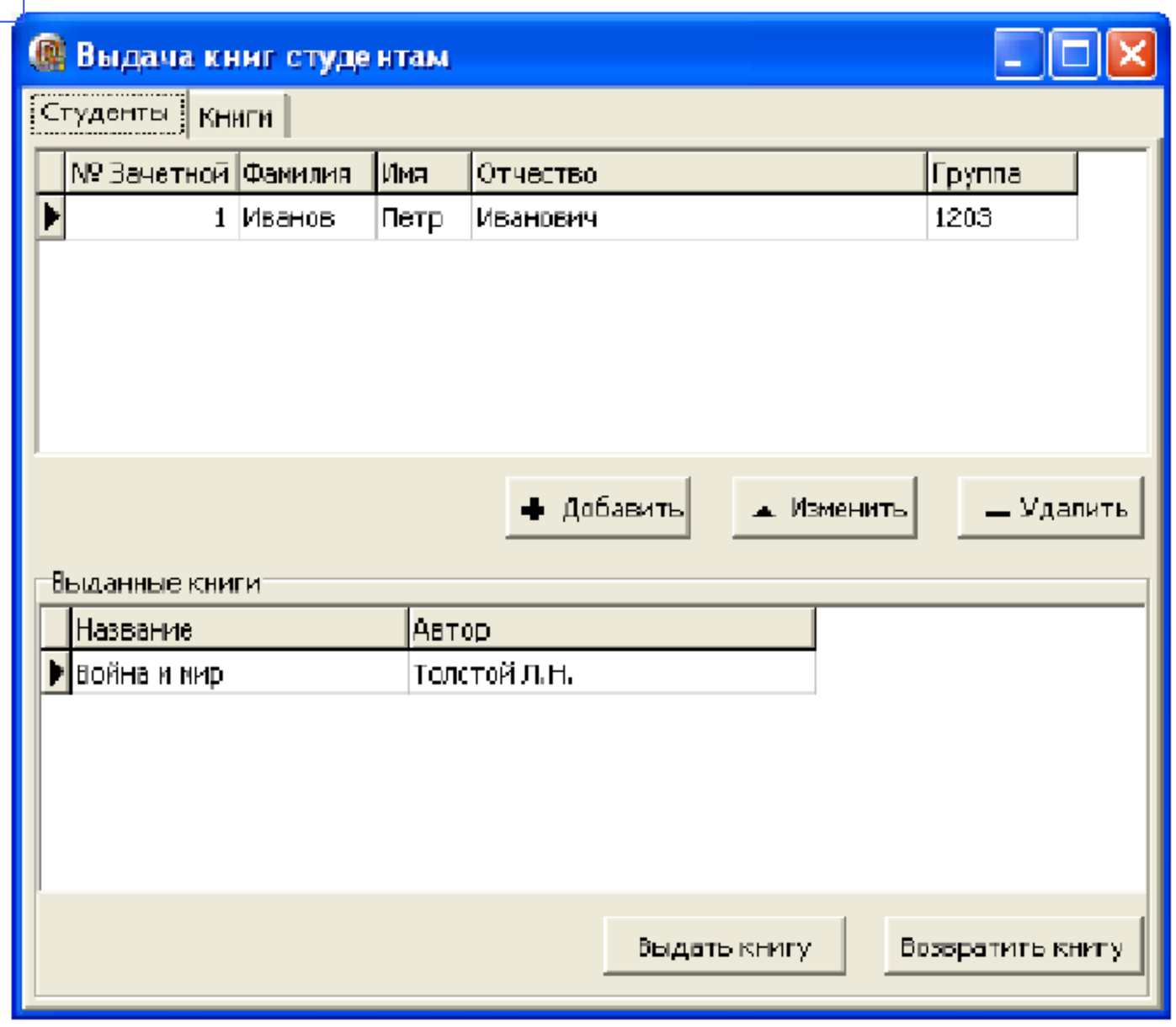

Рисунок 5.11 - Интерфейс выдачи студентам книг

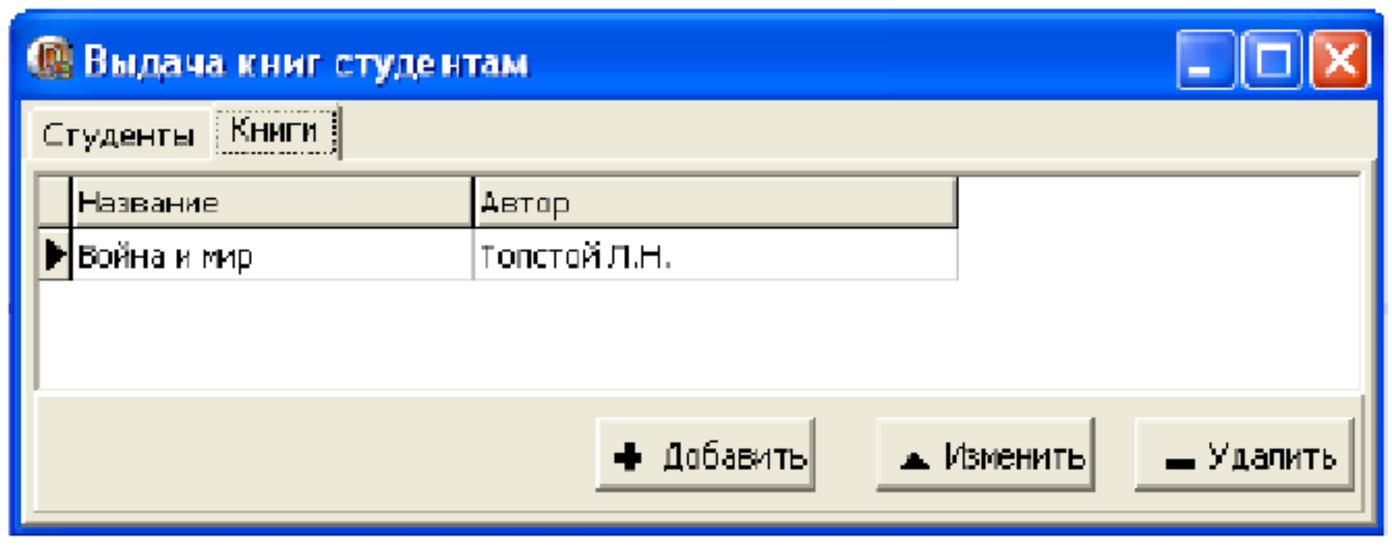

Рисунок 5.12 - Интерфейс редактирования списка книг

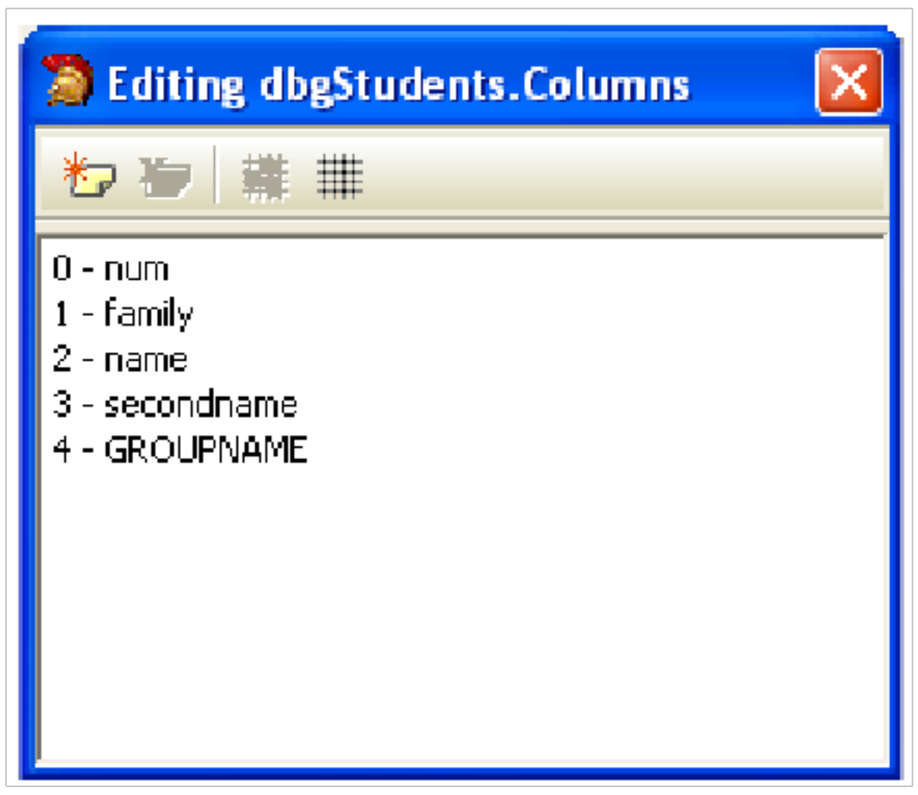

Рисунок 5.13 - Редактор отображаемых полей

13. Добавьте несколько компонентов TBitBtn на компоненты TPanel согласно рисунку 5.11 и 5.12.

14. Для добавления пиктограмм в проект нужно добавить компонент TImageList. Добавление пиктограммы осуществляется путем двойного щелчка по компоненту TImageList – появится окно настройки изображений (рисунок 5.8). В этом окне нажмите кнопку Add, в появившемся диалоге выберите графический файл пиктограммы.

15. Для реализации действий нужно добавить компонент TActionManager. Настройте свойство Images, хранящее пиктограммы.

16. Для добавления действий нужно два раза щелкнуть по компоненту TActionManager – появится редактор действий (рисунок 5.9).

17. Для добавления действия нужно нажать клавишу Ins.

18. Для настройки свойства действия выберите его в редакторе действий – его свойства отобразятся в Области инспектора. Свойства, которые нужно настроить, представлены в таблице 5.32.

Таблица 5.32 - Свойства действия

Свойство	Описание
Caption	Отображаемое название
Category	Категория действия
Name	Имя действия
ImageIndex	Номер пиктограммы из компонента TImageList

19. Также необходимо настроить событие, происходящее при выполнении действия. Для этого в Области инспектора выберите вкладку Events и два раза щелкните по событию OnExecute – появится редактор кода, в котором нужно записать требуемый код.

20. Нужно создать столько действий, сколько кнопок на форме.

21. Для назначения действия кнопки нужно выделить кнопку и в Области инспектора настроить свойство Action, выбрав то действие, которое должно происходить при нажатии кнопки.

Настройка интерфейса остальных форм выполняется по этому же принципу.

Форма редактирования параметров студента

Эта форма должна позволять пользователю задавать параметры студентов при добавлении новой записи и редактировании старой (см. рисунок 5.14).

Форма редактирования параметров книг

Форма должна позволять пользователю задавать параметры книги при редактировании и изменять параметры существующих книг (см. рисунок 5.15).

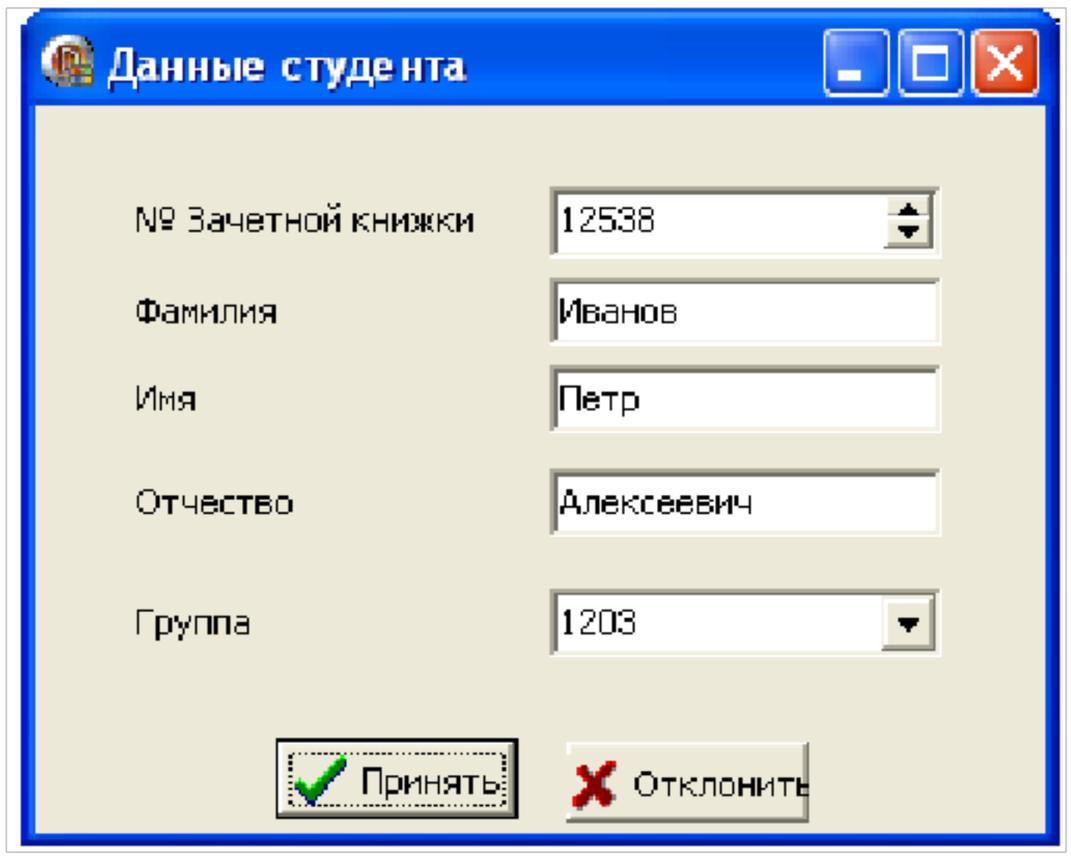

Рисунок 5.14 - Форма редактирования параметров студента

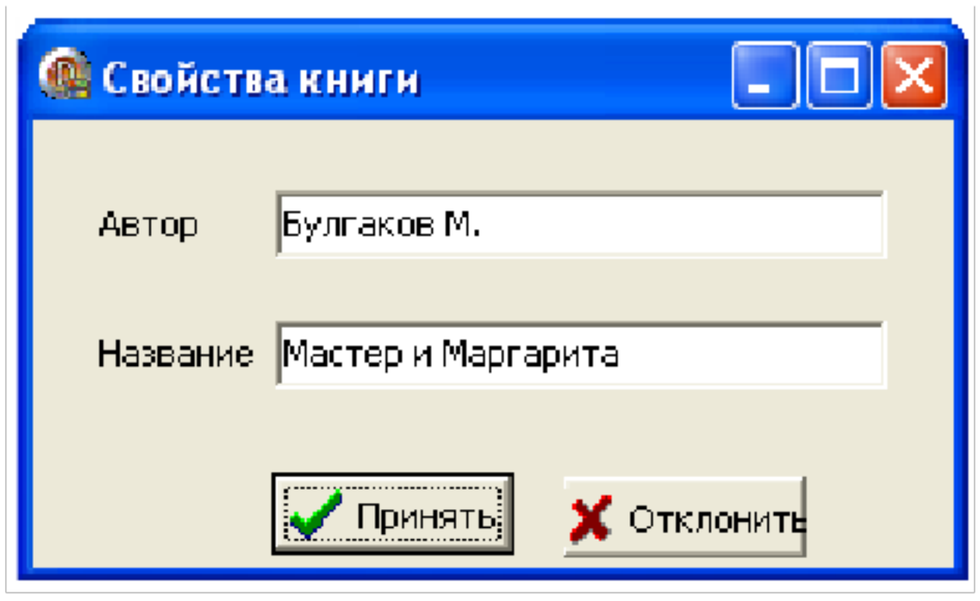

Рисунок 5.15 - Форма редактирования параметров книг

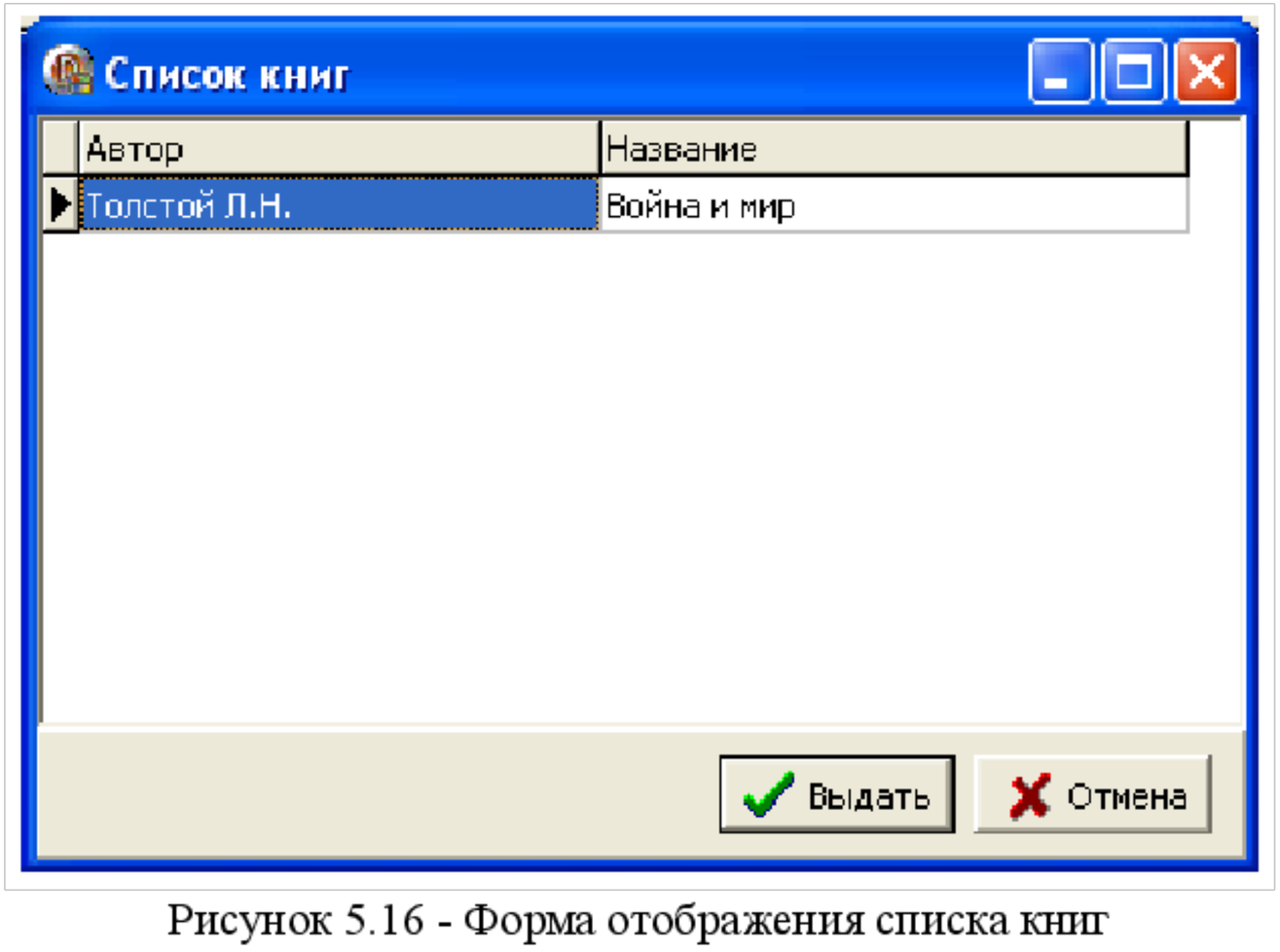

Рисунок 5.16 - Форма отображения списка книг

Форма отображения списка книг

Форма должна отображать список всех книг, для того чтобы можно было выбрать книгу при выдаче ее студенту (см. рисунок 5.16).

После создания форм необходимо подключить классы. Для этого нужно их создать для дальнейшего использования.

Подключение классов

После создания форм к интерфейсу нужно подключить классы. Для этого в главной форме проекта реализуем два события:

OnCreate. По этому событию создается объект TConnection, выполняется подключение к БД и создаются объекты класса бизнес-логики;

OnClose. По этому событию удаляются все созданные объекты.

Для создания событий:

в Конструкторе форм выберите объект, для которого вы хотите создать событие;

откройте вкладку Events в Области инспектора;

найдите требуемое событие и дважды щелкните по нему. При этом откроется редактор исходного кода на месте только что созданного события.

Пример кода для события On Create

```
var
//Объявление переменной класса по работе с ini-файлами
ini: tinifile;
// Объявление переменных для хранения параметров подключения
servername, path: string;
begin
 try
   //Подключение к ini–файлу
ini:=TIniFile.Create(ExtractFilePath(Application.ExeName)+'
```

```
config.ini');
// Чтение имени сервера, к которому необходимо подключиться
   if ini.ReadString('options','servername','')='' then
   begin
     servername:='localhost';
//Отображение диалога ввода имени сервера
     if  InputQuery('Введите  имя  сервера','',servername)
then
     begin
// Запись имени сервера в ini-файл
       ini.WriteString('options','servername',servername);
     end;
   end;
// Чтение имени сервера
   servername:=ini.ReadString('options','servername','');
// Чтение пути к файлу БД, если он пуст
   if ini.ReadString('options','path','')='' then
   begin
// Вывод диалога ввода пути к файлу БД
     if InputQuery('Введите путь к файлу БД','',path) then
     begin
// Запись пути к файлу БД в ini-файл
       ini.WriteString('options','path',path);
     end;
   end;
// Чтение пути к файлу БД из ini-файла
   path:=ini.ReadString('options','path','');
   try
// Создание объекта подключения
      fConnection:=TConnection.Create;
   // Установка параметров подключения
      fConnection.ServerName:=servername;
      fConnection.DataBaseName:=path;
      fConnection.UserName:='SYSDBA';
      fConnection.Password:='masterkey';
   //                Подключение                к                БД
fConnection.Connect;
      except
   //  В  случае  установки  неправильных  параметров  подключения  возникнет
исключительная ситуация
      ShowMessage('Неправильные параметры подключения');
   // Освобождение объекта подключения
      freeandnil(fConnection);
   // Закрытие формы приложения
      fmMain.Close;
   // Закрытие приложения
       Application.Terminate;
   // Выход из события
       Exit;
     end;
   finally
   // Удаление из памяти переменной класса по работе с ini-файлом
     FreeAndNil(ini);
   end;
   // Создание класса слоя бизнес–логики по работе со студентами
   fStudents:=TStudents.Create(fConnection,fConnection.TransactionObject);
   // Установка источника данных для сетки строк, отображающей список студентов
```

```
  dbgStudents.DataSource:=fStudents.DataSource;
// Выбор данных
  fStudents.Select;
// Создание класса слоя бизнес-логики по работе с книгами
fBooks:=TBook.Create(fConnection,fConnection.TransactionObject);
// Установка источника данных для сетки строк, отображающей список книг
  dbgBooks.DataSource:=fBooks.DataSource;
// Выбор данных
  fBooks.Select;
// Создание класса слоя бизнес-логики по работе с выданными
книгами
      fStudentBook      :=TStudentBook.Create(fConnection,      fConnection.TransactionObject);
// Установка события, происходящего при перемещении между записями
        fStudents.OnAfterScroll:=StudentsAfterScroll;
      // Загрузка параметров выбранного студента в поля объекта
        fStudents.LoadCurrent;
      // Установка полей объекта
        fStudentBook.fk_student:=fStudents.id;
      // Выбор данных
        fStudentBook.Select;
      // Установка источника данных для сетки строк, отображающей список выданных книг
        dbgStudentsBook.DataSource:=fStudentBook.DataSource;
      end;
```

Для каждого действия в компоненте TActionManager необходимо реализовать событие OnExecute, в котором будет происходить вызов классов слоя бизнес-логики.

Пример реализации события On Execute

```
Try
// Загрузка параметров выбранного студента в поля объекта
  fStudents.LoadCurrent;
// Создание формы
 fmChangeStudent:=TfmChangeStudent.Create(Application);
  with fmChangeStudent do
  begin
// Заполнение значений компонентов на форме
    seNum.Value:=fStudents.Num;
    edFamily.Text:=fStudents.Family;
    edName.Text:=fStudents.Name;
    edSecondName.Text:=fStudents.SecondName;
// Заполнение списка, отображаемого в компоненте TComboBox
    cbGroup.Items.AddStrings(fStudents.GroupList);
// Установка значения, отображаемого в компоненте TComboBox
cbGroup.ItemIndex:=cbGroup.Items.IndexOf(fStudents.Group);
  end;
// Если пользователь нажал кнопку «Принять»
  if fmChangeStudent.ShowModal=mrOk then
  begin
// Заполнение полей объекта
    with fStudents do
    begin
      Num:=fmChangeStudent.seNum.Value;
                                                  Family:=fmChangeStudent.edFamily.Text;
      Name:=fmChangeStudent.edName.Text;
      SecondName:=fmChangeStudent.edSecondName.Text;
      Group:=fmChangeStudent.cbGroup.Text;
    // Вызов метода, обновляющего запись в таблице
      Update;
```

```
    end;
  end;
 finally
//Удаление из памяти формы
   FreeAndNil(fmChangeStudent);
 end;
```

Сохранение проекта

Для того чтобы сохранить проект, нужно в главном меню выбрать File → Save All. При нажатии этой кнопки появится диалог сохранения файла, в котором нужно указать имя и место расположения файла. Рекомендуется файл проекта называть исходя из названия предметной области (Students, Users, Library), файлы форм называть по функциональности формы, используя при этом префикс fm (fmUser, fmBook, fmGroup), файлы классов называть по именам классов, используя при этом префикс Class (ClassUser, ClassBook, ClassGroup).

Задание

Разработать приложение в Borland Developer Studio в соответствии с выбранной тематикой. Приложение должно работать с реляционной СУБД, например FireBird или Access), а его архитектура – соответствовать трехслойной архитектуре на базе объектно-реляционного отображения с типизированными объектами.

Контрольные вопросы

1. Что такое трехслойная архитектура?
2. Структура проекта Borland Developer Studio.
3. Отношения между классами в Developer Studio.
4. Типовая структура слоя реляционного отображения.
5. Классы слоя бизнес-логики, их особенности.
6. Типизированные объекты.
7. Разработка классов с использованием редактора UML. Достоинства, недостатки.
8. Заполнение данными компонента TDBGrid.
9. Работа с данными через компонент TComboBox.

Работа с литературой:

Рекомендуемые источники информации (№ источника)			
Основная	Дополнительная	Методическая	Интернет-ресурсы
1-3	1-3	1-3	1-2

Оценочные средства: контрольные вопросы

Тема 4. Архитектуры программных систем.

Лабораторная работа № 6. Реализация архитектуры на базе объектно-реляционного отображения с нетипизированными объектами

Форма проведения: Практическое выполнение задания

Цели работы:

- научиться использовать шаблоны проектирования;
- разработать структуру классов трехслойного приложения, используя методологию проектирования, основанную на шаблонах;
- создать класс согласно шаблону для обработки системных событий нетипизированными объектами;
- создать приложение БД, использующее трехслойную архитектуру на базе объектно-реляционного отображения с нетипизированными объектами.

Краткие теоретические сведения

Трехслойная архитектура на базе объектно-реляционного отображения с нетипизированными объектами

Понятие «нетипизированные объекты» связано со слоем графического интерфейса. Классы этого слоя могут работать с классами слоя бизнес-логики только посредством контроллера. Таким образом, интерфейс вызывает методы класса контроллера, а класс контроллера вызывает методы классов слоя бизнес-логики. Интерфейс не связан с классами

бизнес-логики, что позволяет менять интерфейс, не переписывая вызовы методов слоя бизнеслогики (рисунок 6.1).

Рисунок 6.1 - Трехслойная архитектура с контроллером

Для разработки приложения на базе архитектуры с нетипизированными объектами мы будем применять шаблоны проектирования. Ниже описана часть основных шаблонов проектирования, используемых при разработке программных систем.

Шаблоны проектирования

Опытные разработчики объектно-ориентированных систем сформулировали общие принципы и стандартные решения, помогающие в разработке программного обеспечения. Если эти принципы и идиомы систематизировать и структурировать, а также присвоить им имена, то их можно применять в качестве шаблонов (patterns).

Шаблоны проектирования упрощают повторное использование удачных проектных и архитектурных решений. Представление прошедших проверку временем методик в виде паттернов проектирования облегчает доступ к ним со стороны разработчиков новых систем. С помощью шаблонов проектирования можно улучшить качество документации и сопровождения существующих систем, что позволит описать взаимодействия классов и объектов, а также причины, по которым система была построена так, а не иначе. Проще говоря, шаблоны проектирования дают разработчику возможность быстрее найти правильный путь.

В объектно-ориентированной технологии шаблоном проектирования называют именованное описание проблемы и ее решения, которые можно применить при разработке других систем. В идеале, шаблон должен содержать советы по его применению в различных ситуациях, а также описание его преимуществ и недостатков. Многие шаблоны содержат рекомендации по распределению обязанностей между объектами с учетом специфики задачи. Таким образом, шаблон – это именованная пара «проблема/решение», содержащая рекомендации по применению в конкретных ситуациях, которую можно использовать в различных контекстах.

Шаблон можно назвать новым, если он описывает новую идею. Однако сам термин «шаблон» означает стандартную, повторяющуюся сущность. Шаблоны не предназначены для изучения и выражения новых принципов разработки программного обеспечения, скорее, наоборот. Они призваны систематизировать существующие знания, идиомы и принципы: чем шире они используются, тем лучше.

Паттерн состоит из четырех основных элементов:

- имя. Всем шаблонам должны быть присвоены осмысленные имена. Именование шаблонов, методов и принципов имеет следующие преимущества: позволяет зафиксировать понятие в памяти человека, облегчает общение;

- задача. Этот элемент описывает, где следует применять паттерн. Необходимо сформулировать задачу и ее контекст. Может описываться конкретная проблема проектирования, например способ представления алгоритмов в виде объектов;
- решение. Описание элементов дизайна, отношений между ними, функций каждого элемента;
- результаты. Это следствия применения паттерна и разного рода компромиссы.

Шаблон Information Expert (информационный эксперт)

Проблема. Каков наиболее общий принцип распределения обязанностей между объектами при объектно-ориентированном проектировании?

Решение. Назначить обязанность информационному эксперту – классу, у которого имеется информация, требуемая для выполнения обязанности.

В модели системы могут быть определены десятки или сотни программных классов, а в приложении может потребоваться выполнение сотен или тысяч обязанностей. Во время объектно-ориентированного проектирования при формулировке принципов взаимодействия объектов необходимо распределить обязанности между классами. При правильном выполнении этой задачи система становится гораздо проще для понимания, поддержки и расширения. Кроме того, появляется возможность повторного использования уже разработанных компонентов в последующих приложениях.

При распределении обязанностей шаблон Information Expert используется гораздо чаще любого другого шаблона. В нем определены основные принципы, которые уже давно используются в объектно-ориентированном проектировании. Шаблон Expert не содержит неясных или запутанных идей и отражает обычный интуитивно понятный подход. Он заключается в том, что объекты осуществляют действия, связанные с имеющейся у них информацией.

Обратите внимание, что для выполнения обязанности зачастую требуется информация, распределенная между различными классами или объектами. Это предполагает, что должно существовать множество «частичных» экспертов, взаимодействующих при выполнении общей задачи. Например, для вычисления общей суммы продажи, в конечном счете, необходимо взаимодействие трех классов. Если информация распределена между различными объектами, то при выполнении общей задачи они должны взаимодействовать с помощью сообщений.

Преимущества

1. Шаблон Expert поддерживает инкапсуляцию. Для выполнения требуемых задач объекты используют собственные данные.
2. Соответствующее поведение системы обеспечивается несколькими классами, содержащими требуемую информацию. Это приводит к определениям классов, которые гораздо проще понимать и поддерживать.

Шаблон Creator (создатель)

Проблема. Кто должен отвечать за создание нового экземпляра некоторого класса?

Решение. Назначить классу B обязанность создавать экземпляры класса A, если выполняется одно из следующих условий:

- Класс B агрегирует объекты A;
- Класс B содержит объекты A;
- Класс B активно использует объекты A;
- Класс B обладает данными инициализации, которые будут передаваться объектам A при их создании (т. е. при создании объектов A класс B является экспертом);
- Класс B – создатель объектов A.

Если выполняется несколько из этих условий, то лучше использовать класс B, агрегирующий или содержащий класс A.

Создание объектов в объектно-ориентированной системе является одним из наиболее стандартных видов деятельности. Следовательно, при назначении обязанностей, связанных с созданием объектов, полезно руководствоваться некоторым основным принципом. Правильно распределив обязанности при проектировании, можно создать слабо связанные независимые компоненты с возможностью их дальнейшего использования, упростить их, а также обеспечить инкапсуляцию данных и их повторное использование.

Рассмотрение и распределение обязанностей выполняют в процессе создания диаграммы взаимодействий. Затем полученные результаты могут быть реализованы в конкретных методах, помещенных в разделе методов диаграммы классов.

Шаблон Creator определяет способ распределения обязанностей, связанный с процессом создания объектов. В объектно-ориентированных системах эта задача является одной из наиболее распространенных. Основным назначением шаблона Creator является выявление объекта-создателя, который при возникновении любого события должен быть связан со всеми созданными им объектами. При таком подходе обеспечивается низкая степень связанности.

В некоторых случаях в качестве создателя выбирается класс, который содержит данные инициализации, передаваемые объекту во время его создания. На самом деле это пример использования шаблона Expert. В процессе создания инициализирующие данные передаются с помощью метода инициализации некоторого вида, такого как конструктор языка Java с параметрами.

Преимущества

Применение шаблона Creator не повышает степени связанности, поскольку созданный (created) класс, как правило, оказывается видимым для класса-создателя посредством имеющихся ассоциаций.

Шаблон Low Coupling (слабое связывание)

Проблема. Как обеспечить зависимость, незначительное влияние изменений и повысить возможность повторного использования?

Решение. Распределить обязанности таким образом, чтобы степень связанности оставалась низкой.

Степень связанности (coupling) – это мера, определяющая, насколько жестко один элемент связан с другими элементами либо каким количеством данных о других элементах он обладает. Элемент с низкой степенью связанности (или слабым связыванием) зависит от не очень большого числа других элементов. Выражение «очень много» зависит от контекста, однако необходимо провести его оценку.

Класс с высокой степенью связанности (или жестко связанный) зависит от множества других классов. Однако наличие таких классов нежелательно, поскольку оно приводит к возникновению следующих проблем:

- изменения в связанных классах приводят к локальным изменениям в данном классе;
- затрудняется понимание каждого класса в отдельности;
- усложняется повторное использование, поскольку для этого требуется дополнительный анализ классов, с которыми связан данный класс.

В шаблоне Low Coupling описывается принцип, о котором нельзя забывать на протяжении всех стадий работы над проектом. Он является объектом постоянного внимания. Шаблон Low Coupling представляет собой средство, которое разработчик применяет при оценке всех принимаемых в процессе проектирования решений.

Шаблон Low Coupling поддерживает независимость классов, что, в свою очередь, повышает возможности повторного использования и обеспечивает более высокую эффективность приложения. Его нельзя рассматривать изолированно от других шаблонов, таких как Expert и High Cohesion. Скорее, он обеспечивает один из основных принципов проектирования, применяемых при распределении обязанностей.

Подкласс жестко связан со своим суперклассом. Поэтому, принимая решение о наследовании свойств объектов, следует учитывать, что отношение наследования повышает степень связанности классов.

Не существует абсолютной меры для определения слишком высокой степени связывания. Важно лишь понимать степень связанности объектов на текущий момент и не упустить тот момент, когда дальнейшее повышение степени связанности может привести к возникновению проблем. В целом, следует руководствоваться таким принципом: классы, которые являются достаточно общими по своей природе и с высокой вероятностью будут повторно использоваться в дальнейшем, должны иметь минимальную степень связанности с другими классами.

Крайним случаем при реализации шаблона Low Coupling является полное отсутствие связывания между классами. Такая ситуация тоже нежелательна, поскольку базовой идеей объектного подхода является система связанных объектов, которые «общаются» между собой посредством передачи сообщений. При слишком частом использовании принципа слабого связывания система будет состоять из нескольких изолированных сложных активных объектов, самостоятельно выполняющих все операции, и множества пассивных объектов, основная функция которых сводится к хранению данных. Поэтому при создании объектно-ориентированной системы

должна присутствовать некоторая оптимальная степень связывания между объектами, позволяющая выполнять основные функции посредством взаимодействия этих объектов.

Преимущества

1. Изменения компонентов мало сказываются на других объектах.
2. Принципы работы и функции компонентов можно понять, не изучая другие объекты.
3. Удобство повторного использования.

Шаблон High Cohesion (высокое зацепление)

Проблема. Как обеспечить возможность управления сложностью?

Решение. Распределение обязанностей, поддерживающее высокую степень зацепления.

Зацепление (функциональное зацепление) – это мера связанности и сфокусированности обязанностей класса. Считается, что элемент обладает высокой степенью зацепления, если его обязанности тесно связаны между собой и он не выполняет непомерных объемов работы. В роли таких элементов могут выступать классы, подсистемы и т. д.

Класс с низкой степенью зацепления выполняет много разнородных функций или несвязанных между собой обязанностей. Такие классы создавать нежелательно, поскольку они приводят к возникновению следующих проблем:

- трудность понимания назначения класса;
- сложности при повторном использовании;
- сложности поддержки;
- ненадежность, постоянная подверженность изменениям.

Классы со слабым зацеплением, как правило, являются слишком «абстрактными» или выполняют обязанности, которые можно легко распределить между другими объектами.

На практике уровень зацепления не рассматривают изолированно от других обязанностей и принципов, обеспечиваемых шаблонами Expert и Low Coupling.

Как и о принципе слабого связывания, о высокой степени зацепления следует помнить в течение всего процесса проектирования. Этот шаблон необходимо применять при оценке эффективности каждого проектного решения.

Следующие сценарии иллюстрируют различную степень функционального зацепления.

1. Очень слабое зацепление. Только один класс отвечает за выполнение множества операций в самых различных функциональных областях.
2. Слабое зацепление. Класс несет единоличную ответственность за выполнение сложной задачи из одной функциональной области.
3. Среднее зацепление. Класс имеет среднее количество обязанностей из одной функциональной области и для выполнения своих задач взаимодействует с другими классами.
4. Сильное зацепление. Класс имеет несложные обязанности в нескольких различных областях, логически связанных с концепцией этого класса, но не связанных между собой. Пример. Существует класс Company, который несет полную ответственность: за знание сведений обо всех сотрудниках компании; всю финансовую информацию. Эти две области не слишком связаны между собой, однако обе логически связаны с понятием «компания». К тому же предполагается, что такой класс содержит небольшое число открытых методов и требует незначительных объемов кода для их реализации.

Как правило, класс с высокой степенью зацепления содержит сравнительно небольшое число методов, которые функционально тесно связаны между собой, и не выполняет слишком много функций. Он взаимодействует с другими объектами для выполнения более сложных задач.

Преимущества

1. Повышаются ясность и простота проектных решений.
2. Упрощаются поддержка и доработка.
3. Зачастую обеспечивается слабое связывание.
4. Улучшаются возможности повторного использования, поскольку класс с высокой степенью зацепления выполняет конкретную задачу.

Шаблон Controller (контроллер)

Проблема. Кто должен отвечать за обработку входных системных событий?

Системное событие (system event) – это событие высокого уровня, генерируемое внешним исполнителем (событие с внешним входом). Системные события связаны с системными операциями (system operation), т. е. операциями, выполняемыми системой в ответ на события.

Решение. Делегирование обязанностей по обработке системных сообщений классу, удовлетворяющему одному из следующих условий:

- класс представляет всю систему в целом, устройство или подсистему (внешний контроллер);
- класс представляет сценарий некоторого прецедента, в рамках которого выполняется обработка всех системных событий, и обычно называется Handler, Coordinator или Session (контроллер прецедента или контроллер сеанса);
- для всех системных событий в рамках одного сценария прецедента используется один и тот же класс-контроллер.

Заметим, что в этот перечень не включаются классы, реализующие окно, аплет, приложение, вид и документ. Такие классы не выполняют задачи, связанные с системными событиями. Они обычно получают сообщения и делегируют их контроллерам.

Контроллер(controller) – это объект, не относящийся к интерфейсу пользователя и отвечающий за обработку системных событий. Контроллер определяет методы для выполнения системных операций.

Большинство систем получает внешние события. Обычно они связаны с графическим интерфейсом пользователя. Кроме того, системе могу передаваться внешние сообщения, например при обработке телекоммуникационных сигналов или сигналов от датчиков в системах управления.

Во всех случаях при использовании объектно-ориентированного подхода для обработки внешних событий применяются контроллеры. Шаблон Controller обеспечивает наиболее типичные проектные решения для этого случая. Как видно из рисунка 6.1, контроллер – это своеобразный вид интерфейса между уровнями предметной области и графического представления.

Типичной ошибкой при создании контроллеров является возложение на них слишком большого числа обязанностей. Обычно контроллер должен лишь делегировать функции другим объектам и координировать их деятельность, а не выполнять эти действия самостоятельно.

Контроллеры делятся на два типа.

1. Внешний контроллер(facade controller). Представляет всю систему, устройство или подсистему. Основная идея сводится к выбору некоторого класса, имя которого охватывает все слои приложения. Этот класс обеспечивает главную точку вызова всех служб из интерфейса пользователя и обращения к остальным слоям. Этот класс может представлять физический объект, телекоммуникационный переключатель, всю программную часть системы или любые другие понятия, выбранные разработчиком для представления системы в целом. Внешние контроллеры удобно использовать в том случае, когда существует лишь несколько системных событий или системные сообщения невозможно перенаправить другим контроллерам, наподобие системы обработки сообщений.

2. Контроллер прецедента (use case controller). Для каждого прецедента должен существовать отдельный контроллер. Контроллеры прецедентов следует использовать в том случае, когда применение внешнего контроллера приводит к слабой степени зацепления или высокой степени связывания. Контроллер прецедента вводится в том случае, если на существующий контроллер возлагается слишком много дополнительных обязанностей. Контроллеры прецедентов следует применять при наличии большого числа системных событий, распределенных между различными процессами.

Преимущества

1. Улучшение условий для повторного использования компонентов. Применение этого шаблона обеспечивает обработку процессов предметной области на уровне реализации объектов, а не на уровне интерфейса. Обязанности контроллера могут быть технически реализованы в объектах интерфейса, однако в этом случае программный код и логические решения, относящиеся к процессам предметной области, будут жестко связаны с элементами интерфейса, например с окнами. При этом снижается эффективность повторного использования компонентов в других приложениях, поскольку процессы предметной области ограничены рамками интерфейса (например, связаны с оконным объектом), что может оказаться неприемлемым в других приложениях. Делегирование выполнения системных операций специальному контроллеру облегчает повторное использование логики обработки подобных процессов в последующих приложениях.

2. Контроль состояния прецедента. Иногда необходимо удостовериться, что системные операции выполняются в некоторой определенной последовательности. Например, нужно гарантировать, чтобы операция makePayment выполнялась только после операции endSale, для

чего необходимо накапливать информацию о последовательности событий. Для этой цели удобно использовать контроллер, особенно контроллер прецедента.

Применение шаблона Information Expert

Шаблон информационный эксперт отвечает на вопрос, какой класс должен отвечать за ту или иную функциональность. Ответом на этот вопрос является то, что класс должен отвечать за ту или иную функциональность, в случае если у него есть достаточно данных для решения задач функциональности.

В примере из лабораторной работы №5 у нас есть два объекта взаимодействующих между собой – это Студенты и Книги. Очевидно, что у них есть определенная функциональность, эти объекты должны уметь хранить данные о себе, создаваться, изменяться и удаляться. Согласно шаблону Information Expert мы создаем эти классы (TSudents, TBook) и определяем эту функциональность. Теперь возьмем процесс выдачи книг и возврата их от студентов. Нам необходимо отображать список книг, выданных студенту, если мы добавим эту функциональность классу TStudents или классу TBook, то тогда мы их нагрузим лишними функциями. Оптимальным решением данной проблемы является созданием нового класса TStudentBook, в функциональность которого входят эти функции.

Применение шаблона Creator

Шаблон Creator отвечает на вопрос, кто должен создавать экземпляр нового объекта. Класс должен создавать экземпляр нового объекта, если выполняется одно или несколько условий:

- Класс B агрегирует объекты A;
- Класс B содержит объекты A;
- Класс B активно использует объекты A;
- Класс B обладает данными инициализации, которые будут передаваться объектам A при их создании (т. е. при создании объектов A класс B является экспертом);
- Класс B – создатель объектов A.

В качестве примера возьмем базовый класс для слоя бизнес-логики – класс TDataPrepare. Наследники этого класса должны обращаться к классу слоя объектно-реляционного отображения TExecuteObject для выполнения SQL-запросов. Существуют два вида запросов: возвращающие данные и не возвращающие данные. Запросы, возвращающие данные, используются для отображения одного или несколько объектов из реляционной таблицы. Запросы, не возвращающие данные, используются для модификации данных в реляционных таблицах. Таким образом, у нас есть проблема: кто должен создавать объект класса TExecuteObject. Поскольку разные объекты отображают данные из различных таблиц БД, то получается, что каждому объекту класса TDataPrepare необходимо иметь по одному экземпляру объекта TExecuteObject. Но тут появляется другая проблема при выполнении запросов, не возвращающих данные, текущий набор данных закрывается и при попытке другого запроса текущий набор данных закроется, и его необходимо будет заново открыть, что увеличит объем данных, передаваемых по сети. Тогда ответим на другой вопрос: кто должен отвечать за создание и удаления объекта, выполняющего такие запросы? Существуют два варианта: класс TDataPrepare или другой – новый класс. Решение данной проблемы зависит от того, могут ли быть отправлены одновременно два и более таких запросов. Если да, то тогда необходимо, чтобы этот объект создавал класс TDataPrepare, если нет, то тогда необходимо создать новый класс, обладающий такой функциональностью.

Использование шаблона High Cohesion

Шаблон High Cohesion позволяет управлять сложностью классов. Чем проще класс, тем проще его понять, использовать, модифицировать, документировать. Это не означает, что не должно быть сложных классов. Сложность должна быть достаточной для решения проблемы, не более.

В качестве примера возьмем процесс выдачи книг и возврата их от студентов. Нам необходимо:

- отображать список книг, выданных студенту;
- выдавать студенту новые книги;
- принимать от студента книги.

Если мы добавим эту функциональность классу TStudents или классу TBook, то тогда мы их нагрузим лишними функциями, поскольку эти классы могут существовать без этого процесса. Например, есть книги, которые не выдают студентам, и есть студенты, которые не берут книги в библиотеке.

Оптимальным решением данной проблемы является создание нового класса TStudentBook, в функциональность которого будет входить реализация этих функций. Данные, необходимые для процесса выдачи или возврата книг, нужно брать у классов TStudents и TBook. Таким образом, у нас появилось три класса, каждый из которых нагружен только ему присущей функциональностью.

Применение шаблонаController

Шаблон Controller позволяет централизованно обрабатывать системные события, возникающие в приложении. При использовании этого шаблона интерфейс приложения посылает сообщения классам слоя бизнес-логики через класс Contoller. Использование этого шаблона подразумевает создание одного или нескольких классов, которые обрабатывают системные события. Под системными событиями подразумевается функциональные действия пользователей.

Согласно шаблону Creator класс TController должен создавать и уничтожать объекты слоя бизнес-логики, поэтому необходимо в этом классе создать поля, представленные в таблице 6.1.

Таблица 6.1 - Поля класса TController

Название	Описание
fStudents: TStudents	Класс по работе со студентами
fBooks: TBook	Класс по работе с книгами
fStudentBook: TStudentBook	Класс, выдающий и забирающий книги

Согласно шаблону High Cohesion эти классы имеют высокое зацепление, поскольку каждый класс работает только с одним объектом в БД.

Для обработки входных системных событий необходимо создать методы, каждый из которых будет обрабатывать только одно системное событие (см. таблицу 6.2).

Таблица 6.2 - Методы класса TController

Название	Описание
AddStudent (num: integer; family, name, secondname, group: string)	Метод добавляет студента
AddBook (Author, Name: string)	Метод добавляет книгу
AddStudentBook	Метод выдает книгу студенту
UpdateStudent (num: integer; family, name, secondname, group: string)	Метод изменяет выбранного студента
UpdateBook (Author, Name: string)	Метод изменяет выбранную книгу
DeleteStudent	Метод удаляет выбранного студента
DeleteBook	Метод удаляет выбранную книгу
DeleteStudentBook	Метод принимает книгу от студента
StudentParams: tParams	Функция возвращает параметры текущего студента
BookParams: tParams	Функция возвращает параметры текушей книги
GroupList: TStringList	Функция возвращает список групп студентов
StudentsAfterScroll(DataSet: TDataSet)	Событие, происходящее при перемещении по набору данных, отображающих студентов

Для обеспечения отображения данных необходимо создать свойства, возвращающие ссылки на источники наборов данных (таблица 6.2). В результате диаграмма класса примет вид представленный, на рисунке 6.2.

