Министерство образования Республики Беларусь Учреждение образования

«Белорусский государственный университет информатики и радиоэлектроники»

Кафедра электронных вычислительных машин

**А. В. Отвагин, Н. А. Павлёнок**

# ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ И ПРОГРАММИРОВАНИЕ

Лабораторный практикум

для студентов специальности 1-40 02 01

«Вычислительные машины, системы и сети» всех форм обучения

Минск БГУИР 2011

УДК 004.4’2+004.51(076.5) ББК 32.973.26-018.2я73

О-80

Р е ц е н з е н т:

ведущий научный сотрудник лаборатории №222 Объединенного института проблем информатики Национальной академии наук Беларуси,

кандидат технических наук А. А. Дудкин

О-80

## Отвагин, А. В.

Объектно-ориентированное проектирование и программирование: лаб. практикум для студ. спец. 1-40 02 01 «Вычислительные машины, системы и сети» всех форм обуч. / А. В. Отвагин, Н. А. Павлёнок. – Минск : БГУИР, 2011. – 44 с. : ил.

ISBN 978-985-488-571-1.

Лабораторный практикум содержит методические указания и задания к лабора- торным работам, предусмотренным учебной программой дисциплины «Объектно- ориентированное проектирование и программирование». Рассмотрены средства авто- матизации проектирования объектно-ориентированного программного обеспечения, создание графического интерфейса пользователя и использование средств разработки программ на платформе Microsoft.NET.

**УДК 004.4’2+004.51(076.5) ББК 32.973.26-018.2я73**

**ISBN 978-985-488-571-1** © Отвагин А. В., Павлёнок Н. А., 2011

© УО «Белорусский государственный университет информатики

и радиоэлектроники», 2011

2

СОДЕРЖАНИЕ

[Лабораторная работа №1. СИСТЕМА RATIONAL ROSE 4](#_TOC_250041)

* 1. [Основные сведения о системе Rational Rose 4](#_TOC_250040)
  2. [Пользовательский интерфейс Rational Rose 5](#_TOC_250039)
  3. [Представление вариантов использования 5](#_TOC_250038)
  4. [Задание на лабораторную работу 8](#_TOC_250037)
  5. [Контрольные вопросы 10](#_TOC_250036)

[Лабораторная работа №2. ДИАГРАММЫ RATIONAL ROSE 11](#_TOC_250035)

* 1. [Основные виды диаграмм UML, реализуемые в Rational Rose 11](#_TOC_250034)
  2. [Диаграммы взаимодействия компонентов системы 11](#_TOC_250033)
  3. [Диаграммы последовательности 12](#_TOC_250032)
  4. [Диаграммы коопераций 14](#_TOC_250031)
  5. [Задание на лабораторную работу 14](#_TOC_250030)
  6. [Контрольные вопросы 14](#_TOC_250029)

[Лабораторная работа №3. ОПИСАНИЕ ПРОЕКТА ПО 15](#_TOC_250028)

* 1. [Описание статической структуры проекта диаграммой классов 15](#_TOC_250027)
  2. [Диаграммы состояний 17](#_TOC_250026)
  3. [Структуризация классов программного средства 18](#_TOC_250025)
  4. [Задание на лабораторную работу 19](#_TOC_250024)
  5. [Контрольные вопросы 19](#_TOC_250023)

[Лабораторная работа №4. РЕИНЖИНИРИНГ 20](#_TOC_250022)

* 1. [Диаграммы компонентов 20](#_TOC_250021)
  2. [Диаграммы развертывания 21](#_TOC_250020)
  3. [Средства генерации программного кода 23](#_TOC_250019)
  4. [Средства обратного проектирования 23](#_TOC_250018)
  5. [Задание на лабораторную работу 24](#_TOC_250017)
  6. [Контрольные вопросы 24](#_TOC_250016)

[Лабораторная работа №5. МНОГООКОННЫЙ ИНТЕРФЕЙС 25](#_TOC_250015)

* 1. [Основные компоненты многооконного интерфейса 25](#_TOC_250014)
  2. [Архитектура приложений для графического интерфейса 26](#_TOC_250013)
  3. Создание базового приложения для графического интерфейса, обработка событий ... 27 5.4. Создание меню 29
  4. [Задание на лабораторную работу 30](#_TOC_250012)
  5. [Контрольные вопросы 30](#_TOC_250011)

Лабораторная работа №6. АРХИТЕКТУРА «МОДЕЛЬ–ПРЕДСТАВЛЕНИЕ– КОНТРОЛЛЕР» 30

* 1. Общие сведения об архитектуре «Модель–представление–контроллер» 30
  2. [Обобщённое логическое представление архитектуры MVC на UML 31](#_TOC_250010)
  3. Пример использования шаблона «Модель–представление–контроллер» для

реализации простого элемента управления 32

* 1. [Задание на лабораторную работу 35](#_TOC_250009)
  2. [Контрольные вопросы 35](#_TOC_250008)

[Лабораторная работа №7. ДИАЛОГОВЫЕ ОКНА 35](#_TOC_250007)

* 1. [Назначение и виды диалоговых окон 35](#_TOC_250006)
  2. [Создание диалогового окна 36](#_TOC_250005)
  3. [Контроль пользовательского ввода 37](#_TOC_250004)
  4. [Интернационализация приложений 39](#_TOC_250003)
  5. [Задание на лабораторную работу 42](#_TOC_250002)
  6. [Контрольные вопросы 42](#_TOC_250001)

[Литература 43](#_TOC_250000)

## Лабораторная работа №1 СИСТЕМА RATIONAL ROSE

*Цель работы*: изучить возможности и пользовательский интерфейс сис- темы поддержки проектирования программного обеспечения Rational Rose. Создать представление вариантов использования для индивидуального вариан- та задания на разработку программного обеспечения.

## Основные сведения о системе Rational Rose

Важным достижением развития методологии ООП явилось осознание то- го, что процесс написания программного кода может быть отделен от процесса проектирования структуры программы. Прежде чем начать программирование классов, их свойств и методов, необходимо определить сами эти классы, свой- ства и методы, необходимые для придания им требуемого поведения, а также взаимосвязи между классами. Эта совокупность задач решается в процессе об- щего анализа требований к будущей программе, а также анализа конкретной предметной области ее применения. Все эти обстоятельства привели к появле- нию специальной методологии, получившей название методологии объектно- ориентированного анализа и проектирования (ООАП).

Объектно-ориентированный анализ и проектирование (Object-Oriented Analysis/ Design, OOA/D) – технология разработки программных систем, в ос- нову которых положена объектно-ориентированная методология представления предметной области в виде объектов, являющихся экземплярами соответст- вующих классов.

