовка модуля для хранения класса

В общем случае модуль представляет собой совокупность программных ресурсов, предназначенных для использования другими модулями и программами. Программные ресурсы – это: константы, типы, подпрограммы, переменные.

Под модульным программированием понимается процесс разработки программы из нескольких логически завершенных единиц – модулей. Будем считать, что модуль – файл Си-программы, транслируемый независимо от других файлов (модулей).

Модульное программирование на уровне файлов – это возможность разделить полный текст программы на несколько файлов и транслировать их независимо друг от друга. Принцип МОДУЛЬНОГО ПРОГРАММИРОВАНИЯ – представление текста программы в виде нескольких файлов, *каждый из которых транслируется отдельно.*

С модульным программированием мы сталкиваемся в двух случаях:

- когда сами пишем модульную программу;
- когда используем стандартные библиотечные функции.

Согласно принципам скрытия информации в Borland C++ обычно текст модуля разделяют на *заголовочный файл* интерфейса, который содержит объявления классов, функций, переменных и т.п., и *файл реализации,* в котором содержится описание функций. Стандартное расширение файлов реализации – **.** `cpp`. Стандартное расширение заголовочных хэдер-файлов – .h.

Модуль сам по себе *не является выполнимой программой*. Чтобы можно было использовать модуль, его необходимо подключить к программе.

Подключение файлов/модулей в Borland C++ производится с помощью команды препроцессора `#include`:

```
#include <stdlib.h>
#include "matr.h"
```

В первом случае поиск файла производится в папках/директориях инсталлирования среды программирования (здесь C++ Builder), а во втором случае – поиск в директории, содержащей текущий файл.

11.1. Технология создания модуля для хранения класса в среде C++ Builder

Технология создания модуля для хранения класса в среде C++ Builder может быть описана следующими шагами:

1. Создать заготовку модуля. Для этого выполнить команду:
   File ▸ New ▸ Unit C++ Builder.

В результате появятся шаблон – заготовки для двух файлов с именами «по умолчанию»: Unit*i*. h – файла и Unit*i*.cpp – файла. Как это видно из представления, имена *.h и *.cpp – одинаковые.

2. Сохранить новый модуль в папке проекта (команда **File ▸ Save as...**) под заданным именем.

Как правило, имя модуля совпадает с именем класса, определение которого в нем содержатся. Потому для данной задачи следует в качестве имени модуля взять имя, повторяющее имя класса-потомка Fonoteka, но не совпадающее[6]. Шаблоны-заготовки представлены на рис. 13. Здесь слева заготовка cpp-файла, а справа – для h-файла. Эти заготовки содержат только команды препроцессора.

Если посмотреть на заготовку h-файла, можно увидеть в нем команду условной компиляции `#ifndef` и команду определения `#define`. У этих команд в качестве параметра используется имя уникальной для данного модуля псевдопеременной `FonotekaH`, которое складывается из названия модуля и символа «H». Оператор `#ifndef` позволяет проверить, определен или нет идентификатор `FonotekaH` в данный момент. Оператор препроцессора `#define FonotekaH` определяет эту уникальную псевдопеременную для данного модуля, после чего она станет определенной при компиляции.

Операторы препроцессора `#ifndef` и `#endif` заставляют транслятор обходить данный файл в том случае, если он уже транслировался и известен программе. Теперь, сколько бы раз не появлялся `#include "Fonoteka.h"`, компиляция будет выполняться один раз. Завершает модуль оператор `#endif`, который ограничивает область действия `#ifndef` и является необходимым для него.

```
#pragma hdrstop

#include "Fonoteka.h"
//подключение библиотек
#pragma package(smart_init)
//описание методов класса
```

```
#ifndef FonotekaH
#define FonotekaH
//подключение библиотек
//интерфейсы структур
//интерфейсы классов
#endif
```

Рис. 13. Шаблон-заготовки cpp-файла (слева), а справа – для h-файла

3. Вписать интерфейсы структур и интерфейсы классов в заготовку h-файла, расположив их после команды `#define` и перед командой `#endif`.

[6] Если модуль содержит несколько классов, то желательно, чтобы имя модуля характеризовало классы наполняющие его. Например, это могут быть классы одного семейства, тогда модуль может называть так же как семейство.

4. Если в параметрах методов класса используются типы классов, связанных с визуальными компонентами (например, `TMemo`), то в h-файл следует добавить библиотеку `<vcl.hpp>`. Расположить эту команду лучше всего перед описанием структур.
5. Вписать описание методов классов в заготовку cpp-файла. Располагать эти описания можно в самом конце заготовки, после команды #pragma package(smart_init). Методы, заявленные в интерфейсной части класса, описываются по обычным правилам описания процедур и функций. Для связи функций с классом, методами которого они являются, название класса указывается перед именем самой функции:

```
<тип результата> <имя класса>::<имя функции>(<список параметров>)
    {
        <тело функции>
    }
```

6. Если при реализации методов используются библиотеки C++, то их следует подсоединить в cpp-файле с помощью команды `#include`.
7. Добавить модуль к проекту командой **Project ▸ Add to Project**…

11.2. Подключение личного модуля

Подключить модуль к форме, на которой предполагается его использование можно одним из двух способов:

1. С помощью написания команды препроцессора:

```
#include "Fonoteka.h"
```

2. Воспользоваться командой C++ Builder:
   a. Сделать активным тот модуль (или форму), где предполагается использование класса.
   b. Включить хэдер-файл модуля в список команд препроцессора с помощью команды: **File ▸ User Unit …**

12. Использование методов класса для выполнения действий

Доступ к свойствам и методам класса обеспечивается через переменную классового типа, которую называют *экземпляром* класса[7]. Описание переменной типа класса (в таком случае принято говорить «создание экземпляра класса») в общем случае отличается от описания обычной переменной. Это связано с тем, что *классовая переменная должна создаваться с помощью конструктора*.

[7] Здесь программу можно представлять, как взаимодействие объектов. Объекты могут быть разных типов (то есть классов). Но могут даже группы объектов одного типа. Итак программа это группы объектов одного типа; объекты взаимодействуют друг с другом.

Формат описания может быть следующим:

1) <имя класса> <имя переменной>(<список параметров конструктора>);
2) <имя класса> <имя переменной>; // если конструктор без параметров
3) <имя класса>* <имя переменной>; //указатель на класс
   <имя переменной> = new <имя класса>(<список параметров конструктора>);
4) <имя класса>* <имя переменной>; //указатель на класс
   <имя переменной> = new <имя класса>; // если конструктор без параметров

Доступ к свойствам и методам класса обеспечивается через созданный экземпляр класса одним из двух способов:

1) Если переменная описана как простая классовая переменная, то вызов методов производится с использованием точечной нотации. Синтаксис:

<имя переменной типа класса>.<имя метода>(<список параметров>);

2) Если переменная описана как указатель, то вызов методов производится с использованием знака →. Синтаксис:

<имя переменной типа класса>→<имя метода>(<список параметров>);

Пример. Дан класс st, интерфейс которого представлен на рис. 14.

```
class  st
  {
    char   *s;
       public:
           st(char *);
           st(string &);
           st() { s=NULL; }
           st(int);
          ~st();
   char* Fun1(char);
   void  Fun2(char *);
   st &Fun3(st&);
};
```

Рис. 14. Интерфейса класса st

Пусть для доступа к методам класса созданы различные экземпляры класса с помощью операторов:

```
st   x ("abcd");
st   y (5);
st   d (5б,54);
```

```
st   z;
st   *p;
p = new st;
st   *w;
w = new st (56);
```

Требуется дать характеристику класса и характеристику процесса создания экземпляров классов. Подготовить вызовы методов `Fun1` и `Fun2`. Причем, вызов `Fun2` произвести от имени экземпляра с именем `p`, а вызов `Fun1` – от имени экземпляра `x`. Строку, полученную в результате выполнения функции `Fun1`, вывести на экран.

Решение.

а) Характеристика класса.

В поле класса для описания свойства объекта используется переменная `s` – строкового типа. В классе описано три конструктора с разными типами параметров: параметр-строка, параметр типа `string`, параметр-переменная целого типа. А также имеется конструктор без параметров. Кроме этого в нем имеется один деструктор и три метода с именами `Fun1`, `Fun2`, `Fun3`. У функции `Fun1` через параметр передается один символ – входная переменная. Результатом работы функции является строка, которая возвращается через заголовок функции. Функция `Fun2` не возвращает значение через заголовок, имеет один параметр типа указателя на строку, который может быть, как входным, так и выходным. Функция `Fun3` через заголовок возвращает ссылку на переменную с типом класса `st`, в качестве параметра используется также ссылка на переменную с типом `st`.

б) Характеристика процесса создания экземпляров класса.

Следует заметить, что при создании экземпляра класса, компилятор C++ автоматически выбирает нужный конструктор по значению параметра, указанного в описании переменной. Например, если параметр – целая переменная, берется конструктор с параметром – целая переменная; если такой конструктор в классе не находится – выдается сообщение об ошибке.

Переменная `x` описана как простая классовая переменная. Для создания этого экземпляра используется конструктор с параметром – строкой символов (`"abcd"`).

Переменная `y` описана как простая классовая переменная. Для создания этого экземпляра используется конструктор с параметром – переменной целого типа (целочисленное значение 5).

Переменная d описана как простая классовая переменная. Для создания этого экземпляра нужен конструктор с двумя параметрами – переменными целого типа (целочисленные значения 56 и 54). Такого конструктора нет. Компилятор выдает сообщение об ошибке: "*Could not find a match for d::d(int,int)".*

Переменная z описана как простая классовая переменная. Для создания этого экземпляра используется конструктор без параметров.

Переменная p описана как указатель на классовую переменную. Для создания этого экземпляра используется оператор new с вызовом конструктора без параметров.

Переменная w описана как указатель на классовую переменную. Для создания этого экземпляра используется оператор new с вызовом конструктора с параметром – переменной целого типа (целочисленное значение 56).

в) Вызов методов класса.