Таблица 6.3 - Свойства TController

Название	Описание
StudentDataSource: TDataSource	Источник данных набора отображающих студентов
BookDataSource: TDataSource	Источник данных набора отображающих книги
StudentBookDataSource: TDataSource	Источник данных отображающий выданные книги

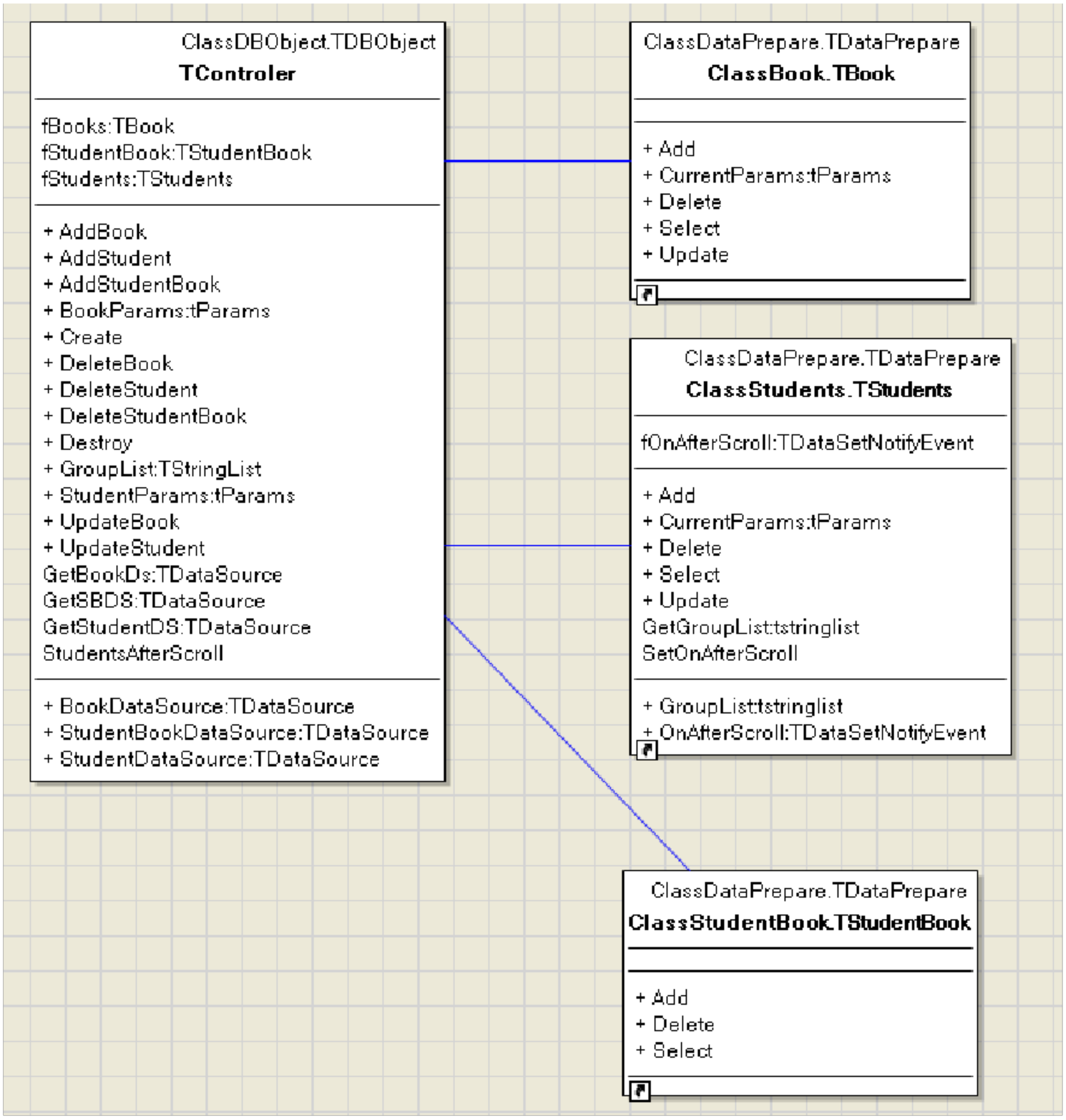

Рисунок 6.2 - Диаграмма классов слоя бизнес-логики

Задание и порядок выполнения работы

Разработать приложение в соответствии с тематикой. Приложение должно работать с реляционной СУБД, а его архитектура соответствовать трехслойной архитектуре на базе объектно-реляционного отображения с нетипизированными объектами. При разработке приложения необходимо применить шаблоны проектирования.

Контрольные вопросы

1. Что такое шаблон проектирования?
2. Какова структура шаблона проектирования?
3. Каков принцип разработки классов на основе шаблонов проектирования?
4. Шаблон проектирования Information Expert.
5. Шаблон проектирования Creator.
6. Шаблон проектирования Low Coupling.
7. Шаблон проектирования High Cohesion.
8. Шаблон проектирования Controller.
9. Чем отличается архитектура с типизированными объектами от архитектуры с нетипизированными объектами?

Работа с литературой:

Рекомендуемые источники информации (№ источника)

Основная	Дополнительная	Методическая	Интернет-ресурсы
1-3	1-3	1-3	1-2

Оценочные средства: контрольные вопросы

Тема 5. Технология MDA.

Лабораторная работа № 7. Разработка простого MDA-приложения

Форма проведения: Практическое выполнение задания

Цели работы:

- ознакомиться с архитектурой, управляемой моделью MDA, технологией ECO и языком OCL;
- научиться создавать простое приложение по технологии MDA.

Краткие теоретические сведения

Архитектура, управляемая моделью MDA

Концепция Model Driven Architecture (MDA) – *архитектура, управляемая моделью* – была разработана в независимой некоммерческой организации Object Management Group (OMG) – консорциум объектного управления.

Она объединяет сотни компаний-производителей программного и аппаратного обеспечения.

В технологии MDA основным элементом проектирования считается модель. *Модель – это описание реальной системы, учитывающее определенные характеристики или аспекты моделируемого объекта, процесса или явления.*

В конкретном проекте модель MDA представляет собой законченное и формализованное представление создаваемого программного продукта или его смысловой части. Высокая точность и непротиворечивость описания необходима, чтобы автоматизировать процесс перевода модели в программный (обычно исходный) код.

Понятие *«управление моделью»* подразумевает прямое использование модели при любых действиях по проектированию, разработке и развертыванию системы. При внесении изменений в архитектуру приложения модель считается первичной. Пусть, например, требуется пополнить описание класса новым полем или методом. В классическом (не модельном) подходе к разработке модификация описания класса выполнялась бы в исходном коде, ручным кодированием. В технологии MDA происходит модификация визуальной диаграммы, на которой класс представлен в виде графического элемента.

На базе такой диаграммы исходный код с измененным описанием класса генерируется автоматически.

Под *архитектурой, управляемой моделью,* понимается полная и законченная архитектура приложения, целиком формируемая с помощью модели.

В крупных проектах с моделью обычно работает не программист, а системный аналитик. Он отвлекается от мелких деталей реализации и перестает мыслить в терминах отдельных операторов языка программирования.

Хороший проектировщик способен охватить и продумать структуру больших функциональных логических блоков и основные взаимосвязи между ними.

Далее подобная модель детализируется и транслируется в исходный код на выбранном языке.

Технология MDA ориентирована на создание приложений, которые независимы от платформы, операционной системы и языков программирования. Она позволяет строить масштабируемые приложения из компонентов, которые могут использоваться повторно и многократно. Сам процесс разработки выполняется, как явствует из названия, под управлением модели.

Модель приложения – это взаимосвязанный набор визуальных диаграмм, наглядно описывающих внутреннюю структуру системы и принципы ее функционирования. Модель приложения не привязана к конкретному языку или конкретной среде программирования.

Визуальные диаграммы чаще всего строятся с помощью унифицированного языка моделирования UML.

Методология MDA описывает не столько моделирование, сколько метамоделирование, предусматривающее большую гибкость в конкретных, прикладных подходах к моделированию.

Метамоделирование – способ описания моделей, определяющий механизмы построения конкретных моделей программных систем с помощью базового словаря и набора ограничений, налагаемых на создаваемые модели.

Технология ECO

Корпорация Borland разработала собственную версию (реализацию) концепции MDA, которая называется Borland MDA (BMDA). Сегодня вместо термина BMDA применяется новый термин ECO. Технология ECO (*Enterprise Core Objects* – ключевые корпоративные объекты), реализующая концепцию MDA, стала наиболее важным улучшением последних версий среды Delphi. Она представлена в Delphi 2006 в виде третьей версии ECO III. Каждый компонент ECO представляет собой своеобразную программную обертку положений концепции MDA. Он является промежуточным слоем между средствами визуального проектирования программных моделей и их конкретной реализацией на языках программирования Delphi и C#.

Технология ECO переводит процесс согласования требований на модельный уровень. Диаграммы UML обычно хорошо воспринимаются и заказчиком, и исполнителем. Поэтому, хотя использование визуального языка моделирования в MDA не обязательно, почти 100 % проектов MDA создаются с применением языка UML.

Построение готового приложения происходит в несколько этапов. Сначала строятся платформно-независимые модели PIM (*Platform Independent Model*), затем выполняется их перенос в платформно-специфичные модели PSM (*Platform Specific Model*). Доступные на сегодняшний день продукты автоматизируют эти шаги на 50–70 %. Перенос моделей PSM в код конкретного языка программирования автоматизирован почти на 100 %.

Еще один подход, настоятельно рекомендуемый группой OMG при использовании подхода MDA, заключается в выделении логики приложения в отдельную, по возможности платформно-независимую, структуру. Достигается это использованием языка объектных ограничений OCL, который сегодня считается частью языка UML. Команды OCL встраиваются непосредственно в модель UML, описывая и уточняя аспекты ее работы. Удобство OCL еще и в том, что его выражения напоминают естественный язык (предложения, записанные на английском). Это позволяет сохранять при построении моделей контакт с людьми (например, представителями заказчика), не являющимися специалистами в области программирования.

Физические объекты, структура которых описана с помощью языка UML, а особенности существования – с помощью языка OCL, функционируют в процессе работы приложения в специально выделенной области оперативной памяти, так называемом *объектном пространстве*. Технология ECO способна автоматически отображать модель не только в программный код, но и в код на языке SQL. Он генерируется для построения схемы базы данных, хранящей копию объектного пространства. Поддержка разных СУБД является в ECO одной из ключевых технологий. С ее помощью удается сохранять текущее состояние объектного пространства приложения в базе данных и затем, после перезапуска приложения, загружать его и продолжать работу с прерванного места.

Среда Delphi на основе подготовленной модели автоматически формирует схему базы данных с учетом версии и специфики конкретной СУБД.

Она автоматически создает в ней все таблицы, нужные для хранения объектного пространства и взаимосвязей между классами, даже если это требует введения дополнительных таблиц. По мере совершенствования модели следует периодически синхронизировать схему базы данных с внесенными в модель модификациями, для этого достаточно нажать одну кнопку. Такой процесс называется в технологии ECO *эволюционным развитием базы данных*.

Стыковка объектного пространства с пользовательским интерфейсом реализуется в ECO с помощью *дескрипторов*. Дескриптор ECO – это стыковочная точка (компонент промежуточного уровня), связывающая с помощью выражений OCL элементы модели UML (классы) с программным кодом и внутренней структурой обычного приложения Delphi. Дескрипторы задают способы формирования наборов объектов в объектном пространстве по критериям, заданным выражениями OCL. Они также обеспечивают доступ к ним элементов пользовательского интерфейса, а программному коду предлагают ссылки на объекты ECO.

Как правило, ручное кодирование на языке Delphi требуется, если классы модели, построенной с помощью технологии ECO, включают методы, которые необходимо запрограммировать явно.

Серьезную роль в создании приложений ECO играет язык OCL. Он предоставляет программные средства обращения к пространству ECO и содержит средства создания,

модификации и уничтожения объектов, навигации по этому пространству и выполнения всевозможных вычислительных задач.

Язык объектных ограничений OCL

Архитектура MDA ставит на первое место модель приложения, и поэтому она непосредственно связана с языком, на котором такие модели создаются (язык унифицированного моделирования UML). Один из самых серьезных и справедливо критикуемых недостатков языка моделирования UML – предоставление разработчику только визуальных средств моделирования.

Эти средства абстрактны, поэтому далеко не всегда способны точно и формально отразить тот или иной нюанс функционирования проектируемой системы. Именно необходимость формализации описания условий и ограничений, накладываемых на элементы диаграмм классов, вызвало появление OCL (Object Constraint Language). Конечно, такие условия могут быть сформулированы и на естественном языке, однако он не будет являться строгим и может допускать неоднозначные трактовки.

Язык OCL не является языком программирования, т. е. не позволяет создать программу из своих операторов или описать логику выполнения каких-либо действий. Он создавался как формальный текстовый язык, дополняющий графические возможности языка UML. Выражение OCL обычно привязано к определенному классу и задает множество экземпляров этого класса. Команды OCL выполняют также фильтрацию этого множества или, например, определяют число его элементов.

Язык OCL содержит развитые средства манипуляции над множествами объектов, поэтому он хорошо подходит для решения задач, где обычно применяется язык запросов к реляционным базам данных SQL. Язык OCL позволяет организовывать запросы с большей эффективностью и на более высоком, модельном уровне абстракции, нежели язык SQL.

Выражение OCL фактически задает условие, которому должны удовлетворять все экземпляры соответствующего объекта UML. Результатом выполнения выражения OCL является множество объектов, удовлетворяющих этому условию.

При этом средствами языка OCL невозможно изменить сами элементы модели (классы, атрибуты, отношения). Среда, вычисляющая выражение OCL, просто определяет результирующее значение, которое может впоследствии использоваться (хотя это и не обязательно).

Язык OCL был разработан в корпорации IBM, в 1997 году вышла спецификация языка версии 1.1, в разработке и согласовании которой приняли участие такие компании, как Rational, Microsoft, Oracle, Hewlett-Packard и ряд других. Сильной стороной языка OCL оказалась независимость от платформы реализации и легкая адаптация к разным средствам программирования.

MDI-контейнеры

MDI – Multiple Document Interface (многодокументный интерфейс). В приложениях с MDI в основном (родительском) окне можно открыть более одного дочернего окна. Данная возможность обычно используется в электронных таблицах или текстовых редакторах.

Каждое MDI приложение имеет три основные составляющие:

- одну (и только одну) родительскую форму MDI;
- одну и более дочерних форм MDI;
- основное меню MDI.

Родительская форма должна быть стартовой, она нуждается, по крайней мере, в одной дочерней форме. Дочерние формы MDI – это простые формы, за исключением того, что их видимая часть ограничена размерами родительского окна. При минимизации такого окна оно помещается не в панель задач, а остается внутри родительского окна (на панель задач попадет только родительское окно).

Создание простого MDA-приложения

Рассмотрим пример создания простого MDA-приложения. Имеются две сущности Кафедра и Преподаватель. Представим экземпляры этих сущностей в таблицах формы. В этих же таблицах можно будет модифицировать значения отдельных полей представленных в них объектов (экземпляров Кафедра и Преподаватель).

Для разработки приложения применятся среда разработки Borland Developer Studio 2006. Приложение разрабатывается для платформы .NET Framework.

Основные этапы разработки приложения

1. Формируется модель UML будущего приложения. Создаются классы, описываются их атрибуты и методы, настраиваются взаимосвязи между ними.

2. В проект добавляются и настраиваются невизуальные компоненты, связывающие созданную модель UML с прикладной частью проекта.

3. Пространство ECO связывается с базой данных. В ней будет долговременно храниться его копия: все объекты ECO и взаимосвязи между ними.

4. Проектируется пользовательский интерфейс. Задействуются компоненты, обеспечивающие связь интерфейса с моделью UML.

5. Создается переносимая логика приложения на языке OCL. Элементы управления связываются с выражениями OCL для выполнения типичных стандартных действий (добавление, редактирование и удаление объектов ECO).

Обзор возможностей Borland Developer Studio 2006 для разработки MDA-приложения

1. После запуска BDS 2006 на экране открывается главное окно среды разработки (рисунок 7.1). Проект Delphi представляет собой группу файлов, содержащих исходный текст программы и вспомогательные данные. Все они объединены тематикой решаемой задачи.

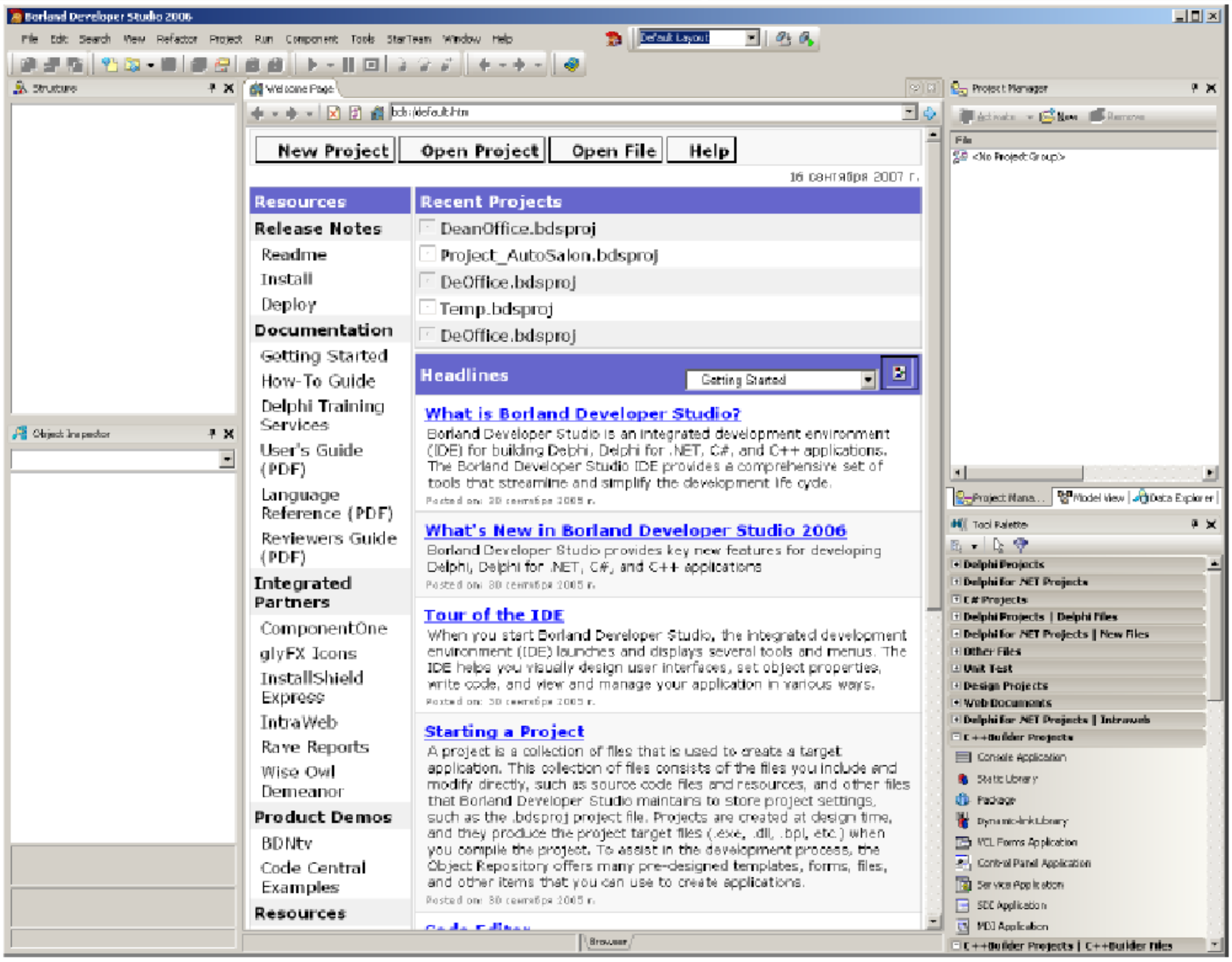

Рисунок 7.1 - Начало работы в среде BDS

2. Создадим пустую заготовку приложения ECO командой File > New > Other. Выберем значок ECO WinForms Application во вкладке Delphi for.NET Projects (рисунок 7.2).

3. В диалоговом окне введем имя проекта (например, projDeanOffice) и его местоположение (каталог). Нажмем кнопку OK.

4. Среда Delphi сформирует пустую заготовку приложения ECO и откроет окно Проектировщика. В нем расположена начальная пустая форма приложения. Структура автоматически созданной модели (пустой заготовки) доступна в окне просмотра модели, открываемом командой View > Model > View либо выбором вкладки Model View в правом верхнем окне (см. рисунок 7.3). В этом окне в дополнение к самому проекту (projDeanOffice) и главной форме (WinForm) можно увидеть еще несколько автоматически добавленных элементов. Среди них имеются следующие:

- элемент projDeanOfficeEcoSpace представляет объектное пространство ECO. Это основное хранилище, в котором располагаются экземпляры классов создаваемой модели во время

работы программы. Это пространство также ответственно за хранение объектов ECO, например, в базе данных или файле XML;

- элемент Package_1 представляет пакет классов UML, которые мы будем создавать.

Переименуем элемент Package_1 в удобное для нас название packModel.

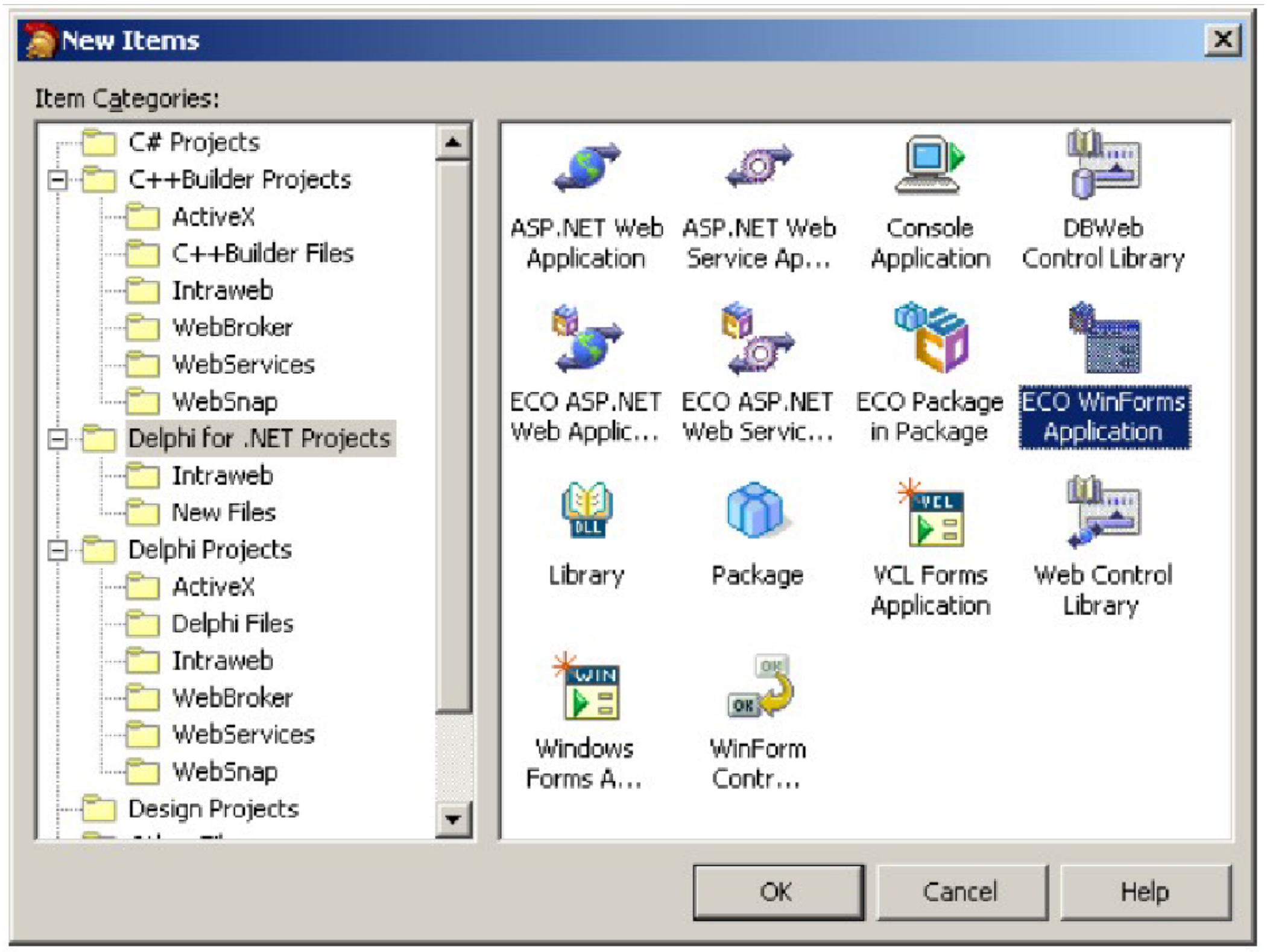

Рисунок 7.2 - Создание заготовки проекта ECO

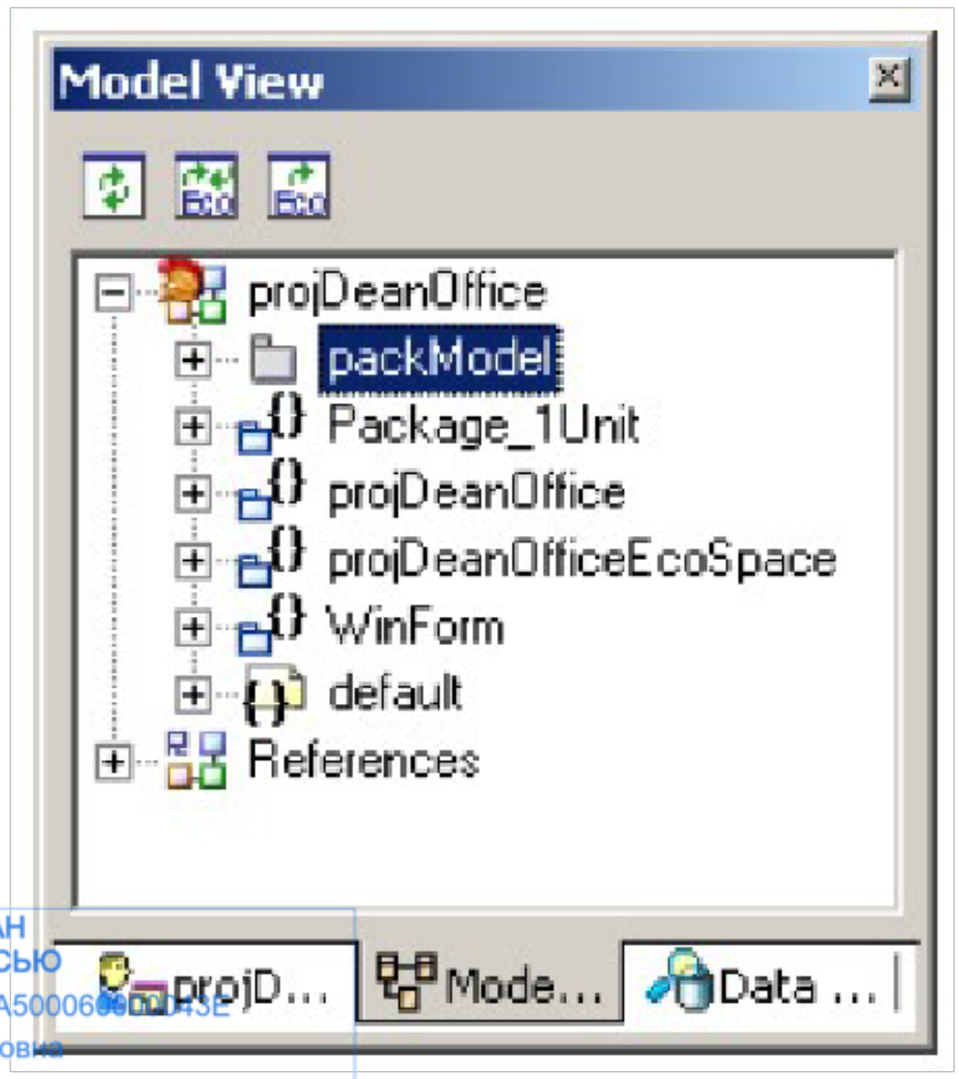

Рисунок 7.3 - Окно просмотра модели

Создание модели UML

1. Для создания модели дважды щелкнем мышью на строке packModel в окне просмотра модели. Откроется окно packModel [diagram] диаграммы классов ECO. Это окно напоминает окно построения обычных диаграмм классов UML, только при работе с ним применяются элементы, специфичные именно для технологии ECO.

2. Разместим на диаграмме первый класс, отражающий сущность Деканат. Для этого выберем на палитре инструментов Tool Palette инструмент ECO Class в категории UML ECO Class Diagram и щелкнем в подходящей точке пространства моделирования. В ней появится графический элемент, изображающий класс.

3. Назовем класс clChair (Кафедра). Пробелы в имени класса ECO не допускаются.

4. Добавим атрибуты класса командой контекстного меню Add > Attribute. Создадим таким образом три атрибута ChairName (Название кафедры), ChairHeadSNP (ФИО заведующего кафедрой), ChairSecrSNP (ФИО секретаря кафедры), рисунок 7.4. Для задания типа атрибута выделим нужный атрибут и в окне Properties установим необходимое значение поля Type в категории General. В данном случае все три атрибута имеют тип String.

5. Переименуем названия класса и атрибутов в понятные названия на русском языке. Для этого в свойстве Alias (категория General) класса и его атрибутов введем русскоязычные названия.

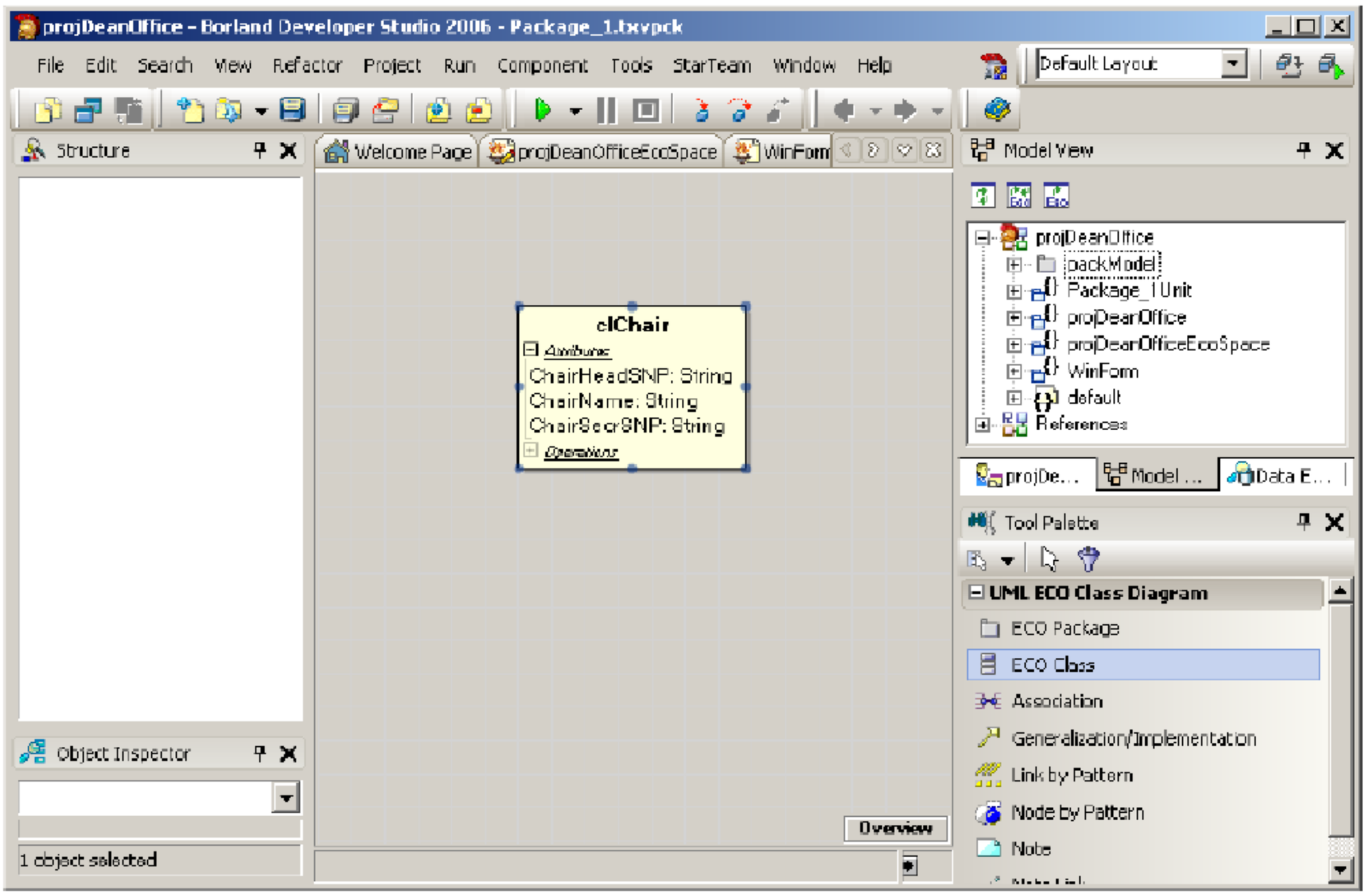

Рисунок 7.4 - Добавление класса clChair с его атрибутами

6. По аналогии добавим к модели еще один класс clLecturer (Преподаватель) с атрибутами LecturerSNP (ФИО преподавателя) и LectAcadDegree (Ученая степень) типа String. Переименуем элементы нового класса на диаграмме, изменив значения свойства Alias.

7. Настроим связи между созданными классами. Эта связь будет представлять отношение ассоциации. Выберем на палитре инструментов инструмент Generalization / Implementation. Щелкнем мышью на представлении класса Кафедра, протянем связь к классу Преподаватель и снова щелкнем мышью. В результате между двумя классами сформируется ассоциативное отношение.

8. Настроим мощность ассоциативного отношения. В нашем случае одному экземпляру класса Кафедра соответствует множество экземпляров класса Преподаватель (от 1 и более). Для этого в окне Properties выделенной связи выберем категорию End1, соответствующую классу Кафедра, и в поле Multiplicity установим значение 1, а в категории End2 (для класса Преподаватель) этому же полю – значение 1..*. Теперь дадим сторонам связи названия. В свойство Name для сторон End1 и End2 введем roleChair и roleLecturers соответственно (рисунок 7.5). Таким

образом, мы назвали роли каждого класса в ассоциативной связи. Эти имена пригодятся нам при кодировании на языке OCL.

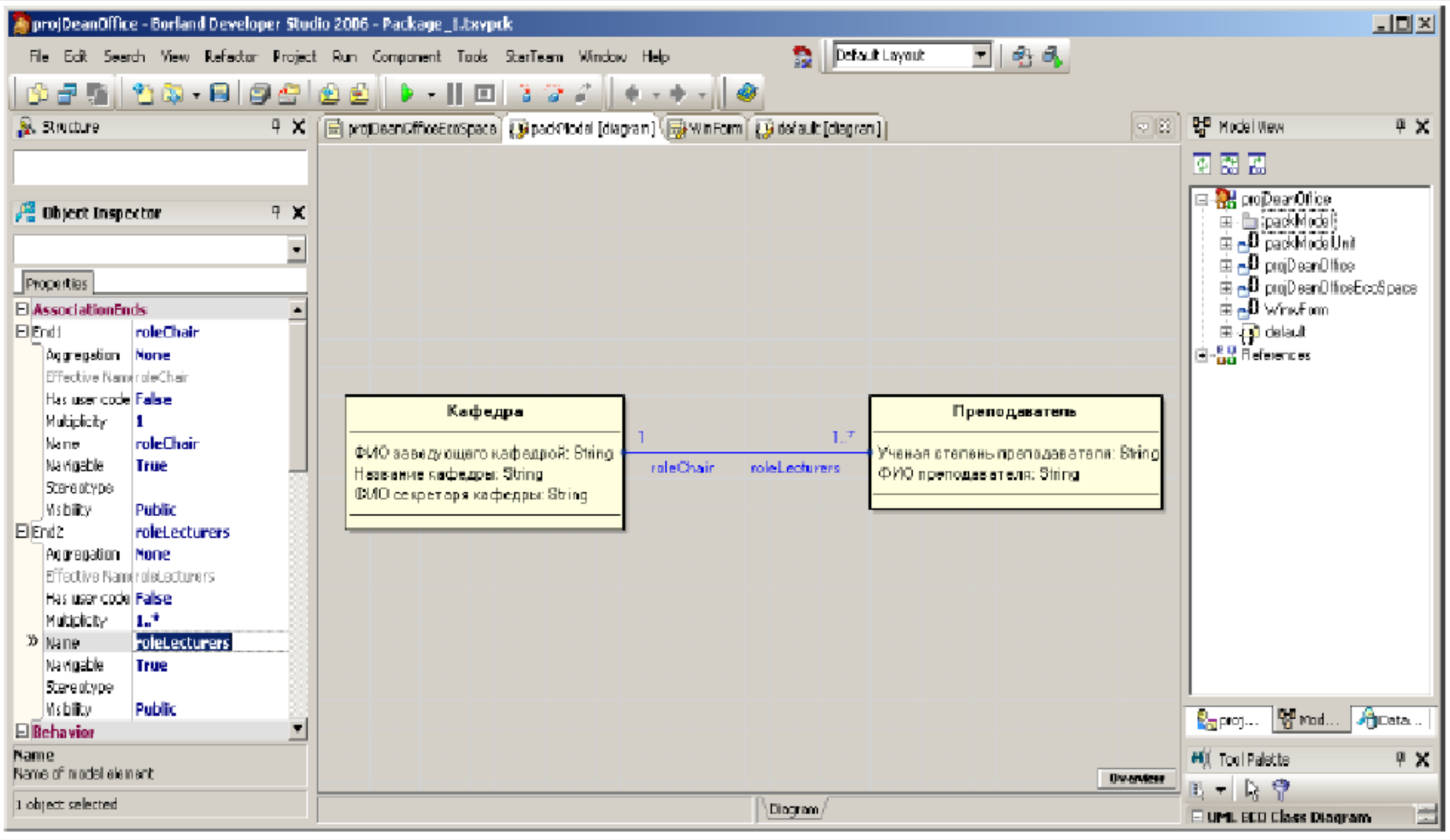

Рисунок 7.5 - Настройка связи между классами

На этом этапе создание простейшей модели закончено. Теперь надо создать базу данных и подключить ее к проекту.

Создание БД и настройка ECO компонент

1. Перейдем на закладку projDeanOfficeEcoSpace рабочего окна Delphi (рисунок 7.6). На этой закладке настраивают особенности функционирования объектного пространства проекта. Пока объектное пространство пусто – оно работает стандартным способом. Элементы модели доступны в других окнах в виде диаграмм UML.

2. Выберем в палитре инструментов компонент BdpConnection из категории Borland Data Provider, предназначенный для связи с СУБД, и добавим его в проект.

3. Настроим этот компонент на доступную базу данных. В качестве учебного примера пропишем путь к пустой базе данных DeOffDB.gbd выбранной СУБД.

4. В свойстве ConnectionString компонента BdpConnection выберем имя соединения IBConn1.

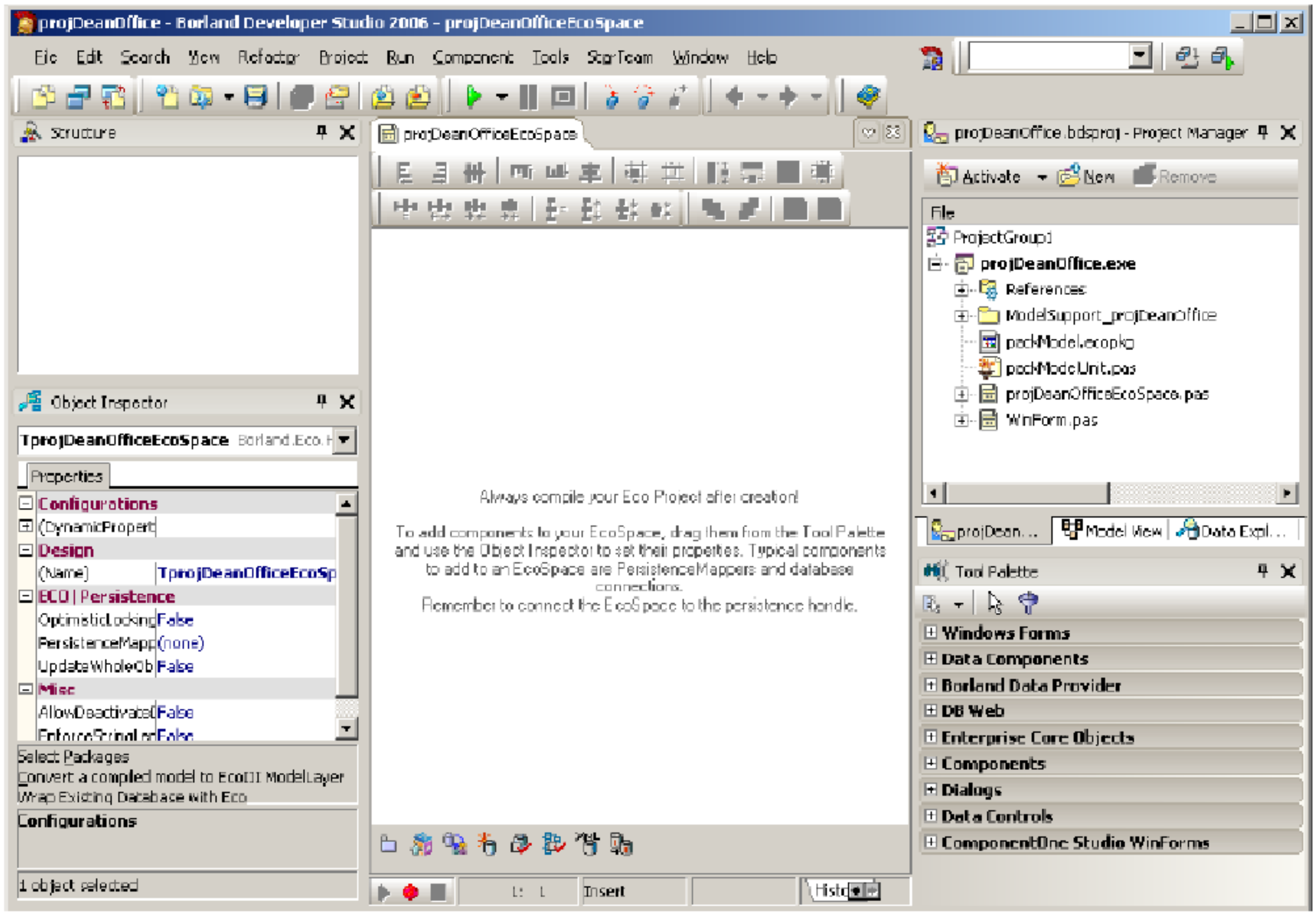

Рисунок 7.6 - Окно настройки объектного пространства

5. Настроим автоматическую связь модели MDA с выбранной СУБД. Синхронизация содержимого пространства ECO с данными на внешних носителях (файлах или базах данных) выполняется компонентами, наследующими базовые характеристики класса PersistenceMapper. Этот класс отвечает за отображение структуры объектного пространства в схемы представления данных, например реляционные. Добавим к проекту компонент PersistenceMapperBdp из категории Enterprise Core Objects. Он предназначен для раскладки объектного пространства в схемы баз данных, доступ к которым происходит по технологии BDP.NET. Этот компонент располагается в окне projDeanOfficeEcoSpace рядом с объектом BdpConnection1 (экземпляром компонента BdpConnection).

6. Свяжем компонент PersistenceMapperBdp через свойство Connection с объектом BdpConnection1 (рисунок 7.7). Компонент PersistenceMapperBdp полностью ответствен за все аспекты автоматического сохранения и загрузки копии объектного пространства модели в выбранную базу данных. Фактически, добавив компонент PersistenceMapperBdp к проекту, мы реализовали все основные аспекты взаимодействия приложения MDA с базой данных.

7. Сгенерируем схему базы данных. Компонент PersistenceMapperBdp самостоятельно создает в выбранной базе данных специальные таблицы, поля и отношения между ними. В этих таблицах хранятся объекты пространства ECO. В контекстном меню компонента PersistenceMapperBdp выберем пункт, обеспечивающий автоматическую настройку всех нужных свойств для конкретной СУБД. В случае использования СУБД Interbase нужный пункт имеет вид Interbase [dialect3] setup, в случае СУБД Microsoft SQL Server – SQL Server setup и так далее. В нашем случае выбираем пункт Interbase [dialect3] setup, так как используется СУБД Firebird 2.0 (рисунок 7.8).