Унифицированный язык моделирования (Unified Modeling Language, UML) является графическим языком для визуализации, специфицирования, конструирования и документирования систем, в которых большая роль принад- лежит программному обеспечению. С помощью UML можно разработать де- тальный план создаваемой системы, содержащий не только ее концептуальные элементы (системные функции, бизнес-процессы), но и конкретные особенно- сти (например, классы, написанные на специальных языках программирования, схемы баз данных, программные компоненты многократного использования). Текущей версией UML является 2.0, однако в качестве международного стан- дарта ISO/IEC принят UML 1.4.2.

Rational Rose – семейство объектно-ориентированных CASE-средств фирмы Rational Software Corporation, предназначенное для автоматизации про- цессов анализа и проектирования ПО, а также для генерации кодов на различ- ных языках программирования и выпуска проектной документации.

В основе работы Rational Rose лежит построение диаграмм и специфика- ций UML, определяющих архитектуру системы, её статические и динамические аспекты.

Основные структурные компоненты Rational Rose:

1. Репозиторий, представляющий собой базу данных проекта.
2. Графический интерфейс пользователя.
3. Средства просмотра проекта – браузер, обеспечивающий навигацию по проекту, в том числе по иерархиям классов и подсистем, переключение от одного вида диаграмм к другому и т. д.
4. Средства контроля проекта и сбора статистики, позволяющие нахо- дить и устранять ошибки по мере развития проекта.
5. Генератор документов, формирующий тексты выходных документов на основе информации, содержащейся в репозитории.
6. Средства автоматической генерации кодов программ, формирующие на основе информации в диаграммах классов и компонентов файлы заголовков и файлы описаний классов и объектов.
7. Анализатор кодов С++ (реализован в виде отдельного программного модуля), который на основе определяемых пользователем исходных текстов на языке C++ может создавать модули проектов Rational Rose.

В результате разработки проекта с помощью CASE-средств Rational Rose могут быть сформированы следующие документы:

* + - диаграммы UML, представляющие собой модель разрабатываемого ПО;
    - спецификации классов, объектов, атрибутов и операций;
    - заготовки текстов программ.

## Пользовательский интерфейс Rational Rose

Основными элементами интерфейса являются:

1. Браузер проекта – используется для быстрой навигации по модели.
2. Окно документации – применяется для работы с текстовым описанием модели.
3. Панели инструментов (главная и стандартная) – применяются для бы- строго доступа к наиболее распространённым командам.
4. Рабочая область изображения диаграммы – используется для просмот- ра и редактирования одной или нескольких диаграмм UML.
5. Журнал – применяется для просмотра ошибок и отчётов о результатах выполнения различных команд.

В модели Rational Rose поддерживается четыре представления (views) – представление вариантов использования, логическое представление, представ- ление компонентов и представление размещения. В первой лабораторной рабо- те рассматривается представление вариантов использования.

## Представление вариантов использования

Понятие варианта использования, или прецедента (Use Case) является ос- новным элементом разработки и планирования проекта. Вариант использования представляет собой последовательность действий (транзакций), выполняемых системой в ответ на событие, инициируемое некоторым внешним объектом (действующим лицом). Прецедент описывает типичное взаимодействие между пользователем и системой.

Действующее лицо или актёр (Actor) – это роль, которую пользователь играет по отношению к системе. Действующим лицом может выступать не только пользователь, но также другая система, взаимодействующая с данной, и время. Время становится действующим лицом, если от него зависит запуск ка- кого-либо события в системе.



Рассмотрим пример диаграммы использования, иллюстрирующий работу банкомата (рис. 1.1).





Рис. 1.1. Диаграмма вариантов использования для системы «Банкомат»



Действующими лицами в данной системе являются клиент и банковская система. Проектируемая система «Банкомат» должна выполнять следующие действия: перевести деньги, сделать вклад, снять деньги со счёта, изменить код, показать баланс, осуществить оплату. На диаграмме показано взаимодействие между вариантами использования и действующими лицами, отражены требова- ния к системе с точки зрения пользователя. Можно сказать, что варианты ис- пользования – это функции, выполняемые системой, а актёры – заинтересован- ные лица (stakeholders), инициирующие варианты использования либо полу- чающие от него информацию.

При разработке диаграмм вариантов использования желательно придер- живаться следующих правил.

1. Действующие лица находятся вне сферы действия проектируемой сис- темы, а значит, и связи между ними моделировать не стоит.
2. Нежелательно соединять коммуникационной связью два варианта ис- пользования, так как диаграмма должна описывать доступные системе вариан- ты использования, а не порядок их следования.
3. Вариант использования должен быть инициирован действующим ли- цом (должна быть сплошная стрелка, идущая от актёра к прецеденту).

Начинать строить диаграмму лучше всего с перечисления всех событий внешнего мира, на которые система должна реагировать.

Более конкретные детали вариантов использования описываются в специ- альном документе, называемом *поток событий* (Flow of Events). Его целью яв- ляется документирование процесса обработки данных в рамках варианта ис- пользования.

Поток событий обычно включает:

* краткое описание;
* предусловия (pre-conditions);
* основной поток событий;
* альтернативный поток событий (один или несколько);
* постусловия (post-conditions).

Краткое описание. Каждый прецедент должен иметь связанное с ним ко- роткое описание выполняемого действия. Например: *Вариант использования*

*«Перевести деньги» позволяет клиенту или служащему банка переводить деньги с одного счёта на другой.*

Предусловия. Предусловия прецедента – это такие условия, которые должны быть выполнены, прежде чем прецедент начнёт выполняться сам. На- пример, выполнение другого варианта использования, наличие у пользователя прав доступа и т. д.

Основной и альтернативный потоки событий. Поток событий поэтапно описывает, что должно выполняться в варианте использования:

* способ запуска варианта использования;
* различные пути выполнения варианта использования;
* нормальный поток событий;
* отклонения от нормального потока (альтернативные потоки);
* потоки ошибок;
* способ завершения варианта использования.