Так как экземпляр класса с именем p описан как указатель на класс, вызов метода нужно произвести с помощью знака «→»:

```
p→Fun2("abc");
```

Так как экземпляр класса с именем x описан как простая классовая переменная, вызов метода нужно произвести с помощью знака «.» (точка):

```
cout<<x.Fun1('u');
```

Поскольку Fun1 возвращает результат через заголовок, полученный результат можно сразу отправлять на печать.

13. Некоторые элементы обеспечения функциональности приложения

Для обеспечения требуемой функциональности программы, пользователю необходимо участвовать в диалогах и в зависимости от ситуации выполнить различные действия: согласиться с выбранным критерием; отказаться от действия; перейти в другое окно и т.д. С этой целью в C++ Builder используют *элементы событийно – ориентированного программирования* и систему диалогов.

13.1. Элементы событийно-ориентированного программирования

В качестве первоочередных (возможных) элементов управления в C++ Builder предусмотрены специальные визуальные компоненты, которые получили названия «Кнопки». Кнопки представлены в стандарт-

ных компонентах тремя классами – TButton, TBitBtn и TSpeedButton. Компоненты TButton и TBitButton реализованы в виде оболочек Windows-элементов управления, причем класс TBitBtn является наследником TButton и добавляет к возможностям родительского класса вывод изображений на поверхность кнопок и автоматическую установку стиля компонента на основе некоторого набора сочетаний изображений и подписей. Данные компоненты поддерживают также возможность автоматического разбиения подписей на слова и вывод их в несколько строк.

Помимо этого все объекты из библиотеки визуальных компонент (VCL, Visual Component Library) C++ Builder, равно как и объекты реального мира, имеют свой набор свойств и свое поведение – набор откликов на события, происходящие с ними.

Вообще вся идеология систем Windows основана на событиях. Функциональность приложения, создаваемого в среде C++ Builder, обеспечивается таким же механизмом. Нажатие на кнопку, выбор пункта меню, нажатие на клавишу мыши и т.д, является событием, на основе которого Windows создает специальную структуру данных. Данные приходят в оконную функцию приложения, затем C++ Builder направляет их тому компоненту, который и вызвал событие. Обработчиком события компонента будет процедура, вызываемая автоматически в ответ на событие, произошедшее в системе, и связанное с этим компонентом [3, 4, 9].

С каждым визуальным компонентом связана целая палитра возможных событий (она представлена в окне инспектора объектов, на вкладке Events). Сообщения о манипуляциях пользователя с мышью или клавиатурой (те самые события), производимых в пределах окна программы, унаследованы классом TForm от его родителя TControl, и являются общими для всех элементов управления C++ Builder. Данные сообщения позволяют, например, определить вид события в рамках компонента, например, нажатие кнопки мыши (событие OnClick) и двойное нажатие (событие OnDblClick).

При *щелчке кнопкой мыши* генерируются помимо OnClick еще два события: OnMouseDown – при нажатии кнопки мыши и OnMouseUp – при отпускании кнопки мыши. Эти события генерируются в порядке их возникновения:

- OnMouseDown;
- OnClick;
- OnMouseUp.

При *двойном щелчке кнопки мыши* помимо события `OnDblClick` генерируются и другие события в следующем порядке:

- `OnMouseDown`;
- `OnClick`;
- `OnMouseUp`;
- `OnDblClick`;
- `OnMouseDown`;
- `OnMouseUp`.

И еще одна группа событий унаследована формами от визуальных компонентов-оболочек, наследников класса `TWinControl`. Данные события позволяют установить факт нажатия пользователем кнопок клавиатуры, причем, определяются три раздельных процесса: нажатие на клавишу без ее отпускания (событие `OnKeyDown`), отпускание клавиши (событие `OnKeyUp`) и нажатие на кнопку (событие `OnKeyPress`).

13.2. Создание обработчика событий

Для создания события выделяется соответствующий компонент, например, кнопка Button1. В окне Object Inspector (Инспектор Объектов) открывается вкладка Events. В свободном поле (правое поле) строки `OnClick` производится двойной щелчок левой кнопкой мыши. Среда C++ Builder генерирует имя процедуры обработчика события и вставляет его в поле. После этого активизируется окно Code Editor с cpp-файлом. В него среда вставляет заготовку для процедуры, как это показано на листинге 23, которую следует в дальнейшем заполнить соответствующим кодом.

Листинг 23

Заготовка процедуры обработчика события `OnClick` для кнопки `Button1`

```
void __fastcall TForm1:: Button1Click (TObject* Sender)
{
}
```

А в h-файле в секции __published, появился новый член: это прототип функции-метода-обработчика события `OnClick`.

```
void __fastcall Button1Click (TObject* Sender);
```

Заметим, что имена обработчикам событий даются системой то же по определенному правилу: к **имени** компонента добавляется **имя события**. Благодаря этому облегчается работа с cpp-модулем, содержащим много обработчиков. Но пользователь может дать функции-обработчику другое подходящее по логике имя. Это необходимо делать в редактиру-

емом поле (правое) строки `OnClick` на вкладке Events, окне Object Inspector.

Процедура-обработчик события `OnClick` имеет один параметр `Sender` объектового типа `TObject`. Отметим еще одну особенность: перед именами функций-методов класса ставится опция компилятора `_fastcall`. Процедура-обработчик обычно является методом класса формы, которой принадлежит соответствующий интерфейсный элемент, однако это не обязательно.

13.3. Параметр Sender

`Sender` – стандартный входной параметр процедуры-обработчика событий. Через него передается указатель на компонент, который является инициатором данного события, подлежащего обработке. Если процедура-обработчик связана с несколькими компонентами, то перед описанием действий можно проанализировать, какой из компонентов находится в фокусе (`if (Sender == Button1) <оператор>`, где, например, `Button1` – имя переменной-кнопки). Это важно сделать если порядок обработки зависит от того, кто инициировал ее. А сравнение `Sender == Button1` как раз и позволяет узнать ссылку на инициа-тора.

13.4. Опция компилятора _fastcall

Эта опция влияет на процесс компиляции. Она обеспечивает передачу параметров функции в быстрые регистры, что ускоряет вызов функции. Для функций-методов классов эту опцию следует указывать всегда. Для других функций ее можно указывать, а можно и не указывать. Благодаря `_fastcall` передача параметров организуется **не через стек**, а через регистры центрального процессора. Вызовы обработчиков событий происходят очень часто, поэтому экономия времени, затрачиваемого на выборку параметров из памяти стека, оказывается весьма ощутимой. Здесь кроется одна из причин высокого быстродействия приложений, которые компилирует и собирает C++ Builder.

Пример. Требуется получить информацию об авторе произведения, представленного в одной из фонограмм. А именно, получить сведения об авторе, получить список его трудов, и его портрет, который хранится в графическом файле на диске, как это показано на рис. 9.

Решение.

Пусть содержимое фонотеки представлено в сетке StringGrid1 в виде, как это показано на рис. 15. Из описания структур файлов следует, что для каждого автора музыкального произведения в программе предусмотрено три файла. Наименования файлов совпадают с ФИО автора, а различие только в расширении: в файле с именем «*.bgr» хранятся биографические данные, в файле с именем «*.lst» – список его произведений, а в файле с именем «*.bmp» – его графическое изображение.

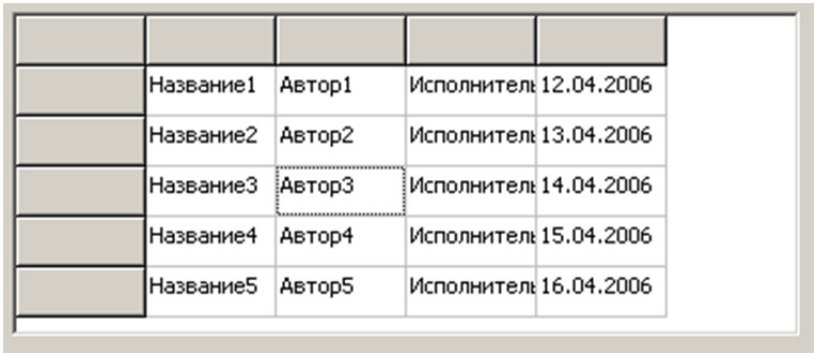


Рис. 15. Представление фонотеки в сетке StringGrid1

По условию задания необходимо обеспечить следующую функциональность приложения: выделил в сетке ячейку с ФИО автора, например, автора с фамилией Автор3 – прочитал содержимое файлов: Автор3.bgr; Автор3.lst, Автор3.bmp. Здесь файл Автор3.bgr является текстовым, в нем содержатся биографические данные Автор3. Прочитать эти данные можно в многострочный редактор, пусть это будет Memo1. Файл Автор3.lst также является текстовым, в нем содержатся список произведений Автор3. Прочитать эти данные можно в список строк, пусть это будет `ListBox1`. А для графического файла потребуется компонент, например, `Image1`.

С компонентом `StringGrid1` связана множество событий, которые представлены на закладке `Events`. Наиболее подходящим для решения задачи – остановить свой выбор на событии `OnSelectCell`, т.к. выбор файлов идет именно от выделенной ячейки в таблице данных.

Переход к обработчику события `OnSelectCell` для объекта `StringGrid1` формирует заготовку, как это показано на листинге 24.

Листинг 24

Заготовка процедуры – обработчика события OnSelectCell для StringGrid1

```
void __fastcall TForm1::StringGrid1SelectCell(TObject
*Sender, int ACol,                int ARow, bool &CanSelect)
```

```
{     //
}
```

Здесь параметр процедуры `ACol` – хранит номер колонки выделенной ячейки, а параметр `ARow` – хранит номер строчки выделенной ячейки. Колонка здесь имеет индекс 2, значение параметра `ARow` – позволяет выбрать конкретную строку и прочитать ФИО автора в выделенной ячейки, на основании чего можно сконструировать имена файлов со сведениями об авторе и прочитать информацию.

Алгоритм процедуры-обработчика можно описать следующими шагами:

1. Сформировать главные имена файлов.
2. Найти путь к файлам.
3. Сформировать полные имена файлов.
4. Загрузить содержимое файлов в соответствующие компоненты.