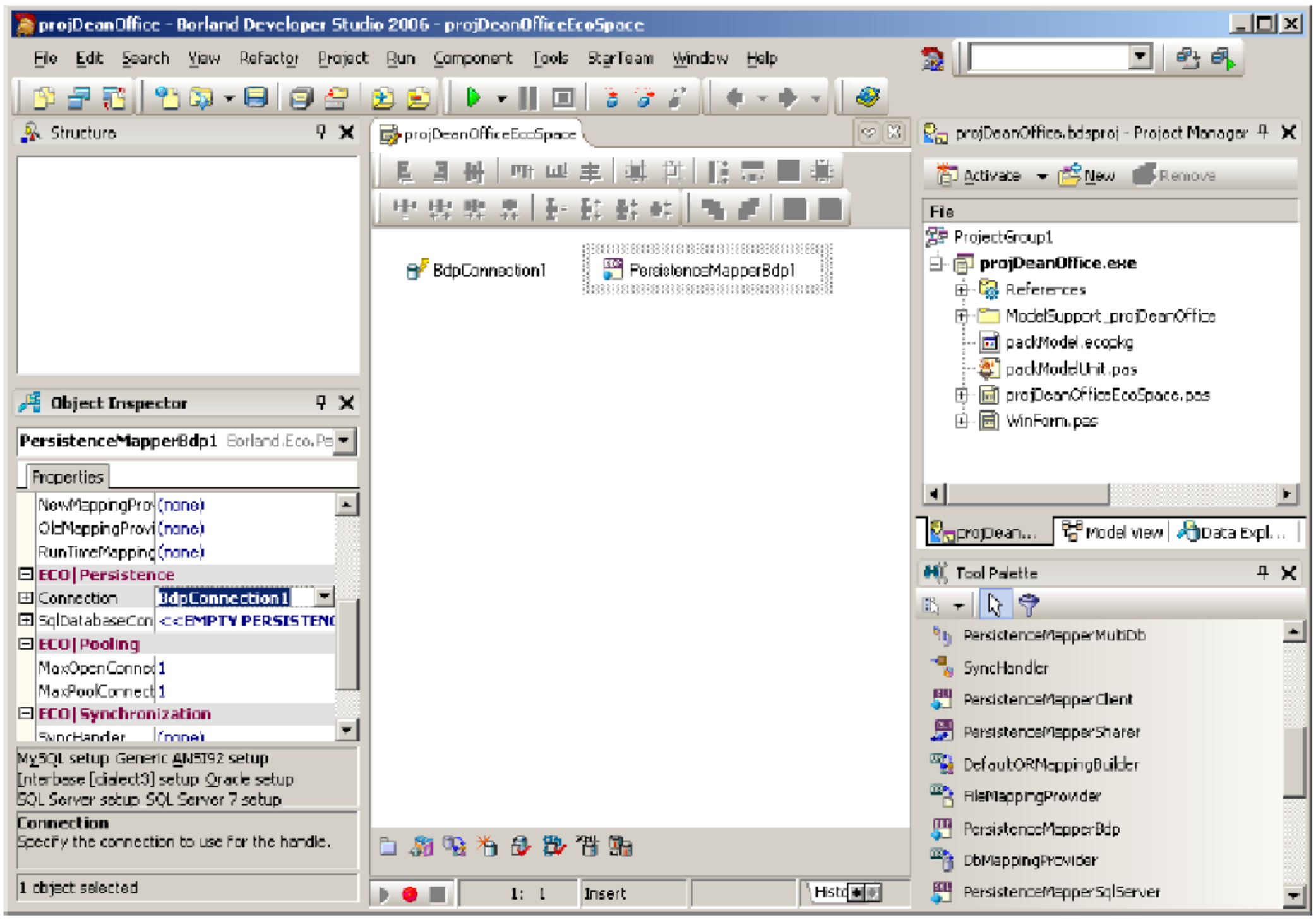

Рисунок 7.7 - Настройка связи ECO с СУБД

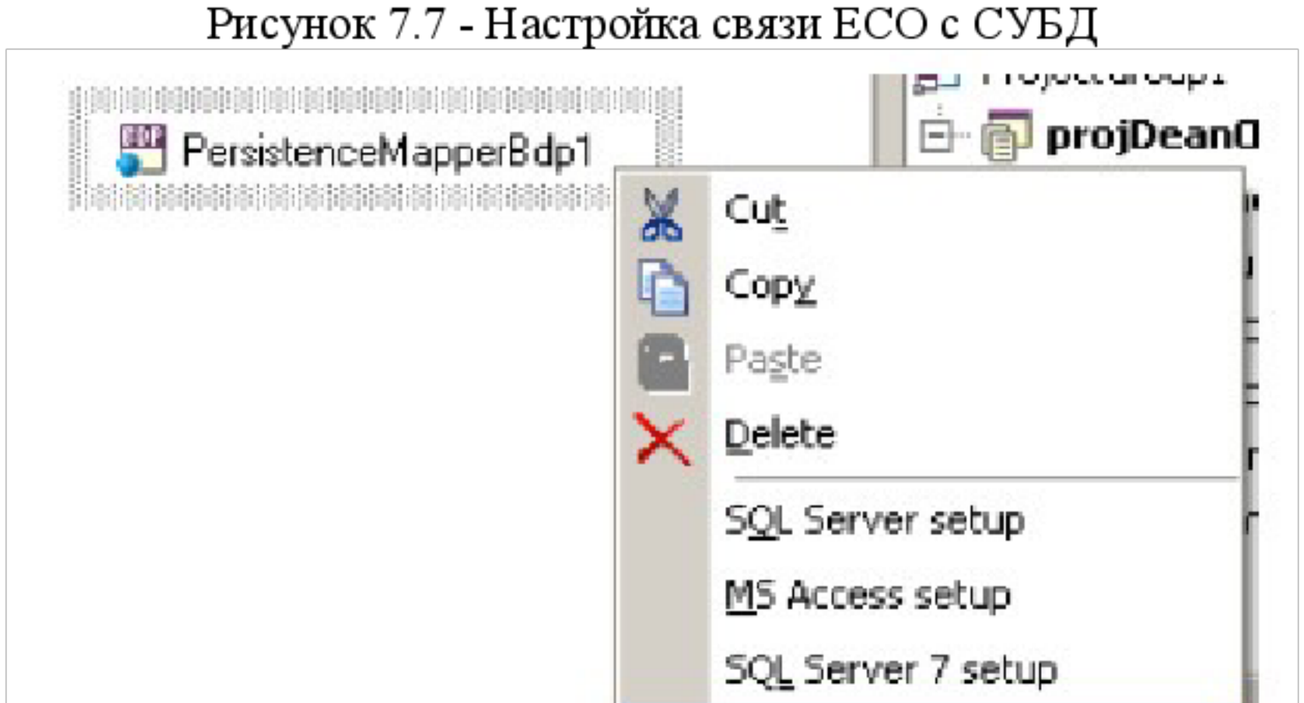

Рисунок 7.8 - Настройка свойств СУБД

8. Сгенерируем таблицы, описывающие структуру объектного пространства. Для этого нажмем кнопку Generate Schema в нижней части окна, представляющего пространство ECO. В текущей базе данных автоматически сформируются таблицы, содержащие поля для хранения модели проекта (рисунок 7.9).

9. В ходе генерации схемы среда Delphi проверит содержимое базы данных. Если в ней имеются таблицы, не имеющие отношения к текущей модели, будет предложено их удалить. По умолчанию режим удаления лишних таблиц выключен.

10. Компонент PersistenceMapperBdp, настроенный на конкретную СУБД, надо задействовать как связующее звено между объектным пространством ECO и СУБД. Для этого обратимся к текущему объектному пространству проекта и щелкнем на свободном месте окна projDeanOfficeEcoSpace.

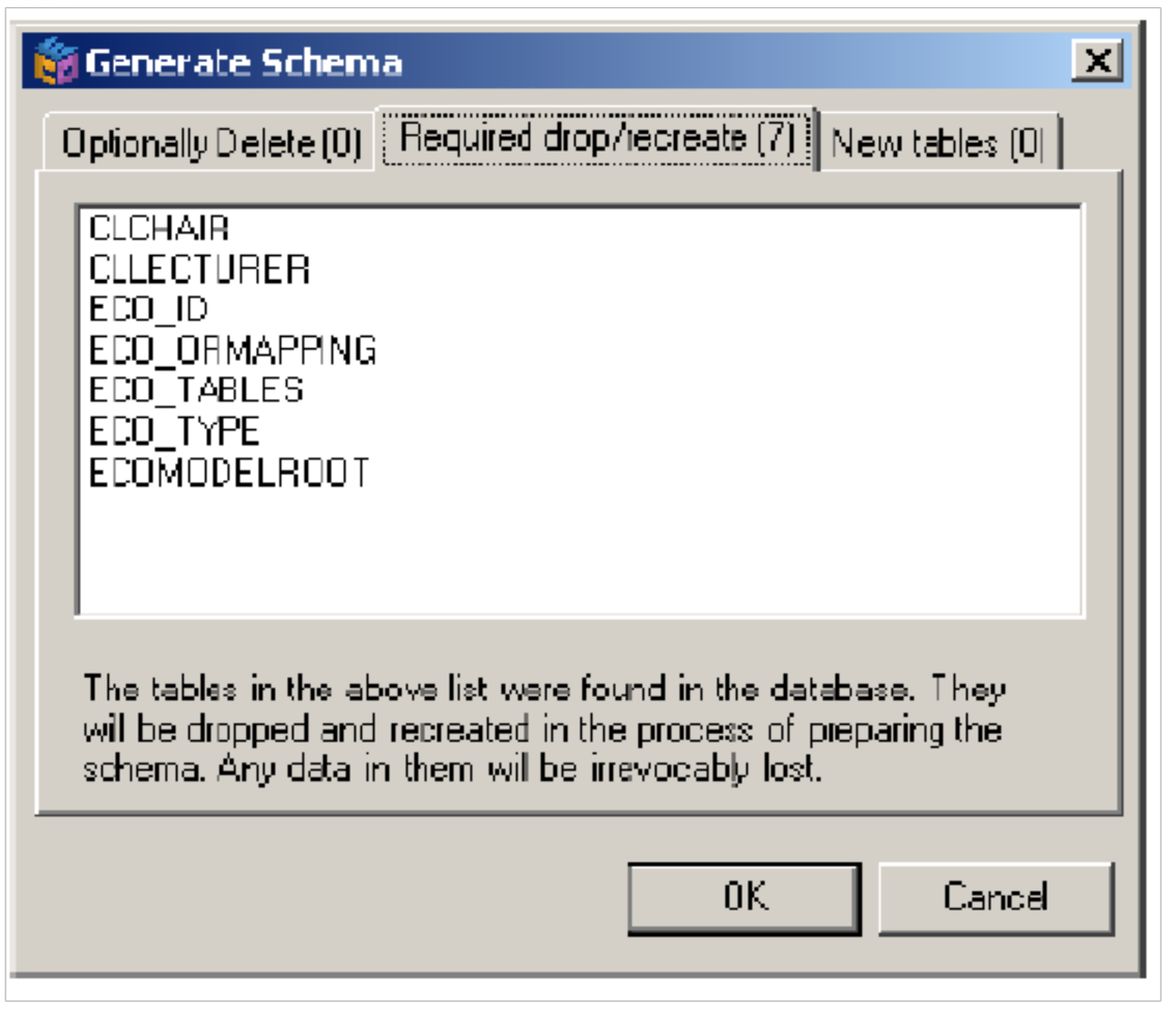

Рисунок 7.9 - Автоматическая генерация схемы модели

В свойстве PersistenceMapper в окне Инспектора объектов Properties для объектного пространства зададим ссылку на объект PersistenceMapperBdp1 (единственный экземпляр компонента PersistenceMapperBdp). Связь объектного пространства с СУБД установлена.

Далее нам необходимо представить создаваемые объекты модели (экземпляры классов Кафедра и Преподаватель) в таблице. Объект следует добавлять в таблицу, удалять его и обновлять БД с помощью кнопок Добавить, Удалить и Сохранить соответственно. Проделаем всю работу исключительно с помощью визуальных средств Delphi, не прибегая к ручному программированию.

Создание интерфейса

1. Перейдем к окну Проектировщика для главной формы проекта WinForm (вкладка Design). Переименуем стандартное название главной формы. Назовем ее wfMain, а сам класс TWinForm переименуем в TMain в свойстве Name категории Design. Свойство Text (категория Appearance) отвечает за название заголовка окна, введем в него строку, например Главная форма. Это окно станет родительским в разрабатываемом приложении, согласно требованиям создания MDI-контейнеров.

2. В свойстве IsMdiContainer (категория Window Style) Главной формы выберем значение True.

3. Создадим новую форму командой File > New > Other. Выберем значок ECO Enabled Windows Form из вкладки New ECO Files (категория Delphi for .NET Projects). В окне Проектировщика появится новая форма. Эта форма будет дочерней, а значит, она будет отображаться внутри главной. Так как в ней будем представлять список преподавателей выбранной кафедры, назовем созданную форму wfLecturer, а сам класс – TLecturer. Введем в свойство формы Text строку Преподаватели.

4. Создадим главное меню на родительской форме для обращения к дочерним формам.

Поместим на Главную форму компонент MainMenu. В верхней части формы появиться строка меню, в начале которой находится область ввода текста. Чтобы создать первый пункт меню, надо щелкнуть в области ввода текста и ввести название пункта меню. Назовем пункт меню Подсистемы. В нижнюю область пункта Подсистемы введем название команды меню – Преподаватели (рисунок 7.10).

5. Выберем форму wfLecturer, вкладку Code. В окне появится код формы. Перед командой Implementation объявим переменную, ответственную за создание дочернего окна Преподаватели:

```
var
callLect: TLecturer;
```

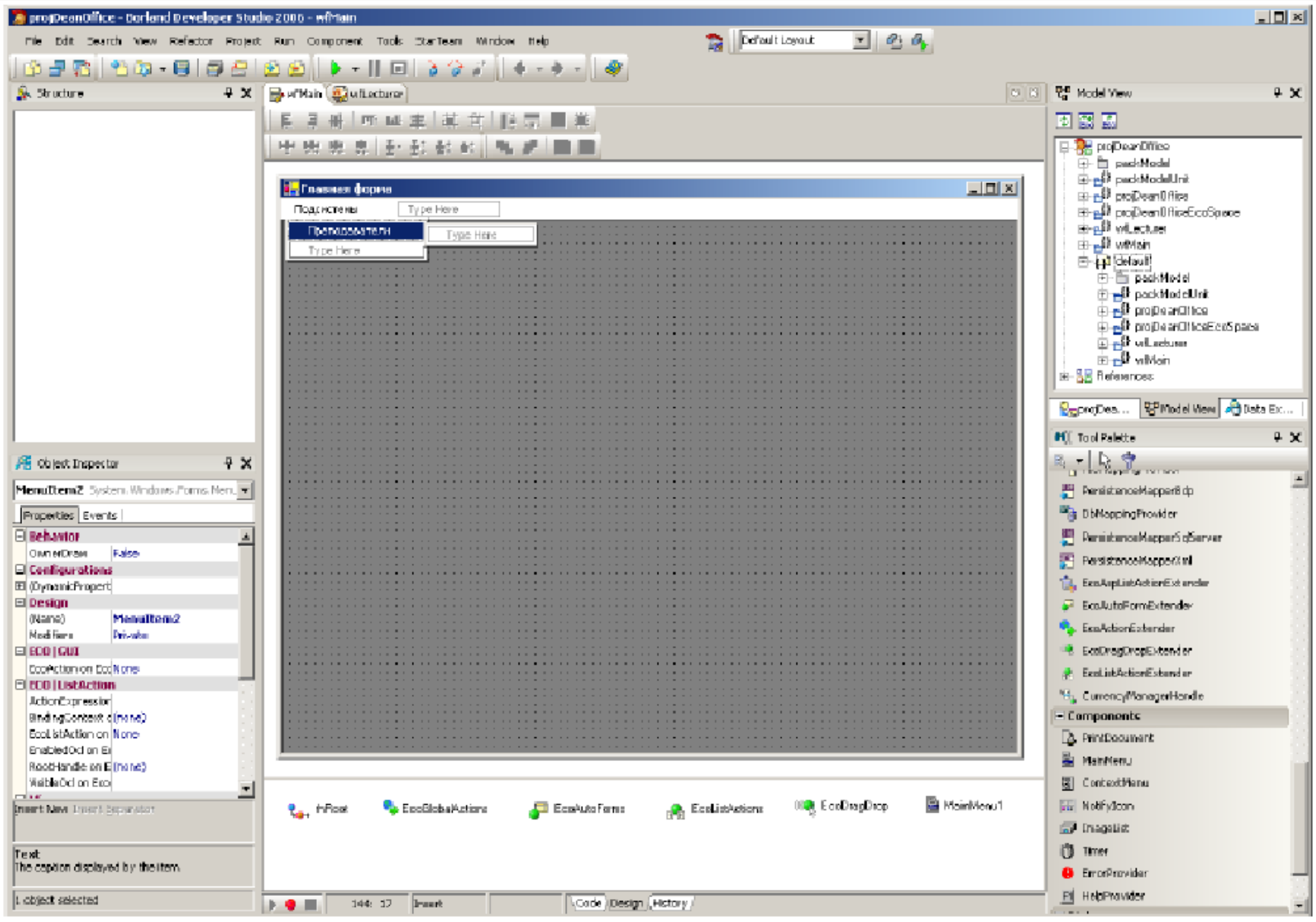

Рисунок 7.10 - Настройка главного меню

6. Перейдем к Главной форме. Чтобы в Главной форме создать дочернюю, подключим к ней форму wfLecturer командой File > Use Unit. В появившемся окне Use Unit выберем форму wfLecturer (Преподаватели). На событие выбора пункта меню Преподаватели пропишем следующие операции:

```
callLect := TLecturer.Create(EcoSpace);
callLect.MdiParent := self;
callLect.Show;
```

7. Запустим приложение. Теперь при нажатии на пункт Преподаватели внутри родительского окна появляется дочернее окно Преподаватели (см. рисунок 7.11).

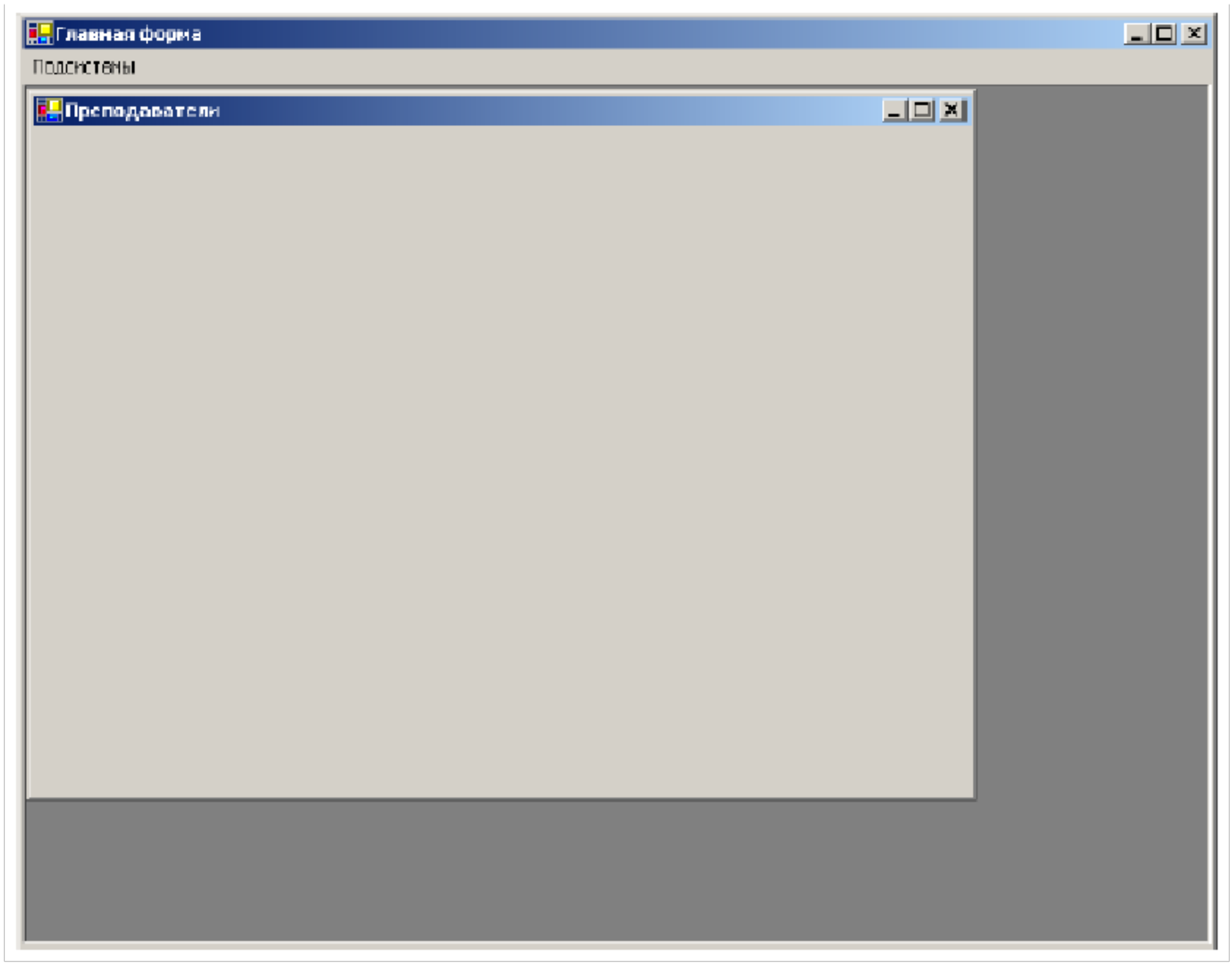

Рисунок 7.11 - Создание дочернего окна внутри родительского

8. Приступим к представлению таблицы с объектами ECO на форме Преподаватели. Воспользуемся готовым компонентом DataGrid из категории палитры инструментов Data Controls. Поместим этот компонент на форму и дадим ему название dgChair (в свойстве Name). Эта таблица будет отвечать за представление экземпляров класса Кафедра (clChair). Исходно таблица должна быть пуста. В свойстве CaptionText (заголовок таблицы) введем название Кафедры.

9. Добавим еще одну таблицу dgLecturer, которая в готовом приложении будет отвечать за отображение экземпляров класса Преподаватель (clLecturer). В свойстве CaptionText введем название Преподаватели.

10. Для работы с таблицами в простом приложении потребуется пять операций: добавление и удаление данных в двух таблицах и сохранение копии ECO пространства из оперативной памяти в базу данных. Редактирование данных будет осуществляться непосредственно в полях таблиц. Добавим на форму пять кнопок – экземпляров класса Button (рисунок 7.12).

11. Настроим автоматическое растягивание таблиц и положение кнопок при изменении размеров окна Преподаватели. Воспользуемся свойством Anchor у объектов формы. Для таблицы Кафедры зададим значения Top, Left, Right в свойстве Anchor, для таблицы Преподаватели – Top, Bottom, Left, Right, для кнопок Button1–Button4 – Top, Right, для кнопки Button5 – Bottom, Right.

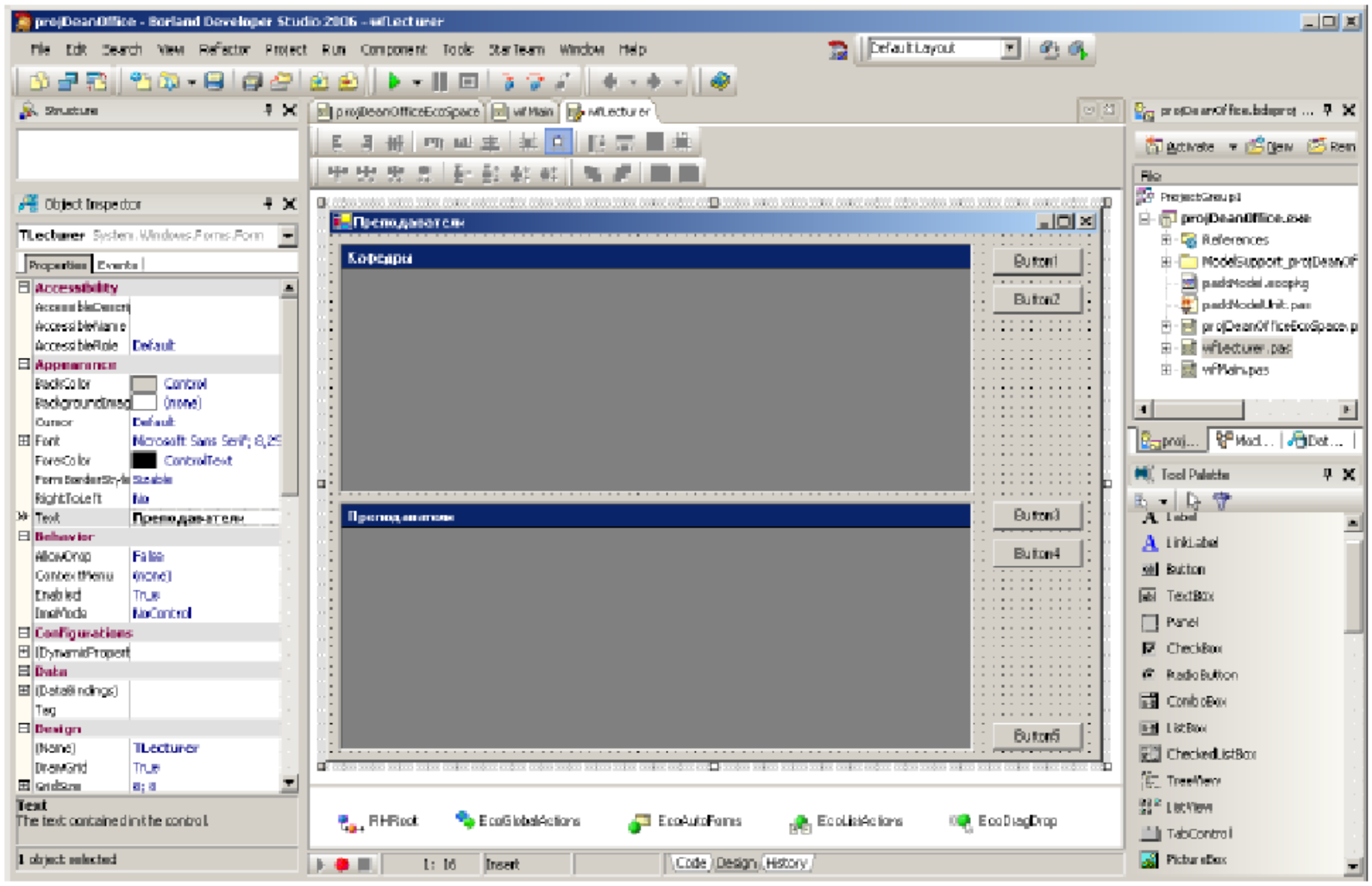

Рисунок 7.12 - Добавление компонентов пользовательского интерфейса

Связывание интерфейса с моделью

1. Связь пользовательского интерфейса с моделями ECO обычно происходит через компонент ExpressionHandle. Он доступен в категории палитры инструментов Enterprise Core Objects. Через подобные идентификаторы (дескрипторы) организуется доступ к объектам модели и пространства ECO во время работы программы.

Идентификаторы ECO связываются друг с другом в цепочки. Во главе такой цепочки расположен корневой идентификатор. Он задает общий контекст работы (доступ к объектному пространству) для всех остальных идентификаторов ECO в программе.

Корневой идентификатор добавляется к проекту автоматически (по умолчанию он называется rhRoot). Все остальные идентификаторы ECO добавляются и настраиваются вручную, что связывает пространство ECO с элементами пользовательского интерфейса с помощью типовых механизмов связывания .NET.

Будем работать с формой Преподаватели. Добавим компонент ExpressionHandle. Он отображается в нижней части окна Проектировщика под формой, где уже имеется набор компонентов ECO. Назовем его ehChair, введем это имя в свойство Name категории Design.

2. В свойстве RootHandle этого дескриптора выберем значение RHRoot – ссылку на родительский, автоматически созданный корневой идентификатор. Так задают место данного идентификатора в цепочке доступа к объектному пространству ECO.

3. В свойстве DataSource таблицы dgChair выберем имя ehChair в списке доступных идентификаторов ECO.

4. Настроим таблицу Преподаватель так, чтобы она отражала список преподавателей, принадлежащих выбранной кафедре в первой таблице. Для начала добавим в проект дескриптор CurrencyManagerHandle (из категории Enterprise Core Objects) и дадим ему имя cmhChair, так как этот компонент будет указывать на текущую строку таблицы Кафедры. Свяжем дескриптор cmhChair с таблицей dgChair через свойство BindingContext. Теперь он отслеживает выделенную строку этой таблицы.

5. Зададим ссылку на объект ehChair в свойстве RootHanler, определяющем корневой идентификатор ECO.

6. Добавим к проекту еще один компонент ExpressionHandle. Назовем его ehLecturer. Дескриптор ehLecturer будет обращаться к экземплярам класса Преподаватель.

7. Зададим ссылку на объект cmhChair в свойстве RootHanler.

8. Потребителем объектов, предоставляемых дескриптором ehLecturer, будет вторая таблица. В ее свойстве DataSource выберем ссылку на объект ehLecturer.

9. Приступим к настройке элементов пользовательского интерфейса. Организуем с помощью визуальных средств Delphi создание новых экземпляров класса Кафедра и добавление их в таблицу. Выделим в окне Проектировщика кнопку Button1. В свойстве EcoListAction (Список стандартных действий ECO) выберем значение Add (Добавить объект).

10. Кнопка Button1 автоматически получила название Add. Переименуем кнопку. Для этого воспользуемся компонентом EcoListActions (он добавляется в проект по умолчанию и находится в нижней части окна Проектировщика), в его свойстве CaptionAdd введем Добавить.

11. Объект ECO, который мы хотим добавить к проекту (сделать доступным через элементы управления на форме), задается в свойстве RootHandle. В нем надо выбрать подходящий поставщик объектов ECO, в нашем случае – идентификатор ehChair.

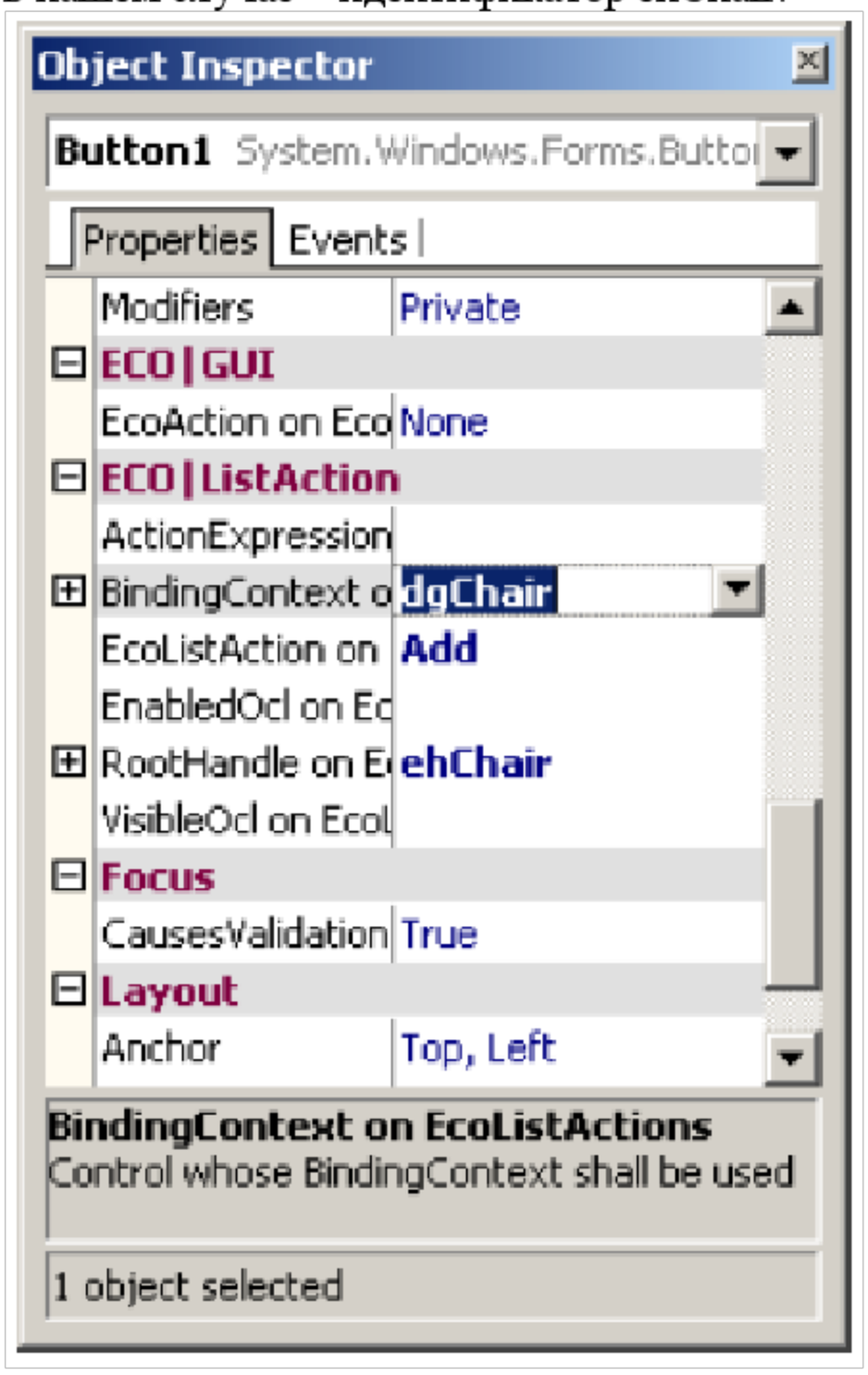

Рисунок 7.13 - Настройка кнопки добавления экземпляра класса Кафедра

12. Свяжем результат действия кнопки (созданный экземпляр класса Кафедра) с визуальным элементом, отображающим этот экземпляр, в нашем случае – с таблицей dgChair. Для этого в свойстве BindingContext (контекст связывания) выберем имя dgChair (см. рисунок 7.13).

13. Теперь добавим операцию удаления экземпляров класса Кафедра. Реализуем это действие по аналогии с операцией добавления. Выделим в окне Проектировщика кнопку Button2. В свойстве EcoListAction выберем значение Delete. Зададим идентификатор ehChair в свойстве RootHandle. Свяжем результат действия кнопки с визуальным элементом – таблицей dgChair. Для этого в свойстве BindingContext у кнопки Delete выберем имя dgChair. Перейдем к компоненту EcoListActions и в его свойстве Caption Delete введем Удалить.

14. Настройка кнопки Button3 (она будет отвечать за добавление экземпляра класса Преподаватель). Значения свойств кнопки: EcoListAction – Add; RootHandle – ehLecturer; BindingContext – dgLecturer. Перейдем к компоненту EcoListActions и в его свойстве Caption Add введем Добавить.

15. Настройка кнопки Button4 (будет отвечать за удаление экземпляра класса Преподаватель). Значения свойств кнопки: EcoListAction – Delete; RootHandle – ehLecturer; BindingContext – dgLecturer. Перейдем к компоненту EcoListActions и в его свойстве Caption Delete введем Удалить.

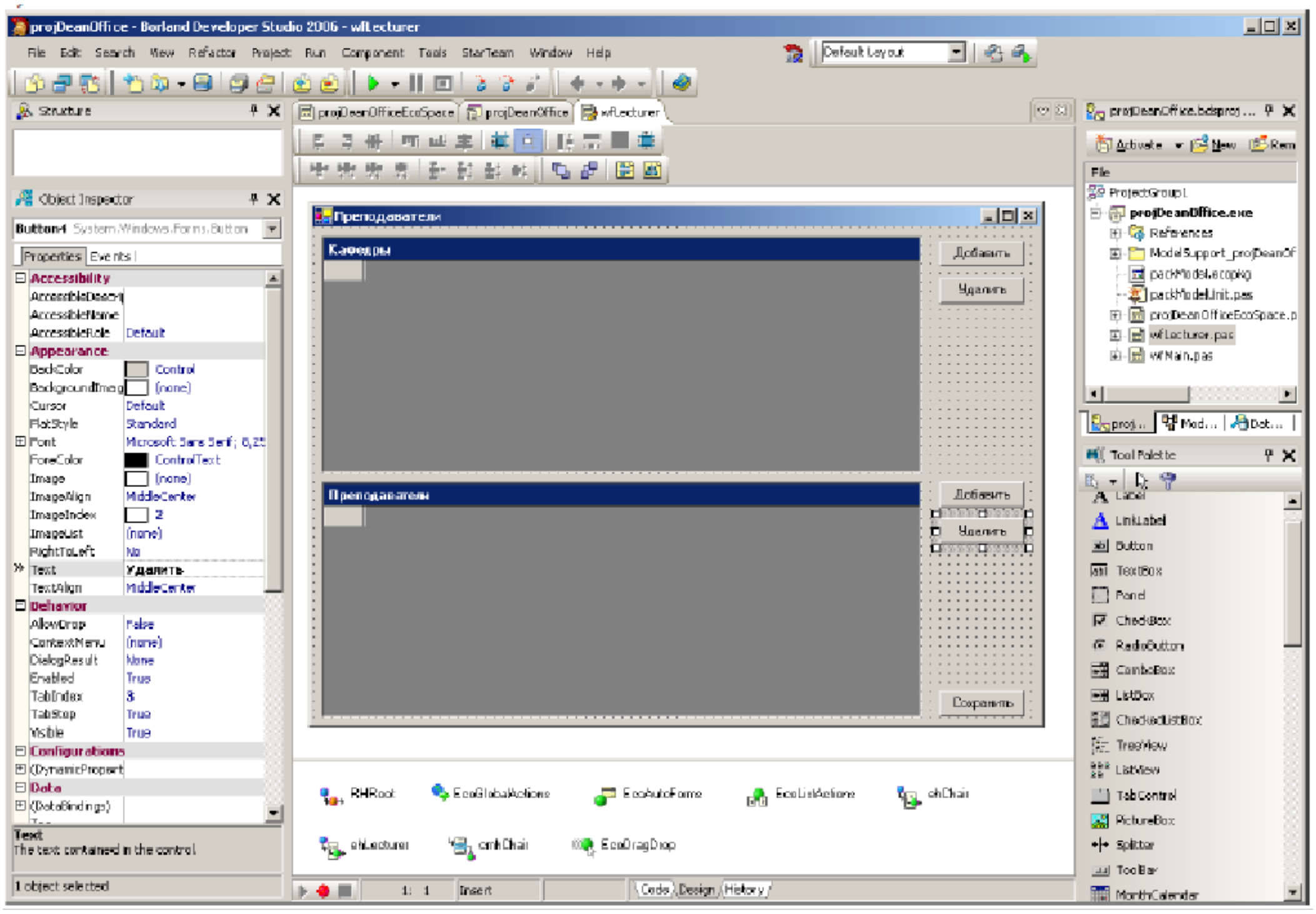

Рисунок 7.14 - Настроенный пользовательский интерфейс приложения

16. Кнопка Button5 будет отвечать за обновление БД. Назовем ее Сохранить. В свойстве EcoAction (категория ECO|GUI) выберем UpdateDatabase. Таким образом, по нажатию этой кнопки выполняется синхронизация содержимого пространства ECO в оперативной памяти и его копии в базе данных. После каждого нажатия кнопки Сохранить содержимое таблицы сохраняется в БД. После нового запуска программы в таблицу пользовательского интерфейса автоматически загружается содержимое модели, сохраненное при последнем выполнении команды UpdateDatabase.

На данном этапе связывание интерфейса с моделью закончено (см. рисунок 7.14).

Создание логики на OCL

1) Действия, которые выполняет дескриптор в программе, определяются значением свойства Expression. В это свойство вводится выражение языка OCL, которое определяет схему создания объектов модели ECO.

Используем дескриптор ehChair для обращения к экземплярам класса Кафедра. Введем в свойство Expression объекта ehChair строку clChair.allinstances. Иначе это можно сделать с помощью редактора OCL - выражений. Введенное выражение задает доступ ко всем текущим экземплярам класса Кафедра в объектном пространстве. В таблице, показанной на форме в окне Проектировщика, сразу же отобразятся поля класса Кафедра (ChairHeadSNP, ChairSecrSNP и ChairName) (в том числе поля родительского класса, если он имеется). Дескриптор ehChair становится поставщиком данных нашей модели ECO для первой таблицы.

2. В качестве выражения OCL объекта ehLecturer введем строку self.roleLecturers в свойство Expression. Здесь roleLecturers – это введенное нами при построении модели имя роли класса Преподаватель (clLecturer) в его ассоциативной связи с классом Кафедра (clChair). Это выражение определяет группу преподавателей, принадлежащих кафедре, которая выбрана в первой таблице. Дескриптор ehLecturer становится поставщиком данных для второй таблицы (см. рисунок 7.15).

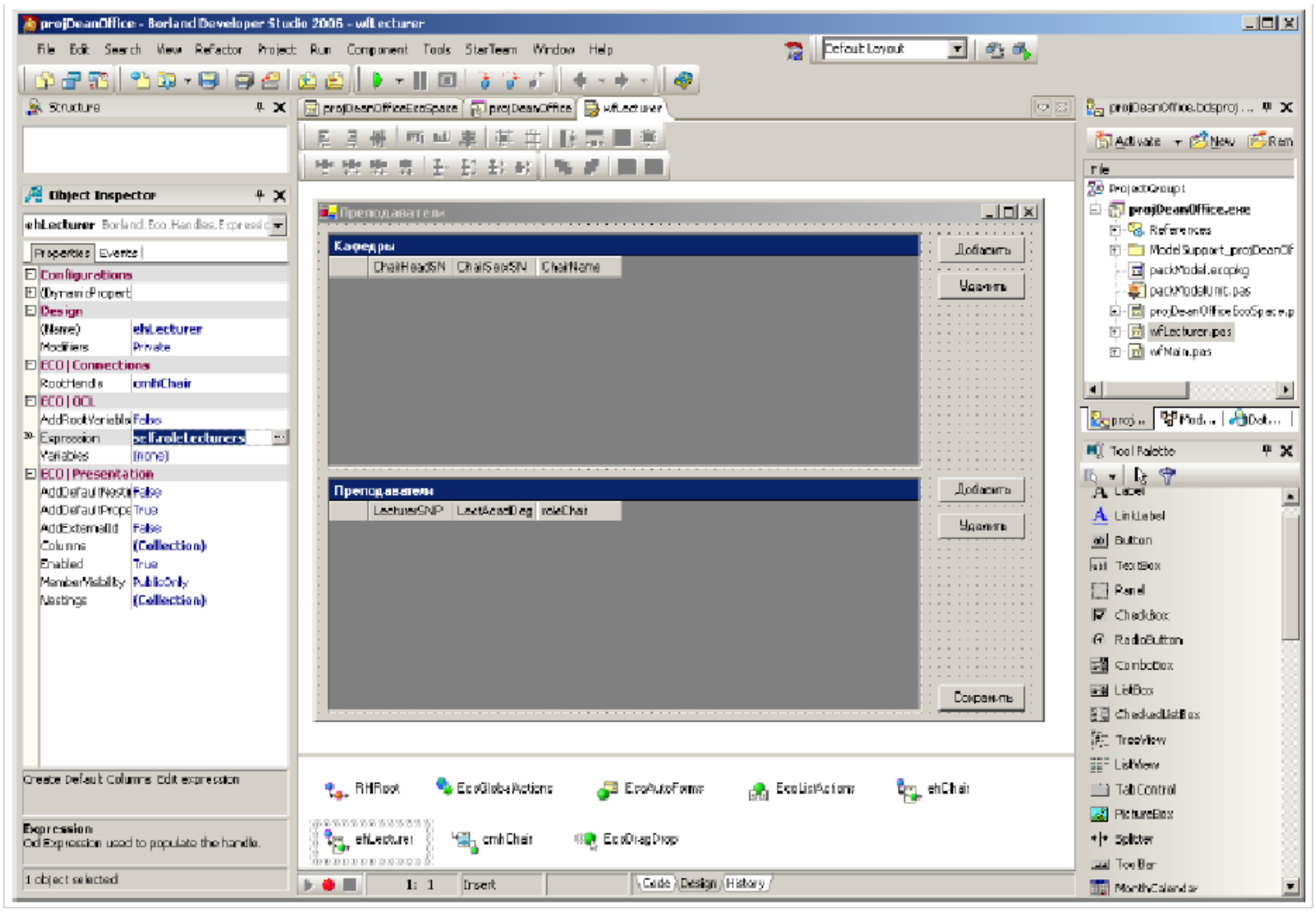

Рисунок 7.15 - Настройка OCL-выражений для взаимосвязанных таблиц

3. Дадим заголовкам полей таблиц русскоязычные названия. Выберем на форме таблицу dgChair. Щелчком на кнопке с многоточием в строке свойства TableStyles откроем окно редактора коллекции стилей таблицы DataGridTableStyle Collection Editor. Создадим новый стиль таблицы, нажав на кнопку Add (см. рисунок 7.16).

Щелкнем по кнопке с многоточием в строке свойства GridColumnStyles. Откроется окно редактора стиля поля таблицы. Нажмем кнопку Add 3 раза, так как нужно переименовать три поля (создать три стиля для имеющихся полей таблицы). Выберем свойство HeaderText первого элемента коллекции и введем в него стоку Название кафедры. Затем в свойстве Width установим ширину поля – 150. И в свойстве MapingName выберем ChairName, т. е. первый столбец настраиваемой таблицы будет содержать информацию атрибута ChairName класса Кафедра (см. рисунок 7.17).

Теперь настроим оставшиеся два элемента коллекции. Эти столбцы будут называться ФИО заведующего кафедрой и ФИО секретаря кафедры. Нажмем кнопку ОК в обоих открытых окнах и убедимся, что поля таблицы получили новые названия.

4. По аналогии настроим стиль таблицы Преподаватели. В ней отобразим два поля: LecturerSNP и LectAcadDegree. Назовем их ФИО преподавателя и Ученая степень.

5. Запустим программу и убедимся, что при нажатии кнопки Добавить в таблице автоматически формируются новые строки, отражающие содержимое новых экземпляров класса из объектного пространства ECO. В них можно ввести нужные значения. Проверим корректное функционирование остальных настроенных операций над объектным пространством (см. рисунок 7.18).