*Пример*: *Поток событий для прецедента «Снять деньги со счёта» Основной поток:*

1. *Вариант использования начинается, когда клиент вставляет карточ- ку в банкомат.*
2. *Банкомат выводит приветствие и предлагает пользователю ввести пин-код.*
3. *Клиент вводит пин-код.*
4. *Банкомат подтверждает введённый пин-код. Если код не подтвер- ждается, выполняется альтернативный поток А1.*
5. *Банкомат выводит список доступных действий*: *положить деньги, снять деньги, перевести деньги.*
6. *Клиент выбирает пункт «Снять деньги».*
7. *Банкомат запрашивает сумму.*
8. *Клиент вводит сумму.*
9. *Банкомат проверяет наличие введённой суммы на счёте. Если денег на счёте недостаточно, выполняется альтернативный поток А2. При возник- новении ошибки выполняется поток ошибки Е1.*
10. *Банкомат вычитает сумму из счёта.*
11. *Банкомат выдаёт требуемую сумму.*
12. *Банкомат возвращает клиенту карточку.*
13. *Банкомат печатает чек.*
14. *Вариант использования завершается.*

Постусловия. Условия, которые должны быть выполнены после заверше- ния прецедента (пометка флажком переключателя, выполнение другого вариан- та использования).

В языке UML на диаграммах прецедентов поддерживается несколько ти- пов связей между элементами:

* коммуникация (communication) – обозначается сплошной линией со стрелкой – связь между актёром и прецедентом, направление стрелки позволя- ет понять, кто инициирует коммуникацию;
* включение (include) – обозначается линией с соответствующим стерео- типом – применяется в тех ситуациях, когда фрагмент поведения системы по- вторяется в нескольких вариантах использования (например, аутентификация клиента в системе «Банкомат» требуется как в прецеденте «Снять деньги со счёта», так и «Сделать вклад» и т. д.);
* расширение (extend) – обозначается аналогично включению – применя- ется при описании изменений в нормальном поведении системы, что позволяет прецеденту использовать функциональные возможности другого прецедента только при необходимости;
* обобщение (generalization) – незакрашенная стрелка – показывает, что у нескольких актёров имеются общие черты. Такие связи необходимы, только ес- ли поведение действующего лица одного типа отличается от поведения актёра другого типа, в противном случае показывать обобщение не следует.

Пример различных связей между прецедентами показан на рис. 1.2.

## Задание на лабораторную работу

В соответствии с вариантом необходимо при помощи Rational Rose по- строить модель программного обеспечения. Процесс создания модели должен включать следующие этапы:

1. Составление глоссария проекта и дополнительных спецификаций.
2. Создание модели вариантов использования.
3. Анализ вариантов использования.
4. Проектирование системы.
5. Реализация системы.





Рис. 1.2. Связи коммуникации, включения, расширения и обобщения



При создании модели вариантов использования должны быть выполнены следующие требования:

* глоссарий проекта должен иметь форму таблицы и храниться в отдель- ном файле;
* дополнительные спецификации также хранятся в отдельном файле;
* на диаграммах прецедентов каждый актёр и прецедент должны сопро- вождаться описанием на русском языке;
* описание актёра должно кратко (1–2 строки) сообщать о роли данного лица в системе;
* описание прецедента должно включать пояснения, предусловия, потоки событий, постусловия;
* описания представляют собой либо присоединённые текстовые файлы, либо текст в поле Documentation спецификации соответствующего элемента диаграммы.

Варианты задания на реализацию программного обеспечения приведены в следующей таблице.

Варианты задания к лабораторным работам

|  |  |
| --- | --- |
| 1. Цифровой диктофон | 6. Холодильник |
| 2. Торговый автомат | 7. Кодовый замок |
| 3. Табло на станции метро | 8. Турникет метро |
| 4. Система автоматизации для пункта проката видеокассет | 9. Система учёта товаров |
| 5. Мини-АТС | 10. Библиотечная система |

Окончание табл.

10

|  |  |
| --- | --- |
| 11. Телефон | 17. Интернет-магазин |
| 12. Стиральная машина | 18. WWW-конференция |
| 13. Таксофон | 19. Каталог ресурсов Интернет |
| 14. Банкомат | 20. Будильник |
| 15. Генеалогическое дерево | 21. Телевизор |
| 16. Система поддержки составления расписания занятий | 22. Домофон |

## Контрольные вопросы

1. Что такое ООАП (OOA/D)?
2. Каковы основные элементы диаграммы вариантов использования?
3. Что такое поток событий, из каких элементов он состоит?
4. Какие типы связей между элементами диаграммы вариантов использования поддерживает UML? Поясните их смысл.

## Лабораторная работа №2 ДИАГРАММЫ RATIONAL ROSE

*Цель работы*: изучить разновидности диаграмм описания программного обеспечения и средства их разработки в системе Rational Rose, освоить инстру- менты проектирования диаграмм. Создать диаграммы взаимодействия для ва- рианта индивидуального задания.

## Основные виды диаграмм UML, реализуемые в Rational Rose

Основной набор диаграмм для моделирования, представляемый стандар- том UML, содержит 8 типов диаграмм:

1. *Диаграммы вариантов использования,* или *Диаграммы прецедентов* (Use Case Diagrams) – для моделирования бизнес-процессов и составления тре- бований к проектируемой системе.
2. *Диаграммы классов* (Class Diagrams) – для моделирования статической структуры классов проектируемой системы и связей между ними.
3. *Диаграммы последовательности* (Sequence Diagrams) – диаграммы, на которых изображается упорядоченное во времени взаимодействие объектов, в частности, участвующие во взаимодействии объекты и последовательность со- общений, которыми они обмениваются.
4. *Диаграммы коопераций* (Collaboration Diagrams) – диаграммы, на ко- торых также изображаются взаимодействия между структурными элементами системы. В отличие от диаграммы последовательности на диаграмме коммуни- кации явно указываются отношения между элементами (объектами), а время как отдельное измерение не используется (применяются порядковые номера вызовов).
5. *Диаграммы состояний* (State Machine Diagrams) – для моделирования поведения объектов системы при переходе из одного состояния в другое.
6. *Диаграммы деятельности* (Activity Diagrams) – для моделирования поведения системы в рамках различных вариантов использования.
7. *Диаграммы компонентов* (Component Diagrams) – статические струк- турные диаграммы, показывающие разбиение системы на структурные компо- ненты и связи между компонентами.
8. *Диаграммы размещения* (Deployment Diagrams) – диаграммы, модели- рующие физическую архитектуру системы.

## Диаграммы взаимодействия компонентов системы

Для описания поведения взаимодействующих групп объектов в UML применяются диаграммы взаимодействия (Interaction Diagrams). Как правило, диаграмма взаимодействия охватывает поведение объектов в рамках только од- ного варианта использования. На диаграммах взаимодействия отображается ряд объектов и те сообщения, которыми они обмениваются между собой.