Полное описание обработчика события `OnSelectCell` представлено на листинге 25.

Листинг 25

Код процедуры – обработчика события `OnSelectCell` для `StringGrid1`

```
void __fastcall TForm1::StringGrid1SelectCell(TObject
*Sender, int ACol,            int ARow, bool &CanSelect)
     {
          //
          AnsiString Ptr = StringGrid1->Cells[2][ARow];
          AnsiString FileBGR = Ptr +".bgr";
          AnsiString FileLST = Ptr +".lst";
          AnsiString FileBMP = Ptr+".bmp";
          AnsiString Way = ExtractFileDir (FileFono);
          FileBGR  = Way+"\\"+FileBGR;
          FileLST  = Way+"\\"+FileLST;
          FileBMP  = Way+"\\"+FileBMP;
          Memo1->Lines->LoadFromFile(FileBGR);
          ListBox1->Items->LoadFromFile(FileLST);
          Image1->Picture->LoadFromFile(FileBMP);
     }
```

13.5. Построение диалогов

Как видно из сценария, описывающего работу пользователя с приложением и детали управления, отражающую логику действий пользователя при решении задачи, проект помимо основной формы должен иметь одну или несколько вспомогательных форм для вывода сообщений пользователю или для предоставления ему возможности

ввода данных. По сути, такие формы представляют собой диалоговые окна, т.к. в любом случае они требуют вмешательства пользователя.

Сложные диалоговые окна разработчику приходится тщательно проектировать и разрабатывать. Состав окна и набор кнопок определяется на стадии разработки проекта. Окно может быть показано в необходимый момент вызовом процедуры `Show`. По завершении работы с окном оно может быть спрятано вызовом процедуры `Hide`.

На рис. 16–18 представлены примеры разработки форм диалога. Чтобы *заблокировать работу программы*, пока открыто окно диалоговое окно, его можно показать в модальном режиме вызовом процедуры `ShowModal`. Тогда при нажатии на кнопку окно появляется сверху других окон проекта. Работать пользователь может только с таким окном. И только по закрытии этого окна может быть получен доступ к другим окнам.

Но при разработке иногда бывает удобным пользоваться готовыми диалоговыми окнами и вызывающими их функциями, которые входят в состав среды C++ Builder.

В среде C++ Buider существуют множество функций, реализующих простые типовые диалоги одного из трех видов:

- посылка сообщения пользователю;
- получение от пользователя ответа;
- ввод строки текста пользователем.

Рассмотрим только некоторые из них.

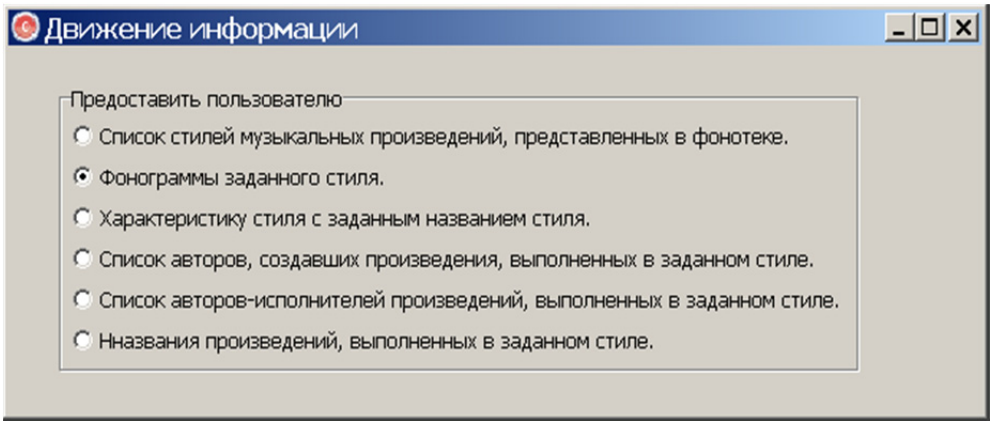


Рис. 16. Предоставление выбора из списка `RadioGroup`

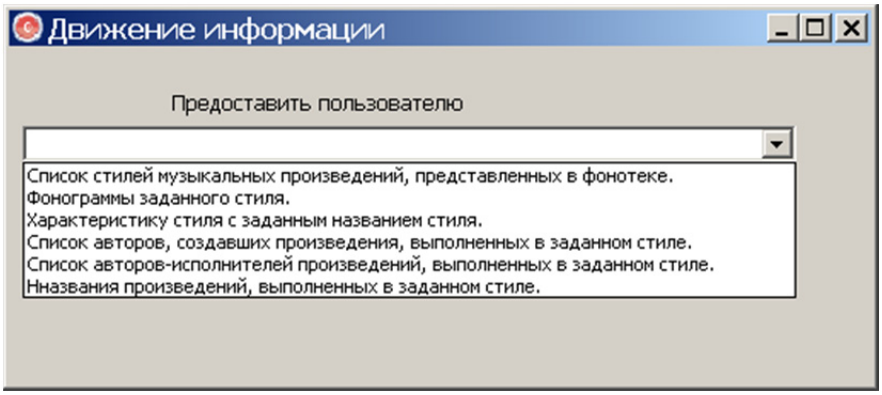


Рис. 17. Предоставление выбора из списка ComboBox

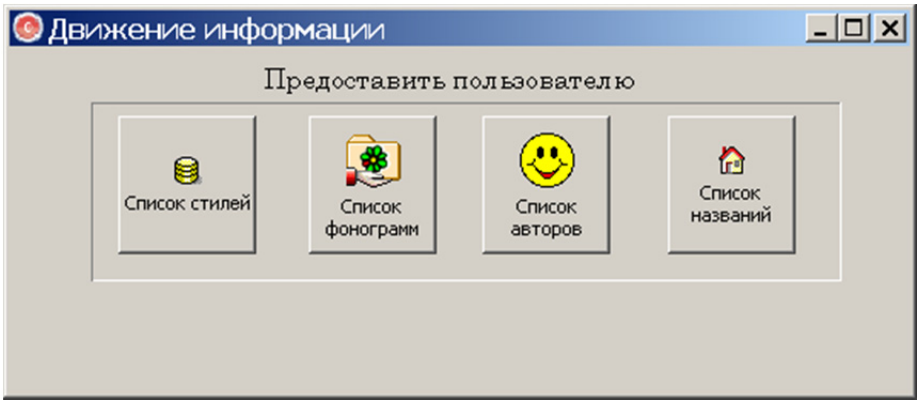


Рис. 18. Предоставление выбора с помощью кнопок BitBtn

13.6. Посылка сообщения пользователю

Простейшей из функций для посылки сообщений пользователю является функция:

```
void ShowMessage(constAnsiString Msg);
```

Эта функция отображает окно сообщения с кнопкой `OK`. Заголовок окна совпадает с именем выполняемого файла приложения. Текст сообщения задается параметром `Msg`.

Пример 1. В программу включен оператор вида:

```
ShowMessage("Неверно указано название музыкального стиля!");
```

Вызванная функция открывает окно с сообщением для пользователя, представленное на рис. 19.

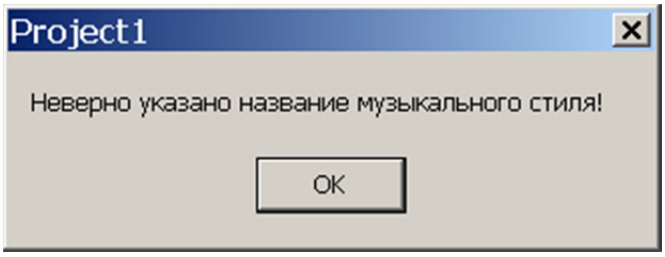


Рис. 19. Интерфейс окна с сообщением пользователю из примера 1

Пример 2. В программу включены операторы вида:

```
int Temp = Memo1->Lines->Count;
ShowMessage ("Число строк в редакторе
"+IntToStr(Temp));
```

Вызванная функция `ShowMessage` в результате выполнения открывает окно с сообщением для пользователя, представленное на рис. 20.

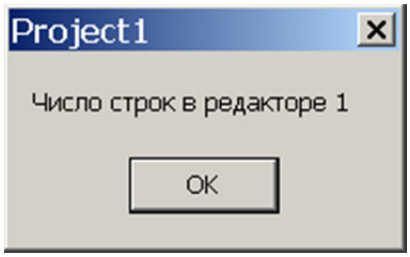


Рис. 20. Интерфейс окна с сообщением пользователю из примера 2

13.7. Ввод строки текста пользователем

Для ввода строк со стороны пользователя можно использовать функции `InputBox` и `InputQuery`. Прототип этой функции выглядит следующим образом:

```
AnsiString InputBox (constAnsiString ACaption,
          const AnsiString APrompt, const AnsiString
          ADefault);
```

Функция `InputBox` пользователю показывает диалоговое окно с заголовком `ACaption`, с подсказкой `APrompt`, и предлагает что-то написать и в окошке редактирования, в котором предварительно загружено значение текста по умолчанию `ADefault`. Если пользователь нажмет в окне ОК, то функция вернет введенную им строку текста. Если же пользователь в диалоге нажал `Cancel`, или нажал `Esc`, или закрыл окно системной кнопкой, то функция вернет строку `ADefault`, даже если перед этим пользователь что-то написал в окошке редактирования. Понять по возвращенному результату, написал ли пользователь какой-то текст, или отказался от ввода, можно, сравнив возвращенный результат со значением `ADefault`. Впрочем, результат останется неизменным и в случае, если пользова-

тель ничего не написал в диалоге, но нажал кнопку OK. Прототип функции InputQuery выглядит следующим образом:

```
bool InputQuery (const AnsiString ACaption,
                 const AnsiString APrompt, AnsiString
                 &Value);
```

Функция InputQuery выполняется подобно InputBox. Отличие: возвращается значение true если пользователь нажал OK, и false – в случае Cancel. Текст передается в переменной *Value*. То есть при использовании такого окна, можно получать и текст, и выполнять разный набор команд в случае закрытия по OK или Cancel.