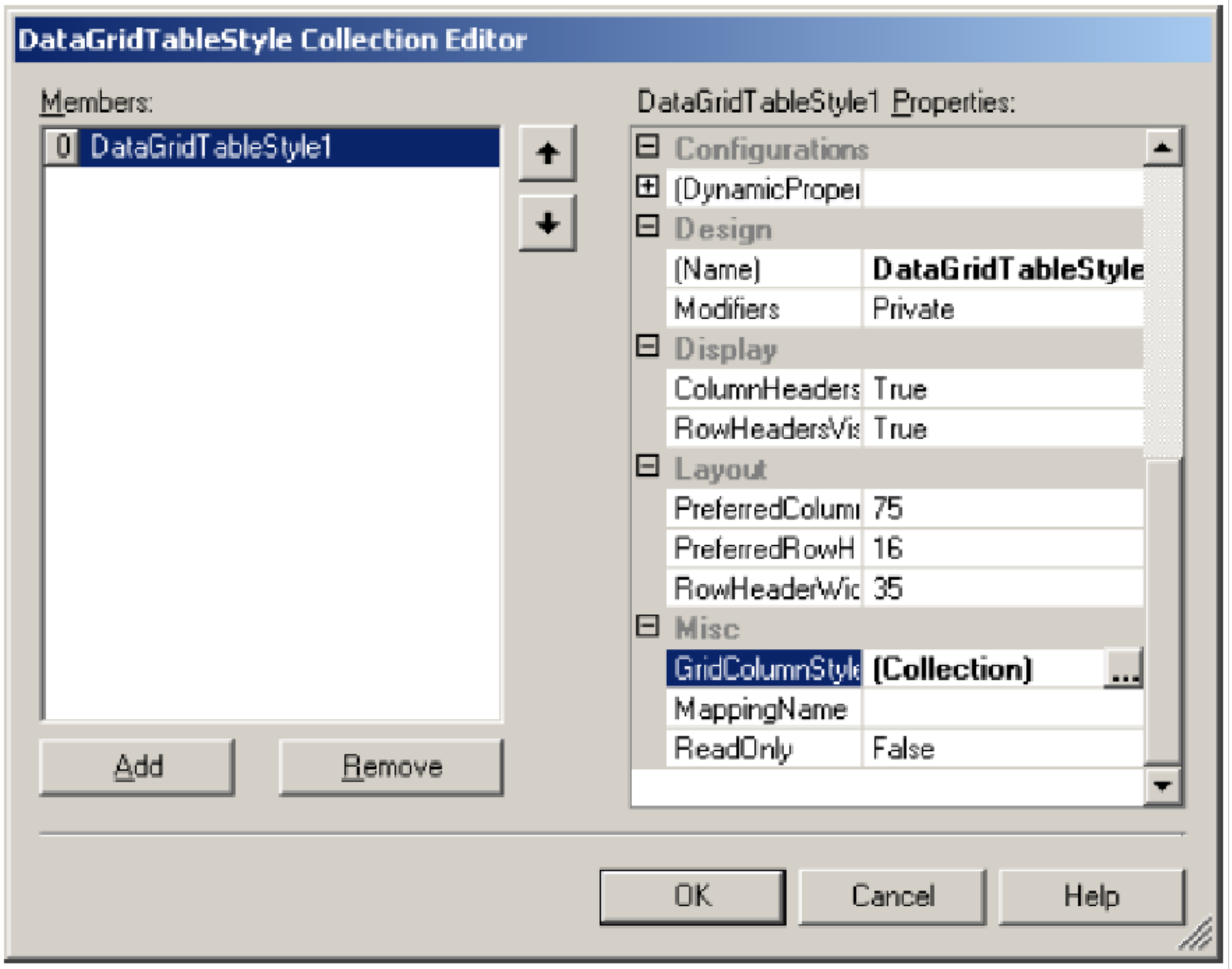

Рисунок 7.16 - Коллекция таблиц компонента gdChair

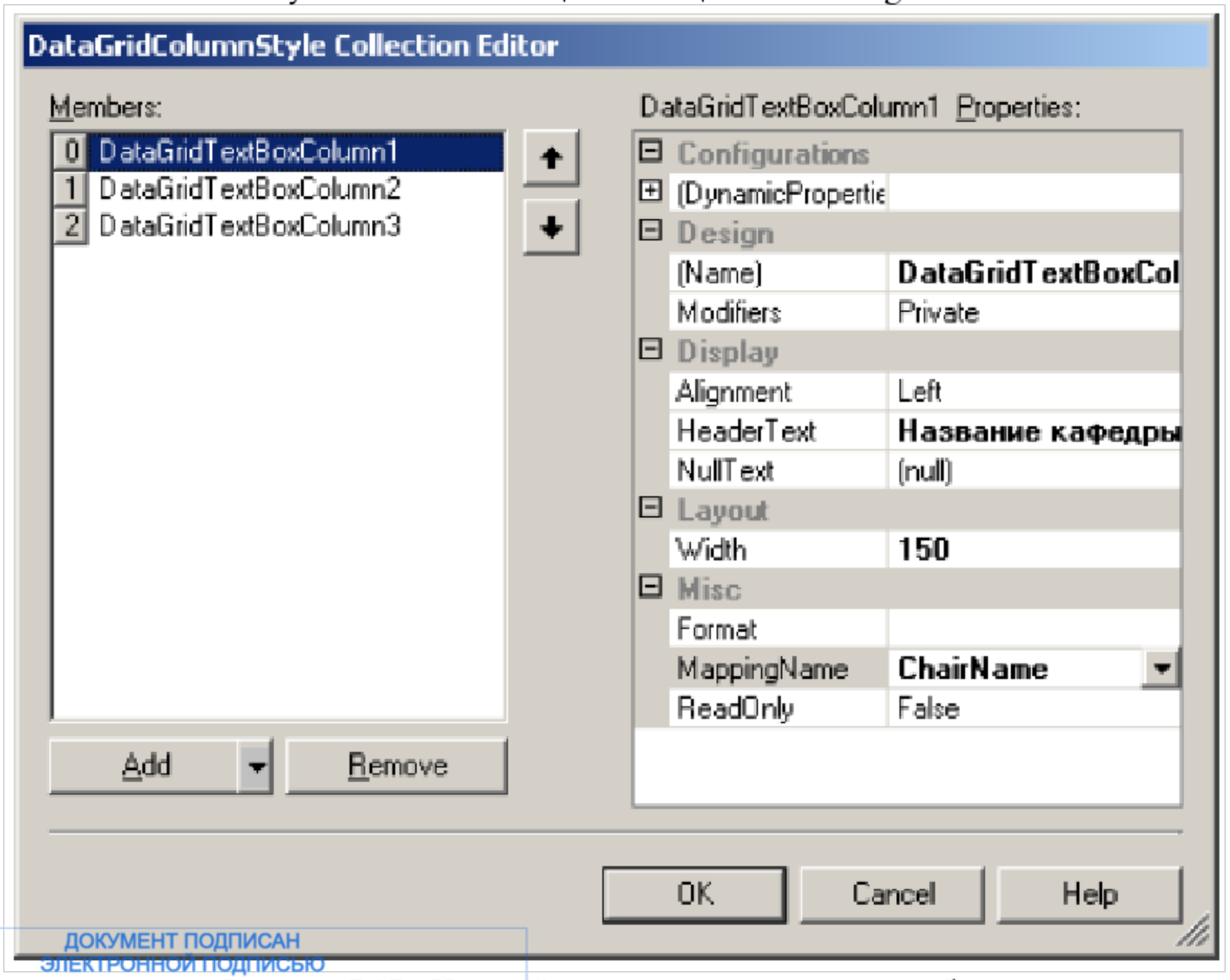

Рисунок 7.17 - Настройка элемента коллекции столбцов

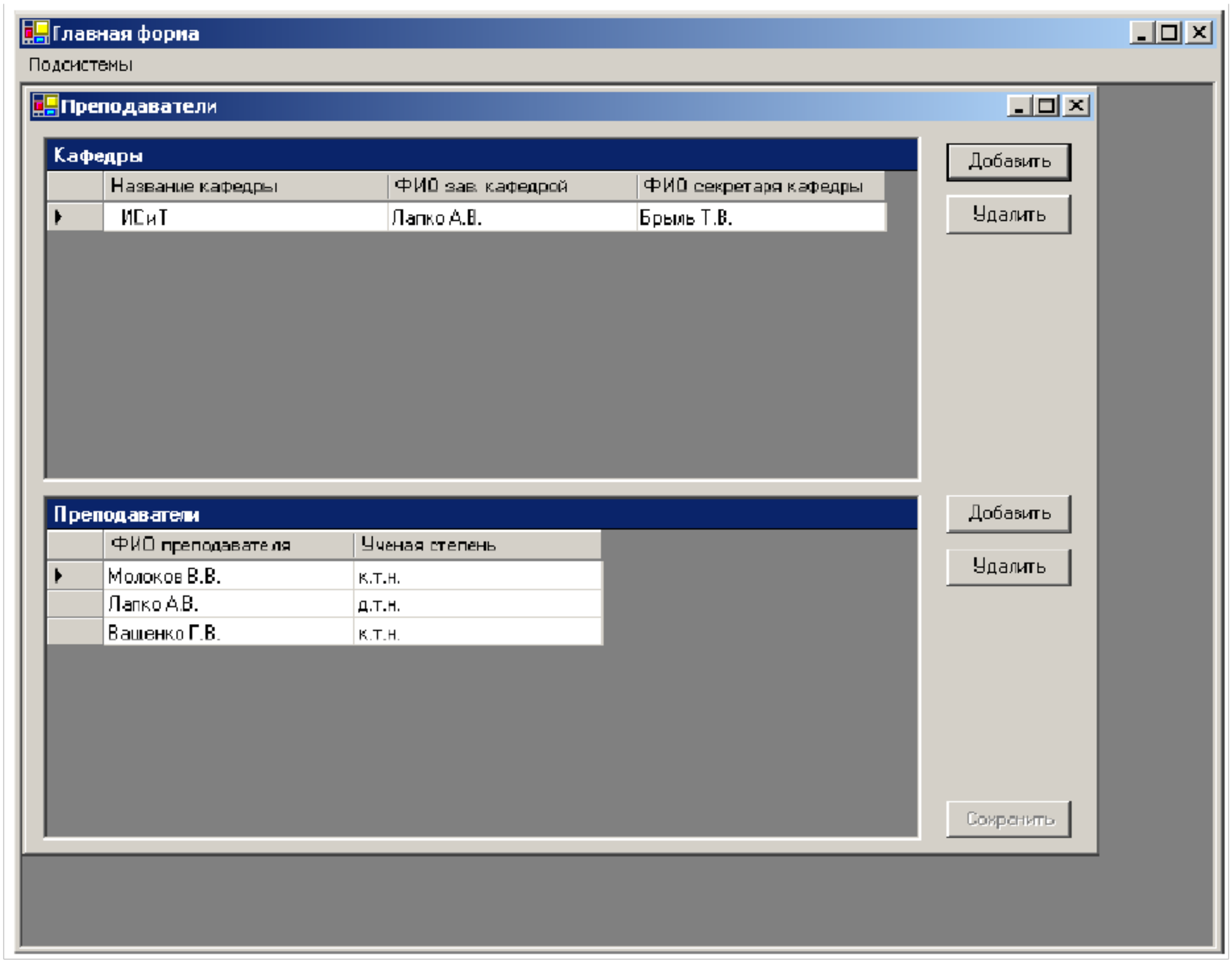

Рисунок 7.18 - Работа с базой данных в режиме таблицы

Задания и порядок выполнения работы

1. Сформировать модель UML. Создать классы, описать их атрибуты. Настроить взаимосвязи между классами.
2. Связать пространство ECO с базой данных.
3. Создать пользовательский интерфейс.
4. Связать интерфейс с моделью UML.
5. Элементы управления связать с выражениями OCL для выполнения типичных стандартных действий (добавление, удаление объектов ECO и сохранение их в базу данных).

Контрольные вопросы

1. Что представляет собой технология MDA?
2. Что такое технология ECO?
3. Зачем нужны дескрипторы ECO?
4. Зачем нужен язык OCL?
5. Из каких этапов складывается процесс построения приложения ECO?
6. Как создаются классы в рамках модели приложения ECO?
7. Как представители класса PersistenceMapper настраиваются на конкретные СУБД?
8. Как автоматически сгенерировать схему базы данных на основе модели ECO?
9. Как элементы пользовательского интерфейса связываются с моделью ECO?
10. Как работает визуальный редактор выражений OCL?

Работа с литературой:

Рекомендуемые источники информации (№ источника)			
Основная	Дополнительная	Методическая	Интернет-ресурсы
1-3	1-3	1-3	1-2

Оценочные средства: контрольные вопросы

Тема 5. Технология MDA.

Лабораторная работа №8. Разработка MDA-приложения с использованием машин состояний

Форма проведения: Практическое выполнение задания
Цель работы – научиться использовать машины состояний при создании MDA-приложений.
Краткие теоретические сведения

Автоматы

Работа объектов программы описывается с помощью *автоматов* (или *машин состояний)*. Существует хорошо развитая *теория автоматов,* в которой изучаются вопросы их построения и анализа работы.

Автомат – это элемент системы, который характеризуется двумя базовыми понятиями: состояние и переход. Каждый автомат в определенный момент времени может находиться в одном из допустимых *состояний.* Число этих состояний, возможно, бесконечно. Задан также набор правил, по которому допускается *переход* из одних множеств состояний в другие.

Схожесть автоматной модели с программированием в том, что переменные (ячейки памяти) в каждый момент работы процессора всегда хранят конкретные значения из допустимых диапазонов значений. Эти значения фактически являются состояниями переменных.

Разрешенные над значениями переменных вычислительные операции приводят к переходу этих переменных в новые состояния (в новые значения).

Этот алгоритмический принцип может быть распространен на большое число управленческих задач. В общем случае автоматный подход позволяет описать любую, сколь угодно сложную последовательность операций.

Познакомимся с *базовыми правилами работы автоматов в языке UML*.

Автомат в определенный момент времени может находиться только в одном состоянии. Это правило соответствует принципу императивного программирования, согласно которому любая переменная в каждый момент времени (такт процессора, шаг работы программы) хранит только одно из множества допустимых значений.

Число состояний автомата конечно.

Из текущего состояния автомат может перейти только в одно следующее состояние. Не исключено, что это состояние выбирается из нескольких доступных вариантов.

В рамках диаграммы у каждого состояния должен быть предшественник (за исключением начального состояния). Это правило запрещает размещение на диаграмме состояний, никак не связанных друг с другом.

В определенном состоянии автомат может находиться сколь угодно долго. Переходы между состояниями считаются мгновенными. На диаграмме состояний не регистрируется путь (история состояний), приведший автомат в текущее состояние (в отличие от диаграмм последовательности и кооперации). Это правило показывает, что по текущему значению переменной нельзя определить те значения (состояния), в которых она находилась ранее.

Состояния

Конкретное состояние (в общем случае – статический срез системы) формируется инструментом State (Состояние). На диаграмме оно изображается в виде прямоугольника с закругленными углами. Состояние имеет *имя,* начинающееся с заглавной буквы. Под ним могут записываться *условия* и *действия*. Условия проверяются, а действия выполняются, когда автомат находится в соответствующем состоянии.

Отметим принятые правила оформления элементов диаграммы состояний. Элемент State представляет не действие, а описание статического состояния, в котором автомат может находиться неограниченно долго. Автомат может переходить из текущего состояния в другие состояния при выполнении определенных действий пользователя или условий, которые должны быть выполнены. Действия и условия отображаются только на линиях переходов между состояниями.

Подавтоматы

Внутри каждого состояния разрешается формировать собственный автомат (подавтомат) – мини-диаграмму состояний. Благодаря этому возможно проектировать систему постепенно, раскрывая отдельные ее элементы по мере необходимости.

Подавтомат – это автомат, вложенный в другой автомат и описывающий поведение конкретного состояния.

Диаграммы состояний

Диаграммы состояний (State Machine Diagram) стали новым типом диаграмм в версии UML

2.0. Они особенно важны для разработчика, использующего среду Delphi. Дело в том, что в версии Delphi 2006 имеется технология моделирования ECO III. Она расширена средствами визуального построения алгоритмов. С помощью этих средств описывается работа разных элементов модели. Ранее для описания модели и генерации исходного кода на языке Delphi применялись лишь статические диаграммы классов. Теперь задействованы и диаграммы состояний.

Диаграмма состояний – это средство, описывающее логику функционирования автоматов (машин состояний).

В основу диаграмм состояний положена концепция автоматов. Доказано, что с помощью таких автоматов можно запрограммировать произвольный алгоритм любой сложности, если его можно также записать на императивном языке программирования.

Диаграмма состояний всегда относится к конкретному классу и описывает его внутреннее функционирование. На диаграмме конкретное состояние отображается с помощью элемента State. Начальное и конечное состояния – элементы Initial (Начальное) и Final (Конечное) – представлены на диаграмме сплошным кружком. Кружок, соответствующий конечному состоянию, обведен каймой. Фактически, это псевдосостояния, не возникающие при реальной работе. Начальное и конечное состояния лишь наглядно задают последовательность входа в первое рабочее состояние и выхода из последнего рабочего состояния.

Отдельное состояние может охватывать последовательность действий. Состояние, охватывающее другие состояния, называют *суперсостоянием,* а вложенные в него состояния – *подсостояниями.* Такой иерархический подход к организации состояний позволяет формировать одинаковые реакции подсостояний на уровне одного суперсостояния. Пусть, например, имеется стандартное аварийное состояние и стандартный переход в него по команде отмены. Для каждого из множества состояний диаграммы можно указать этот переход индивидуально, а можно объединить их в суперсостояние. Тогда переход в аварийное состояние представляется на диаграмме всего одной линией, исходящей из элемента, представляющего суперсостояние.

Каждое из состояний на диаграмме не обязано пассивно ожидать внешнего события, приводящего к переходу основного объекта в следующее состояние. Состояние может вести определенную деятельность. Соответствующая деятельность задается свойством состояния Do activity (Выполнение деятельности). Нужная деятельность уже должна быть задана в проекте, например на одной из диаграмм деятельности. Подобное состояние, когда до него доходит очередь, начинает эту деятельность и ожидает ее завершения, фактически отображая не статическое состояние, а выполнение процесса. Этот процесс может быть прерван, что вызовет переход в другое состояние. Если он закончится успешно, также выполнится соответствующий переход.

В диаграммах состояний UML 2.0 можно описывать историческое состояние. Обычная история состояния и глубокая история представлены как отдельные элементы: Shallow History (Обычная история состояния) и Deep History (Глубокая история).

Элемент Junction (Слияние) визуализирует слияние и разделение потоков управления. Он представляет собой промежуточный узел, по внешнему виду аналогичный начальному элементу Initial. В таком узле собираются, а потом вновь расходятся потоки управления. В диаграммах состояний можно также задействовать условный элемент Choice (Выбор).

Создание MDA-приложений с использованием машин состояний

Для примера возьмем уже созданное нами MDA-приложение в лабораторной работе № 7 и дополним его машиной состояний. Расширим нашу предметную область. Добавим новый класс Дисциплина. Теперь у преподавателей появится список преподаваемых ими дисциплин. Введем ограничение на количество рабочих часов в семестр для преподавателя. Задача: запрограммировать процесс распределения нагрузки на преподавателей.

Использование диаграмм состояний UML позволяют визуально проектировать последовательности смены состояний объектов программы в зависимости от условий.

Модификация модели UML

1. Откроем проект projDeanOffice.

2. Добавим к модельному пространству приложения класс clSubject (Дисциплина). Определим у него класса: поле SubjName (Название дисциплины) типа String, поле SubjType (Тип дисциплины) типа String, поле SubjAmountHours (Количество часов дисциплины) типа Integer.

3. В существующий класс Преподаватель добавим атрибут MaxAmountHours (Ограничение на количество рабочих часов) типа Integer.

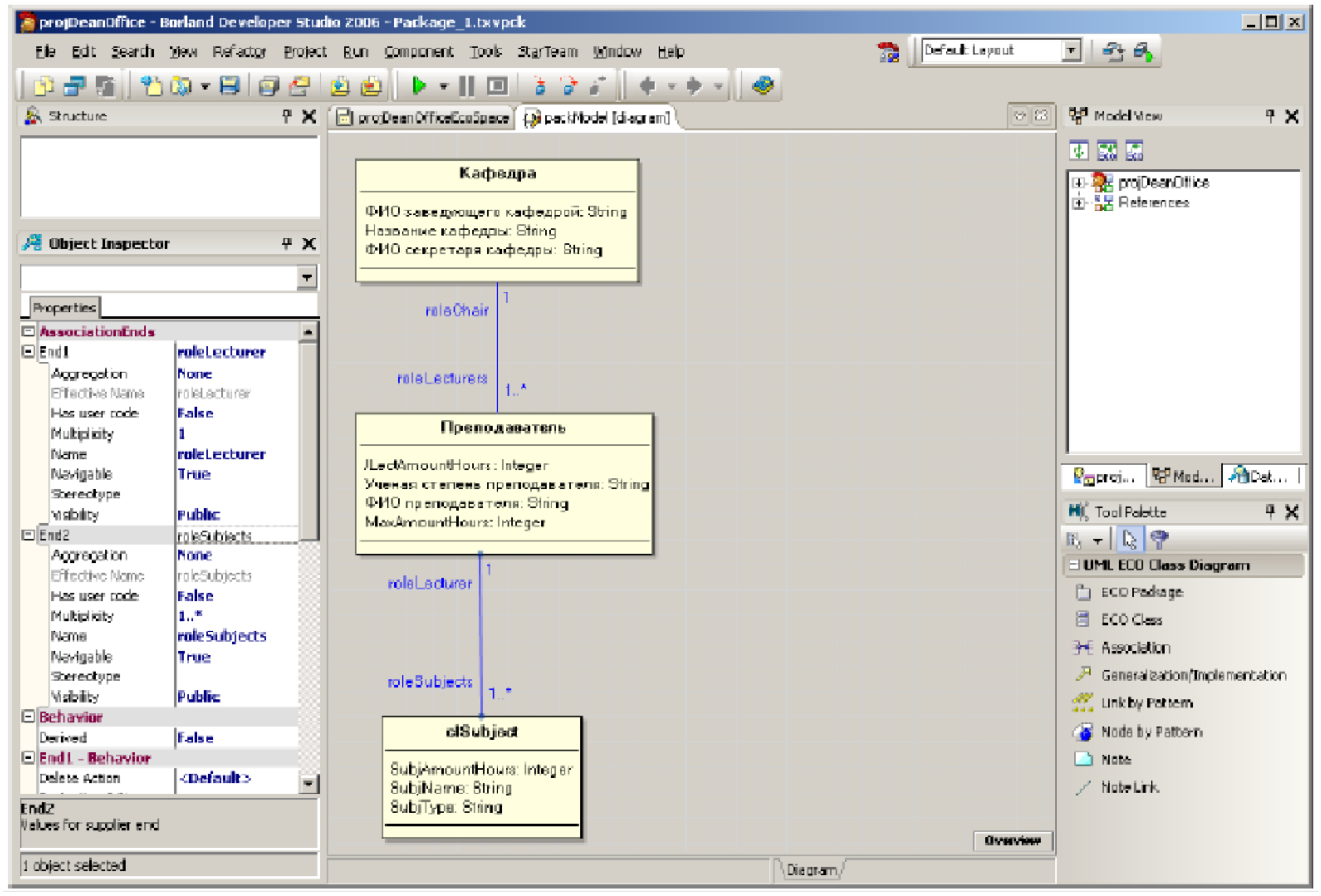

Рисунок 8.1 - Модификация модели UML

4. Теперь добавим вычисляемое поле. Вычисляемое поле – это поле, значение которого не хранится в классе, а рассчитывается непосредственно в момент обращения к нему. Добавим в класс Преподаватель атрибут LectAmountHours (Количество рабочих часов в семестр) типа Integer. Перед именем поля поставим символ /, визуально обозначающий вычисляемое значение. Признаком вычисляемости поля является значение True для свойства Derived (вычисляемое). Значение свойства LectAmountHours будет формироваться выражение OCL. Нужное выражение должно быть значением свойства Derivation OCL (Код OCL для вычисления). Это выражение введем позже, когда создадим машину состояний.

5. Сформируем между созданными классами ассоциативное отношение: «один преподаватель – много дисциплин». Дадим сторонам связи названия. В свойство Name для сторон End1 и End2 введем roleLecturer и roleSubjects соответственно (см. рисунок 8.1).

6. Заполним свойства Alias созданных элементов, чтобы на диаграмме классов отображались русские названия.

Создание машины состояний

В нашем примере дополним класс Дисциплина новыми возможностями. Каждая дисциплина будет иметь определенный статус. Возможно добавление дисциплины преподавателю (состояние Выбран преподаватель).

После чего она может быть принята на рассмотрение (состояние Выбрана дисциплина), затем либо назначена преподавателю (состояние Назначена), либо отклонена (состояние Отклонена). Состояние Отклонена дисциплина возможно также, если сумма часов предложенных преподавателю дисциплин превысит допустимую нагрузку на преподавателя. В случае отклонения дисциплины рассматриваемый экземпляр класса Дисциплина автоматически удаляется. Покажем, как реализовать такой алгоритм исключительно на уровне модели, не прибегая к кодированию на языке Delphi.

1. Перейдем в модельное пространство проекта. Выберем элемент clSubject и с помощью контекстного меню дадим команду Add > ECO State Machine. В рамках Проектировщика откроется новое, первоначально пустое окно. В нем будет построена соответствующая модель. Назовем модель StatesOfSubject.

2. Поместим на диаграмму начальное состояние Initial.

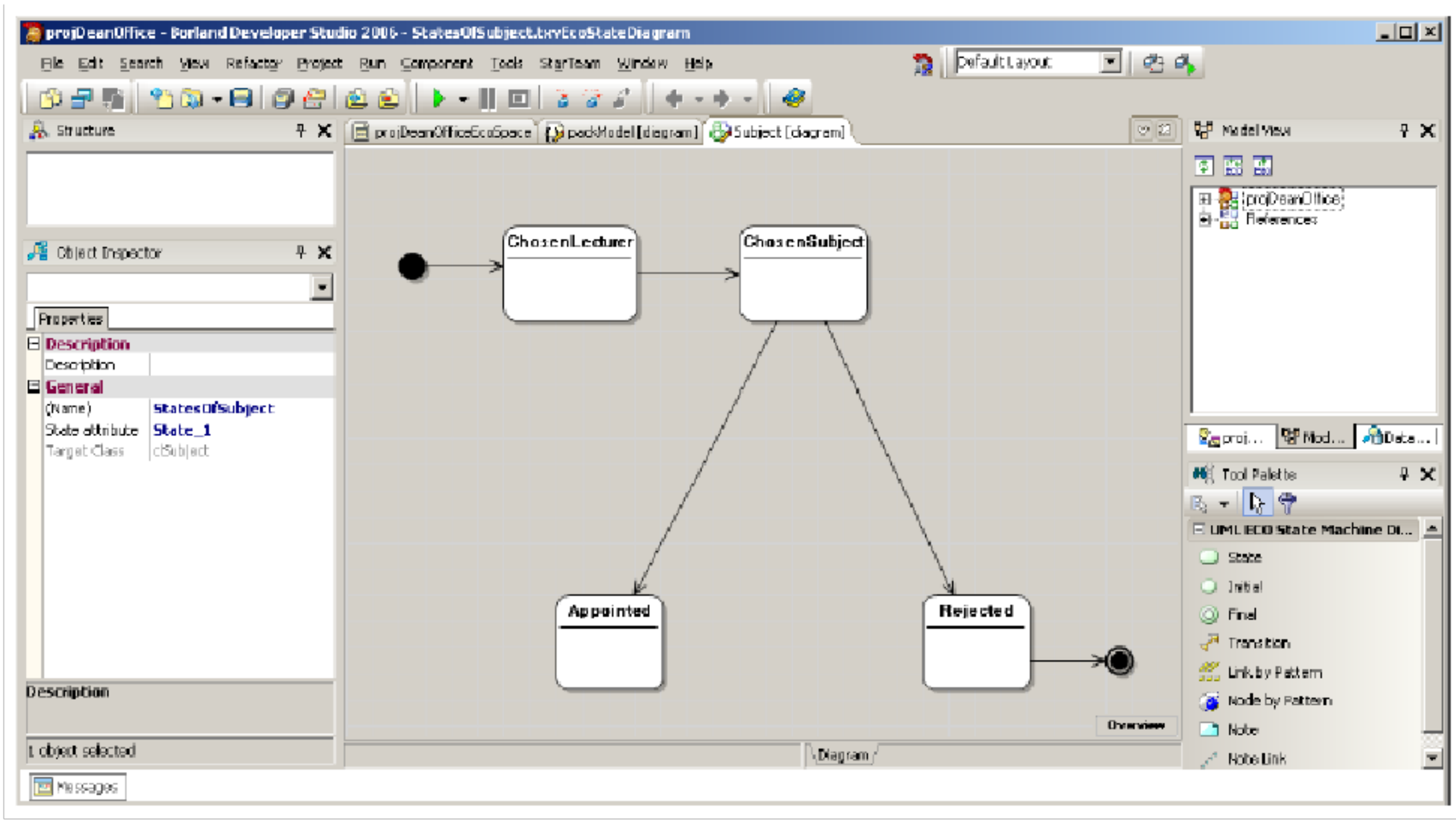

Рисунок 8.2 - Диаграмма машины состояний

3. Добавим на диаграмму четыре основных состояния дисциплины с помощью инструмента State. Дадим им имена: ChosenLecturer (Выбран преподаватель), ChosenSubject (Выбрана дисциплина), Appointed (Назначена), Rejected (Отклонена).

4. Поместим на диаграмму конечное состояние Final.

5. Согласно проектной логике свяжем эти состояния отношениемпереходом с помощью инструмента Transition (рисунок 8.2).

6. Вернемся на диаграмму классов ECO. Видно, что в классе Дисциплина появилось новое свойство State_1 типа String (назовем его SubjectState), обозначающее нашу машину состояний. Теперь можно ввести выражение OCL для вычисляемого поля Количество рабочих часов (LectAmountHours) класса Преподаватель. В свойстве Derivation OCL этого атрибута создадим с помощью встроенного OCL-редактора выражение:

self.roleSubjects->select(SubjectState='Appointed').SubjAmountHours->sum

Оно означает, что в поле Количество рабочих часов каждого объекта Преподаватель будет сумма часов тех дисциплин, которые ему уже назначены (поле SubjectState имеет значение Appointed).

7. Для каждого перехода надо задать триггер.

В терминологии модели ECO *триггер* – это действие, которое производится при выполнении определенного условия и модифицирует текущее состояние (вызывает переход к другому состоянию).

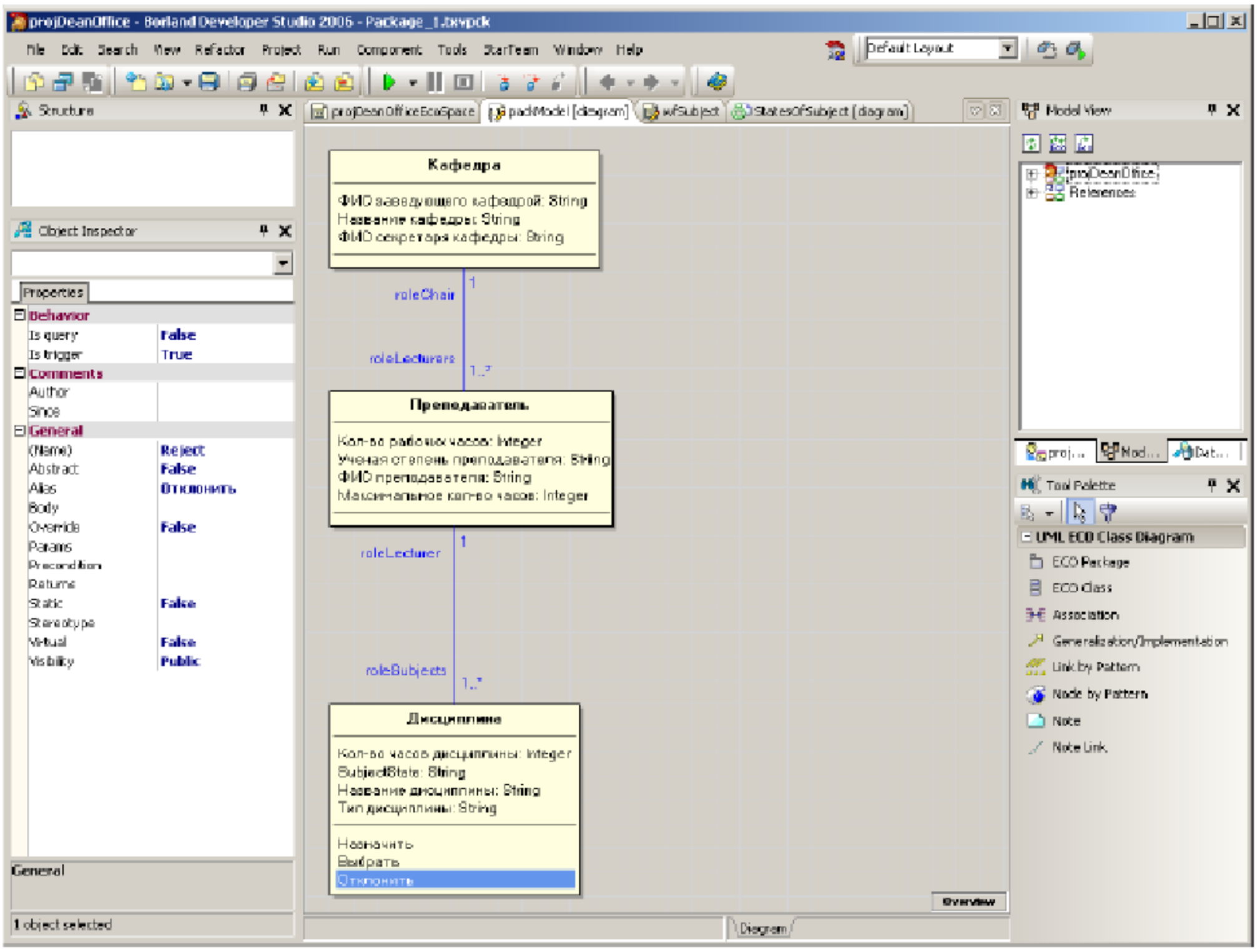

Рисунок 8.3. Список возможных состояний класса Дисциплина

Добавим в класс Дисциплина три триггера. Каждый триггер добавляют командой контекстного меню Add > Trigger. Дадим этим триггерам соответствующие имена: Choose, Appoint и Reject. Их свойства Alias заполним русскоязычными названиями: Выбрать, Назначить и Отклонить (рисунок 8.3).

Для действия Выбран преподаватель создавать триггер не нужно.

Согласно нашей модели считается, что экземпляр класса Дисциплина получает этот статус, как только он сформирован.

8. Распределим триггеры по переходам на диаграмме машины состояний. Будем выбирать по очереди каждую связь (переход между состояниями) и выбирать в ее свойстве Trigger нужное имя триггера из раскрывающегося списка.

9. Зададим правило перехода из состояния Выбрана дисциплина в состояние Назначена. Пока что ограничений или дополнительных требований на этот переход не наложено, и пользователь может бесконтрольно менять состояние дисциплины. Правило перехода задается в свойстве Guard (сторожевое условие) в виде условного выражения OCL:

(self.roleLecturer.LectAmountHours+self.SubjAmountHours) <= self.roleLecturer.MaxAmountHours

Данное выражение означает, что сумма рабочих часов преподавателя, претендента на ведение выбранной дисциплины, и количество часов этой дисциплины должно быть меньше или равно максимальному количеству рабочих часов преподавателя. Теперь возможность назначить дисциплину, если сумма рабочих часов превышает допустимое значение, недоступна, пока преподавателю не увеличат максимальное количество часов или не сократят количество часов для дисциплины.

Обновление базы данных

1. Чтобы сгенерировать исходный код ECO в соответствии с построенной моделью, выберем операцию Regenerate ECO source code из консольного меню окна Model View. Также такая операция производится нажатием кнопки над данным окном. Генерация производится автоматически.

2. Так как мы создали новый класс и добавили новые поля в уже существующий,

необходимо обновить подключенную базу данных. Для этого перейдем на закладку настройки объектного пространства projDeanOfficeEcoSpace. Нажмем кнопку Generate Schema в нижней части окна. В текущей базе данных автоматически сформируются таблицы, содержащие поля для хранения модели проекта. При этом все имеющиеся данные в базе удаляются. Операция Evolve Schema позволяет сохранить существующие данные и дополнить базу созданными элементами в соответствии с построенной моделью.

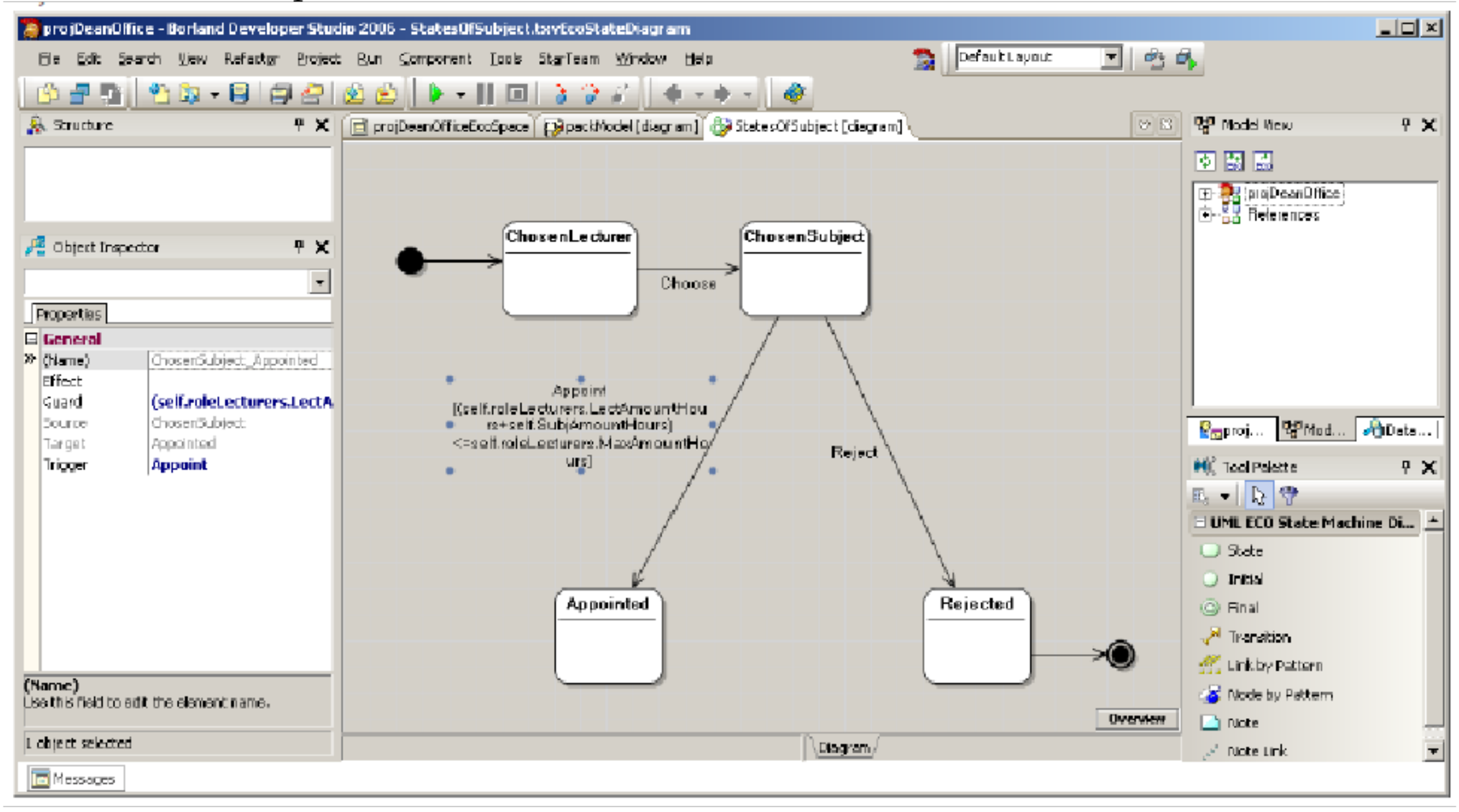

Рисунок 8.4. Добавляем триггеры и сторожевое условие к диаграмме машин состояний

Модификация пользовательского интерфейса

Добавим на форму элементы интерфейса пользователя.

1. Создадим вторую дочернюю форму wfSubject (класс TSubject). Заголовок формы – Дисциплины.

2. Перейдем к вкладке Code. В окне появится код формы. Перед командой Implementation объявим переменную, ответственную за создание дочернего окна Дисциплины:

```
var
callSubj: TSubject;
```

3. Перейдем к Главной форме. Чтобы в Главной форме создать дочернюю, подключим к ней форму wfSubject командой File > Use Unit. В появившемся окне Use Unit выберем форму wfSubject.

4. В главном меню Главной формы добавим подпункт Дисциплины в пункте Подсистемы. На событие выбора пункта меню Дисциплины пропишем следующие операции:

```
callSubj := TSubject.Create(EcoSpace);
callSubj.MdiParent := self;
callSubj.Show;
```

5. Вернемся к форме Дисциплины. Разместим на ней две таблицы: dgLecturer и dgSubject (экземпляры класса DataGrid) для представления экземпляров классов Преподаватель и Дисциплина. Заполним в таблицах свойство CaptionText (название заголовка).

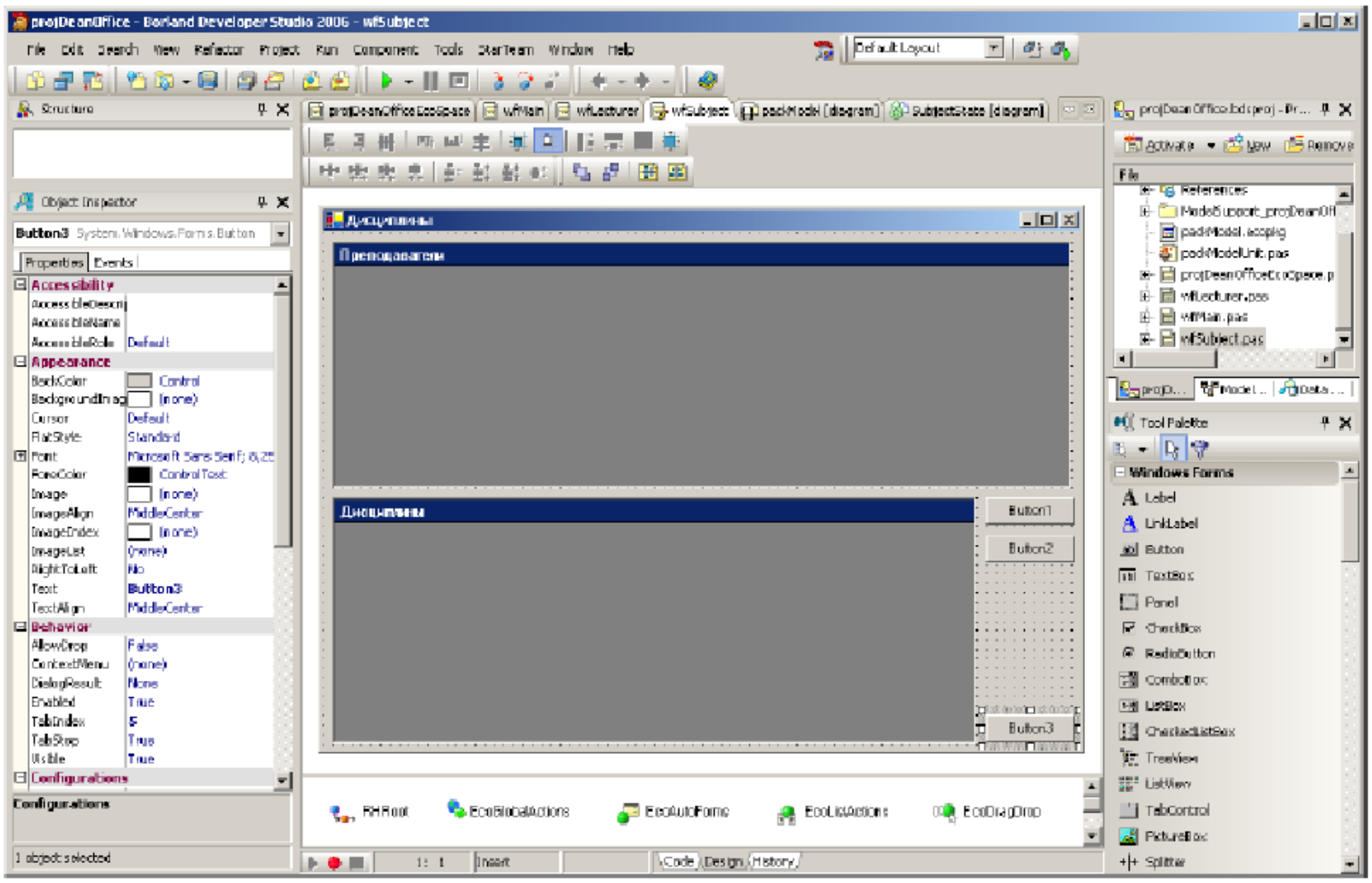

Рисунок 8.5 - Настройка пользовательского интерфейса

6. Добавим две кнопки для работы с таблицей Дисциплины (добавление, удаление объектов) и третью кнопку для обновления базы данных (рисунок 8.5).

Связывание интерфейса с моделью

1. Сформируем три дескриптора ECO: ehLecturer, ehSubject (экземпляры класса ExpressionHandle) для организации доступа к объектам модели и пространства ECO во время работы программы и cmhLecturer (экземпляр класса CurrensyManagerHandle), который будет служить для указания на текущий объект таблицы Преподаватели.

2. Настроим таблицу Дисциплины так, чтобы она отражала список дисциплин, принадлежащих выбранному преподавателю в первой таблице.

Свяжем дескриптор cmhLecturer с таблицей dgLecturer через свойство BindingContext.

Теперь он отслеживает выделенную строку этой таблицы.

3. Настроим дескриптор cmhLecturer на связь с родительским объектом. Зададим ссылку на объект ehLecturer в свойстве RootHandle, определяющем корневой идентификатор ECO.

4. В свойстве RootHandle дескриптора ehSubject выберем значение cmhLecturer. Так задают место данного идентификатора в цепочке доступа к объектному пространству ECO.

5. В качестве выражения OCL введем в свойство Expression объекта ehSubject строку self.roleSubjects. Здесь roleSubjects – это имя роли класса Subject в его ассоциативной связи с классом Преподаватель. Это выражение определяет список дисциплин, предложенных и назначенных преподавателю, который выбран в таблице Преподаватели.

6. Свяжем таблицу Преподаватели с дескриптором ehLecturer через ее свойство DataSource, а таблицу Дисциплины – с дескриптором ehSubject.

Теперь таблицы отображают поля классов Преподаватель и Дисциплина.

Кроме этого, в таблице Дисциплины появилось поле SubjectState, которое будет автоматически отображать текущее состояние дисциплин.

7. Настроим стили таблиц. Воспользуемся свойством таблицы TableStyles. Присвоим полям таблиц русскоязычные названия.

8. Настроим функционирование кнопок. Выделим в окне Проектировщика кнопку Button1. В свойстве EcoListAction выберем значение Add. В свойстве RootHandle выберем подходящий поставщик объектов ECO – идентификатор ehSubject. Свяжем результат действия кнопки (созданный экземпляр класса Дисциплина) с визуальным элементом, отображающим этот экземпляр, в нашем случае – с таблицей dgSubject. Для этого в свойстве BindingContext выберем имя dgSubject. Аналогично настроим кнопку Button2 на операцию удаления. Выберем компонент

EcoListActions, в его свойстве CaptionAdd введем слово Добавить, а в свойстве CaptionDelete – Удалить.

Кнопка автоматически получила название Добавить и Удалить. В свойстве Text кнопки Button3 введем значение Сохранить (см. рисунок 8.6).

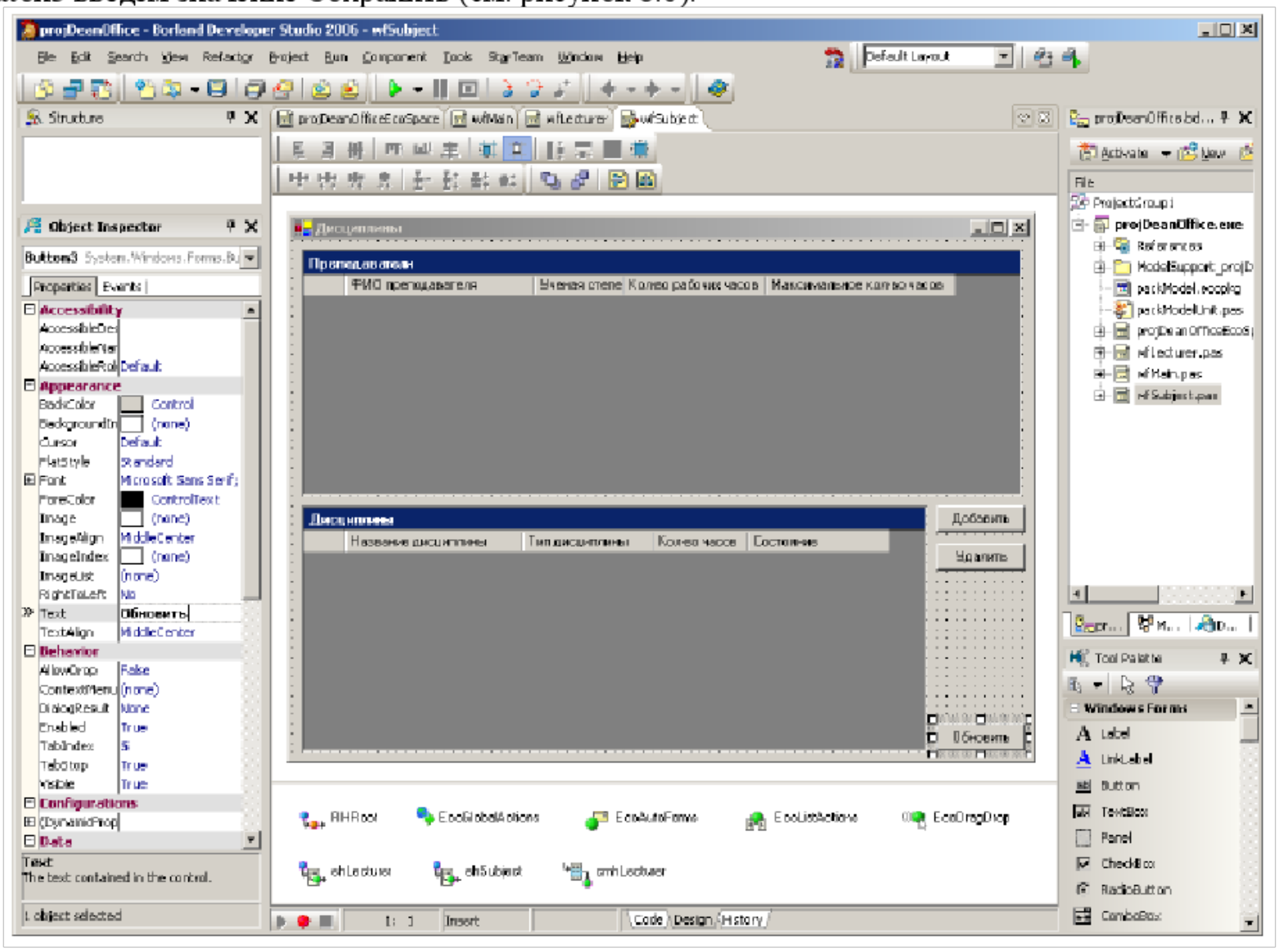

Рисунок 8.6 - Связывание интерфейса с моделью

9. Запустим приложение. Добавим несколько объектов в таблицу Дисциплины для выбранного преподавателя. В поле Состояние таблицы Дисциплины все объекты имеют одинаковое значение ChosenLecturer. На данный момент это состояние неизменяемо. Чтобы иметь возможность изменять состояния, произведем следующую модификацию пользовательского интерфейса.

Применение автоформ

Автоформа ECO предназначена для модификации в отдельном окне объектов ECO, представленных в таблицах. Для вызова автоформы надо задать в свойстве таблицы EcoAutoForm значение True. Если запустить программу и дважды щелкнуть мышью на произвольной строке таблицы, откроется диалоговое окно, содержащее текстовые поля, которые соответствуют столбцам таблицы (атрибутам объекта ECO). В этом окне отображается содержимое полей строки, на которой был выполнен щелчок.