Сообщение (message) – средство, с помощью которого объект- отправитель запрашивает у объекта-получателя выполнение одной из его опе- раций. Можно выделить следующие типы сообщений:

* информационное (informative) – сообщение, снабжающее объект- получатель какой-либо информацией для обновления его состояния;
* запрос (interrogative) – сообщение, запрашивающее выдачу некоторой информации об объекте получателе;
* императивное (imperative) – сообщение, запрашивающее у объекта- получателя выполнение некоторых действий.

Нотация UML предоставляет несколько видов диаграмм взаимодействия, наиболее распространёнными среди которых являются диаграммы последова- тельности (Sequence Diagrams) и диаграммы коопераций (Collaboration Dia- grams).

## Диаграммы последовательности

На диаграмме последовательности основное внимание уделяется временнόй упорядоченности сообщений. На ней изображаются объекты, непо- средственно участвующие во взаимодействии, и сообщения, которыми эти объ- екты обмениваются (рис. 2.1).

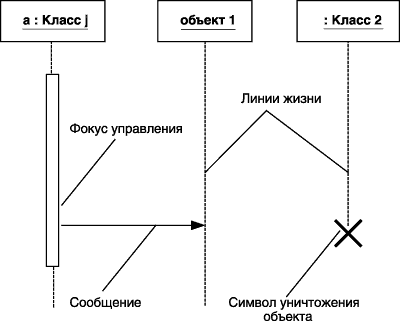


Рис. 2.1. Графические элементы диаграммы последовательности

Каждый объект изображается в виде прямоугольника, от которого начи- нается его *линия жизни*. Линия жизни отражает существование объекта во вре- мени. Большинство объектов существует на протяжении всего времени взаимо- действия, как показано на рис. 2.1, другие могут создаваться во время взаимо-

действия, и тогда их линии жизни начинаются с получения сообщения со сте- реотипом create. Также объекты могут уничтожаться во время взаимодействия (стереотип destroy), и уничтожение объекта обозначается специальным симво- лом.

Имя объекта записывается со строчной буквы, затем через двоеточие сле- дует имя класса, вся запись подчеркивается. Если на диаграмме последователь- ности отсутствует собственное имя объекта, то при этом должно быть указано имя класса – такой объект считается анонимным.

Порядок расположения объектов на диаграмме последовательности опре- деляется исключительно соображениями удобства визуализации их взаимодей- ствия друг с другом.

*Фокус управления* изображается в виде вытянутого прямоугольника и по- казывает промежуток времени, в течение которого объект выполняет какое-либо действие непосредственно или с помощью подчиненной процедуры. Верхняя грань прямоугольника выравнивается по временной оси с моментом начала действия, нижняя – с моментом его завершения.

Сообщения, которыми обмениваются объекты, обозначаются стрелками различного вида и упорядочиваются по времени (поэтому их порядковые номе- ра указывать необязательно). Каждое сообщение на диаграмме последователь- ности ассоциируется с определенной операцией, которая должна быть выпол- нена принявшим его объектом. При этом могут указываться аргументы или па- раметры, значения которых влияют на получение различных результатов. Кро- ме того, значения параметров отдельных сообщений могут содержать условные выражения, образуя ветвление, или альтернативные пути основного потока управления.

Основные виды обозначения сообщений приведены на рис. 2.2.

Рис. 2.2. Основные виды сообщений:

а – вызов процедуры; б – асинхронное сообщение; в – возврат из процедуры

Для изображения ветвления используются две или более стрелки, выхо- дящие из одной точки фокуса управления объекта. Перед порядковым номером сообщения ставится выражение-условие, например [х > 0]. У всех альтернатив- ных ветвей будет один и тот же порядковый номер, но условия на каждой ветви должны быть заданы так, чтобы они не выполнялись одновременно. Для моде- лирования итерации перед номером сообщения в последовательности ставится выражение итерации, например \* [ i : = 1. . n] (или просто \*, если надо обозна- чить итерацию без дальнейшей детализации).

## Диаграммы коопераций

Диаграмма коопераций, в отличие от диаграммы последовательности, ак- центирует внимание на организации объектов, принимающих участие во взаи- модействии. Она представляется в виде графа, вершины которого – объекты, участвующие во взаимодействии, а дуги – связи, соединяющие объекты. Связи дополняются сообщениями, которые объекты принимают и посылают. Таким образом, диаграмма коопераций даёт визуальное представление о потоке управления в контексте структурной организации кооперирующихся объектов.

В основном описанная нотация для диаграмм последовательностей пред- назначена и для диаграмм коопераций, однако существуют некоторые отличия.

1. Путь – для описания связи одного объекта с другим к дальней конце- вой точке этой связи можно присоединить стереотип пути:

* association – соответствующий объект виден на ассоциации;
* self – объект является диспетчером операции;
* global – объект находится в глобальной области действия;
* local – объект находится в локальной области действия;
* parameter – объект является параметром.

1. Порядковый номер сообщения – для обозначения временнόй последо- вательности перед сообщением ставится номер (нумерация начинается с еди- ницы), который должен постепенно возрастать для каждого нового сообщения (2, 3 и т. д.). Для обозначения вложенности используется десятичная нотация Дьюи (1 – первое сообщение; 1.1 – первое сообщение, вложенное в сообщение 1; 1.2 – второе сообщение, вложенное в сообщение 1 и т. д.). Уровень вложен- ности не ограничен. Для каждой связи можно показать несколько сообщений (вероятно, посылаемых разными отправителями), каждое из них должно иметь уникальный порядковый номер.

Как правило, диаграммы коопераций и последовательности служат одной и той же цели и представляют одну и ту же информацию, но с различных точек зрения. Поэтому часто для варианта использования создают диаграммы обоих типов.

## Задание на лабораторную работу

Для прецедентов, созданных при построении диаграммы прецедентов по индивидуальному заданию в лабораторной работе №1, необходимо создать диаграммы последовательности и кооперации. Для каждого прецедента необ- ходимо создать диаграммы, описывающие его основной поток, а также альтер- нативные потоки.

## Контрольные вопросы

1. Перечислите основные виды диаграмм UML.
2. Чем отличаются диаграммы последовательности и коопераций?
3. Что такое сообщение? Какие типы сообщений можно выделить?
4. Поясните понятия «линия жизни» и «фокус управления».

## Лабораторная работа №3 ОПИСАНИЕ ПРОЕКТА ПО

*Цель работы*: расширить описание проекта программного средства сво- его варианта индивидуального задания диаграммами классов, описать поведе- ние экземпляров классов с помощью диаграмм состояний. Выполнить разделе- ние классов проекта на категории согласно их предназначению.