Пример 3.По условию задачи требуется предоставить пользователю окно для ввода ФИО исполнителя музыкальных произведений. Результат ввода разместить в строковой переменной Ptr.

Решение. Воспользоваться функцией InputBox, например, следующим образом:

```
AnsiString Ptr;
Ptr = InputBox("Ввод данных","Введи ФИО исполнителя","Иванов
И.И.");
```

В результате открывается диалоговое окно, как это показано на рис. 21. В нем выделено поле для ввода ФИО. После ввода следует «нажать» кнопку <Ok>.

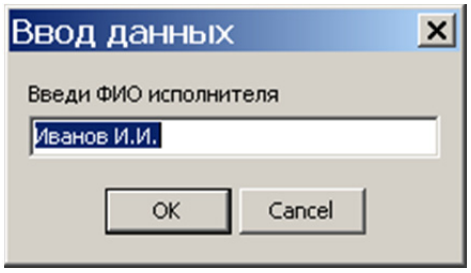


Рис. 21. Интерфейс окна с сообщением пользователю из примера 3

Пример 4. По условию задачи требуется предоставить пользователю окно для ввода ФИО исполнителя музыкальных произведений. Результат ввода разместить в строковой переменной TT.

Решение. Воспользоваться функцией InputQuery, например, следующим образом:

```
UnicodeString TT;
if (InputQuery ("Ввод данных","Введи ФИО исполнителя",TT))
  ShowMessage("Данные приняты!");
else  ShowMessage("Повторите ввод!");
```

В результате открывается диалоговое окно, как это показано на рис. 22. В поле для ввода ФИО пользователь должен вписать ФИО. Если после ввода ФИО будет выполнена команда Ok, в переменную TT будет помещена введенная ФИО. Следует обратить внимание, что переменная TT имеет тип UnicodeString. Такой тип требуется для параметра функции InputQuery.

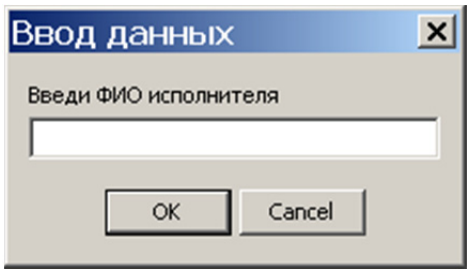


Рис. 22. Интерфейс окна с сообщением пользователю из примера 4

13.8. Получение от пользователя ответа

Когда после выдачи информации от пользователя требуется получить определенный ответ, можно применить функцию MessageDlg, с прототипом:

```
int MessageDlg (const AnsiString Msg, TMsgDlgType DlgType,
TMsgDlgButtons Buttons, int HelpCtx);
```

Вызов MessageDlg отображает диалоговое окно и ожидает ответа пользователя. Сообщение в окне задается параметром функции Msg. Вид отображаемого окна задается параметром DlgType. Возможные значения этого параметра представлены в таблице 9.

Параметр AButtons определяет, какие кнопки будут присутствовать в окне. Тип MsgDlgBtns параметра Buttons является множеством, которое включает различные кнопки. Возможные значения видов кнопок: mbYes, mbNo, mbOK, mbCancel, mbAbort, mbRetry, mbIgnore, mbAll, mbHelp. Например, одиночное значение, задающее единственную кнопку mbOK, при обращении к функции должно быть прописано по месту параметра Buttons как TMsgDlgButtons() << mbOK. Здесь TMsgDlgButtons() – функция, обеспечивающая включение кнопок в диалоговое окно. При необходимости задать на окне несколько кнопок они все должны перечисляться через знак <<, например, TMsgDlgButtons() << mbRetry << mbYes << mbIgnore. Для типовых случаев есть готовые подмножества: mbYesNoCancel, mbOKCancel, mbAbortRetryIgnore.

Таблица 9

Возможные значения параметра `Msg` для задания вида диалогового окна

mtWarning	Окно-предупреждение, содержащее желтый восклицательный знак
mtError	Окно-ошибка, содержащее красный стоп-сигнал
mtInformation	Окно-информация, содержащее голубой символ
mtConfirmation	Окно-подтверждение, содержащее зеленый вопросительный знак
mtCustom	Пользовательское окно

Параметр `HelpCtx` определяет экран контекстной справки, соответствующий данному диалоговому окну. Этот экран справки будет появляться при нажатии пользователем клавиши `F1`. Если вы справку не планируете, при вызове `MessageDlg` надо задать нулевое значение параметра `HelpCtx`.

Функция `MessageDlg` возвращает значение, соответствующее выбранной пользователем кнопке. Возможные возвращаемые значения:

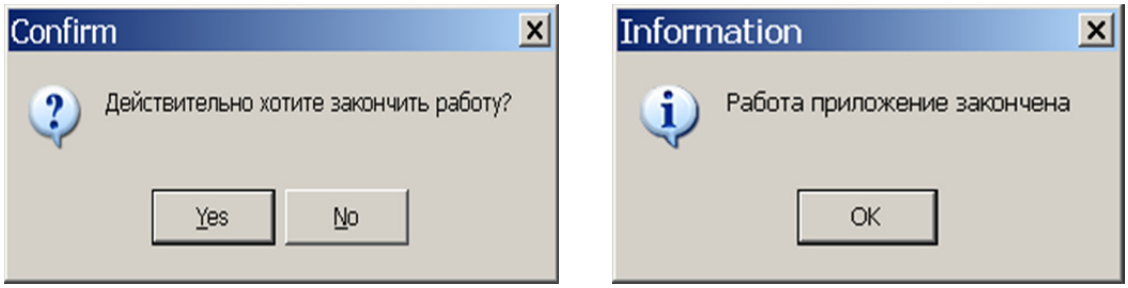
`mrNone`, `mrOk`, `mrCancel`, `mrAbort`, `mrRetry`, `mrIgnore`, `mrYes`, `mrNo`, `mrAll`

Пример 5. Пользователю требуется предоставить возможность для завершения программы. Предусмотреть предупреждение о завершении.

Решение. Можно воспользоваться функцией `MessageDlg`, включив в диалоговое окно две кнопки `mbYes` и `mbNo` с проверкой, какая из кнопок «нажата»:

```
if(MessageDlg("Действительно хотите закончить работу?",
mtConfirmation, TMsgDlgButtons() << mbYes<< mbNo, 0) ==
mrYes)
  {
    MessageDlg("Работа приложение закончена",mtInformation,
              TMsgDlgButtons() << mbOK, 0);
    Close ();
  }
```

Если пользователь нажмет `Ok`, работа программы будет завершена с уведомлением об этом. Окно функции `MessageDlg` представлено на рис. 23 (а), а окно с уведомление, которое открывает функция `MessageDlg` представлено на рис. 23 (б).



(а) (б)

Рис. 23. Интерфейс окна с сообщениями пользователю из примера 5

Пример 6. Требуется пользователю предоставить возможность выбора направленности действий: перейти к новой цели; сменить критерий поиска; продолжить решение задачи.

Решение. Можно воспользоваться функцией `MessageDlg`, включив в диалоговое окно две кнопки `mbRetry`, `mbYes` и `mbIgnore` с проверкой, какая из кнопок «нажата»:

```
int Bt;
   Bt =  MessageDlg (" Сменить цель?", mtCustom,
                     TMsgDlgButtons() << mbRetry << mbYes <<
mbIgnore, 0);
   if(Bt == mrYes)
     {
      ShowMessage("К новой цели!");
      //действия, связанные с переходом к новой цели
      }
   else
      if(Bt == mrRetry)
       {
        ShowMessage("К новому критерию поиска!");
        //действия, связанные со сменой критерия
       }
      else
        ShowMessage("Продолжить!");
```

В результате выполнения функции `MessageDlg` открывается диалоговое окно, как это показано на рис. 24.

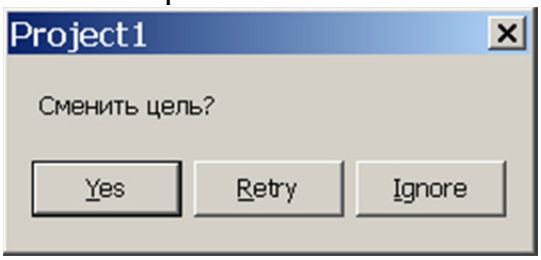


Рис. 24. Интерфейс окна с сообщениями пользователю из примера 6

14. Управление проектом

За редким исключением, задачи под Windows, во время выполнения, представлены на рабочем столе ***в виде окна*** и имеют хотя бы минимальные средства ведения диалога с пользователем. Оконное представление программ отвечает условиям, накладываемым на систему многозадачностью, так как несколько приложений одновременно не могут выводить свою информацию в одно и то же место.

В любой момент времени одно из окон «находится в фокусе», т.е. является «активным», и потоки вводимой информации направляются операционной системой в программу, которой принадлежит это окно. И хотя подавляющая часть ресурсов приложения встроена в саму операционную систему Windows, создание программ для Windows требует от разработчика написания огромного кода, описывающего окна программы, их элементы и взаимодействие этих окон и элементов. Избавиться от огромного труда при создании программ возможно, если использовать одну из визуальных систем программирования, например, Builder.

Все пользовательские программы в среде C++ Builder называются ***приложениями*** (прилагаются к самой среде)[8].

14.1. Хранение проекта

При создании ***приложений*** в среде C++ Builder необходимо хранить множество файлов:

- с кодами/текстами программ (исходные коды, объектные, загрузочные);
- с информацией об окнах среды;
- с информацией о связях между файлами;
- с информацией о внесенных изменениях и др.

Эти файлы специальным образом структурированы и связаны между собой – достаточно удалить один файл из этой связки, и программа-приложение перестанет работать[9]. Набор таких логически связанных файлов называется ***проектом***. Как это было сказано выше, проект C++ Builder состоит из нескольких файлов, логически связанных между собой. И поэтому их целесообразно хранить в ***единой***

[8] Но надо помнить, что с выходом новых версий ОС, с принципиальными изменениями в структуре их интерфейса, меняется и принятая терминология.