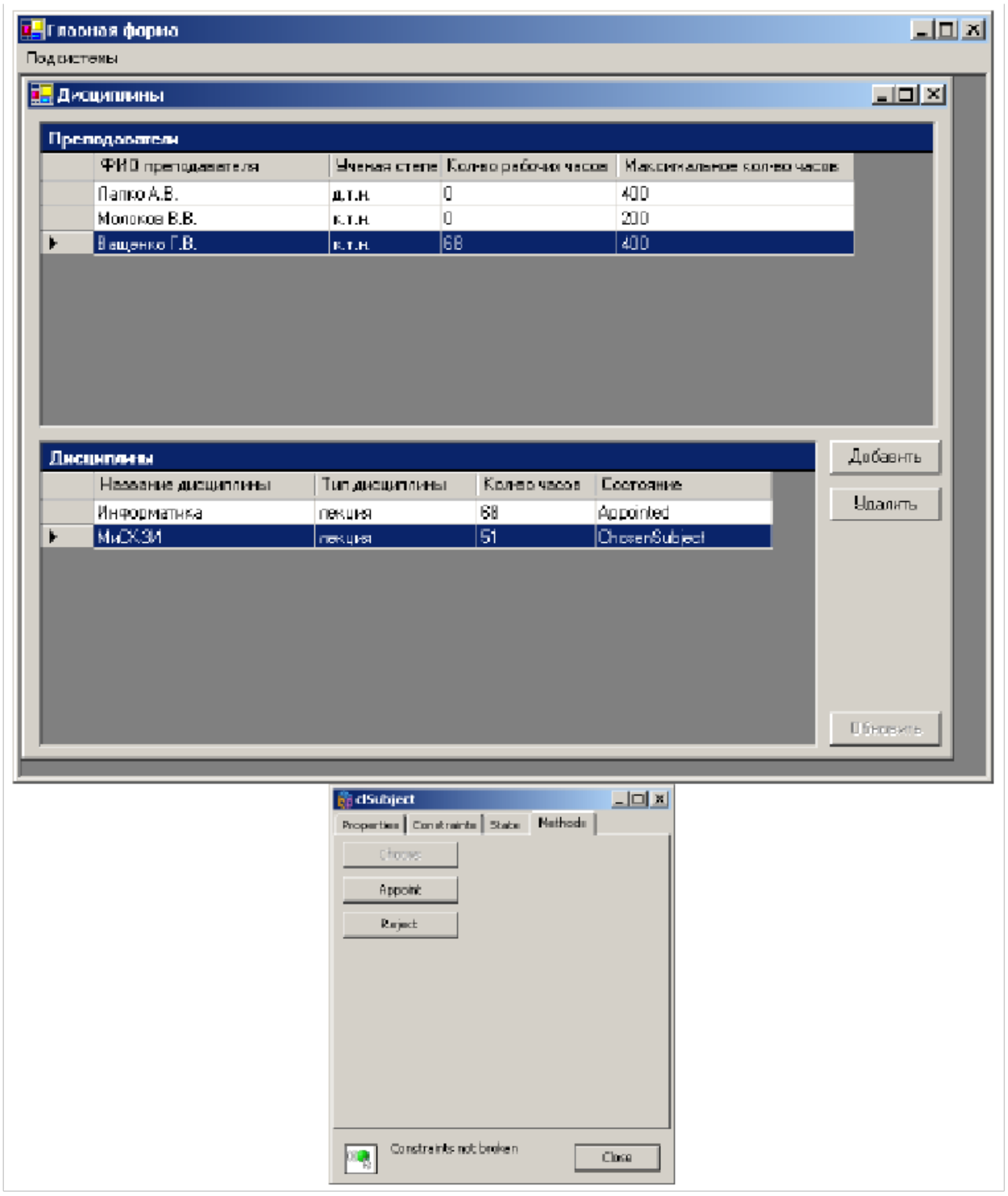

Рисунок 8.7 - Работа с таблицей в режиме автоформы

Для одной таблицы можно открыть несколько окон автоформы одновременно и редактировать значения разных объектов. При этом модифицированные величины корректно отображаются в таблице и фиксируются в объектном пространстве. Данная возможность, как будет показано далее, особо полезна в случаях, когда между объектами таблицы установлены дополнительные связи.

5) Таблицу Дисциплины настроим на работу в режиме автоформы. Для этого в свойстве EcoAutoForm зададим значение True.

6) Запустим приложение. Если щелкнуть в начале произвольной строки таблицы, хранящей список дисциплин, откроется автоформа для ее редактирования. В ней на закладке Methods показаны возможные переходы между состояниями, причем в соответствии с последовательностью переходов: в следующее состояние нельзя перейти, не побывав в предыдущем. Если вавтоформе нажать кнопку Choose (подпись кнопки совпадает с именем соответствующего триггера), список доступных переходов автоматически изменяется. Соответствующие изменения отобразятся и в колонке Состояние таблицы Дисциплины. Причем, переход в состояние Назначить дисциплину ограничен условием: количество рабочих часов преподавателя, которого собираются назначить на ведение выбранной дисциплины, в сумме с количеством часов назначаемой дисциплины должно быть меньше или равно максимальному количеству рабочих часов для преподавателя (см. рисунок 8.7).

Расширение пользовательского интерфейса

Команды-триггеры для удобства можно привязать к пользовательским элементам управления. Создадим в текущем проекте контекстное меню с помощью компонента ContextMenu из категории Components. Пусть в нем будет три пункта: Выбрать, Назначить и Отклонить. Их

необходимо связать с соответствующими триггерами.

1. Добавим на форму компонент ContextMenu, дадим ему имя cmSubject.

2. В свойстве меню RootHandle задается корневой идентификатор ehSubject.

3. Объект управления (таблица Дисциплины), для которого вызывается данное меню, выбирается в свойстве BindingContext.

4. В свойстве EcoListAction выбирается тип действия ECO, выполняемого при выборе данного пункта. Введем значение ExecuteAction (исполняемое выражение OCL).

5. Само выражение OCL следует ввести в свойство ActionExpression. Это выражение формируется с помощью раздела Triggers в редакторе выражений OCL. Так, для пункта Выбрать дисциплину это выражение записывается как строка self.Choose.

6. В свойстве EnabledOCL с помощью этого же редактора формируется выражение OCL – триггерный запрос, который определяет, доступен ли пользователю соответствующий элемент управления (в нашем случае – пункт меню). Если триггер недоступен, то и пункт меню автоматически блокируется. Запрос выбирается в разделе редактора Trigger queries. Например, для пункта меню Назначить он записывается строкой self.Appoint (рисунок 8.8).

7. Выделим таблицу dgSubject. В ее свойстве ContextMenu выберем ссылку на настроенный компонент cmSubject. Привязка и настройка контекстного меню для объектов таблицы Дисциплины закончена.

8. Запустим приложение. Видно, что контекстное меню каждой строки (доступные в ней пункты) автоматически меняется в зависимости от состояния текущей дисциплины (см. рисунок 8.9).

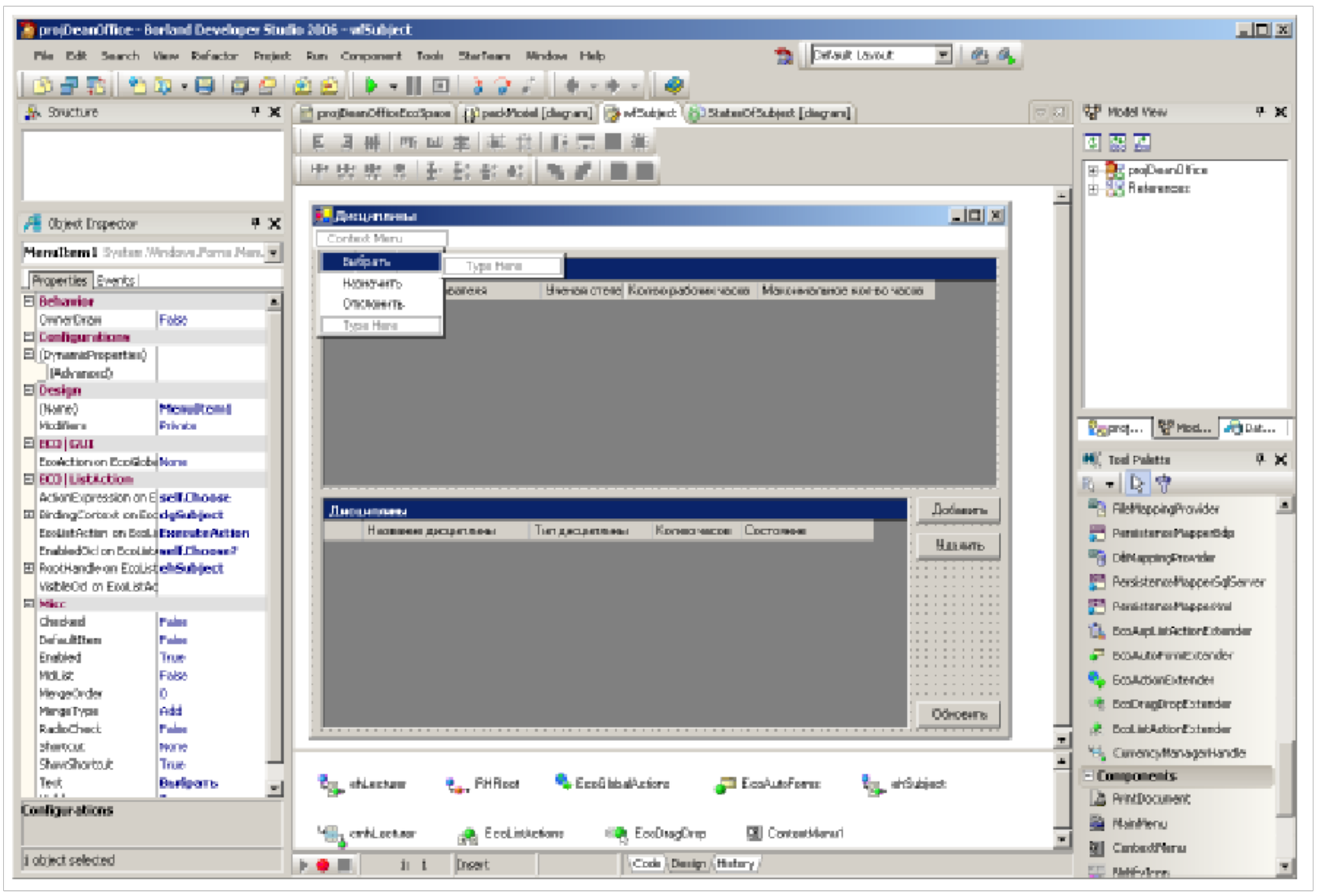

Рисунок 8.8 - Связывание команд-триггеров с компонентом ContextMenu

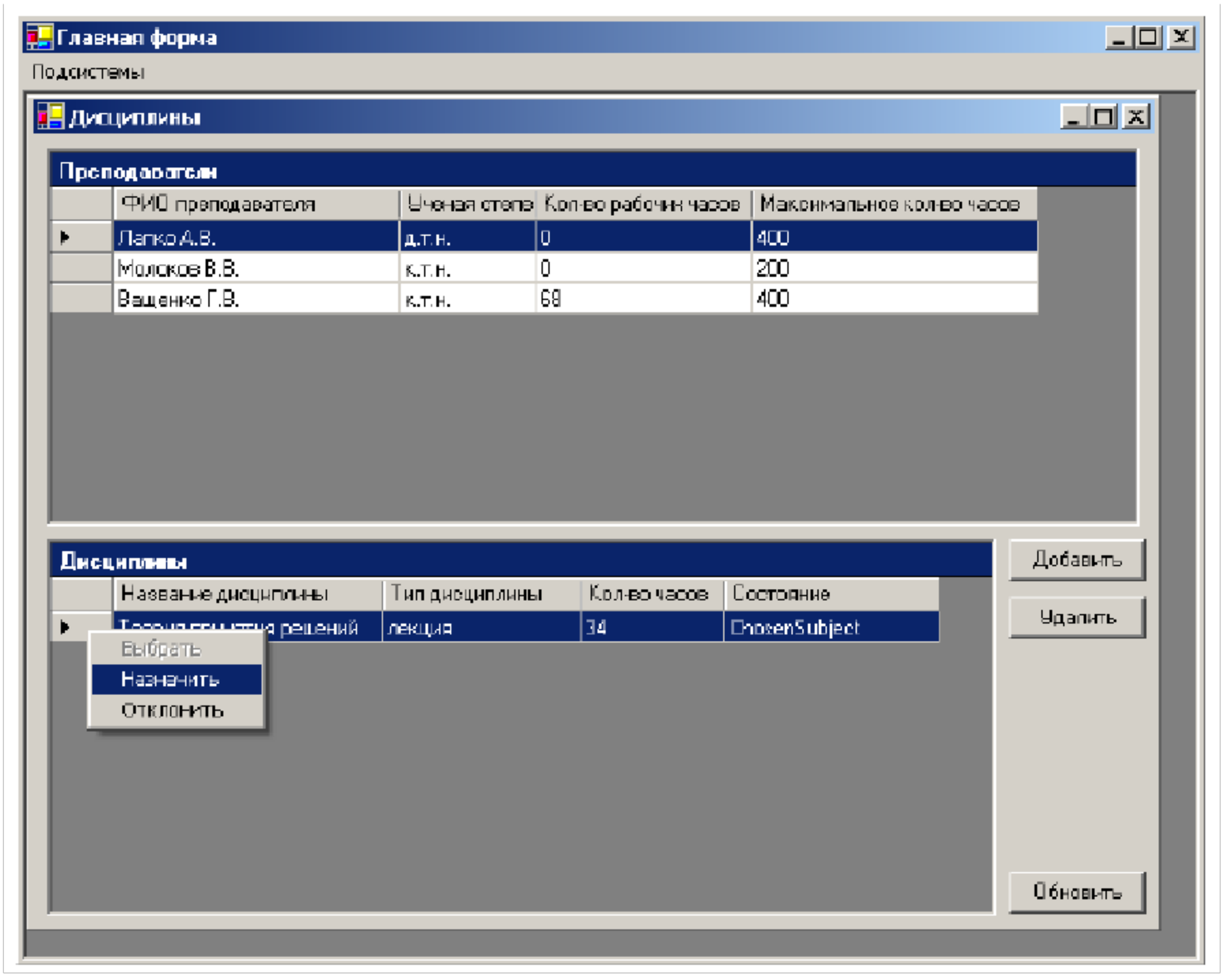

Рисунок 8.9 - Применение контекстного меню в приложении

Задания и порядок выполнения работы

1. Продумать, какой объект из предметной области можно представить в виде последовательности определенных состояний.
2. Расширить модель UML: создать машину состояний для выбранного объекта (или объектов) и модифицировать диаграмму классов.
3. Обновить базу данных, модифицировать пользовательский интерфейс и связать его с моделью.
4. Применить автоформы или контекстное меню.

Контрольные вопросы

1. Что такое автомат?
2. Назовите базовые правила работы автоматов в языке UML.
3. Для чего предназначены диаграммы состояний?
4. Как строятся диаграммы машин состояний?
5. Как настраиваются триггеры в диаграммах состояний?
6. Как задаются правила перехода на языке OCL для триггеров?
7. Как связать триггеры с элементами пользовательского интерфейса?

Работа с литературой:

Рекомендуемые источники информации (№ источника)			
Основная	Дополнительная	Методическая	Интернет-ресурсы
1-3	1-3	1-3	1-2

Оценочные средства: контрольные вопросы

5. КРИТЕРИИ ОЦЕНИВАНИЯ КОМПЕТЕНЦИЙ

Оценка «отлично» выставляется студенту, если теоретическое содержание курса освоено полностью, без пробелов; исчерпывающе, последовательно, четко и логически стройно излагает материал; свободно справляется с задачами, вопросами и другими видами применения знаний; использует в ответе дополнительный материал все предусмотренные программой задания выполнены, качество их выполнения оценено числом баллов, близким к максимальному; анализирует полученные результаты; проявляет самостоятельность при выполнении заданий.

Оценка «хорошо» выставляется студенту, если теоретическое содержание курса освоено полностью, необходимые практические компетенции в основном сформированы, все предусмотренные программой обучения учебные задания выполнены, качество их выполнения достаточно высокое. Студент твердо знает материал, грамотно и по существу излагает его, не допуская существенных неточностей в ответе на вопрос.

Оценка «удовлетворительно» выставляется студенту, если теоретическое содержание курса освоено частично, но пробелы не носят существенного характера, большинство предусмотренных программой заданий выполнено, но в них имеются ошибки, при ответе на поставленный вопрос студент допускает неточности, недостаточно правильные формулировки, наблюдаются нарушения логической последовательности в изложении программного материала.

Оценка «неудовлетворительно» выставляется студенту, если он не знает значительной части программного материала, допускает существенные ошибки, неуверенно, с большими затруднениями выполняет практические работы, необходимые практические компетенции не сформированы, большинство предусмотренных программой обучения учебных заданий не выполнено, качество их выполнения оценено числом баллов, близким к минимальному.

6.МЕТОДИЧЕСКИЕ МАТЕРИАЛЫ, ОПРЕДЕЛЯЮЩИЕ ПРОЦЕДУРЫ ОЦЕНИВАНИЯ ЗНАНИЙ, УМЕНИЙ, НАВЫКОВ И (ИЛИ) ОПЫТА ДЕЯТЕЛЬНОСТИ, ХАРАКТЕРИЗУЮЩИХ ЭТАПЫ ФОРМИРОВАНИЯ КОМПЕТЕНЦИЙ

Промежуточная аттестация проводится в форме экзамена.

Процедура проведения экзамена осуществляется в соответствии с Положением о проведении текущего контроля успеваемости и промежуточной аттестации обучающихся по образовательным программам высшего образования в СКФУ.

В экзаменационный билет включаются три вопроса: по одному вопросу из категорий «знать, уметь, владеть».

Для подготовки по билету отводится 30 минут.

При подготовке к ответу студенту предоставляется право пользования учебно-методическим комплексом дисциплины.

Текущая аттестация студентов проводится преподавателями, ведущими лабораторные занятия по дисциплине.

Практическое выполнение задания со студентом должно представлять собой связанный, логически последовательныйобмен сообщениями на заданную тему, показывать его умение применять определения, правила в конкретных случаях. Основные требования к собеседованию: полноту и правильность ответа;степень осознанности, понимания изученного;языковое оформление ответа.

Метод case-study (от английского case – случай, ситуация) – метод активного проблемно-ситуационного анализа, основанный на обучении путем решения конкретных задач – ситуаций (решение кейсов). Относится к неигровым имитационным активным методам обучения. Непосредственная цель метода case-study – совместными усилиями группы студентов проанализировать ситуацию – case, возникающую при конкретном положении дел, и выработать практическое решение; окончание процесса – оценка предложенных алгоритмов и выбор лучшего в контексте поставленной проблемы. Технология метода заключается в следующем: по определенным правилам разрабатывается модель конкретной ситуации, произошедшей в реальной жизни, и отражается тот комплекс знаний и практических навыков, которые студентам нужно получить; при этом преподаватель выступает в роли ведущего, генерирующего вопросы, фиксирующего ответы, поддерживающего дискуссию, т.е. в роли диспетчера процесса сотворчества. Проверка и оценка знаний должны проводиться согласно дидактическим принципам

обучения. При этом выделяются следующие требования к оцениванию: объективность – создание условий, в которых бы максимально точно выявлялись знания обучаемых, предъявление к ним единых требований, справедливое отношение к каждому; обоснованность оценок – их аргументация; систематичность – важнейший психологический фактор, организующий и дисциплинирующий студентов, формирующий настойчивость и устремленность в достижении цели; всесторонность и оптимальность.

7. УЧЕБНО-МЕТОДИЧЕСКОЕ И ИНФОРМАЦИОННОЕ ОБЕСПЕЧЕНИЕ ДИСЦИПЛИНЫ

7.1.1. Перечень основной литературы:

1. Битюцкая, Н. И. Разработка программных приложений : лабораторный практикум / Н. И. Битюцкая. — Ставрополь : Северо-Кавказский федеральный университет, 2015. — 140 с.

2. Абрамов, Г. В. Проектирование и разработка информационных систем : учебное пособие для СПО / Г. В. Абрамов, И. Е. Медведкова, Л. А. Коробова. — Саратов : Профобразование, 2020. — 169 с. — ISBN 978-5-4488-0730-5. — Текст : электронный // Цифровой образовательный ресурс IPR SMART : [сайт]. — URL: https://www.iprbookshop.ru/88888.html (дата обращения: 10.04.2022). — Режим доступа: для авторизир. пользователей. - DOI: https://doi.org/10.23682/88888

7.1.2. Перечень дополнительной литературы:

1. Баженова, И. Ю. Основы проектирования приложений баз данных : учебное пособие / И. Ю. Баженова. — 3-е изд. — Москва : Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2020. — 324 с. — ISBN 978-5-4497-0682-9. — Текст : электронный // Цифровой образовательный ресурс IPR SMART : [сайт]. — URL: https://www.iprbookshop.ru/97569.html (дата обращения: 10.04.2022). — Режим доступа: для авторизир. пользователей

2. Гвозденко, Н. П. Разработка блок-схем алгоритмов : учебное пособие / Н. П. Гвозденко, С. А. Суслова. — Липецк : Липецкий государственный технический университет, ЭБС АСВ, 2021. — 59 с. — ISBN 978-5-00175-055-0. — Текст : электронный // Цифровой образовательный ресурс IPR SMART : [сайт]. — URL: https://www.iprbookshop.ru/116169.html (дата обращения: 10.04.2022). — Режим доступа: для авторизир. Пользователей

3. Савельев, А. О. Проектирование и разработка веб-приложений на основе технологий Microsoft : учебное пособие / А. О. Савельев, А. А. Алексеев. — 4-е изд. — Москва : Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2022. — 418 с. — ISBN 978-5-4497-1650-7. — Текст : электронный // Цифровой образовательный ресурс IPR SMART : [сайт]. — URL: https://www.iprbookshop.ru/120486.html (дата обращения: 10.04.2022). — Режим доступа: для авторизир. пользователей

7.2.Перечень учебно-методического обеспечения самостоятельной работы обучающихся по дисциплине:

3. Методические указания по выполнению контрольной работы по дисциплине «Технология разработки программного обеспечения».

4. Методические рекомендации для студентов по организации самостоятельной работы по дисциплине «Технология разработки программного обеспечения».

7.3. Перечень ресурсов информационно-телекоммуникационной сети «Интернет», необходимых для освоения дисциплины:

1. http://el.ncfu.ru/ – система управления обучением ФГАОУ ВО СКФУ. Дистанционная поддержка дисциплины «Цифровая грамотность и обработка данных»
2. http://www.un.org - Сайт ООН Информационно-коммуникационные технологии
3. http://www.intuit.ru – Интернет-Университет Компьютерных технологий

8. Перечень информационных технологий, используемых при осуществлении образовательного процесса по дисциплине (модулю), включая перечень программного обеспечения и информационных справочных систем

При чтении лекций используется компьютерная техника, демонстрации презентационных мультимедийных материалов. На семинарских и практических занятиях студенты представляют презентации, подготовленные ими в часы самостоятельной работы.

Информационные справочные системы:

Информационно-справочные и информационно-правовые системы, используемые при изучении дисциплины:

1	КонсультантПлюс - http://www.consultant.ru/

Программное обеспечение:

1	Операционная система: Microsoft Windows 8: 2013-02(3000). Бессрочная лицензия. Договор № 01-эа/13 от 25.02.2013. Окончание бесплатной поддержки – 2023-01 ИЛИ Операционная система: Microsoft Windows 10: 2016-08(20), 2017-10(67), 2018-01(18), 2018-04(6), 2018-05(6), 2019-02(7). Бессрочная лицензия. Договоры № 27-эа/16 от 02.08.2016. и № 0321100021117000009_229123 от 10.10.2017. На текущий момент окончания поддержки не анонсировано.
2	Базовый пакет программ Microsoft Office (Word, Excel, PowerPoint). MicrosoftOfficeStandard 2013: договор № 01-эа/13 от 25.02.2013г., Лицензирование Microsoft Office https://support.microsoft.com/ru-ru/lifecycle/search/16674 Дата начала жизненного цикла 09.01.2013г.; набор обновлений Office 2013 Service Pack1 Дата начала жизненного цикла 25.02.2014г., Дата окончания основной фазы поддержки 10.04.2018; Дополнительная дата окончания поддержки 11.04.2023г.

9. Описание материально-технической базы, необходимой для осуществления образовательного процесса по дисциплине (модулю)

Лабораторные занятия	Учебная аудитория для проведения занятий семинарского типа (практических работ) — компьютерная аудитория	Количество рабочих мест – 12 Оборудование: Персональные компьютеры, объединенные в локальную сеть и имеющие выход в интернет
Самостоятельная работа	Помещения для самостоятельной работы	Компьютеры с выходом в Интернет и обеспечением доступа в электронную информационно-образовательную среду ИСУ СКФУ.

Учебные аудитории для проведения учебных занятий, оснащены оборудованием и техническими средствами обучения. Помещения для самостоятельной работы обучающихся, оснащенные компьютерной техникой с возможностью подключения к сети "Интернет" и обеспечением доступа к электронной информационно-образовательной среде. Специализированная мебель и технические средства обучения, служащие для представления учебной информации.

Материально-техническая база обеспечивает проведение всех видов дисциплинарной и междисциплинарной подготовки, лабораторной, научно-исследовательской работы обучающихся (переносной ноутбук, переносной проектор, компьютеры с необходимым программным обеспечением и выходом в интернет).

Помещения для самостоятельной работы обучающихся оснащены компьютерной техникой с возможностью подключения к сети «Интернет» и обеспечением доступа в электронную информационно-образовательную среду образовательной организации.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Пятигорский институт (филиал) СКФУ

Методические указания
по организации самостоятельной работы обучающихся
по дисциплине «**ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**»

Направление подготовки/специальность **09.04.02 Информационные системы и технологии**
Направленность (профиль)/специализация **Технологии работы с данными и знаниями, анализ информации**
Форма обучения **очная, заочная**
Год начала обучения **2023**
Изучается в 3 семестре

СОДЕРЖАНИЕ

1. ЦЕЛЬ И ЗАДАЧИ ОСВОЕНИЯ ДИСЦИПЛИНЫ 3
2. КОМПЕТЕНЦИИ ОБУЧАЮЩЕГОСЯ, ФОРМИРУЕМЫЕ В РЕЗУЛЬТАТЕ ИЗУЧЕНИЯ ДИСЦИПЛИНЫ 3
3.ТЕХНОЛОГИЧЕСКАЯ КАРТА САМОСТОЯТЕЛЬНОЙ РАБОТЫ СТУДЕНТА 3
4. СОДЕРЖАНИЕ САМОСТОЯТЕЛЬНОЙ РАБОТЫ 4
5. ВОПРОСЫ К ЭКЗАМЕНУ 145
6. КРИТЕРИИ ОЦЕНИВАНИЯ КОМПЕТЕНЦИЙ 146
7. УЧЕБНО-МЕТОДИЧЕСКОЕ И ИНФОРМАЦИОННОЕ ОБЕСПЕЧЕНИЕ ДИСЦИПЛИНЫ 147

1. ЦЕЛЬ И ЗАДАЧИ ОСВОЕНИЯ ДИСЦИПЛИНЫ

Целью изучения дисциплины «Технологии разработки программного обеспечения» учебного плана подготовки магистров по направлению подготовки 09.04.02 «Информационные системы и технологии», является получение компетенций, достаточных для анализа требований к программным системам, их документирования, проектирования, разработки, тестирования, внедрения, управления программными проектами и управления качеством разработки программных систем.

Задачами курса является приобретение и развитие знаний, умений и навыков для производственно-технологической, организационно-управленческой, проектной и научно-исследовательской деятельности.

2. КОМПЕТЕНЦИИ ОБУЧАЮЩЕГОСЯ, ФОРМИРУЕМЫЕ В РЕЗУЛЬТАТЕ ИЗУЧЕНИЯ ДИСЦИПЛИНЫ

Код, формулировка компетенции	Код, формулировка индикатора	Планируемые результаты обучения по дисциплине (модулю), характеризующие этапы формирования компетенций, индикаторов
ПК-2 способен создания технической документации информационно-методического и маркетингового назначения в сфере информационных технологий и систем	**ИД-1 ПК-2** Создает техническую документации информационно-методического назначения в сфере информационных технологий и систем **ИД-2 ПК-2** Применяет техническую документацию для создания информационных технологий и систем **ИД-3 ПК-2** Использует техническую документацию для решения маркетинговых задач в сфере информационных технологий и систем;	Проводит предпроектное обследование объекта проектирования, анализ научно-технической информации, отечественного и зарубежного опыта
ПК-4 способен выполнять разработку систем управления базами данных, операционных систем, организацию разработки системного программного обеспечения, интеграция разработанного системного программного обеспечения	**ИД-1 ПК-4** Выполняет разработку систем управления базами данных. **ИД-2 ПК-4** Проводить непосредственное руководство процессами разработки программного обеспечения; **ИД-3 ПК-4** Проводить организацию разработки системного программного обеспечения, интеграцию разработанного системного программного обеспечения.	Проводит анализ разработанных программных приложений
ПК-6 способен проводить организационное сопровождение разработки, отладки, модификации и поддержки информационных технологий и систем	**ИД-1 ПК-6** Проводить организационное сопровождение процессом разработки ПО **ИД-2 ПК-6** Выполняет управление проектами в области ИТ любого масштаба в условиях высокой неопределенности **ИД-3 ПК-6** Проводить отладку, модификацию и поддержку информационных технологий и систем	Проводит техническое проектирование и сопровождение программных приложений
ПК-10 способен выполнять управление аналитическими работами и подразделением	**ИД-1 ПК-10** Выполняет разработку новых инструментов и методов управления проектами в области ИТ. **ИД-2 ПК-10** Проводить разработку новых инструментов; **ИД-3 ПК-10** Использует методы управления проектами в области	Разрабатывает отдельные компоненты информационной системы

	ИТ	

3.ТЕХНОЛОГИЧЕСКАЯ КАРТА САМОСТОЯТЕЛЬНОЙ РАБОТЫ ОБУЧАЮЩЕГОСЯ ОЧНОЙ ФОРМЫ ОБУЧЕНИЯ

Коды реализуемых компетенций	Вид деятельности студентов	Средства и технологии оценки	Объем часов, в том числе (астр.)		
			СРС	Контактная работа с преподавателем	Всего
3 семестр					
ИД-1ПК-2, ИД-2 ПК-2, ИД-1 ПК-4, ИД-2 ПК-4, ИД-1 ПК-9, ИД-2 ПК-9, ИД-1 ПК-10, ИД-2 ПК-10, ИД-1 ПК-12, ИД-2 ПК-12	Самостоятельное изучение литературы и источников	Собеседование	14,355	1,595	15,95
ИД-1ПК-2, ИД-2 ПК-2, ИД-1 ПК-4, ИД-2 ПК-4, ИД-1 ПК-9, ИД-2 ПК-9, ИД-1 ПК-10, ИД-2 ПК-10, ИД-1 ПК-12, ИД-2 ПК-12	Подготовка к лабораторным занятиям	Собеседование	3,645	0,405	4,05
ИД-1ПК-2, ИД-2 ПК-2, ИД-1 ПК-4, ИД-2 ПК-4, ИД-1 ПК-9, ИД-2 ПК-9, ИД-1 ПК-10, ИД-2 ПК-10, ИД-1 ПК-12, ИД-2 ПК-12	Вопросы к экзамену	Собеседование	18	2,0	20
ИД-1ПК-2, ИД-2 ПК-2, ИД-1 ПК-4, ИД-2 ПК-4, ИД-1 ПК-9, ИД-2 ПК-9, ИД-1 ПК-10, ИД-2 ПК-10, ИД-1 ПК-12, ИД-2 ПК-12	Оценочные средства для курсовой работы	Собеседование	18	2,0	20
Итого за 3 семестр			**54**	**6,0**	**60**
Итого			**54**	**6,0**	**60**

ЗАОЧНОЙ ФОРМЫ ОБУЧЕНИЯ

Коды реализуемых компетенций	Вид деятельности студентов	Средства и технологии оценки	Объем часов, в том числе (астр.)		
			СРС	Контактная работа с преподавателем	Всего
4 семестр					
ИД-1ПК-2, ИД-2 ПК-2, ИД-1 ПК-4, ИД-2 ПК-4, ИД-1 ПК-9, ИД-2 ПК-9, ИД-1 ПК-10, ИД-2 ПК-10, ИД-1 ПК-12, ИД-2 ПК-12	Самостоятельное изучение литературы и источников	Собеседование	45,81	50,9	50,9
ИД-1ПК-2, ИД-2 ПК-2, ИД-1 ПК-4, ИД-2 ПК-4, ИД-1 ПК-9, ИД-2 ПК-9, ИД-1 ПК-10, ИД-2 ПК-10, ИД-1 ПК-12, ИД-2 ПК-12	Подготовка к лабораторным занятиям	Собеседование	1,215	0,135	1,35
ИД-1ПК-2, ИД-2 ПК-2, ИД-1 ПК-4, ИД-2 ПК-4, ИД-1 ПК-9, ИД-2 ПК-9, ИД-1 ПК-10, ИД-2 ПК-10, ИД-1 ПК-12, ИД-2 ПК-12	Вопросы к экзамену	Собеседование	18	2,0	20
ИД-1ПК-2, ИД-2 ПК-2, ИД-1 ПК-4, ИД-2 ПК-4, ИД-1 ПК-9, ИД-2 ПК-9, ИД-1 ПК-10, ИД-2 ПК-10, ИД-1 ПК-12, ИД-2 ПК-12	Оценочные средства для курсовой работы	Собеседование	18	2,0	20
Итого за 4 семестр			**83,025**	**9,225**	**92,25**
Итого			**83,025**	**9,225**	**92,25**

4. СОДЕРЖАНИЕ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

Тема самостоятельного изучения № 1
Введение в технологии разработки программного обеспечения

Вид деятельности студентов: самостоятельное изучение литературы
Итоговый продукт самостоятельной работы: конспект
Средства и технологии оценки: собеседование
Краткие теоретические сведения

1 ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1.1 Основные этапы развития технологии разработки

Технологией программирования называют совокупность методов и средств, используемых в процессе разработки программного обеспечения. Как любая другая технология, технология программирования представляет собой набор технологических инструкций, включающих:

- указание последовательности выполнения технологических операций;
- перечисление условий, при которых выполняется та или иная операция;
- описания самих операций, где для каждой операции определены исходные данные, результаты, а также инструкции, нормативы, стандарты, критерии, методы оценки и т. п. (см. рисунок 1.1).

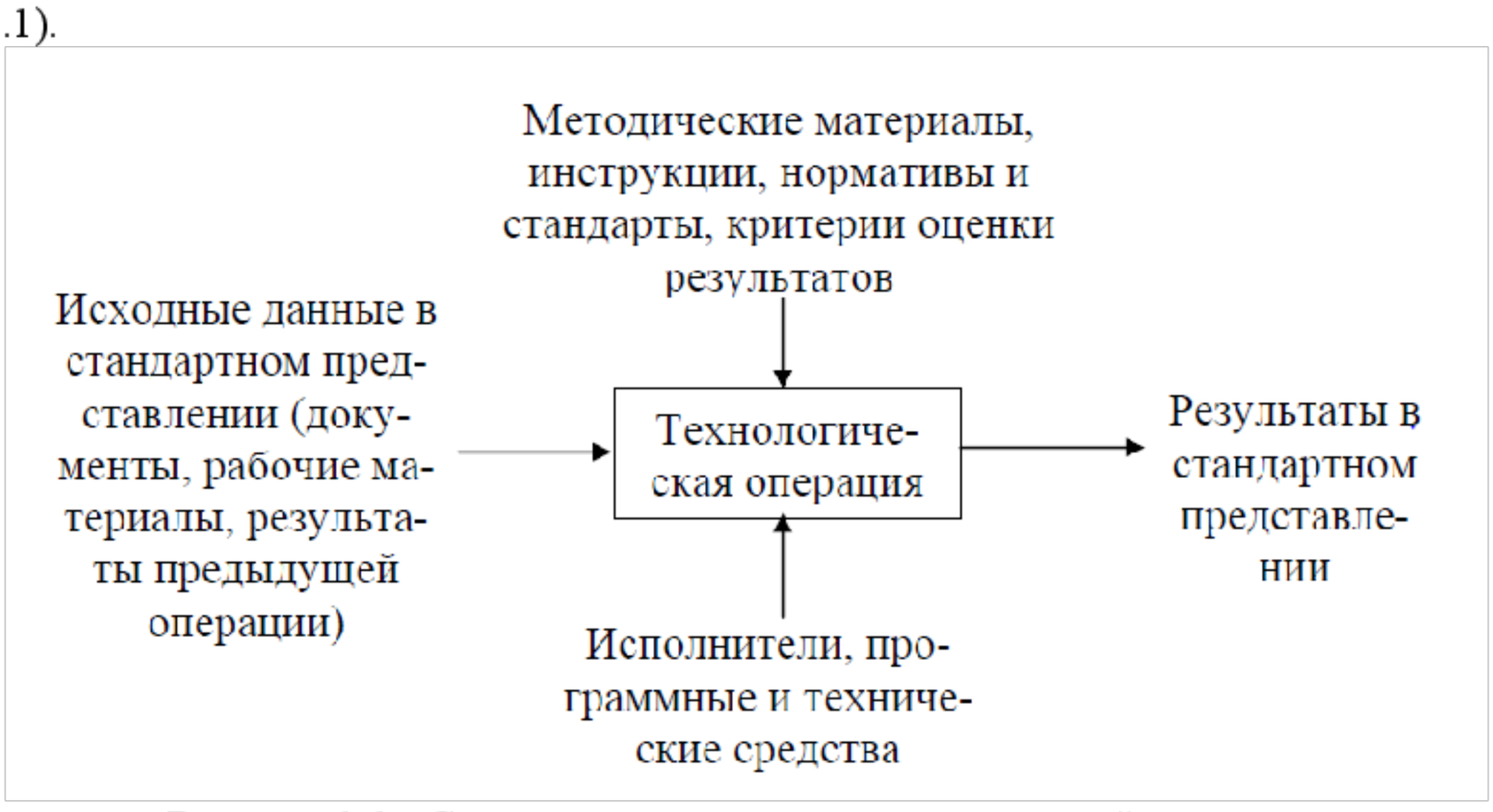

Рисунок 1.1 - Структура описания технологической операции

Кроме набора операций и их последовательности, технология также определяет способ описания проектируемой системы, точнее модели, используемой на конкретном этапе разработки.

Различают технологии, используемые на конкретных этапах разработки или для решения отдельных задач этих этапов, и технологии, охватывающие несколько этапов или весь процесс разработки. В основе первых, как правило, лежит ограниченно применимый *метод*, позволяющий решить конкретную задачу. В основе вторых – базовый метод или *подход*, определяющий совокупность методов, используемых на разных этапах разработки, или проектируемой системы, точнее модели, используемой на конкретном этапе разработки.

Чтобы разобраться в существующих технологиях программирования и определить основные тенденции их развития, целесообразно рассматривать эти технологии в историческом контексте, выделяя основные этапы развития программирования как науки.

1.1.1 Этап 1. «Стихийное» программирование

Этот этап охватывает период от момента появления первых вычислительных машин до середины 60-х гг. XX в. В это время практически отсутствовали технологии разработки программного обеспечения и программирование фактически было искусством. Первые

программы имели простейшую структуру. Они состояли из собственно программы на машинном языке и обрабатываемых ею данных (рисунок 1.2). Сложность программ в машинных кодах ограничивалась способностью программиста одновременно мысленно отслеживать последовательность выполняемых операций и местонахождение данных при программировании.

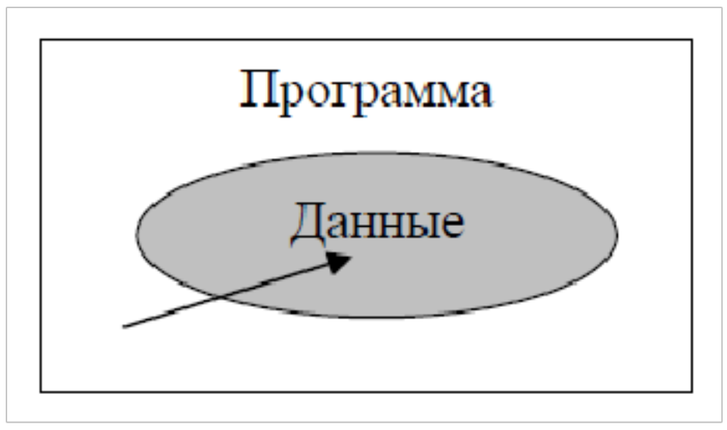

Рисунок 1.2. Структура первых программ

Появление ассемблеров позволило вместо двоичных или шестнадцатеричных кодов использовать символические имена данных и мнемоники ко- дов операций. В результате программы стали более «читаемыми».

Создание языков программирования высокого уровня, таких как FORTRAN и ALGOL, существенно упростило программирование вычислений, снизив уровень детализации операций. Это, в свою очередь, позволило увеличить сложность программ.

Революционным было появление в языках средств, позволяющих оперировать подпрограммами. (Идея написания подпрограмм появилась гораздо раньше, но отсутствие средств поддержки в первых языковых средствах существенно снижало эффективность их применения.) Подпрограммы можно было сохранять и использовать в других программах. В результате были созданы огромные библиотеки расчетных и служебных подпрограмм, которые по мере надобности вызывались из разрабатываемой программы.

Типичная программа того времени состояла из основной программы, области глобальных данных и набора подпрограмм (в основном библиотечных), выполняющих обработку всех данных или их части (рисунок 1.3).

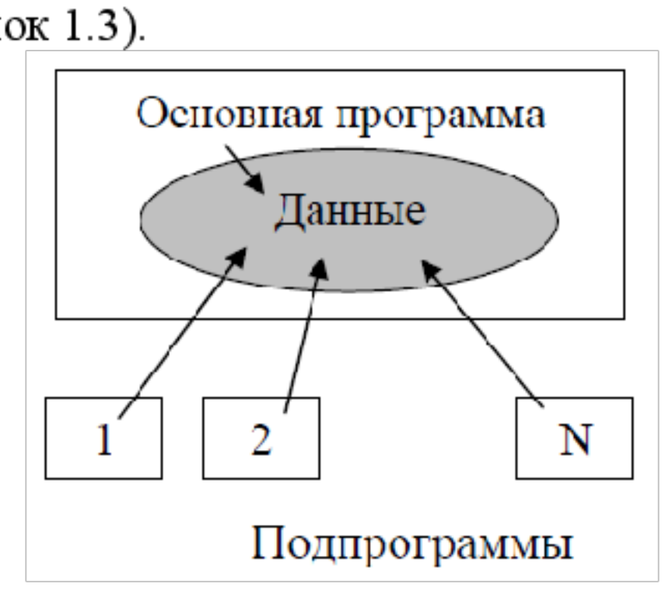

Рисунок 1.3 - Принцип работы программ с глобальной областью данных

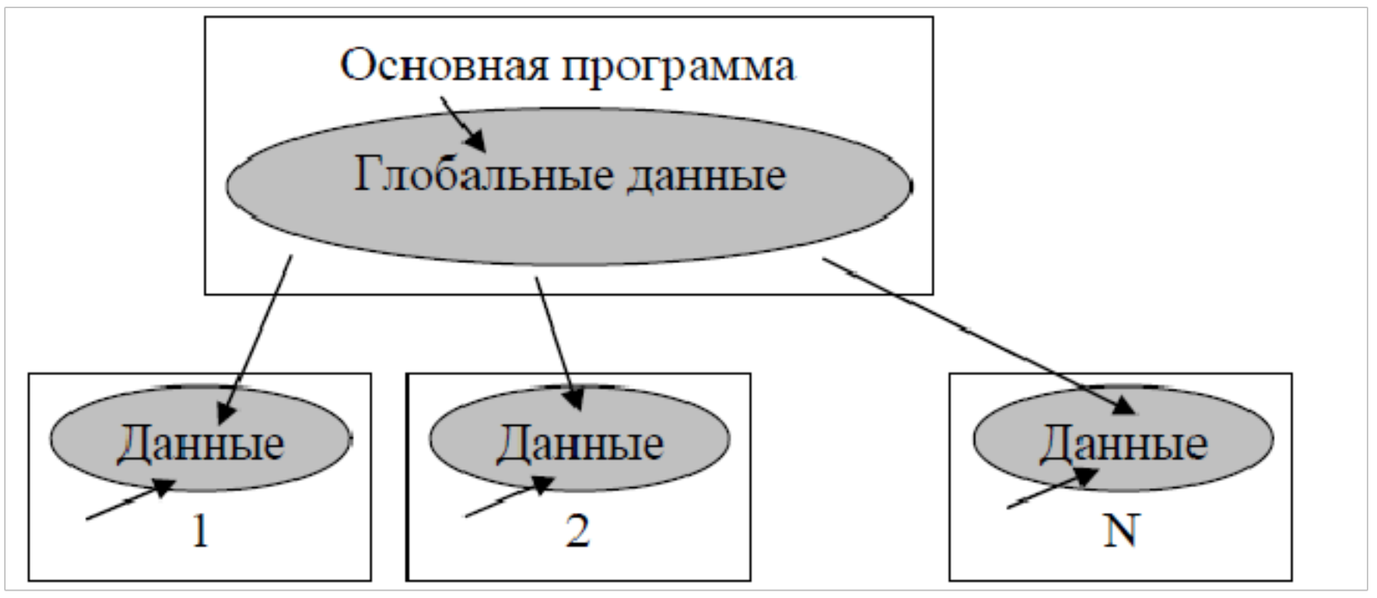

Рисунок 1.4 - Принцип работы программы, использующей подпрограммы с локальными данными

Слабым местом такой архитектуры было то, что при увеличении количества подпрограмм возрастала вероятность искажения части глобальных данных какой-либо подпрограммой. Например, подпрограмма поиска корней уравнения на заданном интервале по методу деления

отрезка пополам меняет величину интервала. Если при выходе из подпрограммы не предусмотреть восстановления первоначального интервала, то в глобальной области окажется неверное значение интервала. Чтобы сократить количество таких ошибок, было предложено в подпрограммах размещать локальные данные (рисунок 1.4).

Сложность разрабатываемого программного обеспечения при использовании подпрограмм с локальными данными по-прежнему ограничивалась возможностью программиста отслеживать процессы обработки данных, но уже на новом уровне. Однако появление средств поддержки подпрограмм позволило осуществлять разработку программного обеспечения нескольким программистам параллельно.

В начале 60-х гг. XX в. разразился «кризис программирования». Он выражался в том, что фирмы, взявшиеся за разработку сложного программного обеспечения такого, как операционные системы, срывали все сроки завершения проектов. Проект устаревал раньше, чем был готов к внедрению, увеличивалась его стоимость, и в результате многие проекты так никогда и не были завершены.

Объективно все это было вызвано несовершенством технологии программирования. Прежде всего, стихийно использовалась разработка «снизу- вверх» – подход, при котором вначале проектировали и реализовывали сравнительно простые подпрограммы, из которых затем пытались построить сложную программу. В отсутствии четких моделей описания подпрограмм и методов их проектирования создание каждой подпрограммы превращалось в непростую задачу, интерфейсы подпрограмм получались сложными, и при сборке программного продукта выявлялось большое количество ошибок согласования. Исправление таких ошибок, как правило, требовало серьезного изменения уже разработанных подпрограмм, что еще более осложняло ситуацию, так как при этом в программу часто вносились новые ошибки, которые также необходимо было исправлять... В конечном итоге процесс тестирования и отладки программ занимал более 80 % времени разработки, если вообще когда-нибудь заканчивался. На повестке дня самым серьезным образом стоял вопрос разработки технологии создания сложных программных продуктов, снижающей вероятность ошибок проектирования.

Анализ причин возникновения большинства ошибок позволил сформулировать новый подход к программированию, который был назван «структурным».

1.1.2 Этап 2. Структурный подход к программированию (60–70-е гг. XX в.)

Структурный подход к программированию представляет собой совокупность рекомендуемых технологических приемов, охватывающих выполнение всех этапов разработки программного обеспечения. В основе структурного подхода лежит декомпозиция (разбиение на части) сложных систем с целью последующей реализации в виде отдельных небольших подпрограмм. С появлением других принципов декомпозиции (объектного, логического и т. д.) данный способ получил название «процедурной декомпозиции».

В отличие от используемого ранее процедурного подхода к декомпозиции структурный подход требовал представления задачи в виде иерархии подзадач простейшей структуры. Проектирование, таким образом, осуществлялось «сверху-вниз» и подразумевало реализацию общей идеи, обеспечивая проработку интерфейсов подпрограмм. Одновременно вводились ограничения на конструкции алгоритмов, рекомендовались формальные модели их описания, а также специальный метод проектирования алгоритмов – метод пошаговой детализации.