## Описание статической структуры проекта диаграммой классов

Статическая структура проектируемой системы представляется в UML диаграммами классов, а также диаграммами реализаций – диаграммой компо- нентов и размещения. Диаграмма классов определяет структуру проекта, типы классов системы, а также различного рода связи, существующие между ними.

На диаграмме UML класс отображается в виде прямоугольника. Имя класса – текстовая строка. Имя абстрактного класса выделяется *курсивом*. В со- ставном имени класса может присутствовать имя пакета, в который он входит: (PackageName::ClassName).

Атрибут – это именованное свойство класса, включающее описание мно- жества значений, которые могут принимать экземпляры этого свойства. Он представляет собой свойство моделируемой сущности, общее для всех объек- тов данного класса.

Общий формат описания атрибута в UML следующий.

*ExportControl Name : Type = InitialValue*

Свойства атрибутов в Rational Rose устанавливаются в спецификации (Specification) атрибута. Видимость атрибута (в Rational Rose – Export Control) может принимать четыре значения:

– public – атрибут виден и может быть изменён всеми остальными клас-

сами;

* protected – доступен только самому классу и его потомкам;
* private – не виден другим классам;
* implementation – является общим в пределах пакета.

Операции реализуют связанное с классом поведение, иными словами, аб-

стракцию того, что позволено делать с объектом. Формат описания операции следующий.

*ExportControl Name (Argument1 : Type = InitialValue, Argument2: Type = InitialValue): Type*

Можно выделить четыре типа операций:

* операции реализации – реализуют некоторые функции: такие операции соотносятся с сообщениями на диаграммах взаимодействия (каждая операция реализации должна быть легко прослеживаема – операция выводится из сооб-

щения на диаграмме взаимодействия, сообщения исходят из подробного описа- ния потока событий, который составляется на основе варианта использования, который, в свою очередь, отражает требования к системе);



* операции управления – управляют созданием и уничтожением объек- тов (конструкторы и деструкторы);
* операции доступа – реализуют установку и чтение значений атрибутов;
* вспомогательные операции – представляют собой закрытые и защи- щённые операции класса.

Основная нотация UML для описания диаграмм классов приведена на рис. 3.1.



Рис. 3.1. Пример диаграммы классов со связями между ними

Связь представляет собой семантическую взаимосвязь между классами. На диаграммах между классами могут устанавливаться следующие виды свя- зей: ассоциации, зависимости, агрегации, обобщения.

Ассоциация (Association) – это семантическая связь между классами. Ас- социации могут быть двунаправленными (без стрелок) или однонаправленными – Navigable (стрелка с одной стороны). Направление ассоциации определяется по диаграмме взаимодействия – если сообщения на диаграмме отправляются толь- ко одним классом, а принимаются только другим – имеет место однонаправ- ленная ассоциация, в противном случае – двунаправленная.

Зависимость (Dependency) – отражает связь между классами, которая все- гда однонаправлена и показывает, что один класс зависит от определений, сде- ланных в другом классе. Зависимости описывают существующие между клас- сами отношения использования (включая отношения трассировки, уточнения и связывания).

Агрегация (Aggregation) – представляет собой более тесную форму ассо- циации – связь между частью и целым. В дополнение к простой агрегации в UML вводится более сильная её разновидность – композиция. Согласно компо- зиции, объект-часть может принадлежать только единственному объекту- целому, кроме того, жизненный цикл их, как правило, совпадает. Любое удале- ние целого распространяется на его части.

Обобщение (Generalization) определяет связь наследования между двумя классами.

Кроме того, на диаграммах UML связи могут дополняться обозначением множественности, а также именами связей и ролей. Множественность (Multip- licity) показывает, сколько экземпляров класса взаимодействует с помощью данной связи с одним или несколькими экземплярами другого. Индикаторы множественности устанавливаются на обоих концах линии связи.

## Диаграммы состояний

Диаграммы состояний (State Machine Diagrams) отражают все возможные состояния, в которых может находиться конкретный объект, а также процесс смены состояний объекта в результате некоторых событий.

Пример диаграммы состояний программы для оплаты счетов представлен на рис. 3.2. Здесь два специальных состояния – начальное (start) и конечное (stop). Начальное состояние (чёрная точка) соответствует объекту в момент его создания. Очевидно, что на диаграмме состояний может быть только одно на- чальное состояние. Конечное состояние (чёрная точка в кружке) соответствует состоянию объекта непосредственно перед его уничтожением. На диаграмме состояний конечных состояний может быть несколько или не быть таковых вообще.

Процессы, происходящие с объектом в каком-либо состоянии, называются действиями (actions). С каждым состоянием связываются данные пяти типов: дея- тельность, входное действие, выходное действие, событие, история состояния.

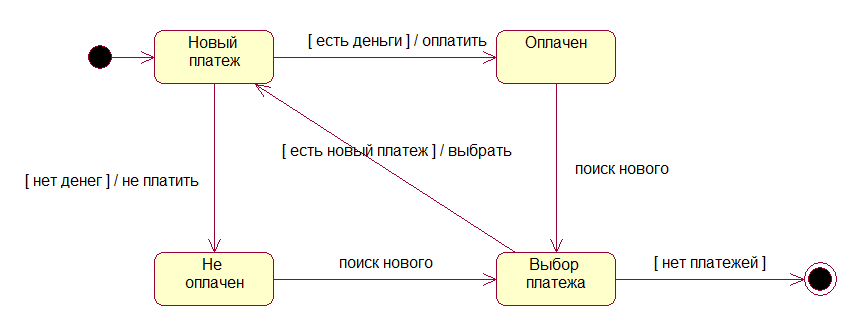


Рис. 3.2. Пример диаграммы состояний

*Деятельность* – поведение, реализуемое объектом в данном состоянии. Это поведение прерываемо и может выполняться либо до своего завершения, либо может быть прервано переходом объекта в другое состояние. Изображает- ся внутри состояния, с предшествующим словом do.

*Входным действием* называется поведение, которое выполняется, когда объект переходит в данное состояние, т. е. его можно рассматривать как часть перехода. Входное действие не прерывается. Ему предшествует спецификатор типа entry. Выходное действие аналогично входному, но ему предшествует сло- во exit. Если поведение объекта включает отсылку сообщения, описанию дея- тельности предшествует значок «^».

Переход объекта из одного состояния в другое отображается на диаграм- ме состояний в виде стрелки. Переходы могут быть рефлексивными. Специфи- кация перехода может включать события, аргументы, ограждающие условия, действия, посылаемые события.