[9] Но некоторые из файлов генерируются заново при каждой компиляции проекта или при отсутствии требуемых. Поэтому такие файлы, конечно же, могут быть удалены для сокращения занимаемого места на диске или при архивировании. Но надо быть очень осторожным и точно знать какие из файлов можно удалять.

папке, дабы не растерять файлы, относящиеся к единому проекту, на необъятных просторах дискового пространства компьютера.

Организация файлов в виде проекта позволяет среде разработчика разделять элементы различного назначения в различные файлы, следить за изменениями в проекте и учитывать эти изменения в процессе компиляции. При такой организации проекта программист ***работает только с теми файлами, которые требуют модификации***. Так, например, при создании первых проектов начинающий программист работает только с файлом исходного кода программы. В дальнейшем, по мере освоения и при необходимости, он редактирует другие файлы. При этом редактирование может выполняться или непосредственно файлов, если они, для примера, в текстовом формате; или выполнять необходимые настройки в диалоговых окнах среды, а уже среда вносит сделанные изменения в соответствующие файлы.

14.2. Общие сведения о среде RAD Studio

Embarcadero RAD Studio – среда быстрой разработки кроссплатформенных приложений для Microsoft Windows фирмы Embarcadero Technologies.

Версии продукта:

1. Borland Developer Studio (2002 г.) – Первая версия с Delphi 7 Borland Developer Studio 1.0.
2. CodeGear RAD Studio (2006 г.) – добавлены средства разработки баз данных (БД). При использовании RAD Studio приложения могут подключаться к популярным базам данных, используя быстродействующие функции. Можно создавать многозвенные приложения с серверами Windows и поддержкой клиентов в нескольких операционных системах.
3. Embarcadero RAD Studio (2008 г.) – добавлены средства, позволяющие эффективно работать с данными для Windows, Mac OS X, .NET, PHP, веб-решений и мобильных устройств. Включена возможность также подключаться к широкому набору данных и служб.

Редакция RAD Studio XE2 Enterprise предназначена для разработчиков и групп разработки программного обеспечения (ПО), занимающихся созданием клиент-серверных, многоуровневых, облачных, веб-приложений для платформ Windows, Mac OS X, .NET, iOS и веб-решений. RAD Studio Enterprise обеспечивает удобство подключения к различным серверам баз данных и корпоративным источникам дан-

ных, а также предоставляет многоуровневую технологию DataSnap. Текущая версия Embarcadero RAD Studio XE2 объединяет Delphi XE и C++ Builder XE2 в единую интегрированную среду разработки.

Приложения RAD Studio XE2 компилируются в простые и эффективные исполняемые файлы, которые удобно распространять и развертывать.

В RAD Studio XE2 также входят тысячи встроенных расширяемых компонентов, многократно используемых и ускоряющих процесс разработки программ.

14.3. Описание структуры приложения

В папку сохранения проекта система **Embarcadero RAD Studio XE2** на начальном этапе помещает 7 файлов с исходными кодами и ресурсами программы:

- файл Project1.cbproj – отвечает за сборку проекта;
- файл Project1.cpp – главная программа текущего проекта;
- файл Project1.cbproj.local – фиксирует изменения в файлах текущего проекта;
- файл Project1.res – файл с ресурсами проекта;
- Unit1.dfm – файл описания формы;
- файл Unit1.cpp – для записи кодов, связанных с формой;
- Unit1.h – заголовочный файл для формы.

Автоматически создается папка __history – в ней будет накапливаться «история» файлов проекта, то есть их прежние копии.

Вследствие работы над проектом в него могут включаться новые формы. С добавлением каждой новой формы в папку проекта добавляются три связанных файла: Unit*.cpp; Unit*.h; Unit*.dfm. И, соответственно, вносятся изменения в проектные файла (файлы Project1.*). Если в проект добавляется модуль без формы, то к папке проекта добавляются два файла (без *.dfm). Во время компиляции программы система **Embarcadero RAD Studio XE2** создает папку Debug\Win32 в которую помещаются скомпилированные файлы и конечный исполняемый файл для среды Win32.

14.4. Схема связности модулей

Файлы проекта связаны между собой. На рис. 25 представлена схема, которая демонстрирует модульные связи некоторого проекта.

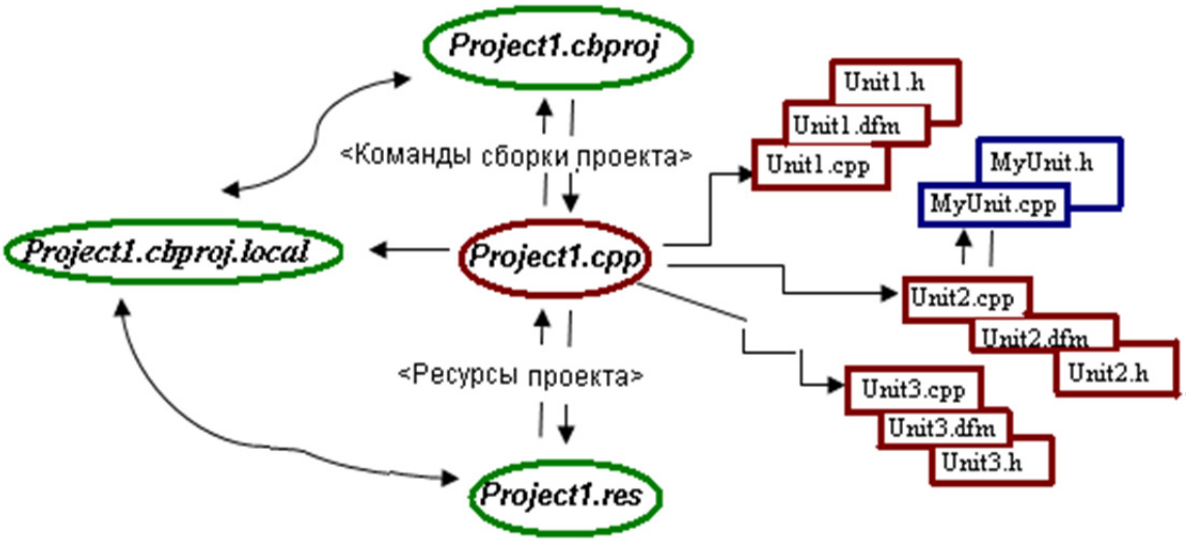


Рис. 25. Схема связности модулей проекта

14.5. Структура главной программы проекта

На рис. 26 приведен код главной программы, который располагается в файле Project*.cpp. В самом начале программы расположены три команды препроцессора, которые подсоединяют необходимые библиотеки и ресурсы. Затем следуют две директивы USERFORM. Директива USEFORM сообщает, какие модули и формы используются в проекте. Здесь же может располагаться и директива USERES компилятора, которая присоединяет файлы ресурсов к выполняемому файлу. При создании проекта автоматически создается файл ресурсов с расширением *.res для хранения курсоров, пиктограммы приложения и др.

После чего располагается главная программа проекта. В заголовке программы указано 4 параметра типов HINSTANCE, LPTSTR, int.

В программе используется объект Application, который создается автоматически при создании приложения в среде C++ Builder. Объект Application имеет тип TApplication – приложение. Этот компонент отсутствует в палитре библиотеки. Он всегда один в приложении. По сути это и есть приложение, которое может и не иметь окон. Объект Application имеет ряд свойств, методов, событий, характеризующих приложение в целом и к нему можно обращаться из модулей приложения.

В главной программе используются следующие методы Application:

- Метод Initialize – для инициализации проекта.
- Метод Run – для запуска выполнения приложения.

- Метод создания форм CreateForm вписывается автоматически для всех создаваемых форм проекта, которые создаются сразу на старте программы.
- Конструкция try … catch обеспечивает общее отлавливание моментов возникновения ошибок приложения и выдачу сообщения на экран, даже если подобные конструкции разработчик не предусмотрел в местах наиболее вероятного их возникновения.

```
#include <vcl.h>
#pragma hdrstop
#include <tchar.h>
//---------------------------------------------------------------------------
USEFORM("Unit1.cpp", Form1);
USEFORM("Unit2.cpp", Form2);
//---------------------------------------------------------------------------
WINAPI _tWinMain(HINSTANCE, HINSTANCE, LPTSTR, int)
{
        try
        {
                Application->Initialize();
                Application->MainFormOnTaskBar = true;
                Application->CreateForm(__classid(TForm1), &Form1);
                Application->CreateForm(__classid(TForm2), &Form2);
                Application->Run();
        }
        catch (Exception &exception)
        {
                Application->ShowException(&exception);
        }
        catch (...)
        {
                try
                {
                        throw Exception("");
                }
                catch (Exception &exception)
                {
                        Application->ShowException(&exception);
                }
        }
        return 0;
}
```

Рис. 26. Главная программа приложения, расположенная в файле ProjectN.cpp

15. Тестирование программы и ее сопровождение

После того как программа разработана и написан ее текст, необходимо провести ее тестирование и отладку. По сути это два взаимно связных процесса.

Отладка программы – это процесс поиска и устранения ошибок в программе, производимый по результатам её прогона на компьютере.

Тестирование – это испытание, проверка правильности работы программы в целом, либо её составных частей.

Основной смысл этих этапов состоит в проверке того, насколько программный продукт в том виде, в котором он получился, соответствует требованиям, установленным в процессе согласования спецификации. Каждая функция или метод класса должны соответствовать требованиям, определенным для них на этапе спецификации.

Перед тем, как приступить к тестированию, нужно разработать его план. Разработка плана тестирования состоит из следующих шагов:

1. описание последовательности действий, которые позволяют проверить как работу отдельных методов, так и их совокупности;
2. описание данных, для которых известны результаты тестирования;
3. описание места расположения тестовых данных.

Оценка результатов тестирования сводится к сравнению выходных данных работы отдельных процедур с контрольными значениями, которые получены в результате выполнения процедуры тестирования.