Поддержка принципов структурного программирования была заложена в основу так называемых процедурных языков программирования. Как правило, они включали основные «структурные» операторы передачи управления, поддерживали вложение подпрограмм, локализацию и ограничение области «видимости» данных. Среди наиболее известных языков этой группы стоит назвать PL/1, ALGOL-68, Pascal, C.

Одновременно со структурным программированием появилось огромное количество языков, базирующихся на других концепциях, но большинство из них не выдержало конкуренции. Какие-то языки были просто забыты, идеи других были в дальнейшем использованы в следующих версиях развиваемых языков.

Дальнейший рост сложности и размеров разрабатываемого программного обеспечения потребовал развития структурирования данных. Как следствие этого в языках появляется возможность определения пользовательских типов данных. Одновременно усилилось стремление разграничить доступ к глобальным данным программы, чтобы уменьшить

количество ошибок, возникающих при работе с глобальными данными. В результате появилась и начала развиваться технология модульного программирования.

Модульное программирование предполагает выделение групп подпрограмм, использующих одни и те же глобальные данные в отдельно компилируемые модули (библиотеки подпрограмм), например, модуль графических ресурсов, модуль подпрограмм вывода на принтер (рисунок 1.5). Связи между модулями при использовании данной технологии осуществляются через специальный интерфейс, в то время как доступ к реализации модуля (телам под- программ и некоторым «внутренним» переменным) запрещен. Эту технологию поддерживают современные версии языков Pascal и С (С++), языки Ада и Modula.

Использование модульного программирования существенно упростило разработку программного обеспечения несколькими программистами. Теперь каждый из них мог разрабатывать свои модули независимо, обеспечивая взаимодействие модулей через специально оговоренные межмодульные интерфейсы. Кроме того, модули в дальнейшем без изменений можно было использовать в других разработках, что повысило производительность труда программистов.

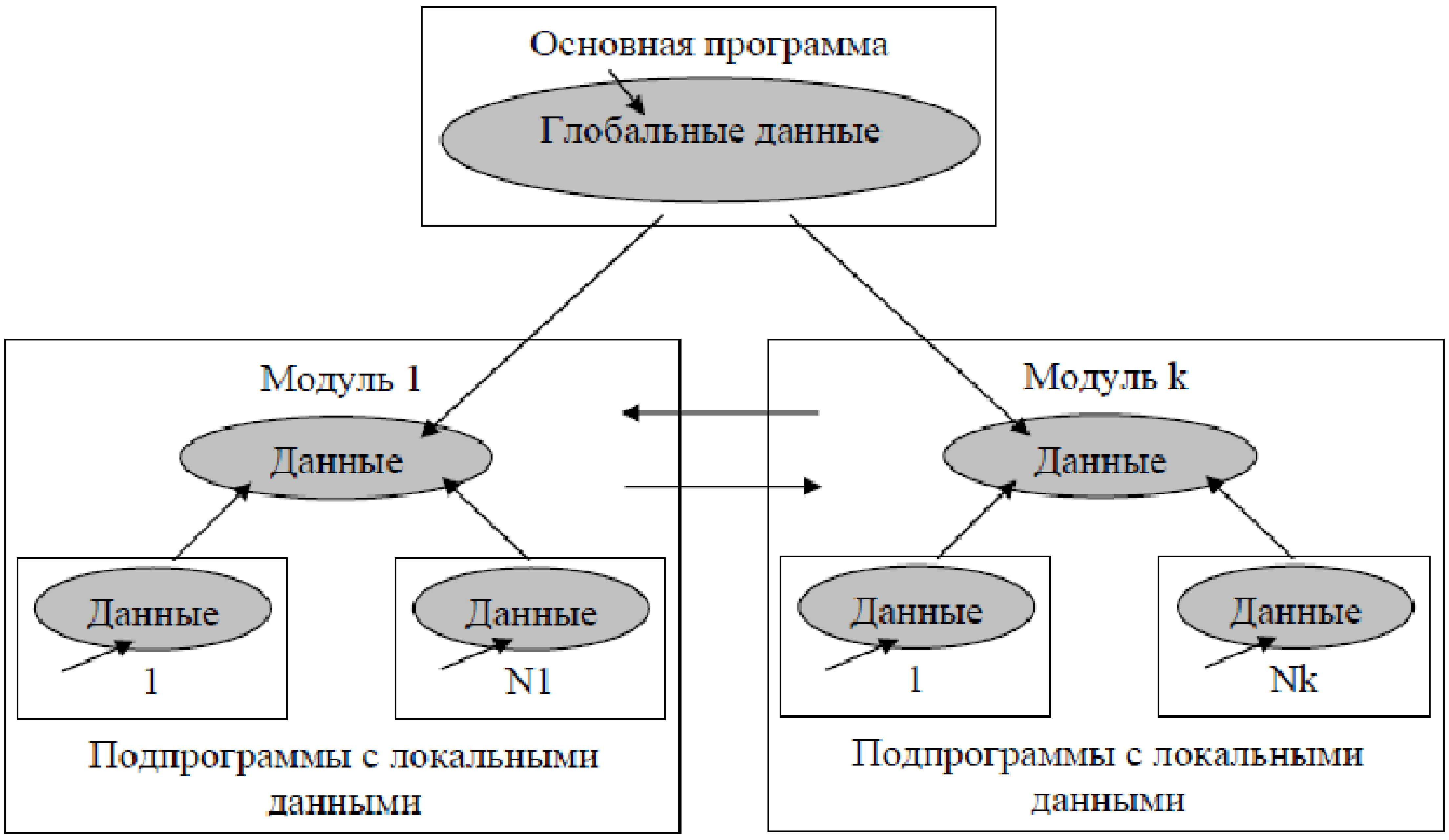

Рисунок 1.5 - Модульная структура программ

Практика показала, что структурный подход в сочетании с модульным программированием позволяет получать достаточно надежные программы, размер которых не превышает 100 000 операторов. Узким местом модульного программирования является то, что ошибка в интерфейсе при вызове подпрограммы выявляется только при выполнении программы (из-за раздельной компиляции модулей обнаружить эти ошибки раньше невозможно). При увеличении размера программы обычно возрастает сложность межмодульных интерфейсов, и с некоторого момента предусмотреть взаимовлияние отдельных частей программы становится практически невозможно. Для разработки программного обеспечения большого объема было предложено использовать объектный подход.

1.1.3 Этап 3. Объектный подход к программированию (с середины 1980-х гг. до нашего времени)

Объектно-ориентированное программирование определяется как технология создания сложного программного обеспечения, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса (рисунок 1.6), а классы образуют иерархию с наследованием свойств. Взаимодействие программных объектов в такой системе осуществляется путем передачи сообщений (рисунок 1.7).

Объектная структура программы впервые была использована в языке имитационного моделирования сложных систем Simula, появившемся еще в 60-х гг. XX в. Естественный для языков моделирования способ представления программы получил развитие в другом специализированном языке моделирования – языке Smalltalk (70-е гг. XX в.), а затем был использован в новых версиях универсальных языков программирования, таких как Pascal, С+

+, Modula, Java.

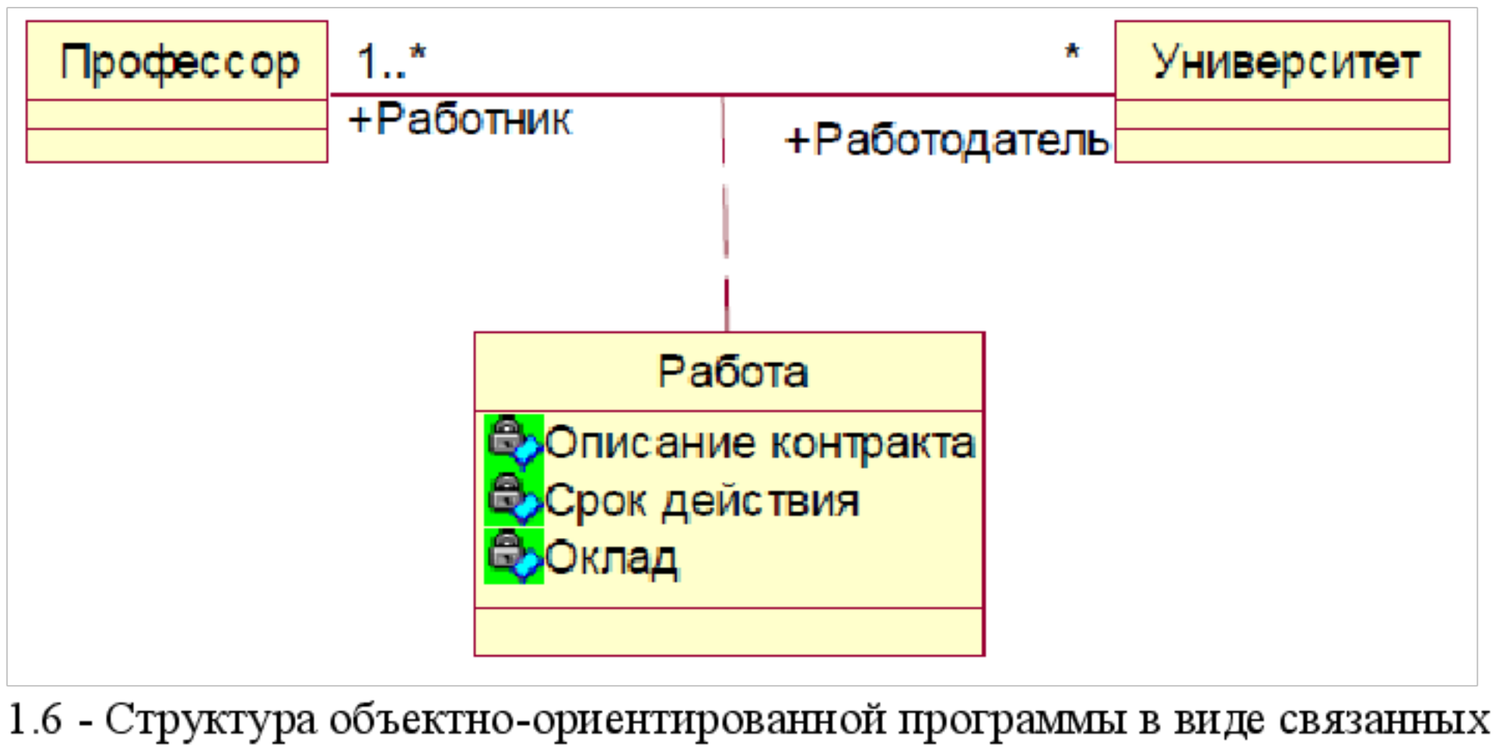

Рисунок 1.6 - Структура объектно-ориентированной программы в виде связанных классов

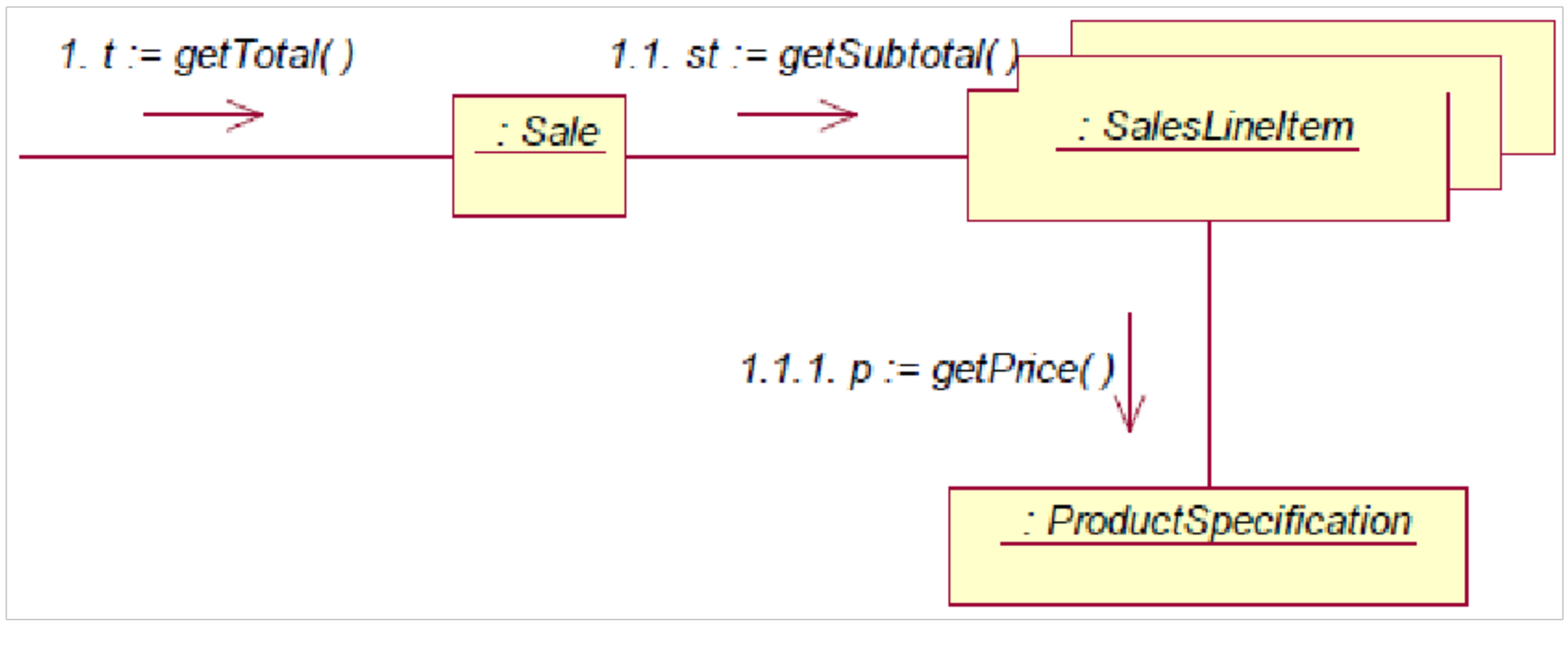

Рисунок 1.7 - Взаимодействие объектов в объектно-ориентированных программах

Основным достоинством объектно-ориентированного программирования по сравнению с модульным программированием является «более естественная» декомпозиция программного обеспечения, которая существенно облегчает его разработку. Это приводит к более полной локализации данных и интегрированию их с подпрограммами обработки, что позволяет вести практически независимую разработку отдельных частей (объектов) программы. Кроме этого, объектный подход предлагает новые способы организации программ, основанные на механизмах наследования, полиморфизма, композиции, наполнения. Эти механизмы позволяют конструировать сложные объекты из сравнительно простых. В результате существенно увеличивается показатель повторного использования кодов и появляется возможность создания библиотек классов для различных применений.

Бурное развитие технологий программирования, основанных на объектном подходе, позволило решить многие проблемы. Так, были созданы среды, поддерживающие визуальное программирование, например, Delphi, C++ Builder, Visual C++ и т. д. При использовании визуальной среды у программиста появляется возможность проектировать некоторую часть, например, интерфейсы будущего продукта, с применением визуальных средств добавления и настройки специальных библиотечных компонентов. Результатом визуального проектирования является заготовка будущей программы, в которую уже внесены соответствующие коды.

Использование объектного подхода имеет много преимуществ, однако его конкретная реализация в объектно-ориентированных языках программирования, таких как Pascal и C++, имеет существенные недостатки:

- фактически отсутствуют стандарты компоновки двоичных результатов компиляции объектов в единое целое даже в пределах одного языка программирования: компоновка объектов, полученных разными компиляторами C++ в лучшем случае проблематична, что приводит к необходимости разработки;
- программного обеспечения с использованием средств и возможностей одного языка программирования высокого уровня и одного компилятора, а значит, требует наличия

исходных кодов используемых библиотек классов;

- изменения реализации одного из программных объектов, как минимум, связано с перекомпиляцией соответствующего модуля и перекомпоновкой всего программного обеспечения, использующего данный объект.

Таким образом, при использовании этих языков программирования сохраняется зависимость модулей программного обеспечения от адресов экспортируемых полей и методов, а также структур и форматов данных. Эта зависимость объективна, так как модули должны взаимодействовать между собой, обращаясь к ресурсам друг друга. Связи модулей нельзя разорвать, но можно попробовать стандартизировать их взаимодействие, на чем и основан компонентный подход к программированию.

1.1.4 Этап 4. Компонентный подход и CASE-технологии (с середины 1990-х гг. до нашего времени)

Компонентный подход предполагает построение программного обеспечения из отдельных компонентов – физически отдельно существующих частей программного обеспечения, которые взаимодействуют между собой через стандартизованные двоичные интерфейсы. В отличие от обычных объектов объекты-компоненты можно собрать в динамически вызываемые библиотеки или исполняемые файлы, распространять в двоичном виде (без исходных текстов) и использовать в любом языке программирования, поддерживающем соответствующую технологию. Сегодня рынок объектов стал реальностью: так, в Интернете существуют узлы, предоставляющие большое количество компонентов, рекламой компонентов забиты журналы. Это позволяет программистам создавать продукты, хотя бы частично состоящие из повторно использованных частей, т. е. применять технологию, хорошо зарекомендовавшую себя в области проектирования аппаратуры.

Компонентный подход лежит в основе технологий, разработанных на базе COM (Component Object Model – компонентная модель объектов), и технологии создания распределенных приложений CORBA (Common Object Request Broker Architecture – общая архитектура с посредником обработки запросов объектов). Эти технологии используют сходные принципы и различаются лишь особенностями их реализации.

Технология COM фирмы Microsoft является развитием технологии OLE (Object Linking and Embedding – связывание и внедрение объектов), которая использовалась в ранних версиях Windows для создания составных документов. Технология COM определяет общую парадигму взаимодействия программ любых типов: библиотек, приложений, операционной системы, т. е. позволяет одной части программного обеспечения использовать функции (*службы*), предоставляемые другой, независимо от того, функционируют ли эти части в пределах одного процесса, в разных процессах на одном компьютере или на разных компьютерах (рисунок 1.8). Модификация COM, обеспечивающая передачу вызовов между компьютерами, называется DCOM (Distri- buted COM – распределенная COM).

По технологии COM приложение предоставляет свои службы, используя специальные объекты – объекты COM, которые являются экземплярами классов COM. Объект COM, так же как обычный объект, включает поля и методы, но в отличие от обычных объектов каждый объект COM может реализовывать несколько интерфейсов, обеспечивающих доступ к его полям и функциям. Это достигается за счет организации отдельной таблицы адресов методов для каждого интерфейса (по типу таблиц виртуальных методов).

При этом интерфейс обычно объединяет несколько однотипных функций.

Кроме того, классы COM поддерживают наследование интерфейсов, но не поддерживают наследования реализации, т. е. не наследуют код методов, хотя при необходимости объект класса-потомка может вызвать метод родителя.

Каждый интерфейс имеет имя, начинающееся с символа I и глобальный уникальный идентификатор IID (Interface IDentifier). Любой объект COM обязательно реализует интерфейс (Unknown (на схемах этот интерфейс всегда располагают сверху). Использование этого интерфейса позволяет получить доступ к остальным интерфейсам объекта.

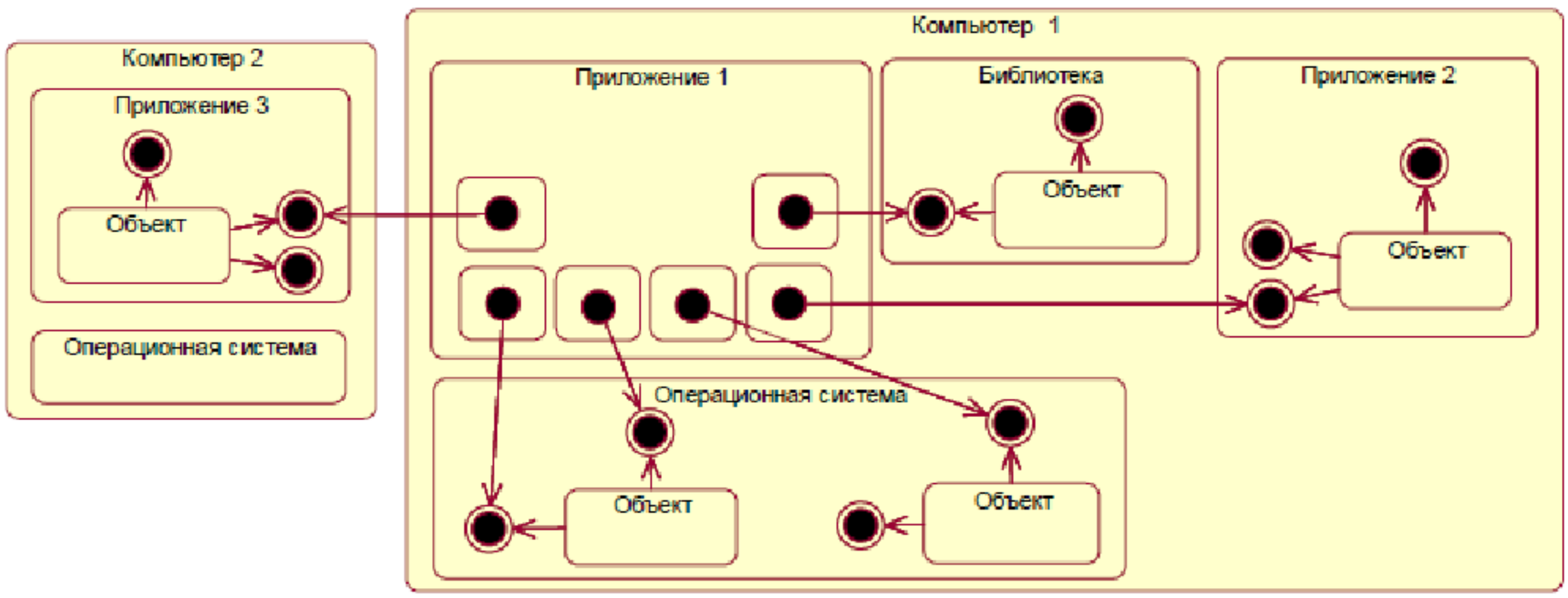

Рисунок 1.8 - Взаимодействие программных компонентов различных типов

Объект всегда функционирует в составе сервера – динамической библиотеки или исполняемого файла, которые обеспечивают функционирование объекта. Различают три типа серверов:

- внутренний сервер: реализуется динамическими библиотеками, которые подключаются к приложению-клиенту и работают в одном с ними адресном пространстве. Это наиболее эффективный сервер, кроме того, он не требует специальных средств;
- локальный сервер: создается отдельным процессом (exe), который работает на одном компьютере с клиентом;
- удаленный сервер: создается процессом, который работает на другом компьютере.

Например, Microsoft Word является локальным сервером. Он включает множество объектов, которые могут использоваться другими приложениями.

Взаимодействие клиента и сервера обеспечивается базовыми механизмами COM или DCOM, поэтому клиенту безразлично местонахождение объекта. При использовании локальных и удаленных серверов в адресном пространстве клиента создается *proxy-объект* – заместитель объекта COM, а в адресном пространстве сервера COM – *заглушка,* соответствующая клиенту. Получив задание от клиента, заместитель упаковывает его параметры и, используя службы операционной системы, передает вызов заглушке. Заглушка распаковывает задание и передает его объекту COM. Результат возвращается клиенту в обратном порядке.

1.1.5 этап 5. Разработка, ориентированная на архитектуру и CASE-технологии (с начала XXI в. до нашего времени)

До конца XX в. все возможные проектные модели употребляли исключительно для документирования промежуточных и заключительных этапов разработки программного обеспечения, для чего применялись различные графические нотации и технологии, которые впоследствии были использованы для создания стандарта объектного моделирования.

В ноябре 1997 г. после продолжительного процесса объединения различных методик группа OMG (Object Management Group) приняла получившийся в результате унифицированный язык моделирования (Unified Model Language – UML) в качестве стандарта. В 2001 г. члены OMG начали работу над новой версией UML, добавляя в нее недостающие элементы и устраняя недостатки, выявленные в UML1. Версия UML2 была принята в 2004 г. С официальной спецификацией UML можно ознакомиться на веб-сайте OMG по адресу www.omg.org.

Идея создания языка UML включала в себя не только реализацию стандарта для документирования и общения разработчиков, но и реализацию возможности использования UML как языка программирования. Этот момент вызывал массу проблем при осуществлении, так как язык визуального моделирования по определению не мог содержать в себе всей выразительности объектно-ориентированных языков в плане проектирования (программирования) динамики и реализации алгоритмов. Нужен был язык, который на более высоком (абстрактном) уровне смог бы обеспечить разработку на UML. Такой язык был создан – это объектный язык ограничений OCL (Object Con- straint Language).

Тенденции развития средств разработки программных систем заключаются в создании

таких средств, которые обеспечили бы не только автоматизацию всех этапов и процессов разработки программных систем, но и связь между результатами этапов. Одним из ключевых соединительных узлов является связь между проектными моделями и программным кодом. Когда разработка программных систем начинается от проектирования ее структуры до последующего кодирования и все изменения в функциях разрабатываемой системы реализуются, начиная с перепроектирования архитектуры, то такая технология называется ориентированной на архитектуру (Model Driven Ar- chitecture – MDA).

Компания Borland, начиная с седьмой версии своей среды разработки (Delphi) уже использует набор компонент, реализующий подход, ориентированный на архитектуру, но в этой версии присутствует еще масса недостатков и недоработок. В последней своей версии (Delphi 2006) компания Borland модернизировала технологию, ориентированную на архитектуру, что позволило использовать ее для разработки промышленных приложений, в том числе и для платформы Microsoft Framework .Net.

1.2 Эволюция моделей жизненного цикла программного обеспечения

1.2.1 Каскадная модель

Первоначально (1970–1985 гг.) была предложена и использовалась *каскадная схема разработки программного обеспечения* (ПО) (рисунок 1.9), которая предполагала, что переход на следующую стадию осуществляется после того, как полностью будут завершены проектные операции предыдущей стадии и получены все исходные данные для следующей стадии.

Именно такую схему и используют обычно при блочно-иерархическом подходе к разработке сложных *технических* объектов, обеспечивая очень высокие параметры эффективности разработки. Однако данная схема оказалась применимой только к созданию систем, для которых в самом начале разработки удавалось точно и полно сформулировать все требования. Это уменьшало вероятность возникновения в процессе разработки проблем, связанных с принятием неудачного решения на предыдущих стадиях. На практике такие разработки встречаются крайне редко.

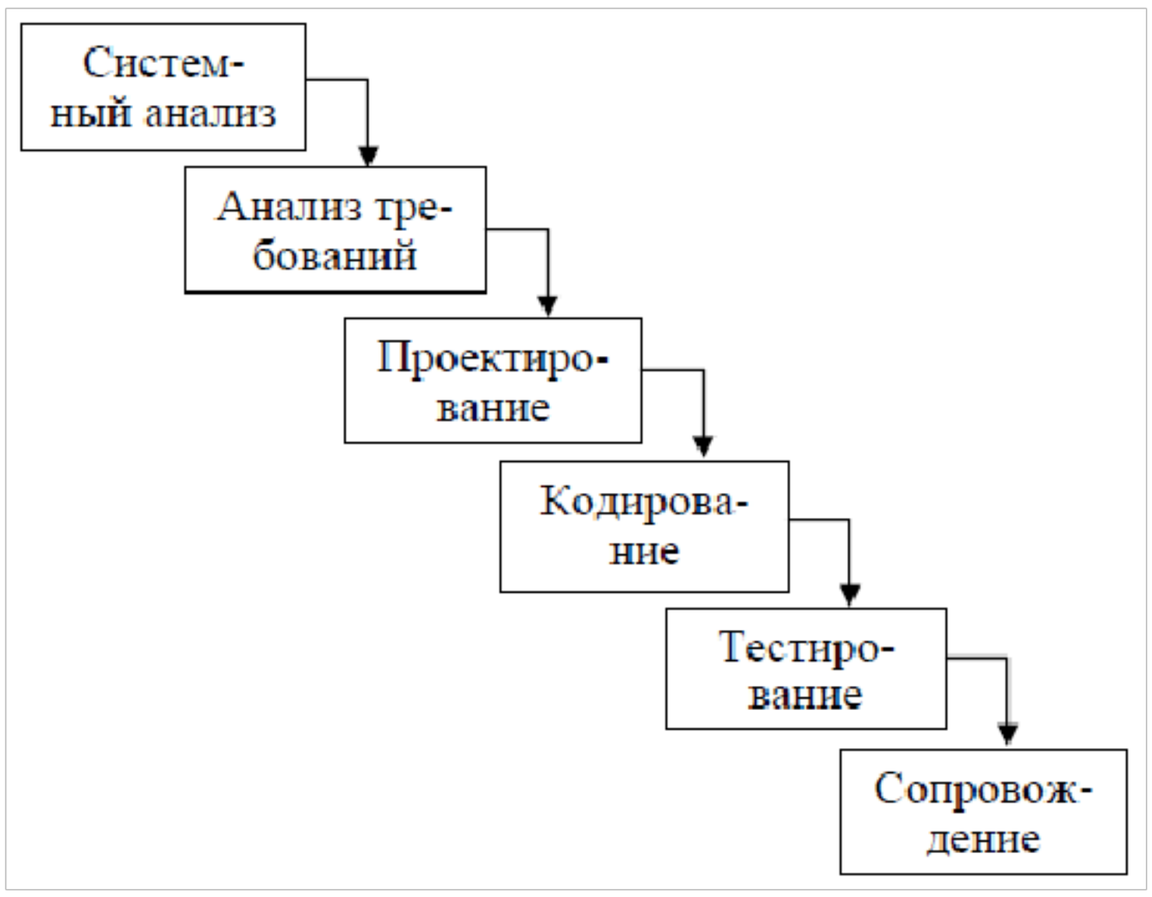

Рисунок - 1.9 Каскадная модель разработки программного обеспечения

Схема, поддерживающая итерационный характер процесса разработки, была названа схемой с промежуточным контролем (рисунок 1.10). Контроль, который выполняется по данной схеме после завершения каждого этапа, позволяет при необходимости вернуться на любой уровень и внести необходимые изменения.

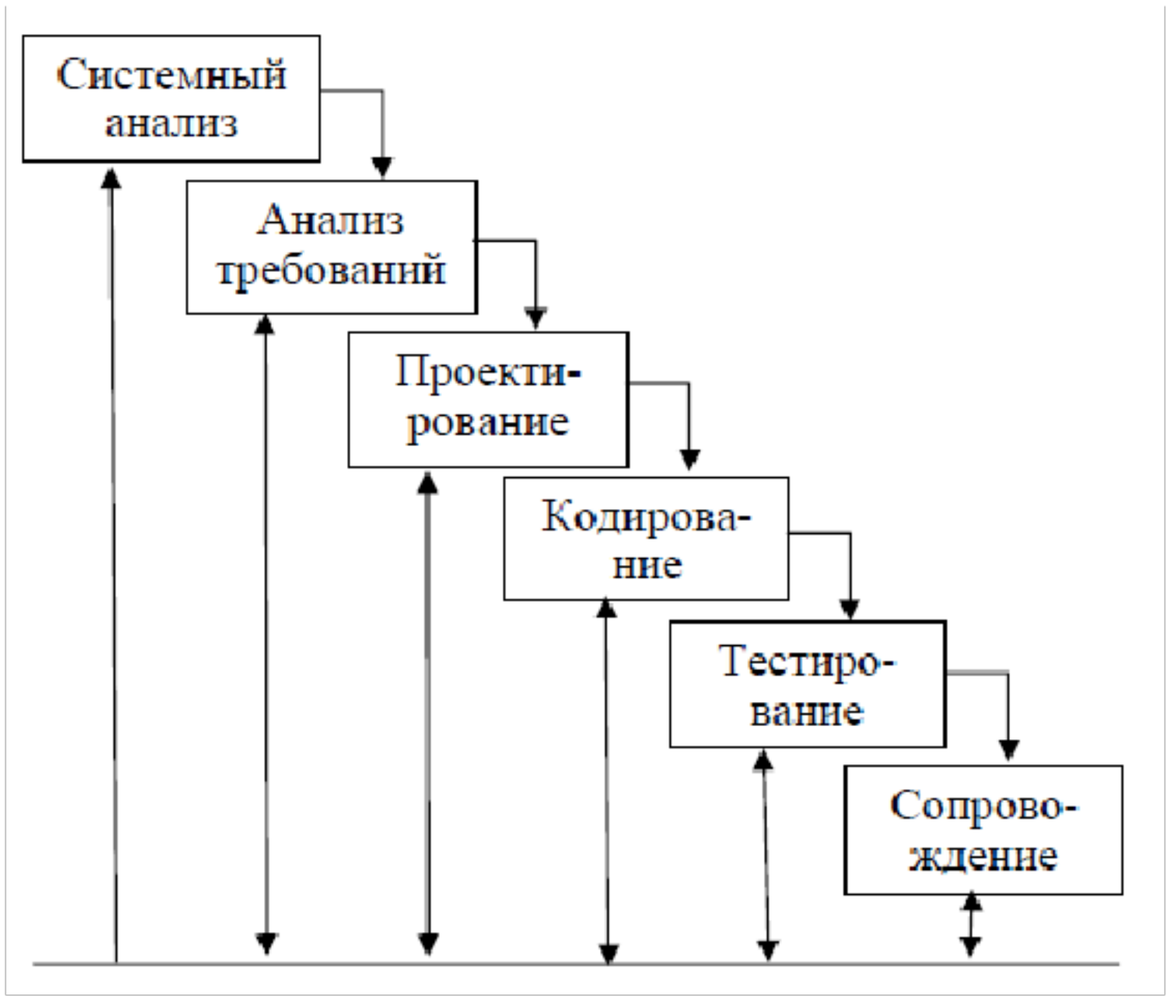

Рисунок 1.10 - Каскадная модель разработки ПО с промежуточным контролем

Охарактеризуем содержание основных этапов.

Подразумевается, что разработка начинается на системном уровне и проходит через анализ, проектирование, кодирование, тестирование и сопровождение. При этом моделируются действия стандартного инженерного цикла.

Системный анализ задает роль каждого элемента в компьютерной системе, взаимодействие элементов друг с другом. Поскольку ПО является лишь частью большой системы, то анализ начинается с определения требований ко всем системным элементам и назначения подмножества этих требований программному элементу. Необходимость системного подхода явно проявляется, когда формируется интерфейс ПО с другими элементами (аппаратурой, людьми, базами данных). На этом же этапе начинается решение задачи планирования проекта ПО. В ходе планирования проекта определяются объем проектных работ и их риск, необходимые трудозатраты, формируются рабочие задачи и план-график работ.

Анализ требований относится к программному элементу – программному обеспечению. Уточняются и детализируются его функции, характеристики и интерфейс.

Все определения документируются в *спецификации анализа*. Здесь же завершается решение задачи планирования проекта.

Проектирование состоит в создании представлений:

- архитектуры ПО;
- модульной структуры ПО;
- алгоритмической структуры ПО;
- структуры данных;
- входного и выходного интерфейса (входных и выходных форм данных).

Исходные данные для проектирования содержатся в *спецификации анализа*, т. е. в ходе проектирования выполняется трансляция требований к ПО во множество проектных представлений: при решении задач проектирования основное внимание уделяется качеству будущего программного продукта.

Кодирование – перевод результатов проектирования в текст на языке программирования.

Тестирование – выполнение программы для выявления дефектов в функциях, логике и форме реализации программного продукта.

Сопровождение – это внесение изменений в эксплуатируемое ПО. Цели изменений:

- исправление ошибок;
- адаптация к изменениям внешней для ПО среды;
- усовершенствование ПО по требованиям заказчика.

Сопровождение ПО состоит в повторном применении каждого из предшествующих шагов (этапов) жизненного цикла к существующей программе, но не в разработке новой программы.

Как и любая инженерная схема, классический жизненный цикл имеет достоинства и

недостатки.

Достоинства классического жизненного цикла:

- получение в конце каждой стадии законченного набора проектной документации, отвечающего требованиям полноты и согласованности;
- простота планирования процесса разработки.

Недостатки классического жизненного цикла:

- реальные проекты часто требуют отклонения от стандартной последовательности шагов;
- цикл основан на точной формулировке исходных требований к ПО (реально в начале проекта требования заказчика определены лишь частично);
- результаты проекта доступны заказчику только в конце работы.

1.2.2 Спиральная модель

Спиральная модель (автор Барри Боэм, 1988) базируется на лучших свойствах классического жизненного цикла и макетирования, к которым добавляется новый элемент – анализ риска, отсутствующий в этих парадигмах.

Как показано на рисунке 1.11, модель определяет четыре действия, представляемые четырьмя квадрантами спирали.

1. Планирование – определение целей, вариантов и ограничений.
2. Анализ риска – анализ вариантов и распознавание/выбор риска.
3. Конструирование – разработка продукта следующего уровня.
4. Оценивание – оценка заказчиком текущих результатов конструирования.

Интегрирующий аспект спиральной модели очевиден при учете радиального измерения спирали. С каждой итерацией по спирали (продвижением от центра к периферии) строятся все более полные версии ПО.

Рисунок 1.11 - Спиральная модель: *1* – начальный сбор требований и планирование проекта; *2* – та же работа, но на основе рекомендаций заказчика; *3* – анализ риска на основе начальных требований; *4* – анализ риска на основе реакции заказчика; *5* – переход к комплексной системе; *6* – начальный макет системы; *7* – следующий уровень макета; *8* – сконструированная система; *9* – оценивание заказчиком

В первом витке спирали определяются начальные цели, варианты и ограничения, распознается и анализируется риск. Если анализ риска показывает неопределенность требований, на помощь разработчику и заказчику приходит макетирование (используемое в квадранте конструирования). Для дальнейшего определения проблемных и уточненных требований может быть использовано моделирование. Заказчик оценивает инженерную (конструкторскую) работу и вносит предложения по модификации (квадрант оценки заказчиком). Следующая фаза планирования и анализа риска базируется на предложениях заказчика. В каждом цикле по спирали результаты анализа риска формируются в виде «продолжать, не продолжать». Если риск слишком велик, проект может быть остановлен.

В большинстве случаев движение по спирали продолжается, с каждым шагом продвигая

разработчиков к более общей модели системы. В каждом цикле по спирали требуется конструирование (нижний правый квадрант), которое может быть реализовано классическим жизненным циклом или макетированием. Заметим, что количество действий по разработке (происходящих в правом нижнем квадранте) возрастает по мере продвижения от центра спирали.

Достоинства спиральной модели:

- наиболее реально (в виде эволюции) отображает разработку программного обеспечения;
- позволяет явно учитывать риск на каждом витке эволюции разработки;
- включает шаг системного подхода в итерационную структуру разработки;
- использует моделирование для уменьшения риска и совершенствования программного изделия.

Недостатки спиральной модели:

- повышенные требования к заказчику;
- трудности контроля и управления временем разработки.

1.2.3 Макетирование

Достаточно часто заказчик не может сформулировать подробные требования по вводу, обработке или выводу данных для будущего программного продукта. С другой стороны, разработчик может сомневаться в приспосабливаемости продукта под операционную систему, в форме диалога с пользователем или в эффективности реализуемого алгоритма. В этих случаях целесообразно использовать макетирование.

Основная цель макетирования – снять неопределенности в требованиях заказчика.

Макетирование (прототипирование) – это процесс создания модели требуемого программного продукта.

Модель может принимать одну из трех форм:

- бумажный макет или макет на основе ПК (изображает или рисует человеко-машинный диалог);
- работающий макет (выполняет некоторую часть требуемых функций);
- существующая программа (характеристики которой затем должны быть улучшены).

Макетирование основывается на многократном повторении итераций, в которых участвуют заказчик и разработчик (рисунок 1.12).

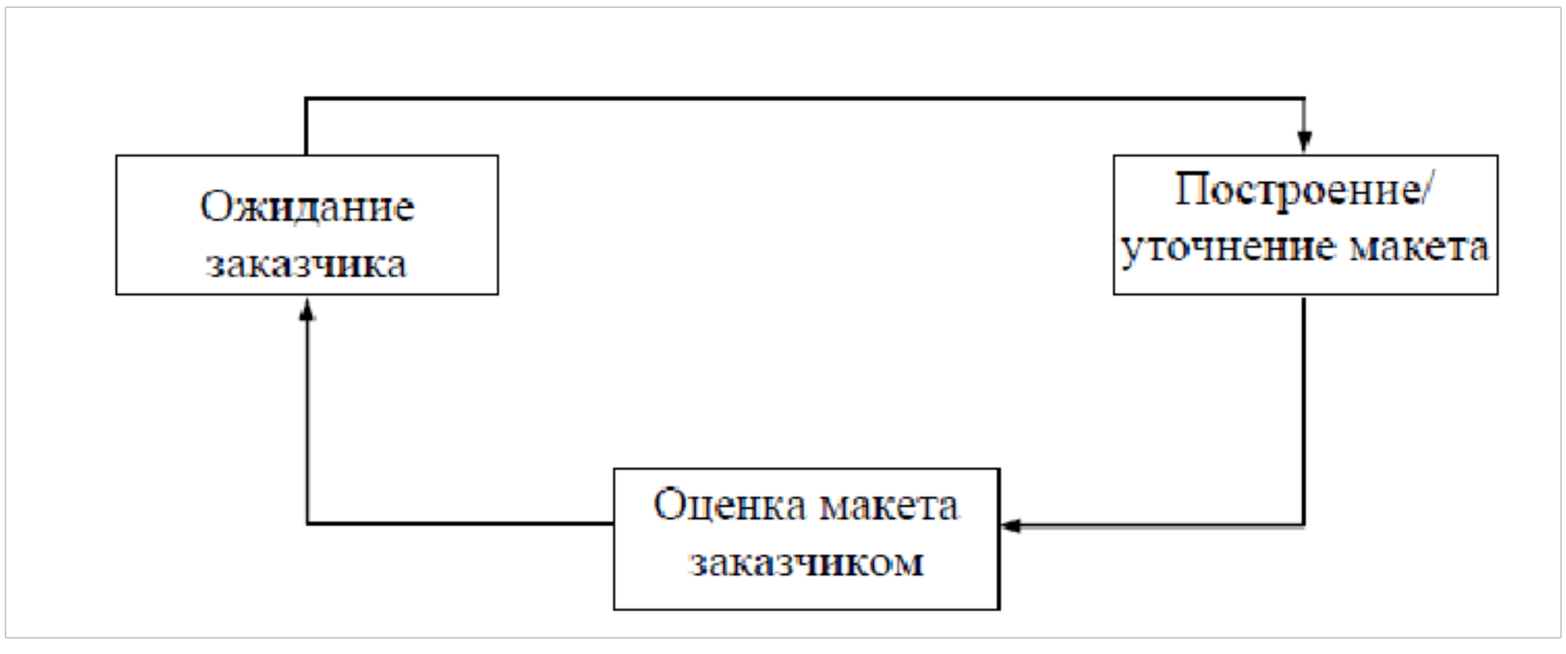

Рисунок 1.12 - Макетирование

Последовательность действий при макетировании представлена на рисунке 1.13. Макетирование начинается со сбора и уточнения требований к создаваемому ПО. Разработчик и заказчик встречаются и определяют все цели ПО, устанавливают, какие требования известны, а какие предстоит доопределить.

Затем выполняется быстрое проектирование. В нем внимание сосредоточивается на тех характеристиках ПО, которые должны быть видимы пользователю.

Быстрое проектирование приводит к построению макета.

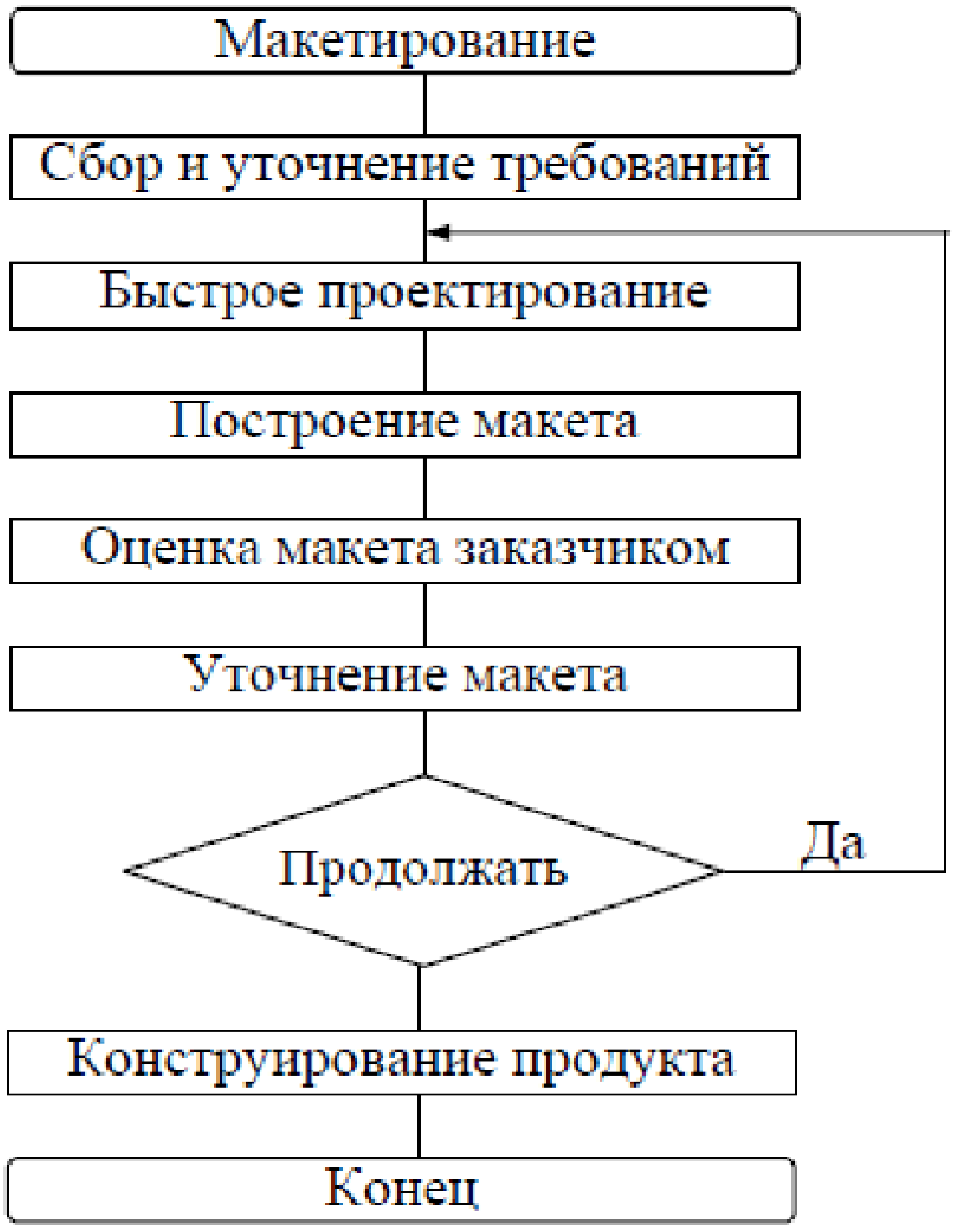

Рисунок 1.13 - Последовательность действий при макетировании

Макет оценивается заказчиком и используется для уточнения требований к ПО. Итерации повторяются до тех пор, пока макет не выявит все требования заказчика и тем самым не даст возможность разработчику понять, что должно быть сделано.

Достоинство макетирования – обеспечивает определение полных требований к ПО.

Недостатки макетирования:

- заказчик может принять макет за продукт;
- разработчик может принять макет за продукт.

Поясним суть недостатков. Когда заказчик видит работающую версию ПО, он перестает сознавать, что детали макета скреплены «жевательной резинкой и проволокой»; он забывает, что в погоне за работающим вариантом оставлены нерешенными вопросы качества и удобства сопровождения ПО. Когда заказчику говорят, что продукт должен быть перестроен, он начинает возмущаться и требовать, чтобы макет «в три приема» был превращен в рабочий продукт. Очень часто это отрицательно сказывается на управлении разработкой ПО. С другой стороны, для быстрого получения работающего макета разра- ботчик часто идет на определенные компромиссы. Могут использоваться не самые подходящие язык программирования или операционная система. Для простой демонстрации возможностей может применяться неэффективный алгоритм. Спустя некоторое время разработчик забывает о причинах, по которым эти средства не подходят. В результате далеко не идеальный выбранный вариант интегрируется в систему.

Очевидно, что преодоление этих недостатков требует борьбы с житейским соблазном – принять желаемое за действительное.