*Событие* (Event) – это то, что вызывает переход. На диаграмме событие размещается вдоль линии перехода. Кроме того, у события могут быть аргу- менты (Arguments). Для отображения события можно использовать как имя операции, так и обычную фразу. На диаграммах состояний могут быть также автоматические переходы, не имеющие событий.

Ограждающие условия (Guard Conditions) определяют возможность пере- хода. На диаграмме они размещаются вдоль линии перехода после события в квадратных скобках. Ограждающие условия необязательны, однако, если суще- ствует несколько автоматических переходов из состояния, необходимо опреде- лить взаимоисключающие ограждающие условия.

При переходе также может указываться действие – вдоль линии перехода, после значка «/». Если действие представляет собой событие, посылаемое дру- гому объекту, на диаграмме ему предшествует значок «^».

*История состояния* указывает, как изменяется поведение объекта в зави- симости от того, сколько раз он находился в этом конкретном состоянии.

Как правило, диаграммы состояний создаются не для всех классов, а только в тех случаях, когда объект класса может существовать в нескольких состояниях и в каждом их них ведёт себя по-разному.

## Структуризация классов программного средства

Структуризация элементов диаграмм выполняется с помощью пакетов. В целом механизм пакетов (Packages) применим к любым элементам модели UML. В отношении классов пакеты применяются для того, чтобы сгруппиро- вать классы, обладающие определённой общностью. Наиболее распространён- ные подходы к группировке:

* по стереотипу – такой подход полезен с точки зрения размещения ком- понентов системы, в этом случае выделяют пакеты с классами-сущностями (Entities), граничными классами (Boundaries) и управляющими классами (Con- trols);
* по функциональности – такой подход даёт преимущество повторного использования, например, в пакете Security могут содержаться все классы, от- вечающие за безопасность приложения, в пакете Error Handling – классы, отве- чающие за обработку ошибок и т. д.

Диаграмма пакетов в UML (рис. 3.3) представляет собой диаграмму, со- держащую пакеты классов и зависимости между ними.



Рис. 3.3. Пример группировки классов в пакеты

Классы анализа группируются в пакеты с учетом следующих правил:

* проектирование граничных классов зависит от возможностей среды разработки пользовательского интерфейса;
* проектирование классов-сущностей производится с учётом соображе- ний производительности (выделение в отдельные классы атрибутов с различ- ной частотой использования);
* проектирование управляющих классов должно способствовать отдале- нию классов, реализующих простую передачу информации от граничных клас- сов к сущностям;
* устойчивые классы, содержащие хранимую информацию, необходимо также идентифицировать.

## Задание на лабораторную работу

В соответствии с индивидуальным заданием лабораторной работы №1 необходимо создать диаграмму классов проектируемой системы. Для классов, обладающих сложным поведением, необходимо построить диаграмму состоя- ний. Созданную диаграмму классов необходимо сгруппировать в диаграмму пакетов.

## Контрольные вопросы

* 1. Какие основные подходы к формированию пакетов вы можете назвать?
  2. Назовите основные элементы диаграммы классов UML. Каков общий формат описания атрибута и операции?
  3. Какие виды связей между классами могут устанавливаться? Поясните  смысл и обозначение каждого типа связей.
  4. Какие основные элементы используются при создании диаграмм состояний UML?

## Лабораторная работа №4 РЕИНЖИНИРИНГ

*Цель работы*: изучить способы представления структуры проекта про- граммного обеспечения в виде диаграмм компонентов и диаграмм развертыва- ния. Сгенерировать шаблон программного кода приложения по результатам выполнения индивидуального задания. Изучить средства обратного проектиро- вания системы Rational Rose.

## Диаграммы компонентов

Диаграмма компонентов описывает особенности физического представ- ления системы. Она позволяет определить архитектуру разрабатываемой сис- темы, установив зависимости между программными компонентами, в роли ко- торых могут выступать исходный и исполняемый код. Основными графически- ми элементами диаграммы компонентов являются компоненты, интерфейсы и зависимости.

Диаграмма компонентов разрабатывается для следующих целей:

* визуализации общей структуры исходного кода программной системы;
* спецификации исполняемого варианта программной системы;
* обеспечения многократного использования фрагментов кода;
* представления концептуальной и физической схем баз данных.

Для представления физических сущностей в языке UML применяется специальный термин – компонент (Сomponent). Компонент реализует некото- рый набор интерфейсов и служит для общего обозначения элементов физиче- ского представления модели. Для графического представления компонента ис- пользуется специальный символ (рис. 4.1).

|  |  |  |
| --- | --- | --- |
|  | Component | |
|  |  |  |
|  |  | |
|  |  |  |
|  |  | |

Рис. 4.1. Изображение компонента

В качестве имен компонентов принято использовать имена исполняемых файлов (с указанием расширения .ехе), динамических библиотек (.dll), Web-страниц (.html), текстовых файлов (.txt или .doc) или файлов справки (.hlp), файлов баз данных (.db) или файлов с исходными текстами программ (.h, .cpp для языка C++, .java для языка Java), скрипты (.pl, .asp) и др.

Способом спецификации различных видов компонентов является явное указание стереотипа компонента перед именем. В языке UML для компонентов определены следующие стереотипы:

* библиотека (library) – компонент, который представляется в форме ди- намической или статической библиотеки;
* таблица (table) – компонент, который представляется в форме таблицы базы данных;
* файл (file) – компонент, который представляется в виде файлов с ис- ходными текстами программ;
* документ (document) – компонент, который представляется в форме до- кумента;
* исполнимый (executable) – компонент, который может исполняться в

узле.

Интерфейс компонента графически изображается окружностью, которая

соединяется с компонентом отрезком линии без стрелок. Имя интерфейса должно начинаться с заглавной буквы «I» и записываться рядом с окружно- стью. Линия означает реализацию интерфейса, а наличие интерфейсов у ком- понента означает, что данный компонент реализует соответствующий набор интерфейсов. Другим способом представления интерфейса на диаграмме ком- понентов является его изображение в виде прямоугольника класса со стереоти- пом «interface» и возможными секциями атрибутов и операций. Как правило, этот вариант обозначения используется для представления внутренней структу- ры интерфейса, которая может быть важна для реализации.

Зависимость служит для представления факта наличия связи между эле- ментами диаграммы, когда изменение одного элемента модели оказывает влия- ние или приводит к изменению другого элемента модели. Отношение зависи- мости на диаграмме компонентов изображается пунктирной линией со стрел- кой, направленной от клиента (зависимого элемента) к источнику (независимо- му элементу).