Для реализации метода тестов должны быть изготовлены или заранее известны точные результаты, как это показано на рис. 27. Их обычно называют эталонными. Вычислять эталонные результаты нужно обязательно *до*, а не *после* получения машинных результатов.

В противном случае имеется **опасность невольной подгонки вычисляемых значений под желаемые**, полученные ранее на машине.

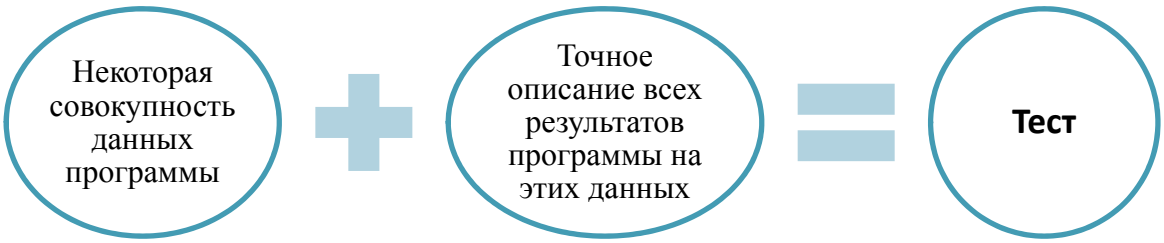


Рис. 27. Формула тестирования

Программу условно можно считать правильной, если её запуск для выбранной системы тестовых исходных данных во всех случаях

дает правильные результаты. Но, как справедливо указывал известный теоретик программирования Э. Дейкстра: «… тестирование может показать лишь наличие ошибок, но не их отсутствие». Нередки случаи, когда новые входные данные вызывают "отказ" или получение неверных результатов работы программы, которая считалась полностью отлаженной.

При отладке программ **важно помнить следующее:**

- в начале процесса отладки надо использовать простые тестовые данные;
- возникающие затруднения следует четко разделять и устранять строго поочередно;
- не нужно считать причиной ошибок машину, так как современные машины и трансляторы обладают чрезвычайно высокой надежностью.

15.1. Характеристика тестовых данных

Тестовые данные должны обеспечить проверку всех возможных условий возникновения ошибок:

- должна быть испытана каждая ветвь алгоритма;
- очередной тестовый прогон должен контролировать нечто такое, что еще не было проверено на предыдущих прогонах;
- первый тест должен быть максимально прост, чтобы проверить, работает ли программа вообще;
- арифметические операции в тестах должны предельно упрощаться для уменьшения объема вычислений;
- количество элементов последовательностей, точность для итерационных вычислений, количество проходов цикла в тестовых примерах должны задаваться из соображений сокращения объема вычислений;
- минимизация вычислений не должна снижать надежности контроля;
- тестирование должно быть целенаправленным и систематизированным, так как случайный выбор исходных данных привел бы к трудностям в определении ручным способом ожидаемых результатов; кроме того, при случайном выборе тестовых данных могут оказаться непроверенными многие ситуации;
- усложнение тестовых данных должно происходить постепенно.

Пример. Подготовить систему тестов для задачи нахождения корней квадратного уравнения $ax^2 + bx + c = 0$.

Решение. Система тестов может быть такой, как это показано на рис. 28.

Номер теста	Проверяемый случай	Коэффициенты			Результаты
		a	b	c	
1	d > 0	1	1	-2	x1 = 1, x2 = –2
2	d = 0	1	2	1	Корни равны: x1 = –1, x2 = –1
3	d < 0	2	1	2	Действительных корней нет
4	a = 0, b = 0, c = 0	0	0	0	Все коэффициенты равны нулю. x – любое число
5	a = 0, b = 0, c ≠ 0	0	0	2	Неправильное уравнение
6	a = 0, b ≠ 0	0	2	1	Линейное уравнение; один корень: x = –0.5
7	a ≠ 0, b ≠ 0, c = 0	2	1	0	x1 = 0, x2 = –0.5

Рис. 28. Система тестов для задачи нахождения корней квадратного уравнения

15.2. Основные этапы тестирования

Процесс тестирования можно разделить на три этапа:

1. Проверка в нормальных условиях. Предполагает тестирование на основе данных, которые характерны для реальных условий функционирования программы.
2. Проверка в экстремальных условиях. Тестовые данные включают граничные значения области изменения входных переменных, которые должны восприниматься программой как правильные данные. Типичными примерами таких значений являются очень маленькие или очень большие числа и отсутствие данных. Еще один тип экстремальных условий – это граничные объемы данных, когда массивы состоят из слишком малого или слишком большого числа элементов.

3. Проверка в исключительных ситуациях. Проводится с использованием данных, значения которых лежат за пределами допустимой области изменений.

Программа должна сообщать о любых данных, которые она не в состоянии обрабатывать правильно. Наихудшая ситуация складывается тогда, когда **программа воспринимает неверные данные как правильные и выдает неверный, но правдоподобный результат.** Такие ошибки сложнее всего выявить.

15.3. Характерные ошибки программирования

Ошибки могут быть допущены на всех этапах решения задачи – от ее постановки до оформления. Разновидности ошибок и возможные причины их возникновения приведены в таблице 10.

Таблица 10

Разновидности ошибок, возникающие в процессе отладки программ

Вид ошибки	Пример
Неправильная постановка задачи	Правильное решение неверно сформулированной задачи, фактически другой задачи
Неверный алгоритм	Выбор алгоритма, приводящего к неточному или неэффективному решению задачи
Ошибка анализа	Неполный учет ситуаций, которые могут возникнуть; логические ошибки
Семантические ошибки	Непонимание порядка выполнения оператора: неверное указание ветви алгоритма после проверки некоторого условия; неверное определение порядка действий; неполный учет возможных условий; пропуск в программе одного или более блоков алгоритма. Ошибки в циклах: неправильное указание начала цикла; неправильное указание условий окончания цикла; неправильное указание числа повторений цикла; бесконечный цикл.
Синтаксические ошибки	Нарушение правил, определяемых языком программирования: пропуск знака пунктуации; несогласованность скобок; неправильное формирование оператора; неверное образование имен переменных; неверное написание служебных слов; отсутствие условий окончания цикла; отсутствие описания массива и т.п.
Ошибки при выполнении операций	Слишком большое число, деление на ноль, извлечение квадратного корня из отрицательного числа и т. п.

Ошибки в данных	Неудачное определение возможного диапазона изменения данных: неправильное задание тип данных; организация считывания меньшего или большего объёма данных, чем требуется; неправильное редактирование данных; использование переменных без указания их начальных значений; ошибочное указание одной переменной вместо другой. Неверное указание типа переменной (например, целочисленного вместо вещественного); деление на нуль; извлечение квадратного корня из отрицательного числа; потеря значащих разрядов числа.
Опечатки	Перепутаны близкие по написанию символы, например, цифра 1 и буквы *I, l*; перепутаны близкие по написанию идентификаторы.
Ошибки ввода-вывода	Неверное считывание входных данных, неверное задание форматов данных

15.4. Использование программных средств для тестирования программ

В современных программных системах (C++ Builder, Delphi и др.) отладка осуществляется часто с использованием специальных программных средств, называемых дебáггерами или отладчиками.

Отлáдчик (англ. debugger) – компьютерная программа, предназначенная для поиска ошибок в других программах. Английский термин debugging ("*отладка*") буквально означает "*вылавливание жучков*". Термин появился в 1945 г., когда один из первых компьютеров – "Марк-1" прекратил работу из-за того, что в его электрические цепи попал мотылек и заблокировал своими останками одно из тысяч реле машины.

Отладчик позволяет исследовать внутреннее поведение программы и обеспечивает следующие возможности:

1) пошаговое исполнение программы с остановкой после каждой команды (оператора);
2) просмотр текущего значения любой переменной или нахождение значения любого выражения, в том числе, с использованием стандартных функций; при необходимости можно установить новое значение переменной;
3) установку в программе "контрольных точек", т.е. точек, в которых программа временно прекращает свое выполнение, так что можно оценить промежуточные результаты, и др.

C++ Builder имеет мощный отладчик программ, помогающий исправлять ошибки программирования. Однако для простых приложений

достаточно иметь минимальные сведения о нем. Команды/кнопки для отладки программы собраны в меню **Run**, как это показано на рис. 29. Некоторые из этих команд вынесены на панель инструментов системы в виде кнопок быстрого доступа.

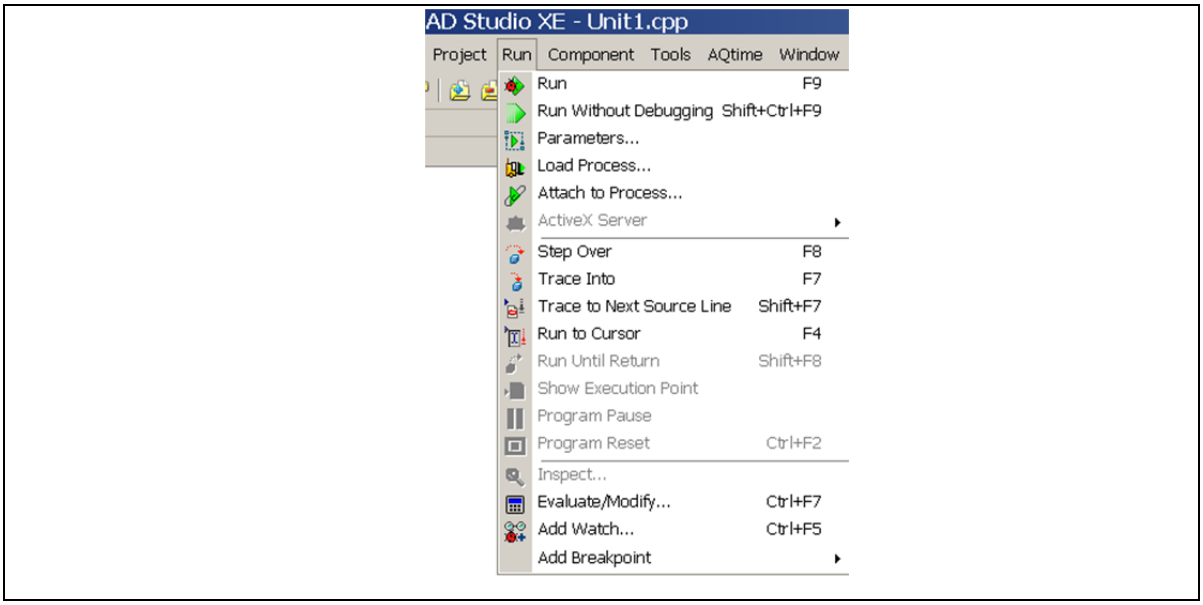


Рис. 29. «Отладочные» команды меню Run

Поясним назначение некоторых из команд:

– Run – выполнение программы с использованием отладчика (F9);

– Run Without Debugging – выполнение программы без использования отладчика (Shift+Ctrl+F9);

– пошаговое исполнение программы с остановкой после каждой команды (оператора) без захода в процедуру, если она указана в операторе, при этом функции пользователя будут выполняться целиком (F8);

– пошаговое исполнение программы с остановкой после каждой команды (оператора) с заходом в функцию, если она вызывается в операторе, здесь пошаговое выполнение будет включать и строки функций (F7);

– продолжение выполнения программы до строки с курсором (F4);

– Add Watch – открывает диалоговое окно, как это показано на рис. 30 для задания имен переменных с целью последующего мониторинга их значений в окне Watch List (Ctrl+F5), как это показано на рис. 30 и 31;

– Add breakpoint ►– установка точек останова; можно установить точку останова прямо в окне редактора: выполнить щелчок мышью

на вертикальной полоске серого цвета, идущей по левому краю окна редактора кода. На ней появится кружок красного цвета (рис. 31). Это и есть точка останова. Она находится в начале строки программы. Поэтому среда, по мере выполнения программы, прекратит выполнение кода точно в начале строки останова, выделит эту строку красным цветом и будет ожидать действий по отладке.

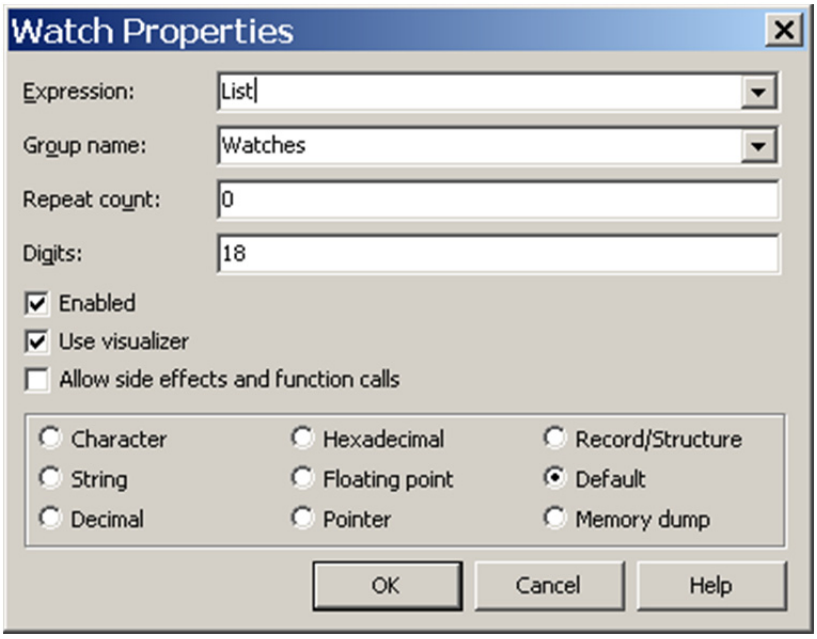


Рис. 30. Диалоговое окно отладчика для задания имен переменных с целью мониторинга

Во время прерывания программы в точке останова можно просматривать содержимое переменных и изменять их значение. Просмотр удобно выполнять, останавливая курсор мыши над соответствующей переменной в тексте программы.

Эксперименты с меню Run могут подсказать еще ряд приемов по отладке программы. Следует постепенно изучать эти приемы, тогда все будет легко и просто.

15.5. Примеры рекомендаций к разработке классов

Пример 1. Требуется разработать класс «Автопарк», предназначенный для хранения динамической информации о наличии автобусов в автобусном парке.

Рекомендации:

1. В информационную модель включить структуру, содержащую сведения о каждом автобусе:
   - номер автобуса;

- фамилию и инициалы водителя;
- номер маршрута.

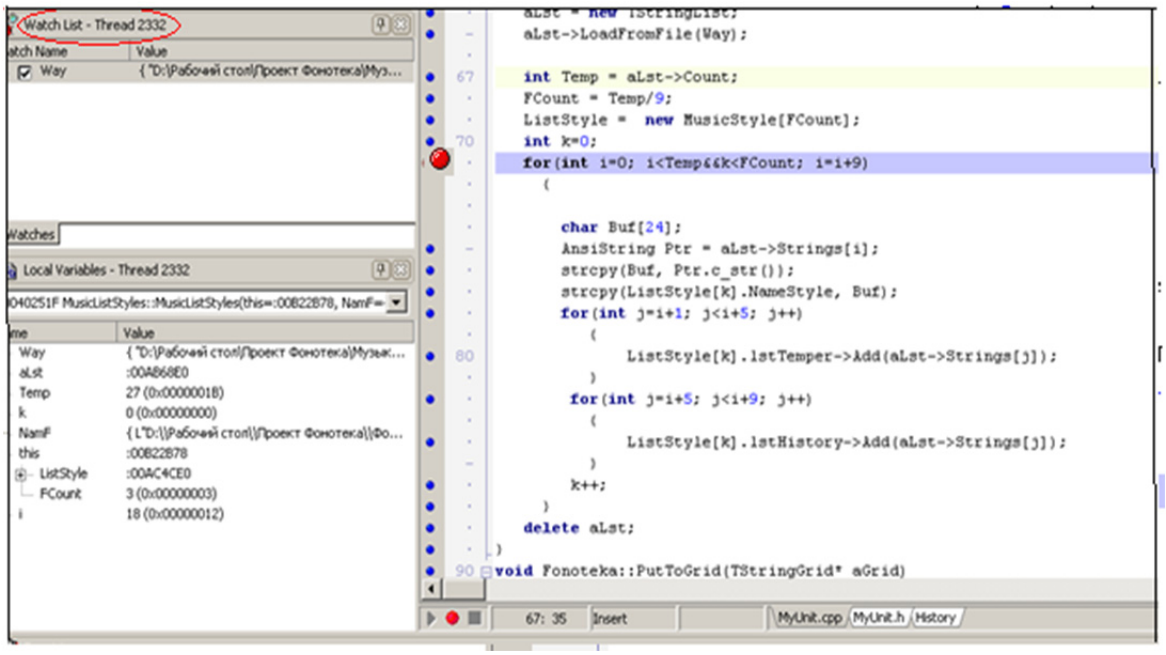


Рис. 31. Фиксация точки останова в окне редактора кода

2. В списке действий предусмотреть:
   - Подготовку двух списков. Один список – для хранения информации об автобусах, находящихся в парке. Другой список – для хранения информации об автобусах, находящихся на маршруте.
   - Начальное формирование данных о всех автобусах, находящихся в парке.
   - Удаление данных об автобусе из списка автобусов, находящихся в парке, при выезде автобуса на маршрут.
   - При выезде автобуса на маршрут запись его данных в список автобусов, находящихся на маршруте.
   - Удаление данных об автобусе из списка автобусов, находящихся на маршруте, при въезде автобуса в парк.
   - При въезде автобуса в парк запись его данных в список автобусов, находящихся в парке.
   - Возможность по запросу получить сведения: об автобусах, находящихся в парке; об автобусах, находящихся на маршруте; и т.п.
3. Можно предоставить графическую иллюстрацию, отражающую меняющееся состояние «парк-маршрут».

Пример 2. Требуется разработать класс «Структурный анализ данных», предназначенный для формирования кластеров из точек, сгруппированных между собой по величине проекций на заданную ось.

Рекомендации:

1. Исходные данные представить в текстовом файле, в который включить информацию о количестве точек, используемых для структурного анализа и координаты этих точек.
2. Выходные данные записать в текстовый файл, в котором предусмотреть помимо координат точек номер кластера, к которому эта точка относится.
3. В списке действий предусмотреть:
   - Отображение в графической области осей координат и набора точек, представленных в файле с исходными данными, как это показано на рис. 32. Для этого следует ввести элементы масштабирования, чтобы точки смогли попасть в отведенную область.
   - Сортировку точек по возрастанию проекций.
   - Решение задачи группирования точек.
   - Возможности для ввода координат произвольной точки (с учетом масштабирования) с последующим представлением ее внутри «своего» кластера.

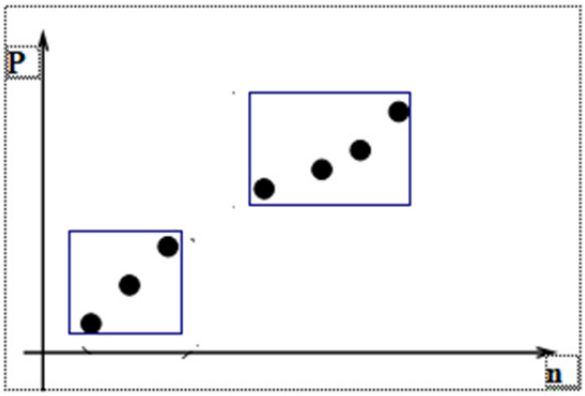


Рис. 32. Графическое представление набора точек с выделением кластерных областей

Пример 4. Требуется разработать класс «Линии уровня», предназначенный для геометрической интерпретации функции двух переменных u=f(x1,x2), если u – есть функция

$$u = (x_1 - 2)^2 + (x_2 - 1)^2 + \frac{0.04}{-\left(\frac{x_1^2}{4}\right) - x_2^2 + 1} + \frac{1}{0.2}(x_1 - 2x_2 + 1)^2$$

Предусмотреть поиск ограниченного множества точек (не более 5), направленных в сторону минимума функции, методом случайного поиска (рис. 33).