1.2.4 Быстрая разработка приложений

Модель быстрой разработки приложений (Rapid Application Development) обеспечивает экстремально короткий цикл разработки. RAD – высокоскоростная адаптация линейной последовательной модели, в которой быстрая разработка достигается за счет использования компонентно-ориентированного конструирования. Если требования полностью определены, а проектная область ограничена, RAD-процесс позволяет группе создать полностью функциональную систему за очень короткое время (60–90 дней).

RAD-подход ориентирован на разработку информационных систем и выделяет

следующие этапы:

1. Бизнес-моделирование. Моделируется информационный поток между бизнес-функциями. Ищутся ответы на следующие вопросы: Какая информация руководит бизнес-процессом? Какая информация генерируется? Кто генерирует ее? Где информация применяется? Кто обрабатывает ее?

2. Моделирование данных. Информационный поток, определенный на этапе бизнес-моделирования, отображается в наборе объектов данных, которые требуются для поддержки бизнеса. Идентифицируются характеристики (свойства, атрибуты) каждого объекта, определяются отношения между объектами;

3. Моделирование обработки. Определяются преобразования объектов данных, обеспечивающие реализацию бизнес-функций. Создаются описания обработки для добавления, модификации, удаления или нахождения (испраления) объектов данных;

4. Генерация приложения. Предполагается использование методов, ориентированных на языки программирования 4-го поколения. Вместо создания ПО с помощью языков программирования 3-го поколения RAD-процесс работает с повторно используемыми программными компонентами или создает повторно используемые компоненты. Для обеспечения конструирования применяются утилиты автоматизации;

5. Тестирование и объединение. Поскольку применяются повторно используемые компоненты, многие программные элементы уже протестированы. Это уменьшает время тестирования (хотя все новые элементы должны быть протестированы).

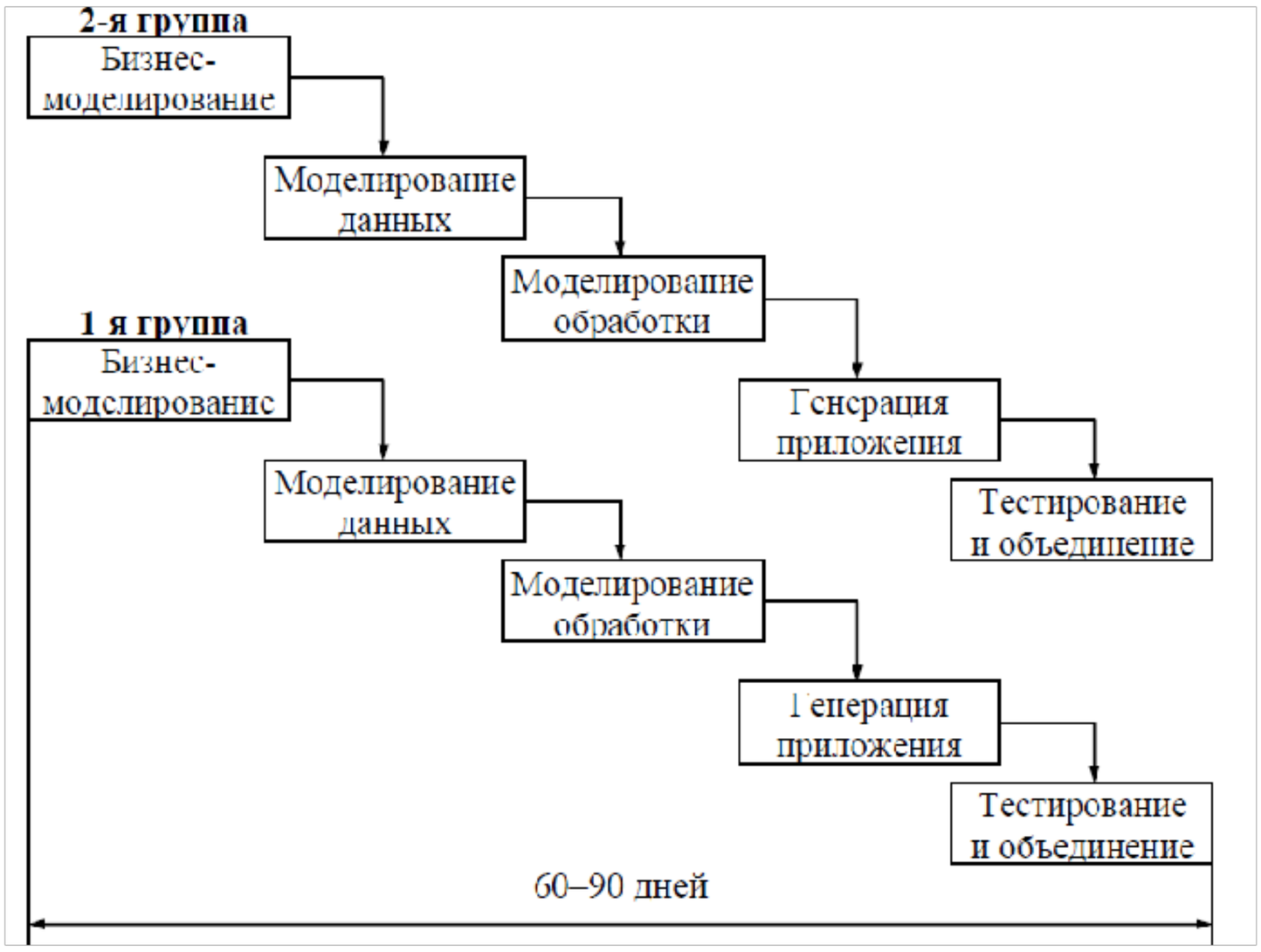

Рисунок 1.14 - Модель быстрой разработки приложений

Применение RAD возможно в том случае, когда каждая главная функция может быть завершена за 3 месяца. Каждая главная функция адресуется отдельной группе разработчиков, а затем интегрируется в целую систему.

Применение RAD имеет и свои недостатки, и ограничения:

- для больших проектов в RAD требуются существенные людские ресурсы (необходимо создать достаточное количество групп);
- RAD применима только для таких приложений, которые могут декомпозироваться на отдельные модули и в которых производительность не является критической величиной;
- RAD неприменима в условиях высоких технических рисков (т. е. при использовании новой технологии).

1.2.5 Компонентно-ориентированная модель

Компонентно-ориентированная модель является развитием спиральной. В этой модели конкретизируется содержание квадранта конструирования – оно отражает тот факт, что в современных условиях новая разработка должна основываться на повторном использовании существующих программных компонентов (рисунок 1.15).

Программные компоненты, созданные в реализованных программных проектах, хранятся в библиотеке. В новом программном проекте, исходя из требований заказчика, выявляются кандидаты в компоненты. Далее проверяется наличие этих кандидатов в библиотеке. Если они найдены, то компоненты извлекаются из библиотеки и используются повторно. В противном случае создаются новые компоненты, они применяются в проекте и включаются в библиотеку.

Достоинства компонентно-ориентированной модели:

- уменьшает на 30 % время разработки программного продукта;
- уменьшает стоимость программной разработки до 70 %;
- увеличивает в полтора раза производительность разработки.

Недостатком такой модели является сложность организации процесса разработки ПО по данной модели.

1.2.6 XP-процесс

Экстремальное программирование (eXtreme Programming, XP) – облегченный (подвижный) процесс (или методология), главный автор которого Кент Бек (1999). XP-процесс ориентирован на группы малого и среднего размера, строящие программное обеспечение в условиях неопределенных или быстро изменяющихся требований. XP-группу образуют до 10 сотрудников, которые размещаются в одном помещении.

Основная идея XP – устранить высокую стоимость изменения, характерную для приложений с использованием объектов, паттернов и реляционных баз данных. Поэтому XP-процесс должен быть высокодинамичным процессом. XP-группа имеет дело с изменениями требований на всем протяжении итерационного цикла разработки, причем цикл состоит из очень коротких итераций. Четырьмя базовыми действиями в XP-цикле являются: кодирование, тестирование, выслушивание заказчика и проектирование. Динамизм обеспечивается с помощью четырех характеристик: непрерывной связи с заказчиком (и в пределах группы), простоты (всегда выбирается минимальное решение), быстрой обратной связи (с помощью модульного и функционального тестирования), смелости в проведении профилактики возможных проблем.

Большинство принципов, поддерживаемых в XP (минимальность, простота, эволюционный цикл разработки, малая длительность итерации, участие пользователя, оптимальные стандарты кодирования и т. д.), продиктованы здравым смыслом и применяются в любом упорядоченном процессе. Просто в XP эти принципы достигают «экстремальных значений» (таблица 1.1).

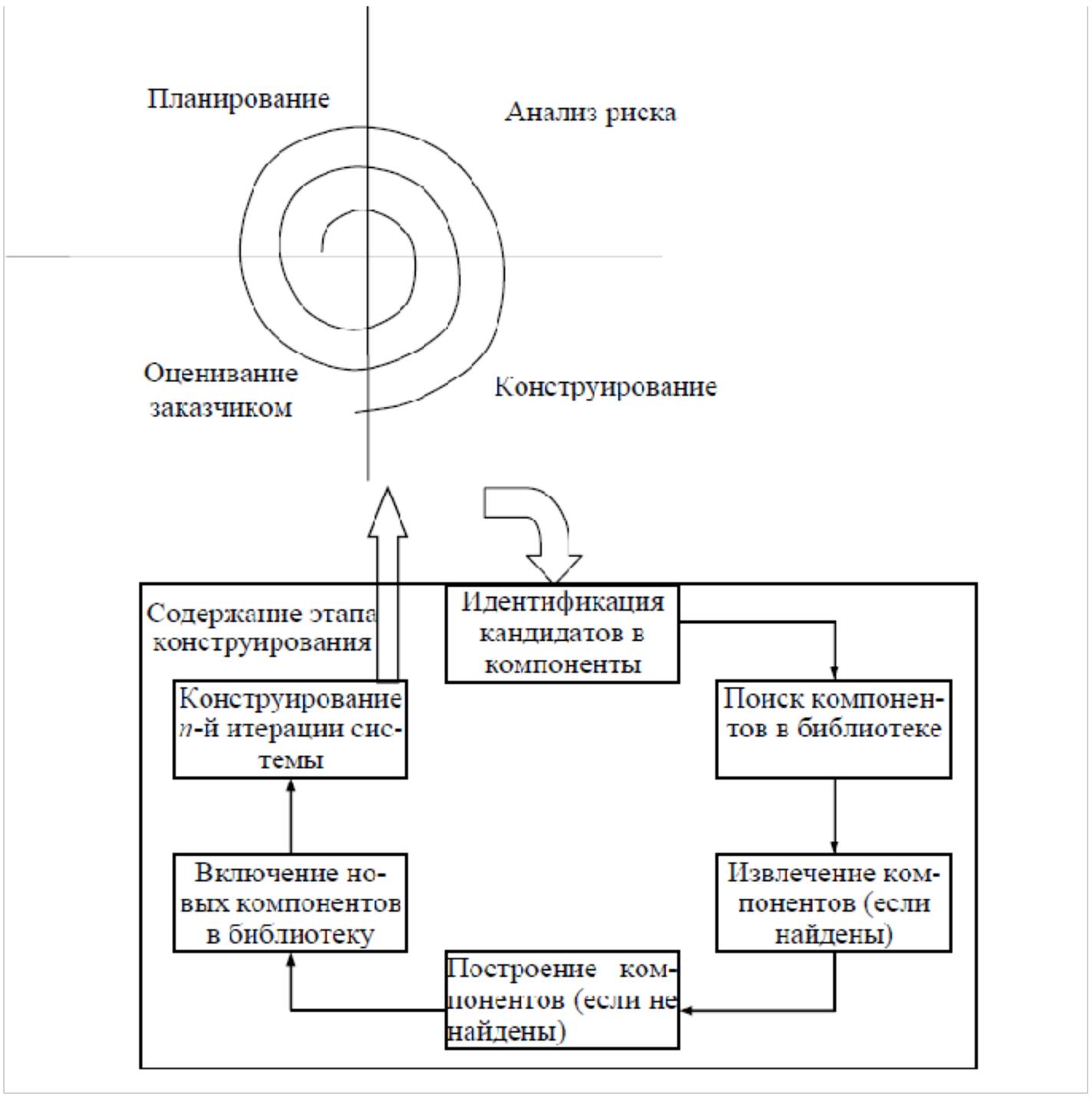

Рисунок 1.15 - Компонентно-ориентированная модель

Таблица 1.1 Экстремумы в экстремальном программировании

Практика здравого смысла	XP-экстремум	XP-реализация
Проверки кода	Код проверяется все время	Парное программирование
Тестирование	Тестирование выполняется все время, даже с помощью заказчиков	Тестирование модуля, функциональное тестирование
Проектирование	Проектирование является частью ежедневной деятельности каждого разработчика	Реорганизация (refactoring)
Простота	Для системы выбирается простейшее проектное решение, поддерживающее ее текущую функциональность	Самая простая вещь, которая могла бы работать
Архитектура	Каждый постоянно работает над уточнением архитектуры. Интегрируется и тестируется несколько раз в день	Метафора
Тестирование интеграции	Итерации являются предельно короткими, продолжаются секунды, минуты, часы, а не недели, месяцы или годы	Непрерывная интеграция
Короткие итерации		Игра планирования

Тот, кто принимает принцип минимального решения за хакерство, ошибается; в действительности XP – строго упорядоченный процесс. Простые решения, имеющие высший приоритет, в настоящее время рассматриваются как наиболее ценные части системы, в отличие от проектных решений, которые пока не нужны, а могут (в условиях изменения требований

и операционной среды) и вообще не понадобиться. Базис XP образуют перечисленные ниже двенадцать методов.

1. Игра планирования (Planning game) – быстрое определение области действия следующей реализации путем объединения деловых приоритетов и технических оценок. Заказчик формирует область действия, приоритетность и сроки с точки зрения бизнеса, а разработчики оценивают и прослеживают продвижение (прогресс).

2. Частая смена версий (Small releases) – быстрый запуск в производство простой системы. Новые версии реализуются в очень коротком (двухнедельном) цикле.

3. Метафора (Metaphor) – вся разработка проводится на основе простой, общедоступной истории о том, как работает вся система.

4. Простое проектирование (Simple design) – проектирование выполняется настолько просто, насколько это возможно в данный момент.

5. Тестирование (Testing) – непрерывное написание тестов для модулей, которые должны выполняться безупречно; заказчики пишут тесты для демонстрации законченности функций. «Тестируй, а затем кодируй» означает, что входным критерием для написания кода является «отказавший» тестовый вариант.

6. Реорганизация (Refactoring) – система реструктурируется, но ее поведение не изменяется; цель – устранить дублирование, улучшить взаимодействие, упростить систему или добавить в нее гибкость.

7. Парное программирование (Pair programming) – весь код пишется двумя программистами, работающими на одном компьютере.

8. Коллективное владение кодом (Collective ownership) – любой разработчик может улучшать любой код системы в любое время.

9. Непрерывная интеграция (Continuous integration) – система интегрируется и строится много раз в день по мере завершения каждой задачи. Непрерывное регрессионное тестирование, т. е. повторение предыдущих тестов, гарантирует, что изменения требований не приведут к регрессу функциональности.

10. 40-часовая неделя (40-hour week) – как правило, работают не более 40 часов в неделю. Нельзя удваивать рабочую неделю за счет сверхурочных работ.

11. Локальный заказчик (On-site customer) – в группе все время должен находиться представитель заказчика, действительно готовый отвечать на вопросы разработчиков.

12. Стандарты кодирования (Coding standards) – должны выдерживаться правила, обеспечивающие одинаковое представление программного кода во всех частях программной системы.

1.3 Стандарты, регламентирующие процесс разработки программного обеспечения

1.3.1 ГОСТ Р ИСО 9000–2001. Системы менеджмента качества. Основные положения и словарь

1.3.1.1 Предисловие

Стандарт разработан Всероссийским научно-исследовательским институтом сертификации (ВНИИС). Представляет собой аутентичный текст международного стандарта ИСО 9000–2000 «Системы менеджмента качества. Основные положения и словарь».

1.3.1.2 Введение

Семейство стандартов ИСО 9000, перечисленных ниже, было разработано для того, чтобы помочь организациям всех видов и размеров внедрять и обеспечивать функционирование эффективных систем менеджмента качества: ГОСТ Р ИСО 9000–2001 описывает основные положения систем менеджмента качества и устанавливает терминологию для систем менеджмента качества.

ГОСТ Р ИСО 9001–2001 определяет требования к системам менеджмента качества для тех случаев, когда организации необходимо продемонстрировать свою способность предоставлять продукцию, отвечающую требованиям потребителей и установленным к ней обязательным требованиям. Направлен на повышение удовлетворенности потребителей;

ГОСТ Р ИСО 9004–2001 содержит рекомендации, рассматривающие как результативность, так и эффективность системы менеджмента качества. Целью этого стандарта является улучшение деятельности организации и удовлетворенность потребителей и других заинтересованных сторон;

ИСО 19011* содержит методические указания по аудиту (проверке) систем менеджмента качества и охраны окружающей среды.

Вместе они образуют согласованный комплекс стандартов на системы менеджмента качества, содействующий взаимопониманию в национальной и международной торговле.

Принципы менеджмента качества. Для успешного руководства организацией и ее функционирования необходимо направлять ее и управлять систематически и прозрачным способом. Успех может быть достигнут в результате внедрения и поддержания в рабочем состоянии системы менеджмента качества, разработанной для постоянного улучшения деятельности с учетом потребностей всех заинтересованных сторон. Управление организацией включает менеджмент качества наряду с другими аспектами менеджмента.

Восемь принципов менеджмента качества были определены для того, чтобы высшее руководство могло следовать им с целью улучшения деятельности организации.

1. Ориентация на потребителя

Организации зависят от своих потребителей, и поэтому должны понимать их текущие и будущие потребности, выполнять их требования и стремиться превзойти их ожидания.

2. Лидерство руководителя

Руководители обеспечивают единство цели и направления деятельности организации. Они должны создавать и поддерживать внутреннюю среду, в которой работники могут быть полностью вовлечены в решение задач организации.

3. Вовлечение работников

Работники всех уровней составляют основу организации, и их полное вовлечение дает возможность организации с выгодой использовать их способности.

4. Процессный подход

Желаемый результат достигается эффективнее, когда деятельностью и соответствующими ресурсами управляют как процессом.

5. Системный подход к менеджменту

Выявление, понимание и менеджмент взаимосвязанных процессов как системы содействуют результативности и эффективности организации при достижении ее целей.

6. Постоянное улучшение

Постоянное улучшение деятельности организации в целом следует рассматривать как ее неизменную цель.

7. Принятие решений, основанное на фактах

Эффективные решения основываются на анализе данных и информации.

8. Взаимовыгодные отношения с поставщиками

Организация и ее поставщики взаимозависимы, и отношения взаимной выгоды повышают способность обеих сторон создавать ценности.

Эти восемь принципов менеджмента качества образуют основу для стандартов на системы менеджмента качества, входящих в семейство ИСО 9000.

1.3.1.3 Область применения

Настоящий стандарт устанавливает основные положения систем менеджмента качества, являющихся объектом стандартов семейства ИСО 9000, и определяет соответствующие термины.

Настоящий стандарт может использоваться:

- организациями, стремящимися добиться преимущества посредством внедрения системы менеджмента качества;
- организациями, стремящимися получить уверенность в том, что их заданные требования к продукции будут выполнены поставщиками;
- пользователями продукции;
- теми, кто заинтересован в едином понимании терминологии, применяемой в менеджменте качества (например, поставщики, потребители, регламентирующие органы);
- теми сторонами, внутренними или внешними по отношению к организации, которые

оценивают систему менеджмента качества или проверяют ее на соответствие требованиям ГОСТ Р ИСО 9001 (например, аудиторы, органы по сертификации/регистрации);

- теми сторонами, внутренними или внешними по отношению к организации, которые консультируют или проводят обучение по системе менеджмента качества, соответствующей данной организации;
- разработчиками соответствующих стандартов.

1.3.1.4 Основные положения систем менеджмента качества

Обоснование необходимости систем менеджмента качества. Системы менеджмента качества могут содействовать организациям в повышении удовлетворенности потребителей.

Потребителям необходима продукция, характеристики которой удовлетворяли бы их потребности и ожидания. Эти потребности и ожидания, как правило, отражаются в технических условиях на продукцию и обычно считаются требованиями потребителей. Требования могут быть установлены потребителем в контракте или определены самой организацией. В любом случае приемлемость продукции в конечном счете устанавливает потребитель. Поскольку потребности и ожидания потребителей меняются, организации также испытывают давление, обусловленное конкуренцией и техническим прогрессом, они должны постоянно совершенствовать свою продукцию и свои процессы.

Системный подход к менеджменту качества побуждает организации анализировать требования потребителей, определять процессы, способствующие получению продукции, приемлемой для потребителей, а также поддерживать эти процессы в управляемом состоянии. Система менеджмента качества может быть основой постоянного улучшения с целью увеличения вероятности повышения удовлетворенности как потребителей, так и других заинтересованных сторон. Она дает уверенность самой организации и потребителям в ее способности поставлять продукцию, полностью соответствующую требованиям.

Требования к системам менеджмента качества и требования к продукции. Семейство стандартов ИСО 9000 проводит различие между требованиями к системам менеджмента качества и требованиями к продукции.

Требования к системам менеджмента качества установлены в ГОСТ Р ИСО 9001. Они являются общими и применимыми к организациям в любых секторах промышленности или экономики независимо от категории продукции. ГОСТ Р ИСО 9001 не устанавливает требований к продукции.

Требования к продукции могут быть установлены потребителями или организацией, исходя из предполагаемых запросов потребителей или требований регламентов. Требования к продукции и в ряде случаев к связанным с ней процессам могут содержаться, например в технических условиях, стандартах на продукцию, стандартах на процессы, контрактных соглашениях и регламентах.

Подход к системам менеджмента качества. Подход к разработке и внедрению системы менеджмента качества состоит из нескольких ступеней, включающих:

- установление потребностей и ожиданий потребителей и других заинтересованных сторон;
- разработку политики и целей организации в области качества;
- установление процессов и ответственности, необходимых для достижения целей в области качества;
- установление и определение необходимых ресурсов и обеспечение ими для достижения целей в области качества;
- разработку методов для измерения результативности и эффективности каждого процесса;
- применение данных этих измерений для определения результативности и эффективности каждого процесса;
- определение средств, необходимых для предупреждения несоответствий и устранения их причин;
- разработку и применение процесса для постоянного улучшения системы менеджмента качества.

Такой подход также применяется для поддержания в рабочем состоянии и улучшения имеющейся системы менеджмента качества.

Организация, принимающая указанный выше подход, создает уверенность в

возможностях своих процессов и качестве своей продукции, а также обеспечивает основу для постоянного улучшения. Это может привести к возрастанию удовлетворенности потребителей и других заинтересованных сторон и успеху организации.

Процессный подход. Любая деятельность или комплекс деятельности, в которой используются ресурсы для преобразования входов в выходы, может рассматриваться как процесс.

Чтобы результативно функционировать, организации должны определять и управлять многочисленными взаимосвязанными и взаимодействующими процессами. Часто выход одного процесса образует непосредственно вход следующего. Систематическая идентификация и менеджмент применяемых организацией процессов и прежде всего обеспечения их взаимодействия могут считаться «процессным подходом».

Назначение настоящего стандарта – побуждать принятие процессного подхода к менеджменту организации.

На рисунке 1.16 приведена основанная на процессном подходе система менеджмента качества, описанная в семействе стандартов ИСО 9000. Он показывает, что заинтересованные стороны играют существенную роль в предоставлении организации входных данных. Наблюдение за удовлетворенностью заинтересованных сторон требует оценки информации, касающейся восприятия заинтересованными сторонами степени выполнения их потребностей и ожиданий. Модель (рисунок 1.16), не показывает процессы на детальном уровне.

Рисунок 1.16 - Модель системы менеджмента качества, основанной на процессном подходе

Политика и цели в области качества. Политика и цели в области качества устанавливаются, чтобы служить ориентиром для организации. Они определяют желаемые результаты и способствуют использованию организацией ресурсов для достижения этих результатов. Политика в области качества обеспечивает основу для разработки и анализа целей в области качества. Цели в области качества необходимо согласовывать с политикой в области качества и приверженностью к постоянному улучшению, а результаты должны быть измеримыми.

Достижение целей в области качества может оказывать позитивное воздействие на качество продукции, эффективность работы и финансовые показатели и, следовательно, на удовлетворенность и уверенность заинтересованных сторон.

Роль высшего руководства в системе менеджмента качества. С помощью лидерства и реальных действий высшее руководство может создавать обстановку, способствующую полному вовлечению работников и эффективной работе системы менеджмента качества. Принципы менеджмента качества могут использоваться высшим руководством как основа для выполнения своей роли в:

- разработке и поддержании политики и целей организации в области качества;

- популяризации политики и целей в области качества во всей организации для повышения осознания, мотивации и вовлечения персонала;

- обеспечении ориентации на требования потребителей во всей организации;

- обеспечении внедрения соответствующих процессов, позволяющих выполнять требования потребителей и других заинтересованных сторон и достигать целей в области качества;

- обеспечении разработки, внедрения и поддержания в рабочем состоянии эффективной системы менеджмента качества для достижения этих целей в области качества;

- обеспечении необходимыми ресурсами;

- проведении периодического анализа системы менеджмента качества;

- принятии решений в отношении политики и целей в области качества;

- принятии решений по мерам улучшения системы менеджмента качества.

Документация. Документация дает возможность передать смысл и последовательность действий. Ее применение способствует:

- достижению соответствия требованиям потребителя и улучшению качества;

- обеспечению соответствующей подготовки кадров;

- повторяемости и прослеживаемости;

- обеспечению объективных свидетельств;

- оцениванию эффективности и постоянной пригодности системы менеджмента качества.

Разработка документации не должна быть самоцелью, а должна добавлять ценность.

В системах менеджмента качества применяются следующие виды документов:

- документы, предоставляющие согласованную информацию о системе менеджмента качества организации, предназначенную как для внутреннего, так и внешнего пользования; к таким документам относятся руководства по качеству;

- документы, описывающие, как система менеджмента качества применяется к конкретной продукции, проекту или контракту; к таким документам относятся планы качества;

- документы, устанавливающие требования; к ним относятся документы, содержащие технические требования;

- документы, содержащие рекомендации или предложения; к ним относятся методические документы;

- документы, содержащие информацию о том, как последовательно выполнять действия и процессы; такие документы могут включать документированные процедуры, рабочие инструкции и чертежи;

- документы, содержащие объективные свидетельства выполненных действий или достигнутых результатов; к таким документам относятся записи.

Каждая организация определяет объем необходимой документации и ее носители. Это зависит от таких факторов, как вид и размер организации, сложность и взаимодействие процессов, сложность продукции, требования потребителей, соответствующие обязательные требования, продемонстрированные способности персонала, а также от глубины, до которой необходимо подтверждать выполнение требований к системе менеджмента качества.

Оценивание систем менеджмента качества. При оценке систем менеджмента качества следует задавать четыре основных вопроса в отношении каждого оцениваемого процесса:

1. Выявлен и определен ли соответствующим образом процесс?
2. Распределена ли ответственность?
3. Внедрены и поддерживаются ли в рабочем состоянии процедуры?
4. Эффективен ли процесс в достижении требуемых результатов?

Совокупные ответы на приведенные выше вопросы могут определить результаты оценивания. Оценка системы менеджмента качества может различаться по области применения и включать такие виды деятельности, как аудит (проверку) и анализ системы менеджмента качества, а также самооценку.

Аудиты (проверки) применяют для определения степени выполнения требований к системе менеджмента качества. Наблюдения аудитов (проверок) используют для оценки эффективности системы менеджмента качества и определения возможностей для улучшения.

Аудиты (проверки), проводимые первой стороной (самой организацией) или от ее имени для внутренних целей, могут служить основой для декларирования организацией о своем соответствии.

Аудиты (проверки), проводимые второй стороной, могут проводиться потребителями организации или другими лицами от имени потребителей.

Аудиты (проверки), проводимые третьей стороной, осуществляются внешними независимыми организациями. Такие организации, обычно имеющие аккредитацию, проводят сертификацию на соответствие требованиям, например требованиям ГОСТ Р ИСО 9001.

ИСО 19011 содержит методические указания по аудиту (проверке).

Одна из задач высшего руководства – проведение регулярного систематического оценивания пригодности, адекватности, эффективности и результативности системы менеджмента качества с учетом политики и целей в области качества. Этот анализ может включать рассмотрение необходимости адаптации политики и целей в области качества в ответ на изменение потребностей и ожиданий заинтересованных сторон. Анализ включает определение потребности в действиях.

При анализе системы менеджмента качества наряду с другими источниками информации используют отчеты по аудитам (проверкам).

Самооценка организации является всесторонним и систематическим анализом деятельности организации и результатов по отношению к системе менеджмента качества или модели совершенства (модели премии по качеству).

Самооценка может дать общее представление о деятельности организации и степени развития системы менеджмента качества. Она может также помочь определить организации области, нуждающиеся в улучшении, и приоритеты.

Постоянное улучшение. Целью постоянного улучшения системы менеджмента качества является увеличение возможности повышения удовлетворенности потребителей и других заинтересованных сторон. Действия по улучшению включают:

- анализ и оценку существующего положения для определений областей для улучшения;
- установление целей улучшения;
- поиск возможных решений для достижения целей;
- оценивание и выбор решений;
- выполнение выбранных решений;
- измерение, проверку, анализ и оценку результатов выполнения для установления того, что достигнуты ли цели;
- оформление изменений.

Результаты анализируют с целью установления дальнейших возможностей для улучшения. Таким образом, улучшение является постоянным действием. Аудиты (проверки) и анализ системы менеджмента качества могут также использоваться для определения возможностей улучшения.

Роль статистических методов. Использование статистических методов может помочь в понимании изменчивости и, следовательно, может помочь организациям в решении проблем и повышении результативности и эффективности. Эти методы также способствуют лучшему применению имеющихся в наличии данных для оказания помощи в принятии решений.

Изменчивость можно наблюдать в ходе и результатах многих видов деятельности, даже в условиях очевидной стабильности. Такую изменчивость можно проследить в измеряемых характеристиках продукции и процессов. Ее наличие можно заметить на различных стадиях жизненного цикла продукции от исследования рынка до обслуживания потребителей и утилизации.

Статистические методы могут помочь при измерении, описании, анализе, интерпретации и моделировании такой изменчивости даже при относительно ограниченном количестве данных. Статистический анализ таких данных может помочь лучше понять природу, масштаб и причины изменчивости, способствуя таким образом решению, и даже предупреждению проблем, которые могут быть результатом такой изменчивости, а также постоянному улучшению.

Методические указания по применению статистических методов в системе менеджмента качества приведены в ИСО/ТО 10017.

Направленность систем менеджмента качества и других систем менеджмента. Система менеджмента качества является частью системы менеджмента организации, которая направлена на достижение результатов в соответствии с целями в области качества, чтобы

удовлетворять потребности, ожидания и требования заинтересованных сторон. Цели в области качества дополняют другие цели организации, связанные с развитием, финансированием, рентабельностью, окружающей средой, охраной труда и безопасностью. Различные части системы менеджмента организации могут быть интегрированы вместе с системой менеджмента качества в единую систему менеджмента, использующую общие элементы. Это может облегчить планирование, выделение ресурсов, определение дополнительных целей и оценку общей эффективности организации. Система менеджмента организации может быть оценена на соответствие собственным требованиям организации. Она может быть также проверена на соответствие требованиям ГОСТ Р ИСО 9001 и ГОСТ Р ИСО 14001. Эти аудиты (проверки) могут проводиться отдельно или совместно.

Взаимосвязь между системами менеджмента качества и моделями совершенства. Подходы систем менеджмента качества, приведенные в семействе стандартов ИСО 9000, и модели совершенства основаны на общих принципах. Оба эти подхода:

- дают возможность организации выявить свои сильные и слабые стороны;
- содержат положения по оцениванию в сравнении с общими моделями;
- обеспечивают основу для постоянного улучшения;
- включают способы внешнего признания.

Различие между подходами систем менеджмента качества семейства ИСО 9000 и моделями совершенства заключается в их областях применения.

Стандарты семейства ИСО 9000 содержат требования к системам менеджмента качества и рекомендации по улучшению деятельности; оценивание систем менеджмента качества устанавливает выполнение этих требований. Модели совершенства содержат критерии, позволяющие проводить сравнительную оценку деятельности организации, и это применимо ко всем видам деятельности и ко всем заинтересованным сторонам. Критерии оценки в моделях совершенства обеспечивают организации основу для сравнения ее деятельности с деятельностью других организаций.

1.3.2 ГОСТ Р ИСО/МЭК ТО 15504

Основы оценки и аттестации зрелости процессов создания и сопровождения программных средств и информационных систем установлены стандартами ИСО/МЭК ТО 15504, разработанными на базе концепций CMM (Capability Maturity Model for Software).

1.3.2.1 Обзор

ИСО/МЭК ТО 15504 предоставляет основу для аттестации процесса жизненного цикла программных средств. Эта основа может быть использована организациями, занимающимися планированием, управлением, наблюдением, контролем и совершенствованием приобретения, поставки, разработки, эксплуатации, развития и поддержки программных средств.

ИСО/МЭК ТО 15504 предоставляет структурный подход к аттестации процесса жизненного цикла программных средств, проводящейся:

- организацией или от ее имени с целью выяснения состояния ее собственных процессов для их усовершенствования;
- организацией или от ее имени с целью определения пригодности ее собственных процессов для выполнения определенного требования или класса требований;
- организацией или от ее имени с целью определения пригодности процессов другой организации для определенного договора или класса договоров.

Такая основа для аттестации процесса способствует самоаттестации; направлена на обеспечение адекватности управления аттестуемыми процессами; принимает во внимание контекст, в котором выполняются аттестуемые процессы; выдает набор рейтингов процесса (профиль процесса), а не результат типа зачет/незачет; подходит ко всем сферам приложений и для организаций любого размера.

Чтобы организация могла улучшить качество своей продукции, она должна иметь проверенный, последовательный и надежный метод для аттестации состояния своих процессов, а также она должна иметь средства использования результатов как часть связной программы усовершенствования.

Использование аттестации процессов внутри организации должно способствовать:

- выработке культуры постоянного совершенствования, а также соответствующих механизмов поддержания этой культуры;
- разработке процессов, отвечающих бизнес-целям;
- оптимизации использования ресурсов.

Это приведет к появлению зрелых организаций, максимально восприимчивых к требованиям потребителя и рынка, имеющих минимальную стоимость полного жизненного цикла своей продукции и, как результат, максимально удовлетворяющих конечного пользователя.

Покупателям также будет выгодно использовать аттестацию процесса. Ее применение при определении зрелости

- уменьшит неопределенность при выборе поставщиков программнонасыщенных систем за счет того, что риски, связанные со зрелостью подрядчика, выявляются еще до заключения договора;
- позволит заранее предусмотреть необходимые меры на случай возникновения рискового события;
- предоставит количественные критерии выбора при сопоставлении потребностей бизнеса, требований и оценочной стоимости проекта со зрелостью конкурирующих поставщиков.

Главные преимущества стандартизированного подхода к аттестации процесса в том, что он:

- станет открытым, общедоступным подходом к аттестации процесса;
- приведет к общему пониманию использования аттестации для усовершенствования процесса и оценки зрелости;
- при приобретении облегчит оценку зрелости поставщика;
- будет контролироваться и регулярно пересматриваться в свете опыта использования;
- будет меняться только международным соглашением;
- будет способствовать согласованию существующих схем.

Подход к аттестации процесса, определенный в ИСО/МЭК ТО 15504, разработан с целью предоставить основу для общего подхода к описанию результатов аттестации процесса, позволяющего в какой-то степени сравнивать аттестации, основанные на разных, но совместимых моделях и методах. Изощренность и сложность, требующиеся от процесса, зависят от его контекста. Например, планирование для группы из пяти человек необходимо гораздо в меньшем объеме, чем для группы из пятидесяти человек. Этот контекст влияет на то, как ведущий аттестатор судит о действии, а также на степень сравнимости профилей различных процессов.

1.3.2.2 Область применения

Аттестация процесса применяется в основном в двух контекстах (рисунок 1.17).

Рисунок 1.17 - Оценка и аттестация процессов жизненного цикла программных средств и информационных систем

В контексте усовершенствования процесса аттестация процесса предоставляет средства охарактеризовать текущую деятельность организационной единицы в терминах зрелости некоторых выбранных процессов. Анализ результатов в свете бизнеспотребностей организации выявляет сильные и слабые стороны процессов, а также присущие им риски. Это, в свою очередь, позволяет определить, эффективны ли эти процессы в достижении своих целей, а также выявить существенные причины низкого качества или превышения бюджета или сроков. Все вместе это позволяет расставить приоритеты при усовершенствовании процессов.

Определение зрелости процесса связано с анализом соответствия заявленной зрелости выбранных процессов целевому профилю зрелости процессов для того, чтобы выявить риски, связанные с выполнением проекта, использующего выбранные процессы. Заявленная зрелость может быть основана на результатах прежних аналогичных аттестаций процесса или на аттестации, проведенной с целью выработки заявленной зрелости.

Две части ИСО/МЭК ТО 15504 (части 7 и 8) посвящены использованию аттестации для усовершенствования процесса и для определения зрелости процесса. Другие части ИСО/МЭК ТО 15504 посвящены различным вопросам, относящимся к аттестации процесса.

ИСО/МЭК ТО 15504 разработан так, чтобы в рамках единого источника удовлетворить потребности потребителей, поставщиков и аттестаторов, а также их индивидуальные требования. Преимущества, вытекающие из использования данного комплекта документов, включают:

Для приобретателей:

- способность определять текущую и потенциальную зрелости процессов жизненного цикла у поставщика;

Для поставщиков:

- способность определять текущую и потенциальную зрелости своих собственных процессов жизненного цикла;
- способность определять области и приоритеты для усовершенствования процессов;
- основу, задающую схему усовершенствования процессов;

Для аттестаторов:

- основу для проведения аттестаций.

1.3.2.3 Состав ИСО/МЭК ТО 15504

ИСО/МЭК ТО 15504 состоит из девяти частей. В данном разделе описана каждая часть и

ее роль в ИСО/МЭК ТО 15504.

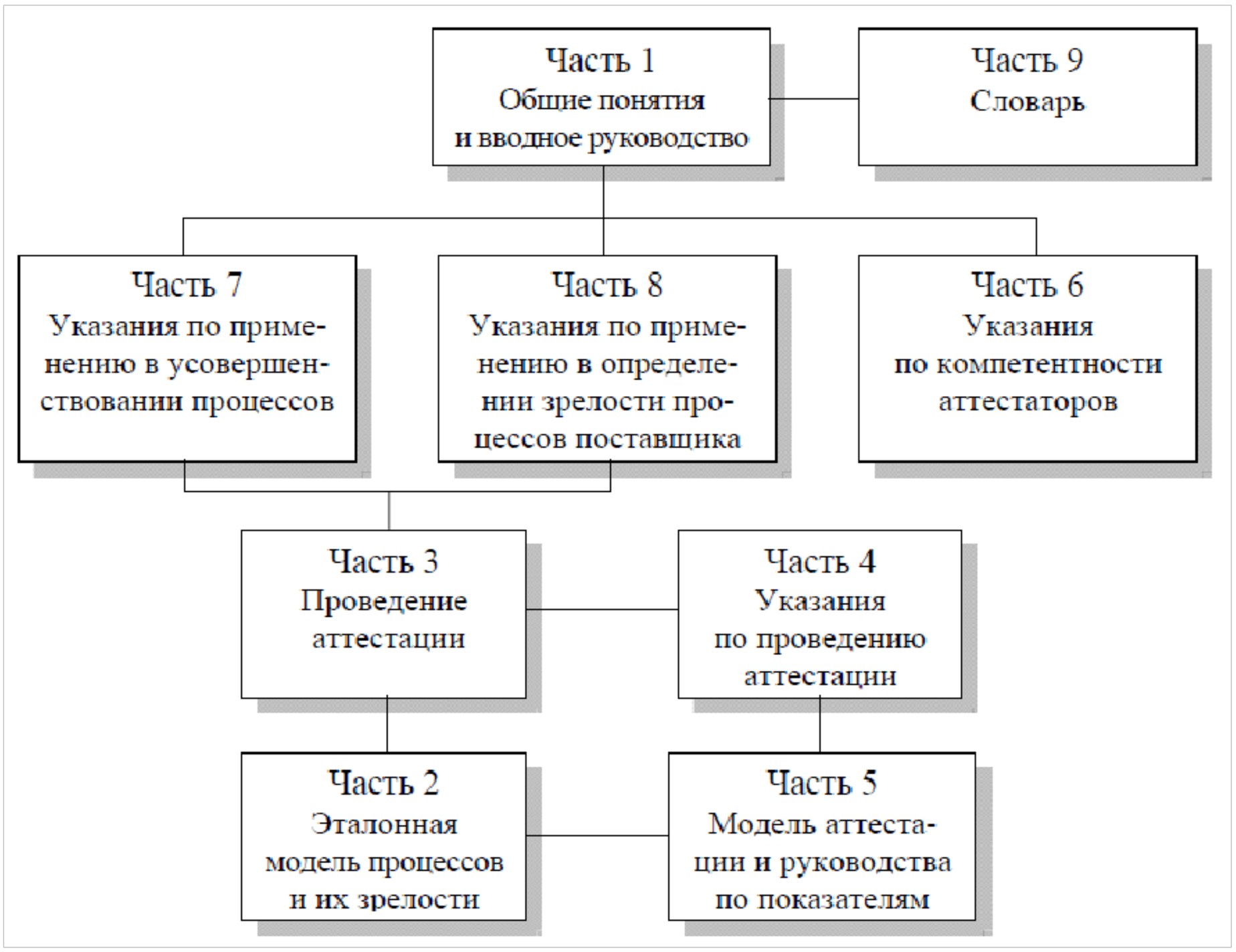

Рисунок 1.18 - Состав ИСО/МЭК ТО 15504

Рисунок 1.18 показывает потенциальную схему использования ИСО/МЭК ТО 15504. Часть 1 (этот документ) служит общей отправной точкой ИСО/МЭК ТО 15504. Читатели, интересующиеся только усовершенствованием процесса или определением зрелости поставщика, должны прочесть части 7 и 8 соответственно, содержащие детальное руководство по этим контекстам использования. Эти части позволят пользователю определить наиболее подходящий для себя способ использования нормативных компонентов ИСО/МЭК ТО 15504 (части 2 и 3). Часть 4 является руководством по применению части 2 и части 3, тогда как часть 5 – это пример аттестационной модели, совместимой с эталонной моделью (часть 2). Пользователи, интересующиеся в основном ролью аттестаторов, должны обратить внимание на часть 6, содержащую руководство по навыкам и компетентности аттестаторов.

В таблице 1.2 перечислены основные классы читателей ИСО/МЭК ТО 15504 и указаны, какие части данного набора документов посвящены первостепенным областям их интересов.

Таблица 1.2 Аудитория ИСО/МЭК ТО 15504

Класс читателей	Интересы	Части, предлагаемые к прочтению

Заказчик (спонсор) аттестации	Как проводится аттестация, какие требуются инструменты и иная поддержка, как инициировать аттестацию	1, 2, 3, 4, 6
Заказчик (спонсор) усовершенствования процесса	Инициирование программы усовершенствования, определение входов для аттестации для целей усовершенствования, использование результатов аттестации для усовершенствования	7
Заказчик (спонсор) определения зрелости процесса	Инициирование программы определения зрелости поставщика, определение целевого профиля зрелости, проверка и использование результатов аттестации при определении зрелости	
Аттестаторы	Проведение согласованной аттестации, развитие навыков и компетентности, необходимых для проведения аттестации	8
Разработчики аттестационных моделей	Разработка модели для проведения аттестации, совместимой с эталонной моделью Разработка метода, который поддерживал бы проведение аттестации, соответствующей требованиям	2, 3, 4, 5, 6
Разработчики методов аттестации	Разработка инструментальных средств, которые помогали бы аттестаторам в сборе, записи, классификации данных в ходе аттестации	2, 3, 4, 5
Разработчики инструментальных средств		2, 3, 4 2, 3, 4, 5

Часть 1 (информационная) является отправной точкой ИСО/МЭК ТО 15504. Она описывает взаимодействие частей набора документов и содержит руководство по их выбору и использованию. Она разъясняет требования, содержащиеся в ИСО/МЭК ТО 15504 и их применимость к проведению аттестаций.

Часть 2 (нормативная) ИСО/МЭК ТО 15504 определяет двумерную эталонную модель для описания процессов и их зрелости, использующуюся при аттестации процессов. Эталонная модель определяет ряд процессов, установленных в терминах их назначения и итогов, а также основу для оценивания зрелости процессов посредством аттестации атрибутов процессов, структурированных по уровням зрелости. Также обозначены требования для установления совместимости различных аттестационных моделей с эталонной моделью.

Часть 3 (нормативная) ИСО/МЭК ТО 15504 определяет требования для проведения аттестации таким образом, чтобы результаты были повторимыми, надежными и согласующимися.

Часть 4 (информационная) ИСО/МЭК ТО 15504 содержит руководство по проведению аттестаций процессов жизненного цикла программных средств, по интерпретации требований ИСО/МЭК ТО 15504-2 и ИСО/МЭК ТО 15504-3 для различных контекстов аттестации. Это руководство охватывает выбор и использование документированного процесса для аттестации, совместимой аттестационной модели (или моделей), а также вспомогательного инструмента или средства аттестации. Это руководство является достаточно общим и применимо для всех организаций для проведения аттестаций с использованием

разнообразных методов и технических приемов для того, чтобы поддерживаться целым рядом средств.

Часть 5 (информационная) ИСО/МЭК ТО 15504 содержит пример модели для проведения аттестации процесса, основанную на эталонной модели из ИСО/МЭК ТО 15504-2 и непосредственно с ней совместимую. Аттестационная модель (или модели) расширяет эталонную модель включением в нее всеобъемлющего набора показателей производительности и зрелости процессов.

Часть 6 (информационная) ИСО/МЭК ТО 15504 описывает компетентность, образование, специальную подготовку и опыт, необходимые аттестаторам для проведения аттестации процессов. В ней приведены механизмы, которые могут быть использованы для демонстрации компетентности и подтверждения образования, специальной подготовки и опыта.

Часть 7 (информационная) ИСО/МЭК ТО 15504 описывает, как определять входы и использовать результаты аттестации, имеющей целью усовершенствование процесса. Это руководство включает примеры применения усовершенствования процесса в различных ситуациях.

Часть 8 (информационная) ИСО/МЭК ТО 15504 описывает, как определять входы и использовать результаты аттестации, имеющей целью определение зрелости процессов. Она охватывает определение зрелости процессов как в простейших ситуациях, так и в более сложных, включающих, например, будущую зрелость. Это руководство по проведению определения зрелости процессов может использоваться либо организацией для определения собственной зрелости, либо потребителем для определения зрелости (потенциального) поставщика.

Часть 9 (нормативная) ИСО/МЭК ТО 15504 является консолидированным словарем терминов, специфически определенных для целей ИСО/МЭК ТО 15504.

1.3.2.4 Связь с другими международными стандартами

ИСО/МЭК ТО 15504 дополняет некоторые международные стандарты и другие модели для оценки зрелости и эффективности организаций и процессов. Данный раздел описывает связь между ИСО/МЭК ТО 15504 и большинством связанных с ним международных стандартов.

ИСО/МЭК ТО 15504 преследует ту же цель, что и серия стандартов ИСО 9000 – обеспечить уверенность в системе управления качеством поставщика, одновременно предоставляя потребителям основу для оценки того, обладают ли потенциальные поставщики производственными возможностями, отвечающими их потребностям. Аттестация процессов дает пользователям возможность оценивать зрелость процессов по непрерывной шкале таким образом, что эти оценки сравнимы и повторимы в отличие от аудитов качества, основанных на ИСО 9001:1994, дающих оценку типа зачет/незачет. Кроме того, основа, описываемая ИСО/МЭК ТО 15504, предоставляет возможность подобрать объем аттестации так, чтобы он охватывал лишь определенные процессы, вызывающие интерес, а не все процессы, используемые организационной единицей.