Зависимости могут отражать связи модулей программы на этапе компи- ляции и генерации объектного кода. Применительно к диаграмме компонентов зависимости могут связывать компоненты и импортируемые этим компонентом интерфейсы, а также различные виды компонентов между собой.

Внутри символа компонента могут изображаться другие элементы графи- ческой нотации, такие как классы (компонент уровня типа) или объекты (ком- понент уровня экземпляра). В этом случае символ компонента изображается та- ким образом, чтобы вместить эти дополнительные символы.

Объекты, которые находятся в отдельном компоненте-экземпляре, изо- бражаются вложенными в символ данного компонента. Подобная вложенность означает, что выполнение компонента влечет выполнение соответствующих объектов.

## Диаграммы развертывания

Диаграммы развертывания предназначены для представления общей конфигурации и топологии распределенной программной системы. Диаграмма развертывания служит средством визуализации элементов и компонентов про- граммы, существующих лишь на этапе ее исполнения (runtime). При этом пред- ставляются только компоненты-экземпляры программы, являющиеся испол- няемыми файлами или динамическими библиотеками, а компоненты, которые

не используются на этапе исполнения, например, тексты программ, на диа- грамме развертывания не показываются.

Диаграмма развертывания содержит графические изображения процессо- ров, устройств, процессов и связей между ними. В отличие от диаграмм логиче- ского представления, диаграмма развертывания является единой для системы в целом, поскольку должна всецело отражать особенности ее реализации. Разра- ботка диаграммы развертывания, как правило, является последним этапом спе- цификации модели программной системы.

При разработке диаграммы развертывания преследуют следующие цели:

* определить распределение компонентов системы по физическим узлам;
* показать физические связи между всеми узлами реализации системы на этапе ее исполнения;
* выявить узкие места системы и реконфигурировать ее топологию для достижения требуемой производительности.

Узел вычислительной системы (node) представляет собой некоторый фи- зически существующий элемент системы, обладающий определенным вычис- лительным ресурсом. Понятие узла может включать в себя не только вычисли- тельные устройства, но и другие механические или электронные устройства, такие как датчики, принтеры, модемы, цифровые камеры, сканеры и манипуля- торы. Графически на диаграмме развертывания узел изображается в форме трехмерного куба.

Кроме изображений узлов на диаграмме развертывания указываются от- ношения между ними. В качестве отношений выступают физические соедине- ния между узлами и зависимости между узлами и компонентами, изображения которых тоже могут присутствовать на диаграммах развертывания.

Пример совмещенной диаграммы компонентов и развертывания приведен на рис. 4.2.



Рис. 4.2. Совмещенная диаграмма компонентов и диаграмма развертывания

## Средства генерации программного кода

Системы поддержки проектирования ПО являются удобным средством генерации исходного кода, объем которого напрямую зависит от полноты опи- сания модели. Чем больше информации разработчик внесет в модель на этапе проектирования, тем эффективнее созданный программный код будет реализо- вать прототип будущей программы.

Процесс генерации кода включает следующие шаги.

1. Проверка корректности модели.
2. Установка свойств генерации кода
3. Выбор класса, компонента или объекта.
4. Генерация исходного кода.

Процесс проверки модели выполняется с помощью команды меню Tools → Check Model. В процессе проверки все отмеченные ошибки выводятся в окне журнала. Наиболее распространёнными ошибками бывают сообщения на диаграммах последовательностей или коопераций, не соотнесённые с каким- либо классом.

Процесс генерации кода полностью настраивается с помощью опций (ме- ню Tools → Options), устанавливаемых для отдельных классов, атрибутов, ком- понентов и прочих элементов модели. Опции определяют правила генерации кода, отдельные элементы, включаемые в состав программных модулей (на- пример, конструкторы по умолчанию), другие свойства кода. Каждый поддер- живаемый язык программирования содержит кроме общих опций свои собст- венные, учитывающие синтаксис и стандарты данного языка.

Процесс генерации кода запускается с помощью соответствующих ко- манд меню Tools для выбранного языка программирования. После генерации кода его можно просмотреть и экспортировать в среду программирования для завершения его разработки, наполнения функциональностью и компиляции.

В процессе генерации кода создаются следующие элементы:

* классы – все классы, представленные в диаграммах модели;
* атрибуты – атрибуты всех классов с учетом типов данных, модификато- ров доступа, значений по умолчанию и операций установки/чтения значения атрибута;
* сигнатуры методов класса с учетом синтаксиса, параметров и возвра- щаемого типа данных;
* отношения между классами – наследование, агрегация, композиция в соответствующем синтаксической представлении;
* документация, созданная в процессе описания и разработки модели и переносимая в код по правилам его документирования.

## Средства обратного проектирования

Средства обратного проектирования предназначены для получения ин- формации из исходного кода программного проекта с целью создания или до- полнения модели приложения на языке UML. Средства обратного проектиро- вания, как и средства генерации кода, существенно зависят от языка, с которым

они работают. Сочетание средств прямого и обратного проектирования позво- ляет разработчику более эффективно вести работу над проектом, своевременно учитывать изменения, вносимые в модель или программный код в процессе уточнения пользовательских требований.

В процессе обратного проектирования система Rational Rose собирает информацию о классах, компонентах, атрибутах, методах, пакетах и отношени- ях между классами.

Для выполнения обратного проектирования и анализа существующего кода необходимо создать новый компонент на диаграмме компонентов проекта. Затем с помощью окна спецификации данного компонента следует добавить в область файлов проекта (Project Files) существующие файлы с исходным кодом. Для этого предназначена кнопка добавления файлов (Add Files). Указанные файлы будут подвергнуты анализу и по их содержимому будут построены представления соответствующих классов. Для компонента следует установить признак того, что он используется в процедуре обратного проектирования. С помощью команды меню Tools для соответствующего языка программирования запускается процесс обратного проектирования (Reverse Engineering).

## Задание на лабораторную работу

По индивидуальному заданию к лабораторной работе №1 необходимо создать диаграмму компонентов проектируемой системы. На основе разрабо- танной диаграммы компонентов следует спроектировать аппаратно- операционную платформу для работы приложения и описать ее диаграммой развертывания. Сгенерировать код на любом доступном в Rational Rose языке программирования. Реализовать некоторые наиболее наглядные методы и про- демонстрировать работу модели проектируемой системы. Внести изменения в структуру программного кода и выполнить процедуру обратного проектирова- ния для модели.

## Контрольные вопросы

1. Назовите основные элементы нотации и их назначение в диаграмме развертывания.
2. Для каких целей разрабатывается диаграмма компонентов?
3. Какие элементы языка программирования используются при генерации кода и обратном проектировании?
4. Перечислите этапы выполнения процедуры обратного проектирования.