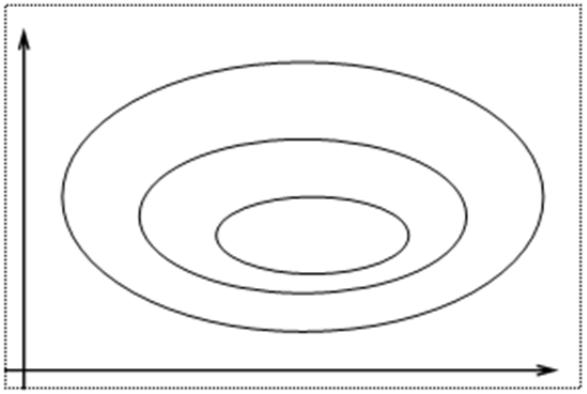


Рис. 33. Графическое представление изолиний

Рекомендации:

1. Известно, что линиями уровня называют линии, вдоль которых функция сохраняет постоянные значения: f(x1, x2)=C. Меняя C, мы получаем систему линий уровня, дающую достаточно полное представление о функции.
2. Исходные данные представить в текстовом файле, в который включить информацию о количестве точек, используемых
3. Выставить на графике точку с координатами $x_0(x_{01},x_{02})$. Здесь x_{01}, x_{02} – заданы.
4. Найти несколько точек, направленных в сторону экстремума (минимума) методом случайного поиска:
   a) случайно выбирается направление (можно разыграть датчиком случайных чисел координаты новой точки, а направление вдоль прямой).
   b) если значение функции (1) в новом состоянии превышает исходное значение или остается неизменным, то есть случайный выбор оказался неудачным, то происходит возврат в исходное состояние и отсюда осуществляется новый шаг и т.д. (4–5 точек).

4) Разработать интерфейсную часть программы. Предусмотреть диалоговые окна с запросами:

Окно 1: запрашивает формулу (1)
запрашивает значения **C** для построения линий уровня
содержит две кнопки: "Построить изображение", "Отмена"

Окно 2: отражает вид формулы
содержит график по типу Рис. 33
содержит три кнопки: "Поиск экстремума", "Отмена", "Изменить c"

Окно 3: представляет собой реакцию на кнопку "Поиск экстремума" и содержит

запрос на диапазон случайного числа, для выбора направления, а также

две кнопки: "Пуск", "Отмена".

Окно 4: повторяет окно 2, то есть содержит график функции, на котором дополнительно вставлены рассчитанные точки и две кнопки: "Изменить диапазон случайных чисел" и "Конец задачи". На кнопке "Изменить" предусмотреть новый ввод и повтор решений.

15.6. Сопровождение программного продукта

Сопровождение программ – это работы, связанные с обслуживанием программ в процессе их эксплуатации. Многократное использование разработанной программы для решения различных задач заданного класса требует проведения дополнительных работ, связанных с доработками программы для решения конкретных задач, проведения дополнительных тестовых просчетов и т.п.

Программа, предназначенная для длительной эксплуатации, должна иметь соответствующую документацию – сопровождение. Главный смысл этого этапа состоит в подготовке дополнительной, сопутствующей документации:

- руководство пользователю;
- проектных документов (описание структуры приложения и схема связности модулей и т.д.);
- спецификации программного продукта;
- руководства по тестированию.

Этот этап выполняется на протяжении всего времени работы над проектом, параллельно с другими этапами.

СПИСОК ЛИТЕРАТУРЫ

1. Павловская Т.А. C/C++. Программирование на языке высокого уровня – СПб.: Питер, 2003. – 461 с: ил.
2. Головач В.В. Дизайн пользовательского интерфейса. [Электронный ресурс]. – Режим доступа: http://www.uibook.ru, свободный
3. Рыбалка С.А., Шкатова Г.И. C++ Builder. Задачи и решения. Учебное пособие. – Томск: Изд-во ТПУ, 2010. – 486 с.
4. Рыбалка С.А., Шкатова Г.И. Методические указания «Языки программирования и методы трансляции» – Томск: изд. ТПУ, 2000. – 88 с.
5. Страуструп Б. Язык программирования C++. – М.: Радио и связь, 1991. – 352 с.
6. Узерелл Ч. Этюды для программистов: Пер. с англ. – М.: Мир, 1982. – 288 с.
7. Мозговой М.В. C++ Мастер-класс, 85 нетривиальных проектов, решений и задач. – СПб: Наука и техника, 2007. – 272 с.
8. Мозговой М.В. Классика программирования: АЛГОРИТМЫ, ЯЗЫКИ, АВТОМАТЫ, КОМПИЛЯТОРЫ. Практический подход. – СПб.: Наука и техника, 2006. – 320 с.
9. Пахомов Б.И. C/C++ и Borland C++ Builder для начинающих. – Спб.: БХВ-Петербург, 2006. – 736 с.: ил.

Приложение 1

Требования к оформлению текста пояснительной записки

Формат текста: Word for Windows – 95/97/2000/2003/XP/2007/2013. Поля: левое – 20 мм, правое – 15 мм, верхнее – 20 мм, нижнее – 25 мм. Ориентация: книжная, выравнивание по ширине. Шрифт: размер (кегль) – 12, тип – Times New Roman. Интервал текста: одинарный. Абзацный отступ: 1,25 см. Заголовки разделов, глав, параграфов должны отделяться от текста интервалами. Заголовок раздела – шрифт Times New Roman 14 ПРОПИСНЫМИ буквами. Обязательно соблюдение красной строки в начале абзаца. Все страницы работы, включая список использованной литературы, оглавление и приложение (если имеется) нумеруются по порядку от листа с содержанием работы до последней страницы. На титульном листе не проставляется номер страницы, на следующей странице (оглавление) ставится номер 2 и т.д. по порядку арабскими цифрами в верхней части листа по центру.

Рисунки, графики и таблицы не должны выходить за параметры страницы. Название и номера рисунков указываются под рисунками, названия и номера таблиц – над таблицами. Формулы выполняются в MS Equation, MathType или встроенном редакторе формул Word 2013.

Нумерация страниц выполняется *внизу*, *по центру*.

Приложение 2

Аннотация

Содержит очень краткое[10] содержание работы, перечень используемых ключевых слов, число страниц пояснительной записки, число рисунков, таблиц, приложений.

Введение

Введение должно содержать общие сведения по теме курсовой работы. Так, если в основе работы лежат списки, то требуется дать информацию о списках (что это, зачем, особенности и т.д.). Если речь идет о множества, то сведения о множествах и т.д. Если речь идет о тестах, то виды тестов, зачем тесты…

Во введении автор обосновывает выбор темы, ее актуальность, место в существующей проблематике, степень ее разработанности и освещенности в литературе, определяются цели и задачи исследования. Желателен сжатый обзор научной литературы.

Заключение

В заключении подводятся итоги исследования, обобщаются полученные результаты, делаются выводы по работе, рекомендации по применению результатов.

Здесь подводятся общие итоги проделанной работы, дается их оценка, делаются общие выводы.

Продумать и изложить перспективы в направлении усовершенствования разрабатываемого приложения, практическую значимость полученных результатов (где можно использовать).

[10] Примерно 10–15 строчек текста

Приложение 3

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего профессионального образования

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Институт	Кибернетики
Кафедра	Прикладная математика
Направление	Прикладная математика и информатика

Разработка класса «*Название класса***»**

КУРСОВАЯ РАБОТА

по дисциплине «Языки и методы программирования»

Студент гр. 8Б31	_____ (Подпись)	_____ (Дата)	Иванов И.И. (Ф.И.О)
Руководитель к.т.н., доцент каф. ПМ	_____ (Подпись)	_____ (Дата)	Г.И. Шкатова (Ф.И.О)
Допустить к защите: к.т.н., доцент каф. ПМ	_____ (Подпись)	_____ (Дата)	Г.И. Шкатова (Ф.И.О)

ТОМСК – 2014 г.

Приложение 4

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего профессионального образования

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Институт	Кибернетики
Кафедра	Прикладная математика
Направление	Прикладная математика и информатика

УТВЕРЖДАЮ:

Зав. кафедрой_____

“_____”_____

ЗАДАНИЕ

На выполнение курсовой работы

Студенту Иванову Ивану Ивановичу

Тема курсовой работы: Разработка класса «*Название класса*».

1. Срок сдачи студентом готовой работы: _____ 201__ г.
2. Исходные данные к работе:
3. Содержание расчетно-пояснительной записки (перечень подлежащих разработке вопросов):
    - Введение
    - Анализ и исследование задачи, построение моделей
    - Разработка интерфейса класса
    - Разработка методов класса
    - Разработка обработчиков событий
    - Схема связности модулей
    - Заключение
    - Приложения
4. Перечень графического материала (с точным указанием обязательных чертежей): концептуальная схема функционирования приложения; структуры входных и выходных файлов; схемы движения информации при работе с экземплярами классов; схема связности модулей; интерфейсы окон приложения.
5.

Дата выдачи задания: _____ 201__ г.

Руководитель_____(подпись)_____

Задание принял к исполнению_____(подпись)_____

Учебное издание

РЫБАЛКА Сергей Анатольевич
ШКАТОВА Галина Ивановна

ЯЗЫКИ И МЕТОДЫ ПРОГРАММИРОВАНИЯ

Учебно-методическое пособие

Компьютерная верстка *В.В. Михалев*

Зарегистрировано в Издательстве ТПУ
Размещено на корпоративном портале ТПУ

Национальный исследовательский Томский политехнический университет
Система менеджмента качества
Издательства Томского политехнического университета
сертифицирована в соответствии с требованиями ISO 9001:2008

ИЗДАТЕЛЬСТВО ТПУ. 634050, г. Томск, пр. Ленина, 30
Тел./факс: 8(3822)56-35-35, www.tpu.ru