ИСО/МЭК ТО 15504 в части «Процессного подхода» и «Системного подхода к административному управлению» отвечает концепции ИСО 2000 года в области системы качества. Требования к системам административного управления (менеджмента) качеством установлены в стандартах:

ISO 9000:2000 Quality management systems – Fundamentals and vocabu-lary (Системы административного управления качеством. Основы и словарь);

ISO 9000:2000 Quality management systems – Requirements (Системы административного управления качеством. Требования);

ISO 9000:2000 Quality management systems – Guidelines for performance improvement (Системы административного управления качеством. Руково- дящие указания по усовершенствованию);

ISO 19011:2000 Quality management systems – Guidelines for auditing quality and environmental management systems.

Эталонная модель описания, оценки и аттестации зрелости процессов деятельности, используемой при выполнении этапов жизненного цикла продукции, проекта или системы (ИСО/МЭК ТО 15504-2), согласована с ИСО/МЭК 12207:1995 «Информационная технология. Процессы жизненного цикла программных средств». Однако эта эталонная модель может быть

использована для разработки модели оценки уровня зрелости процессов других видов деятельности. Изложенные в ИСО/МЭК ТО 15504 рабочие продукты процессов и их характеристики могут быть использованы для оценки и усовершенствования процессов любого вида деятельности. ИСО/МЭК ТО 15504 предоставляет механизм включения в этот перечень дополнительных рабочих продуктов и процессов, необходимых для осуществления конкретного вида деятельности и, следовательно, для оценки уровня зрелости этих дополнительных процессов и вида деятельности в целом.

Концептуальные основы системы административного управления качеством определены в восьми принципах. ИСО определяет принцип административного управления качеством как «всеобъемлющее и фундаментальное правило или убеждение, применяемое при руководстве и управлении организацией, направленное на непрерывное и долгосрочное улучшение ее производительности путем ориентации на потребителей одновременно с удовлетворением потребностей остальных участвующих сторон». Эти принципы административного управления качеством войдут в документ ISO 9000 2000 года. На этих принципах будет базироваться семейство стандартов ISO 9000.

Выполнение этих принципов будет способствовать повышению управленческой культуры, проникновению системы административного управления качеством во все виды деятельности организации (т.е. выполнения концепции всеобщего административного управления качеством, TQM – Total Quality Management) и, как следствие, обеспечению конкурентоспособности создаваемой организацией продукции, проектов, систем и услуг.

Принцип 1. Ориентация организации на потребителя (Customer-Focused Organization)

«Организации зависят от своих потребителей и, таким образом, должны понимать текущие и будущие потребности потребителей, удовлетворять их требования и стремиться превзойти их ожидания». Применение принципа ориентации организации на потребителя приводит к следующим действиям:

- понимание всего спектра потребностей и ожиданий потребителей относительно продуктов, поставок, цен, надежности и т. д.;
- обеспечение взвешенного подхода к потребностям и ожиданиям потребителей и других участвующих сторон (владельцев, персонала, поставщиков, местных сообществ и общества в целом);
- распространение информации об этих потребностях и ожиданиях во всей организации;
- измерение степени удовлетворенности потребителей и влияние на результат;
- управление взаимоотношениями с потребителями.

Полезные применения данного принципа включают:

- обеспечение того, что потребности потребителей и других участвующих сторон осознаются всей организацией, для формулировки политики и стратегии (policy and strategy formulation);
- обеспечение непосредственной связи соответствующих целей и плановых показателей с потребностями и ожиданиями потребителей, для установления целей и плановых показателей (goal and target setting);
- повышение производительности организации для удовлетворения потребностей потребителей, для управления операциями (operational management);
- обеспечение того, что персонал обладает знаниями и опытом, требующимися для удовлетворения потребителей организации, для управления людскими ресурсами (human resource management).

Принцип 2. Лидерство (Leadership)

«Лидеры организаций обеспечивают единство назначения и направления организации. Они должны создать и поддерживать внутреннюю окружающую среду, в которой люди могут в полной мере участвовать в достижении стратегических целей организации».

Применение принципа лидерства приводит к следующим действиям:

- действенность и личный пример;
- понимание изменений во внешней окружающей среде и реагирование на них;
- внимание к потребностям всех участвующих сторон, включая потребителей, владельцев, персонал, поставщиков, местные сообщества и общество в целом;
- выработка ясного видения будущего организации;
- выработка общих ценностей и этики на всех уровнях организации;

- установление доверия и искоренение страха;
- обеспечение персонала требуемыми ресурсами и свободой, необходимыми для того, чтобы действовать ответственно и обоснованно;
- воодушевление, поощрение и признание вклада персонала;
- содействие открытому и честному общению;
- обучение, подготовка и инструктирование персонала;
- выработка достойных целей и плановых показателей;
- реализация стратегии по достижению этих целей и плановых показателей.

Полезные применения данного принципа включают:

- выработку и распространение ясного видения будущего организации, для формулировки политики и стратегии;
- преобразование видения организации в измеримые цели и плановые показатели, для установления целей и плановых показателей;
- стратегические цели организации достигаются полномочным и вовлеченным персоналом, для управления операциями;
- наличие полномочной, мотивированной, хорошо информированной и стабильной рабочей силы, для управления людскими ресурсами.

Принцип 3. Вовлечение персонала (Involvement of People)

«Люди составляют сущность организации на всех уровнях, и их полная вовлеченность способствует применению их способностей на благо организации».

Применение принципа вовлечения персонала приводит к следующим действиям персонала:

- принятие на себя задач и ответственность за их решение;
- активный поиск возможностей усовершенствования;
- активный поиск возможностей повышения собственных квалификации, знаний и опыта;
- свободный обмен знаниями и опытом в группах и коллективах;
- концентрация на создании ценностей для потребителя;
- новаторство и творчество в дальнейшем продвижении стратегических целей организации;
- олицетворение организации перед лицом потребителей, местных сообществ и общества в целом;
- получение удовольствия от своей работы;
- гордость и удовлетворение быть частью организации.

Полезные применения данного принципа включают:

- персонал, эффективно участвующий в усовершенствовании политики и стратегии организации, для формулировки политики и стратегии;
- персонал, принимающий на себя задачи и разделяющий ответственность за их решение, для установления целей и плановых показателей;
- персонал, вовлеченный в соответствующие усовершенствования решений и процессов, для управления операциями;
- персонал, в большей степени удовлетворенный своей работой и активно вовлеченный в собственный рост и развитие на благо организации, для управления людскими ресурсами.

Принцип 4. Процессный подход (Process Approach)

«Желаемый результат достигается более эффективно, когда связанные ресурсы и деятельность управляются как процесс».

Применение принципа процессного подхода приводит к следующим действиям персонала:

- определение процесса достижения желаемого результата;
- выявление и измерение входов и выходов процесса;
- выявление интерфейсов процесса с функциями организации;
- оценка возможного риска, его последствий и влияния процесса на потребителей, поставщиков и другие участвующие в процессе стороны;
- четкое распределение ответственности, полномочий и подотчетности при управлении процессом;
- выявление внутренних и внешних потребителей, поставщиков и других участвующих в процессе сторон;
- при проектировании процессов уделяется внимание шагам процессов, видам

деятельности, потокам, контрольным величинам, потребностям в подготовке персонала, оборудования, методах, информации, материалах и других ресурсах, необходимых для достижения желаемого результата.

Полезные применения данного принципа включают:

- использование определенных процессов во всей организации приведет к более предсказуемым результатам, лучшему использованию ресурсов, сокращению времени цикла и снижению затрат, для формулировки политики и стратегии;
- понимание зрелости процессов способствует выработке достойных целей и плановых показателей, для установления целей и плановых показателей;
- принятие процессного подхода ко всем операциям приводит к снижению затрат, предотвращению ошибок, контролю за отклонениями, сокращению времени цикла и более предсказуемым выходам, для управления операциями;
- установление эффективных по затратам процессов для управления людскими ресурсами, таких как найм, образование и подготовка, способствует приведению этих процессов в соответствие потребностям организации и создает более зрелую рабочую силу, для управления людскими ресурсами.

Принцип 5. Системный подход к административному управлению (System Approach to Management)

«Выявление, понимание и административное управление системой взаимосвязанных процессов для заданной стратегической цели повышает эффективность и результативность организации».

Применение принципа системного подхода к административному управлению приводит к следующим действиям:

- определение системы путем выявления или разработки процессов, влияющих на достижение заданной стратегической цели;
- структурирование системы так, чтобы достичь заданную стратегическую цель наиболее эффективным способом;
- понимание взаимозависимостей между процессами системы;
- непрерывное усовершенствование системы посредством измерения и оценки;
- предварительное установление ограничений по ресурсам.

Полезные применения данного принципа включают:

- создание всеобъемлющих и достойных планов, связывающих входы функций и процессов, для формулировки политики и стратегии;
- цели и плановые показатели конкретных процессов приведены в соответствие с ключевыми стратегическими целями организации, для установления целей и плановых показателей;
- более широкий взгляд на эффективность процессов, что приводит к пониманию причин проблем и своевременным действиям по усовершенствованию, для управления операциями;
- лучшее понимание распределения ролей и ответственности при достижении общих стратегических целей, уменьшая таким образом межфункциональные барьеры и улучшая коллективную работу, для управления людскими ресурсами.

Принцип 6. Непрерывное усовершенствование (Continual Improvement)

«Непрерывное усовершенствование должно быть постоянной стратегической целью организации».

Применение принципа непрерывного усовершенствования приводит к следующим действиям:

- превращение непрерывного усовершенствования продуктов, процессов и систем в стратегическую цель каждого сотрудника организации;
- применение базовых понятий последовательного (инкрементного) и скачкообразного усовершенствования;
- проведение периодических аттестаций степени достижения установленных критериев высшего качества для выявления областей потенциального усовершенствования;
- непрерывное повышение эффективности и результативности всех процессов;
- поощрение действий, основанных на предотвращении проблем;
- предоставление каждому члену организации необходимых образования и подготовки по методам и инструментальным средствам непрерывного усовершенствования, таким, как: цикл «планирование – исполнение – проверка – корректирующие действия» (Plan – Do – Check – Act);

- решение проблем;
- реинжиниринг процессов;
- нововведение в процессах;
- установление мер и целей для направления и отслеживания усовершенствований;
- признание усовершенствований.

Полезные применения данного принципа включают:

- создание и осуществление более конкурентоспособных бизнес-планов путем объединения непрерывного усовершенствования со стратегическим и бизнес-планированием, для формулировки политики и стратегии;

- установление реалистичных и достойных целей усовершенствования и предоставление ресурсов для их достижения, для установления целей и плановых показателей;

- вовлечение персонала организации в непрерывное усовершенствование процессов, для управления операциями;

- обеспечение всего персонала организации инструментальными средствами, возможностями и мотивацией для совершенствования продуктов, процессов и систем, для управления людскими ресурсами.

Принцип 7. Основанный на фактах подход к принятию решений (Factual Approach to Decision Making)

«Эффективные решения базируются на анализе данных и информации».

Применение данного принципа приводит к следующим действиям:

- осуществление измерений и сбор данных и информации, относящихся к стратегической цели;

- обеспечение существенной точности, надежности и доступности данных и информации;

- анализ данных и информации с применением обоснованных методов;
- понимание значения соответствующих статистических методик;

- принятие решений и осуществление действий на базе логического анализа, уравновешенного опытом и интуицией.

Полезные применения данного принципа включают:

- стратегии, основанные на соответствующих данных и информации более реалистичны и достижимы с большей вероятностью, для формулировки политики и стратегии;

- применение соответствующих сравнительных данных и информации для установления реалистичных и достойных целей и плановых показателей, для установления целей и плановых показателей;

- данные и информация являются базисом для понимания производительности процессов и систем для направления усовершенствований и предотвращения будущих проблем, для управления операциями;

- анализ данных и информации из таких источников, как опросы персонала, предложения сотрудников, целевых групп для направления формулировки политики управления людскими ресурсами, для управления людскими ресурсами.

Принцип 8. Взаимовыгодные отношения с поставщиками (Mutually Beneficial Supplier Relationship)

«Организация и ее поставщики взаимозависимы, и взаимовыгодные отношения повышают способность обоих производить ценности».

Применение принципа взаимовыгодных отношений с поставщиками приводит к следующим действиям:

- выявление и выбор ключевых поставщиков;

-установление с поставщиками отношений, которые бы уравновешивали краткосрочные выгоды и долгосрочные соображения для организации и общества в целом;

- создание ясного и открытого общения;
- инициация совместных разработок и усовершенствования продуктов и процессов;
- совместное установление ясного понимания потребностей потребителей;
- обмен информацией и будущими планами;
- признание усовершенствований и достижений поставщиков.

Полезные применения данного принципа включают:

- здание конкурентоспособных преимуществ путем органзации стратегических союзов или партнерских отношений с поставщиками, для формулировки политики и стратегии;

- установление более достойных целей и плановых показателей с помощью вовлечения и участия поставщиков на ранних этапах, для установления целей и плановых показателей;

- создание отношений с поставщиками и управление этими отношениями для обеспечения надежных, своевременных и свободных от дефектов поставок, для управления операциями;

- выработку и повышение зрелости поставщиков посредством проведения подготовки поставщиков и совместных усилий по усовершенствованию, для управления людскими ресурсами.

Для комплексного обеспечения нормативной базы практической реализации концепции ИСО/МЭК 15504 с учетом принципов системы административного управления качеством необходимо предусмотреть также выполнение требований нижеприведенных международных стандартов:

ISO/TR 10006:1997 Quality management – Guidelines to quality in project management (Административное управление качеством. Руководящие указания по качеству при административном управлении проектами);

ISO 10007:1997 Quality management – Guidelines for configuration management (Административное управление качеством. Руководящие указания административного управления конфигурацией);

ISO/IEC TR 16326:1999 Software engineering – Guide for the application of ISO/IEC 12207 to project management (Программная инженерия. Руководство по приложению ИСО/МЭК 12207 к административному управлению проектами);

ISO/IEC TR 15271:1998 Information technology – Guide for ISO/IEC

12207 (Software Life Cycle Processes) (Информационная технология. Руководство по применению ИСО/МЭК 12207);

ISO/IEC TR 15846:1998 Information technology – Software life cycle processes – Configuration Management (Информационная технология. Процессы жизненного цикла программных средств. Управление конфигурацией); ISO/IEC 12119:1994 Information technology – Software packages – Quality requirements and testing (Информационная технология. Пакеты программных средств. Требование к качеству и тестирование);

ISO/IEC TR 14759:1999 Software engineering – Mock up and prototype – A categorization of software mock up and prototype models and their use (Программная инженерия. Макетирование и прототипирование);

ISO/IEC 9126:1991 Information technology – Software product evaluation – Quality characteristics and guidelines for their use (Информационная технология. Оценка программного продукта. Характеристики качества и руководящие указания по их применению);

ISO/IEC 14598:1999 Information technology – Software product evaluation

– Part 1–5 (Информационная технология. Оценка программной продукции); ISO/IEC 15026:1998 Information technology – System and software integrity levels (Информационная технология. Уровни целостности программных средств и систем);

ISO/IEC 14764:1999 Information technology – Software maintenance (Информационная технология. Сопровождение программных средств);

ISO/IEC TR 14471:1999 Information technology – Software engineering – Guidelines for the adoption of CASE tools (Информационная технология. Программная инженерия. Руководящие указания по адаптации инструментальных средств CASE);

ISO/IEC TR 9294:1990 Information technology – Guidelines for the management of software documentation (Информационная технология. Руководящие указания по административному управлению программной документацией);

ISO/IEC 6592:2000 Information technology – Guidelines for the documentation of computer-based application systems (Информационная технология. Руководящие указания по документированию прикладных информационных систем);

ISO/IEC 15910:1999 Information technology – Software user documentation process (Информационная технология. Пользовательская документация программных средств);

ISO/IEC 14756:1999 Information technology – Measurement and rating of performance of computer-based software systems (Информационная технология.

Измерение и определение рейтинга программных средств информационных систем).

Необходимо рассмотреть возможность применения ИСО/МЭК ТО

15504 в индустрии различных областей промышленности, отвечающих требованиям CALS-технологии (стандарты серии 10303).

Особое внимание необходимо обратить на создание единого и гармонизированного понятийно-терминологического аппарата в области индустрии программной продукции и информационных технологий.

1.3.3 ГОСТ Р ИСО/МЭК 12207–99. Информационная технология. Процессы жизненного цикла программных средств

Разработан Всероссийским научно-исследовательским институтом стандартизации (ВНИИстандарт) Госстандарта России. Настоящий стандарт содержит полный аутентичный текст международного стандарта ИСО/МЭК 12207–95 «Информационная технология. Процессы жизненного цикла программных средств».

1.3.3.1 Введение

Программные средства являются неотъемлемыми частями информационных технологий и традиционных систем, таких как транспортные, военные, финансовые и система здравоохранения. При этом подразумевается усиление роли стандартов, процедур, методов, средств (инструментария) и внешних условий для разработки и сопровождения программных средств (программного обеспечения). Подобная многоплановость подходов создает значительные трудности при управлении программными средствами и в технологиях программирования, особенно при интеграции продуктов и услуг. Требуется определенное упорядочение вопросов создания программных средств при переходе от подобной многоплановости к общей структуре, которая может быть использована профессионалами для «разговора на одном языке» при создании и управлении программными средствами. Настоящий стандарт устанавливает такую общую структуру.

Данная структура охватывает жизненный цикл программных средств от концепции замыслов через определение и объединение процессов до заказа и поставки программных продуктов и услуг. Кроме того, данная структура предназначена для контроля и модернизации данных процессов.

Процессы, определенные в настоящем стандарте, образуют множество общего назначения. Конкретная организация, в зависимости от своих целей, может выбрать соответствующее подмножество процессов для выполнения своих конкретных задач. Поэтому настоящий стандарт следует адаптировать для конкретной организации, проекта или приложения. Настоящий стандарт предназначен для использования как в случае отдельно поставляемых программных средств, так и для программных средств, встраиваемых или интегрируемых в общую систему.

1.3.3.2 Область применения

Назначение. Настоящий стандарт устанавливает, используя четко определенную терминологию, общую структуру процессов жизненного цикла программных средств, на которую можно ориентироваться в программной индустрии. Настоящий стандарт определяет процессы, работы и задачи, которые используются: при приобретении системы, содержащей программные средства, или отдельно поставляемого программного продукта; при оказании программной услуги, а также при поставке, разработке, эксплуатации и со- провождении программных продуктов. Понятие программных средств также охватывает программный компонент программно-аппаратных средств.

Настоящий стандарт также определяет процесс, который может быть использован при определении, контроле и модернизации процессов жизненного цикла программных средств.

Область распространения. Настоящий стандарт применяется при приобретении систем, программных продуктов и оказании соответствующих услуг, а также при поставке, разработке, эксплуатации и сопровождении программных продуктов и программных компонентов программно-аппаратных средств как в самой организации, так и вне ее. Стандарт содержит также те аспекты описания системы, которые необходимы для обеспечения понимания сути программных продуктов и услуг.

Стандарт также применяется при двусторонних отношениях и может в равной степени применяться, если обе стороны принадлежат к одной и той же

организации. Диапазон применения может простираться от неформального соглашения о сотрудничестве до официально заключаемого контракта (договора). Стандарт может использоваться одной из сторон для самоконтроля.

Стандарт не распространяется на готовые программные продукты, если они не входят в поставляемый продукт.

Стандарт предназначен: для заказчиков систем, программных продуктов и услуг; поставщиков; разработчиков; операторов; персонала сопровождения; администраторов проектов; администраторов, отвечающих за качество, и пользователей программных продуктов.

Адаптация настоящего стандарта. Настоящий стандарт определяет набор процессов, работ и задач, предназначенных для адаптации к условиям конкретных программных проектов. Процесс адаптации заключается в исключении неприменяемых в условиях конкретного проекта процессов, работ и задач.

Соответствие. Соответствие настоящему стандарту определяется как выполнение всех процессов, работ и задач, выбранных из настоящего стандарта в процессе адаптации, для конкретного программного проекта. Осуществление процесса или работы считается завершенным, когда выполнены всетребуемые для них задачи в соответствии с предварительно установленными в договоре критериями и требованиями.

Любая организация (например, национальная, промышленная ассоциация, компания), применяющая настоящий стандарт в качестве условия обеспечения торговых сделок, обязана определить и опубликовать минимальный набор требуемых процессов, работ и задач, который обеспечивает проверку соответствия поставщика настоящему стандарту.

Ограничения. Настоящий стандарт описывает архитектуру процессов жизненного цикла программных средств, но не определяет детали реализации или выполнения работ и задач, входящих в данные процессы.

Стандарт не предназначен для определения наименований, форматов или подробного содержания выпускаемой документации. Стандарт может требовать разработки документов одного класса или типа, например различных планов, но не предусматривает, чтобы такие документы разрабатывались или комплектовались раздельно или совместно. Решение этих вопросов оставлено на усмотрение пользователей настоящего стандарта.

Стандарт не предопределяет конкретной модели жизненного цикла или метода разработки программного средства. Пользователи, применяющие настоящий стандарт, должны сами выбирать модель жизненного цикла применительно к своему программному проекту и распределять процессы, работы и задачи, выбранные из настоящего стандарта, на данной модели; выбирать и применять методы разработки программных средств и выполнять работы и задачи, соответствующие конкретному программному проекту.

Стандарт не имеет противоречий с существующими в организациях стратегиями, стандартами или процедурами. Однако любые возникающие конфликтные ситуации должны быть разрешены, а любые противоречащие условия и ситуации должны быть упомянуты в примечаниях как исключения при применении настоящего стандарта.

В тексте настоящего стандарта слово «должны» используется для выражения соглашения между двумя или более сторонами; слово «должна» – для выражения объявления цели или намерения одной из сторон; слово «следует» – для выражения рекомендации из имеющихся возможных вариантов; слово «может» – для обозначения образа действий, допускаемого в рамках ограничений настоящего стандарта.

В тексте настоящего стандарта приведен ряд перечней задач, однако ни один из перечней нельзя считать исчерпывающим, и они приведены в качестве примеров.

1.3.3.3 Прикладное применение настоящего стандарта

В настоящем разделе определяются процессы жизненного цикла программных средств, которые могут быть реализованы при заказе, поставке, разработке, эксплуатации и сопровождении программных продуктов. Целью настоящего раздела является создание «путеводителя» для пользователей, который поможет ориентироваться в стандарте и осмысленно применять его.

Построение стандарта

В настоящем стандарте работы, которые могут выполняться в жизненном цикле программных средств, распределены по пяти основным, восьми вспомогательным и четырем организационным процессам. Каждый процесс жизненного цикла разделен на набор работ; каждая работа разделена на набор задач. Все процессы жизненного цикла описаны ниже (рисунок 1.19).

Основные процессы жизненного цикла

Раздел 5 состоит из пяти процессов, которые реализуются под управлением основных сторон, вовлеченных в жизненный цикл программных средств. Под основной стороной понимают одну из тех организаций, которые инициируют или выполняют разработку, эксплуатацию или сопровождение программных продуктов. Основные стороны – заказчик, поставщик, разработчик, оператор и персонал сопровождения программных продуктов. Основными процессами являются:

1. Процесс заказа (подразд. 5.1). Определяет работы заказчика, т. е. организации, которая приобретает систему, программный продукт или программную услугу.

2. Процесс поставки (подразд. 5.2). Определяет работы поставщика, т. е. организации, которая поставляет систему, программный продукт или программную услугу заказчику.

3. Процесс разработки (подразд. 5.3). Определяет работы разработчика, т. е. организации, которая проектирует и разрабатывает программный продукт.

4. Процесс эксплуатации (подразд. 5.4). Определяет работы оператора, т. е. организации, которая обеспечивает эксплуатационное обслуживание вычислительной системы в заданных условиях в интересах пользователей.

5. Процесс сопровождения (подразд. 5.5). Определяет работы персонала сопровождения, т. е. организации, которая предоставляет услуги по сопровождению программного продукта, состоящие в контролируемом изменениипрограммного продукта с целью сохранения его исходного состояния и функциональных возможностей. Данный процесс охватывает перенос и снятие с эксплуатации программного продукта.

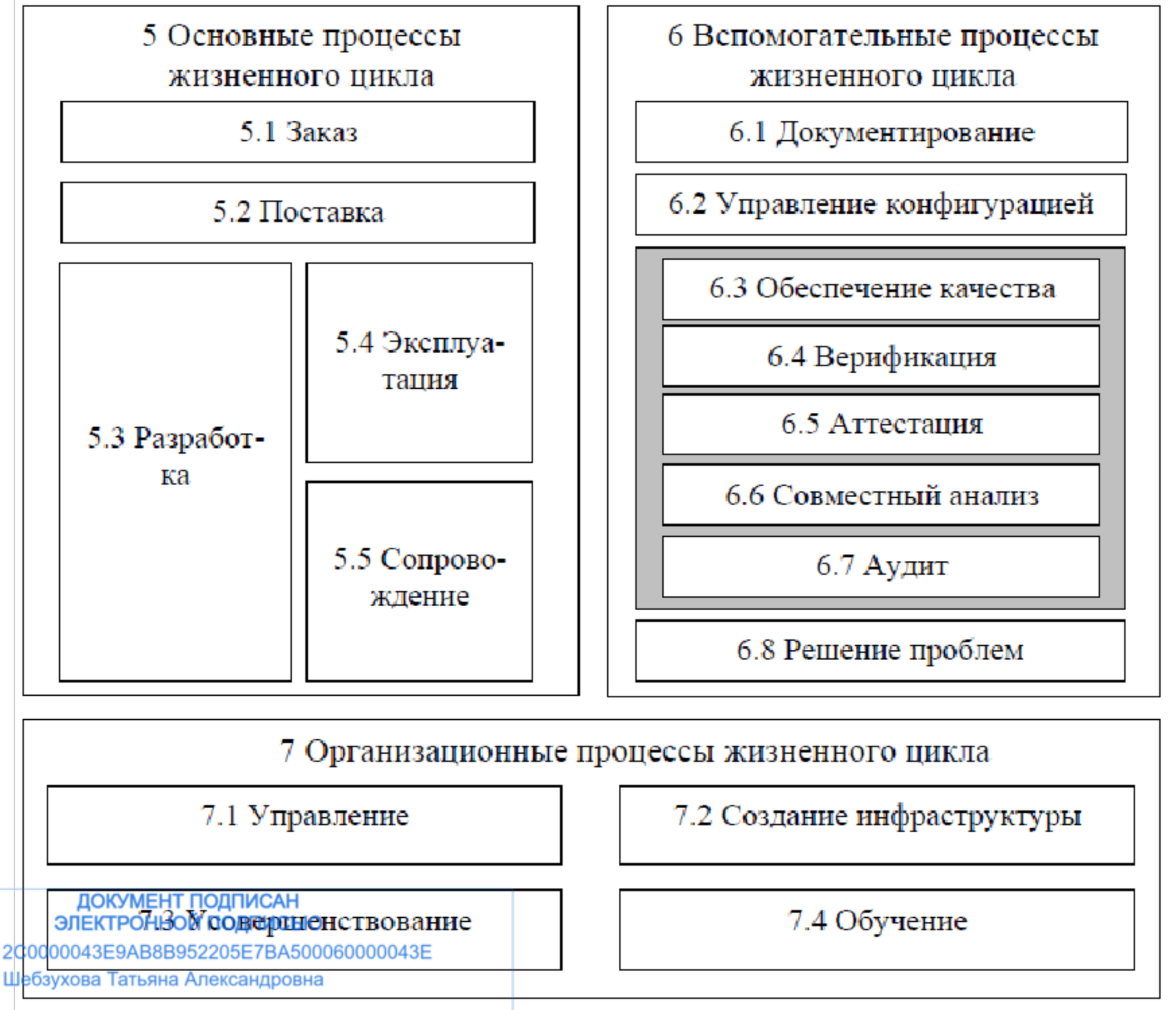

Рисунок 1.19 - Структура стандарта

Вспомогательные процессы жизненного цикла

Раздел 6 состоит из восьми процессов. Вспомогательный процесс является целенаправленной составной частью другого процесса, обеспечивающей успешную реализацию и качество выполнения программного проекта. Вспомогательный процесс, при необходимости, инициируется и используется другим процессом. Вспомогательными процессами являются:

1. Процесс документирования (подразд. 6.1). Определяет работы по описанию информации, выдаваемой в процессе жизненного цикла.

2. Процесс управления конфигурацией (подразд. 6.2). Определяет работы по управлению конфигурацией.

3. Процесс обеспечения качества (подразд. 6.3). Определяет работы по объективному обеспечению того, чтобы программные продукты и процессы соответствовали требованиям, установленным для них, и реализовывались в рамках утвержденных планов. Совместные анализы, аудиторские проверки, верификация и аттестация могут использоваться в качестве методов обеспечения качества.

4. Процесс верификации (подразд. 6.4). Определяет работы (заказчика, поставщика или независимой стороны) по верификации программных продуктов по мере реализации программного проекта.

5. Процесс аттестации (подразд. 6.5). Определяет работы (заказчика, поставщика или независимой стороны) по аттестации программных продуктов программного проекта.

6. Процесс совместного анализа (подразд. 6.6). Определяет работы по оценке состояния и результатов какой-либо работы. Данный процесс может использоваться двумя любыми сторонами, когда одна из сторон (проверяющая) проверяет другую сторону (проверяемую) на совместном совещании.

7. Процесс аудита (подразд. 6.7). Определяет работы по определению соответствия требованиям, планам и договору. Данный процесс может использоваться двумя сторонами, когда одна из сторон (проверяющая) контролирует программные продукты или работы другой стороны (проверяемой).

8. Процесс решения проблемы (подразд. 6.8). Определяет процесс анализа и устранения проблем (включая несоответствия), независимо от их характера и источника, которые были обнаружены во время осуществления разработки, эксплуатации, сопровождения или других процессов.

Организационные процессы жизненного цикла

Раздел 7 состоит из четырех процессов. Организационные процессы применяются в какой-либо организации для создания и реализации основной структуры, охватывающей взаимосвязанные процессы жизненного цикла и соответствующий персонал, а также для постоянного совершенствования данной структуры и процессов. Эти процессы, как правило, являются типовыми, независимо от области реализации конкретных проектов и договоров; однако уроки, извлеченные из таких проектов и договоров, способствуют совершенствованию организационных вопросов. Организационными процессами являются:

1. Процесс управления (подразд. 7.1). Определяет основные работы по управлению, включая управление проектом, при реализации процессов жизненного цикла.

2. Процесс создания инфраструктуры (подразд. 7.2). Определяет основные работы по созданию основной структуры процесса жизненного цикла.

3. Процесс усовершенствования (подразд. 7.3). Определяет основные работы, которые организация (заказчика, поставщика, разработчика, оператора, персонала сопровождения или администратора другого процесса) выполняет при создании, оценке, контроле и усовершенствовании выбранных процессов жизненного цикла.

4. Процесс обучения (подразд. 7.4). Определяет работы по соответствующему обучению персонала.

План конспекта:

1. Основные этапы развития технологии разработки.
2. Эволюция моделей жизненного цикла программного обеспечения.
3. Стандарты, регламентирующие процесс разработки программного обеспечения.

Работа с литературой:

Рекомендуемые источники информации

(№ источника)			
Основная	Дополнительная	Методическая	Интернет-ресурсы
1-3	1-3	1-3	1-2

Тема самостоятельного изучения № 2
Анализ проблемы и постановка задачи.

Вид деятельности студентов: самостоятельное изучение литературы
Итоговый продукт самостоятельной работы: конспект
Средства и технологии оценки: собеседование
Краткие теоретические сведения

2 АНАЛИЗ ПРОБЛЕМЫ И ПОСТАНОВКА ЗАДАЧИ

2.1 Введение в системный анализ

Системный анализ – система понятий, методов (среди которых должен быть метод декомпозиции) и технологий для изучения, описания, реализации систем различной природы и характера, междисциплинарных проблем; это система общих законов, методов, приемов исследования таких систем.

Любую предметную область также можно определить как системную.

Предметная область – раздел науки, изучающий предметные аспекты системных процессов и системные аспекты предметных процессов и явлений. Это определение можно считать системным определением предметной области.

Пример. Информатика – наука, изучающая информационные аспекты системных процессов и системные аспекты информационных процессов. Это определение можно считать системным определением информатики.

Системный анализ тесно связан с синергетикой.

Синергетика – междисциплинарная наука, изучающая общие идеи, методы и закономерности организации (изменения структуры, ее пространственно-временного усложнения) различных объектов и процессов, инварианты этих процессов. «Синергетика» в переводе с греческого означает «совместный», «согласованно действующий».

Системный анализ тесно связан и с философией. Философия дает общие методы содержательного анализа, а системный анализ дает общие методы формального, межпредметного анализа предметных областей, выявления и описания, изучения их системных инвариантов.

Можно дать и философское определение системного анализа: системный анализ – это прикладная диалектика.

Системный анализ предоставляет к использованию в различных науках, системах следующие методы и процедуры:

- абстрагирование и конкретизация;
- анализ и синтез;
- индукция и дедукция;
- формализация; структурирование; макетирование; алгоритмизация; моделирование; программное управление;
- распознавание, классификация и идентификация образов;
- экспертное оценивание и тестирование и другие методы и процедуры.

2.1.1 Системные ресурсы

Имеются следующие основные типы ресурсов в природе и в обществе.

Вещество – наиболее хорошо изученный ресурс, который в основном представлен таблицей Д. И. Менделеева и пополняется не так часто. Вещество выступает как отражение постоянства материи в природе, как мера однородности материи.

Энергия – не полностью изученный тип ресурсов, например, мы не владеем управляемой термоядерной реакцией. Энергия выступает как отражение изменчивости материи, переходов из одного вида в другой, как мера необратимости материи.

Информация – мало изученный тип ресурсов. Информация выступает как отражение порядка, структурированности материи, как мера порядка, самоорганизации материи (и

социума). Сейчас под этим термином мы будем понимать некоторые сообщения, ниже мы рассмотрим его более детально.

Человек – выступает как носитель интеллекта высшего уровня и является в экономическом, социальном, гуманитарном смысле важнейшим и уникальным ресурсом общества, выступает как мера разума, интеллекта и целенаправленного действия, мера социального начала, высшей формы отражения материи (сознания).

Организация (или организованность) выступает как форма ресурсов в социуме, группе, которая определяет его структуру, включая институты человеческого общества и его надстройки, выступает как мера упорядоченности ресурсов. Организация системы связана с наличием некоторых причинно-следственных связей в этой системе. Организация системы может иметь различные формы, например, биологическую, информационную, экологическую, экономическую, социальную, временную, пространственную и она определяется причинно-следственными связями в материи и социуме.

Пространство – мера протяженности материи (события), распределения ее (его) в окружающей среде.

Время – мера обратимости (необратимости) материи, событий. Время неразрывно связано с изменениями действительности.

Можно говорить о различных полях, в которые «помещен» любой человек: материальном, энергетическом, информационном, социальном, – их пространственных и временных характеристиках.

Пример. Рассмотрим простую задачу – пойти утром на занятия в вуз.

Эта часто решаемая студентом задача имеет все аспекты:

- материальный, физический аспект – студенту необходимо переместить некоторую массу, например, учебников и тетрадей на нужное расстояние;
- энергетический аспект – студенту необходимо иметь и затратить нужное количество энергии на перемещение;
- информационный аспект – необходима информация о маршруте движения и месторасположении вуза, кроме того, нужно обрабатывать инфор- мацию по пути своего движения;
- человеческий аспект – перемещение, в частности переезд на автобусе, невозможен без человека, например, без водителя автобуса;
- организационный аспект – необходимы подходящие транспортные сети и маршруты, остановки и т. д.;
- пространственный аспект – перемещение на определенное расстояние;
- временной аспект – на данное перемещение будет затрачено время (за которое произойдут соответствующие необратимые изменения в среде, в отношениях, в связях).

Все типы ресурсов тесно связаны. Более того, они невозможны друг без друга: актуализация одного из них ведет к актуализации другого.

Пример. При сжигании дров в печке выделяется тепловая энергия, тепловая энергия используется для приготовления пищи, пища используется для получения биологической энергии организма, биологическая энергия используется для получения информации (например, решения некоторой задачи), перемещения во времени и в пространстве. Человек и во время сна расходует свою биологическую энергию на поддержание информационных процессов в организме; более того, сон – продукт таких процессов.

Социальная организация и активность людей совершенствуют информационные ресурсы, процессы в обществе, последние, в свою очередь, совершенствуют производственные отношения.

Если классическое естествознание объясняет мир исходя из движения, взаимопревращений вещества и энергии, то сейчас реальный мир, объективная реальность могут быть объяснены лишь с учетом сопутствующих системных, особенно, системно-информационных процессов.

2.2 Анализ проблемы и моделирование предметной области с использованием системного подхода

2.2.1 Основные положения

Анализ проблем – это процесс осознания реальных проблем и потребностей пользователей и предложения решений, позволяющих удовлетворить эти потребности.

Цель анализа проблемы состоит в том, чтобы добиться лучшего понимания решаемой проблемы до начала разработки.

Чтобы выявить причины (или проблемы, стоящие за проблемой), необходимо опросить людей, которых непосредственно затрагивает данная проблема. Выявление актантов системы является ключевым шагом в анализе проблемы.

При этом необходимо проанализировать и понять область проблемы и исследовать разнообразные области решений. Как правило, возможных решений множество, и нам необходимо найти то, которое наиболее соответствует решаемой проблеме.

Чтобы иметь возможность провести анализ проблемы, полезно определить, что же собой представляет проблема. По определению Гауса и Вайнберга (Cause, Weinberg, 1989) проблема – это разница между желаемым и воспринимаемым.

Это определение достаточно разумно, по крайней мере, оно устраняет часто встречающееся среди разработчиков заблуждение, что подлинная проблема заключается в том, что пользователь не понимает, в чем состоит проблема! Согласно данному определению, если пользователь ощущает нечто как проблему, это и есть настоящая проблема, и она достойна внимания.

При анализе проблемы необходимо осуществить следующие пять этапов:

1) достигнуть соглашения об определении проблемы;
2) выделить основные причины – проблемы, стоящие за проблемой;
3) выявить заинтересованных лиц и пользователей;
4) определить границу системы решения;
5) выявить ограничения, которые необходимо наложить на решение.

2.2.2 Этап 1. Достижение соглашения об определении проблемы

Один из простейших способов заключается в том, чтобы просто записать проблему и выяснить, все ли согласны с такой постановкой (таблица 2.1).

В рамках этого процесса зачастую полезно рассмотреть преимущества предлагаемого решения, причем их следует описывать на языке клиентов/пользователей. Это обеспечивает дополнительную содержательную основу для понимания реальной проблемы. Рассматривая с точки зрения клиента эти преимущества, мы также достигаем лучшего понимания их взгляда на проблему в целом.

Таблица 2.1 - Стандартная форма постановки проблемы

Элемент	Описание
Проблема	[Описание проблемы]
Воздействует на	[Указание лиц, на которых оказывает влияние данная проблема]
Результатом чего является	[Описание воздействия данной проблемы на заинтересованных лиц и бизнес-деятельность]
Выигрыш от	[Указание предлагаемого решения]
Может состоять в следующем	[Список основных предоставляемых решением преимуществ]

2.2.3 Этап 2. Выделение основных причин, стоящих за проблемой

Для понимания реальной проблемы и ее причин можно использовать множество методов. Одним из них является метод анализа корневых причин, представляющий собой семантический способ нахождения причин, лежащих в основе рассматриваемой проблемы или ее проявления.

Рассмотрим реальный пример. Компания GoodsAreUs, занимающаяся торговлей по каталогу, производит и рассылает на дом множество недорогих товаров различных наименований. Решив заняться проблемой недостаточной прибыльности, компания использует рекомендуемую ей программой обеспечения качества методику «качество – во всем» (total quality management, TQM). Применив данный подход, компания практически сразу обратила внимание на ущерб

от несоответствия (cost of nonconformance), который представляет собой стоимость всего, что идет не так, как надо, и приводит к бесполезным затратам. Этот ущерб включает в себя переделки, остатки, неудовлетворенность клиента, текучесть кадров и другие негативные факторы. Проанализировав ущерб от несоответствия, компания заподозрила, что наибольший вклад в него вносят «остатки».

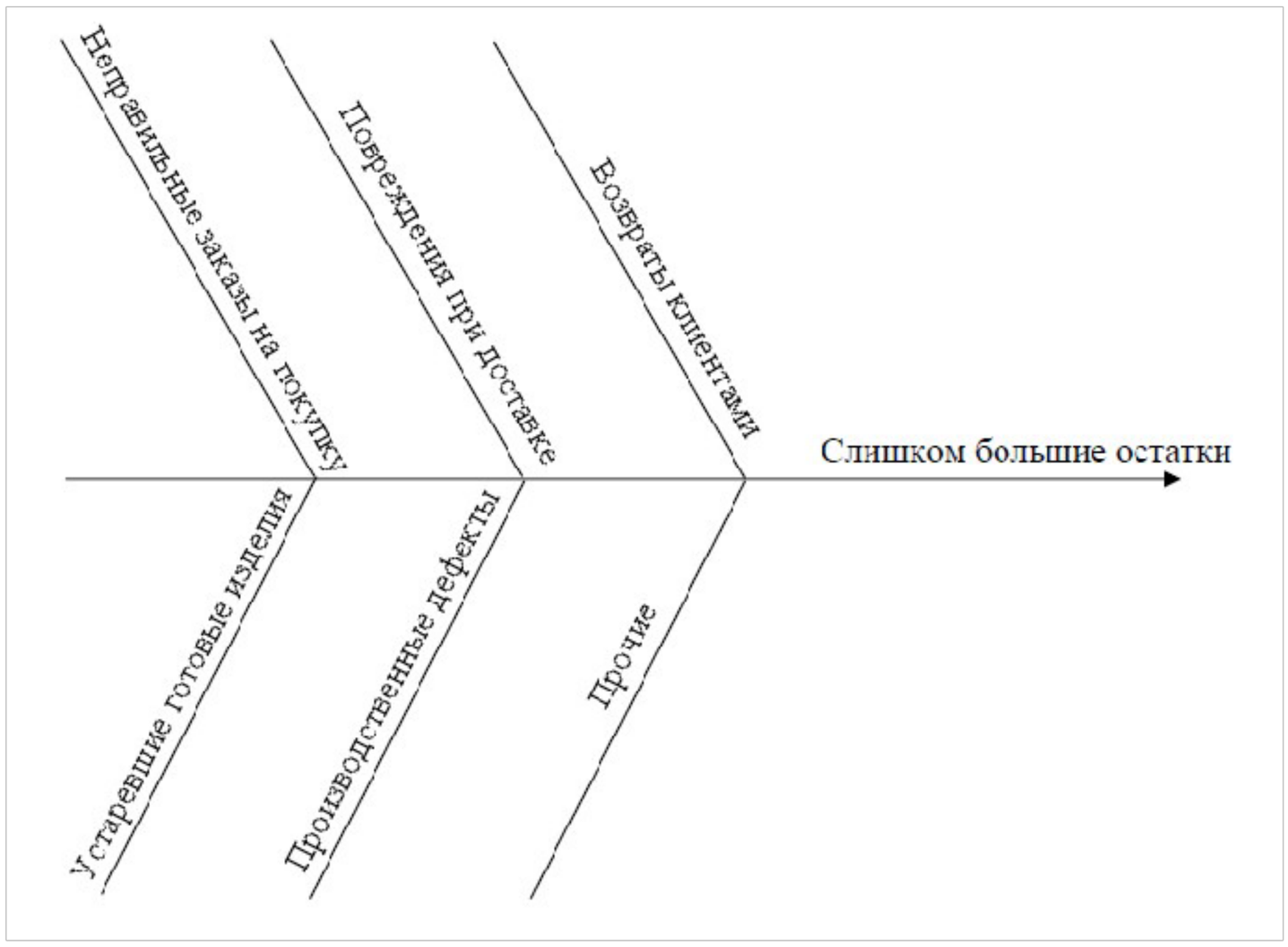

Рисунок 2.1 - Диаграмма в форме рыбного скелета для отображения корневых причин

Следующим шагом должно стать определение того, какие факторы оказывают влияние на величину остатков. TQM советует для обнаружения проблем, стоящих за проблемой, использовать диаграмму в форме рыбного скелета (рисунок 2.1). В нашем случае компания выявила много источников, вносящих свой вклад в остатки. Каждый источник указан как одна из «косточек» на диаграмме.

Способ выявления корневых причин зависит от конкретного случая.

Существует несколько способов выявления причин:

- опрос сотрудников, непосредственно занимающихся этим делом;
- «мозговой штурм» с участием тех, кто знаком с данной областью;
- метод упрощенной спецификации приложений;
- совместная разработка приложений;
- пользовательский сценарий и сеансы разработки схем выбора.

2.2.3.1 Устранение корневых причин

Качественные данные свидетельствуют, что многие корневые причины просто не стоят того, чтобы их устранять, поскольку затраты на их устранение превысят причиняемый проблемой ущерб.

Предложение заменить существующую систему или разработать новую, должно быть хорошо аргументированным. Для этого нужно предоставить стоимостное обоснование такой системы, оценив затраты на разработку и дивиденды от эксплуатации этой системы.

Продолжая анализ, можно применить новую диаграмму в виде рыбного скелета для определения того, какие типы ошибок вносят наибольший вклад в проблему неправильности заказов. Полученные данные можно затем использовать для определения функций новой системы программного обеспечения, которая призвана устранить эти ошибки. Раз мы приняли решение, что проблема неправильных заказов на покупку достойна решения, можно создать для нее формальную постановку (таблица 2.2).

Таблица 2.2 - Постановка проблемы ввода заказов на покупку

Элементы	Описание
Проблема воздействует на	Выполнение неправильных заказов на покупку, выполняющий заказы персонал, клиентов, производство, продажи и обслуживание клиентов
Результатом чего является	Увеличение остатков, повышение производственных затрат, неудовлетворенность клиентов, уменьшение прибыли
Выигрыш от	Новой системы, направленной на решение данной проблемы
Может состоять в следующем	Повышение точности заказов на покупку в точке ввода, совершенствование учета данных о покупках, в конечном счете – увеличение прибыли

После постановки проблемы, ее следует передать для ознакомления заказчикам и заинтересованным лицам, чтобы они внесли свои комментарии. Затем постановка проблемы доводится до сведения всех членов команды разработчиков с тем, чтобы все работали в направлении достижения общей цели.

2.2.4 Этап 3. Выявление заинтересованных лиц и пользователей

При решении любой сложной проблемы, как правило, приходится удовлетворять потребности различных групп заинтересованных лиц. Эти группы обычно имеют различные точки зрения на проблему и различные потребности, которые должны быть учтены в решении.

Заинтересованные лица – это все, на кого реализация новой системы или приложения может оказать материальное воздействие.

Понимание потребностей пользователей и других заинтересованных лиц является ключевым фактором в выработке успешного решения.

Первая категория заинтересованных лиц – это пользователи системы.

Их потребности легко учесть, поскольку они будут непосредственно привлекаться к определению и использованию системы. Вторую категорию составляют непрямые пользователи, а также те, на кого воздействуют только бизнес последствия разработки и те, кто воздействует на разработку. Потребности заинтересованных лиц, не являющихся пользователями, также необходимо выявить и учесть.

В зависимости от того, в какой предметной области работает команда, выявление заинтересованных лиц может оказаться как тривиальным, так и нетривиальным этапом анализа проблемы. Часто достаточно провести простой опрос среди тех, кто принимает решения, а также опросить потенциальных пользователей и другие заинтересованные стороны. В этом процессе могут оказаться полезными следующие вопросы.

Кто является пользователями системы?

Кто является заказчиком (экономическим покупателем) системы?

На кого еще окажут влияние результаты работы системы?

Кто будет оценивать и принимать систему, когда она будет представлена и развернута?

Существуют ли другие внутренние или внешние пользователи системы, чьи потребности необходимо учесть?

Кто будет заниматься сопровождением новой системы?

Не забыли ли мы кого-нибудь?

В нашем примере замены системы заказов на покупку основными и наиболее очевидными пользователями являются служащие, занимающиеся вводом заказов на покупку. Они определенно являются заинтересованными лицами, так как их производительность, удобство, комфорт, выполнение работы и ее результаты зависят от системы. Кого еще из заинтересованных лиц можно выделить?

На руководителя отдела приема заказов система также оказывает непосредственное воздействие, но он взаимодействует с системой не напрямую, а посредством различных интерфейсов пользователя и форм отчетов. Главный финансист компании также, очевидно, принадлежит к заинтересованным лицам, так как ожидается, что система повлияет на производительность, качество предоставляемых услуг и прибыльность компании. Наконец,