## Лабораторная работа №5 МНОГООКОННЫЙ ИНТЕРФЕЙС

*Цель работы*: изучить структуру и основные элементы многооконного интерфейса, архитектуру приложений для графического интерфейса пользова- теля, средства разработки приложений. Создать базовое приложение много- оконного интерфейса со средствами ввода пользовательской информации через диалоговое окно и выводом ее в графическом представлении.

## Основные компоненты многооконного интерфейса

Графический интерфейс пользователя предполагает использование изо- бражений различных объектов для представления входных и выходных данных программы. Программа с графическим интерфейсом всегда работает в рамках определенной оконной подсистемы. Программа выводит определенные кнопки, пиктограммы, диалоговые окна в своем окне, а пользователь управляет ею пу- тем перемещения указателя по окну и выбора объектов управления.

В дизайне пользовательского интерфейса можно условно выделить деко- ративную и активную составляющие. К первой относятся элементы, отвечаю- щие за эстетическую привлекательность программного изделия. Активные эле- менты подразделяются на операционные и информационные образы моделей вычислений и управляющие средства пользовательского интерфейса, посредст- вом которых пользователь управляет программой. Управляющие средства раз- личных классов программных изделий могут значительно различаться.

Наиболее распространенным классом пользовательских интерфейсов на сегодняшний момент является графический интерфейс WIMP. Интерфейс WIMP (Window – окно, Icon – пиктограмма, Menu – меню, Pointer – указатель) характеризуется следующими особенностями.

1. Вся работа с программами, файлами и документами происходит в ок- нах – определенных очерченных рамкой частях экрана.
2. Все программы, файлы, документы, устройства и другие объекты пред- ставляются в виде значков – пиктограмм. При открытии пиктограммы превра- щаются в окна.
3. Все действия с объектами осуществляются с помощью меню. Хотя меню появилось на первом этапе становления графического интерфейса, оно не име- ло в нем главенствующего значения, а служило лишь дополнением к командной строке. В чистом WIMP – интерфейсе меню становится основным элементом управления.
4. Широко используются манипуляторы для указания на объекты. Мани- пулятор становится основным элементом управления. С помощью манипулято- ра УКАЗЫВАЮТ на любую область экрана, окна или иконки, ВЫДЕЛЯЮТ ее, а уже потом через меню или с использованием других технологий осуществля- ют управление ими.

Основными компонентами графического интерфейса типа WIMP являются:

* + *менеджер окон*, который управляет выводом окна на экран, принимает ввод с клавиатуры, мыши и других устройств, и передает пользовательские со- общения приложениям;
  + *интерфейс графического устройства (GDI),* библиотека программных интерфейсов приложения для графических устройств вывода. Включает функ- ции для вывода линий, текста, фигур и геометрических преобразований. Ме- неджер окон вызывает эти программные интерфейсы приложения, но приложе- ния пользователя могут вызвать их самостоятельно;
  + *драйверы графических устройств*, которые собирают и форматируют данные для аппаратно-зависимых драйверов. Эти драйверы вызывают драйверы ядра, которые взаимодействуют с аппаратными средствами ЭВМ;
  + *уровень абстракции аппаратуры* содержит специальные функции для приложений, например, функции создания и удаления процессов, а также обес- печивает связь логических графических устройств с физическими.

Общая архитектура графического интерфейса приведена на рис. 5.1.



Рис. 5.1. Архитектура и компоненты графического интерфейса

## Архитектура приложений для графического интерфейса

Общая архитектура приложения для многооконного графического интер- фейса представляет собой цикл обработки сообщений о событиях, происходя- щих во внешней информационной среде приложения. Событие – это действие, которое программист может обработать в программном коде, обеспечивая ре- акцию на него. События могут быть сгенерированы действиями пользователя, такими как щелчок мыши или нажатие клавиши, кодом программы или дейст- виями системы.

Управляемые событиями приложения выполняют код в процессе реакции на событие. Каждая форма и элемент управления содержат предопределенный набор событий, обработку которых можно программировать. Если происходит

одно из этих событий и существует код связанного с ним обработчика событий, этот код будет вызван.

Типы событий, посылаемых объектом, различны, но существует много ти- пов событий, которые присущи большинству элементов управления. Например, большинство объектов обработает событие нажатия кнопки мыши в окне эле- мента управления. Если пользователь нажимает кнопку мыши на форме, вы- полняется код в обработчике событий нажатия формы.

Многие события происходят одновременно с другими событиями. На- пример, в процессе появления события двойного щелчка происходят события нажатия кнопки мыши, её отпускания и щелчка мыши.

## Создание базового приложения для графического интерфейса, обработка событий

Разработку базового приложения для графического интерфейса Windows проиллюстрируем на примере работы со средой проектирования Visual Studio. В качестве компонентов, используемых для разработки интерфейса приложе- ния, будет рассмотрена библиотека компонентов Windows Forms.

Для создания скелета приложения удобно использовать мастер, встроен- ный в среду Visual Studio. Для этого следует запустить среду и выбрать в меню последовательность действий Файл (File) → Новый (New) → Проект (Project) → Язык Visual C++ (Visual C++) → Общая среда выполнения (CLR). После этого мастер предложит выбрать тип приложения, например, приложе- ние на основе библиотеки компонентов Windows Forms (Windows Forms Appli- cation). В поле Имя (Name) необходимо ввести имя проекта (например, Test) для создания необходимой инфраструктуры исходных файлов и других элемен- тов проекта. Откроется окно проекта Test с вкладкой Form1.h. В этом окне ото- бразится дизайн основного окна приложения (рис. 5.2). На данной форме мож- но размещать элементы управления, используемые для передачи команд поль- зователя приложению. Кроме того, как и любой элемент управления, форма са- ма по себе способна реагировать на пользовательский ввод, обрабатывая сооб- щения о событиях.

Для создания кода обработки сообщения о событиях, происходящих для конкретного элемента управления, следует выбрать его мышью и нажать пра- вую кнопку мыши. Выполнив эти действия для формы, получим доступ к всплывающему меню, в котором следует выбрать пункт Свойства (Properties). В этом окне можно изменять свойства самого элемента управления, а также до- бавлять код для обработки сообщения о событии. В качестве примера создания обработчика событий рассмотрим событие активации формы. Для создания за- готовки соответствующего обработчика следует выбрать в окне свойств собы- тие Активация (Activated) и выполнить двойной щелчок по нему левой кнопкой мыши (рис. 5.3).

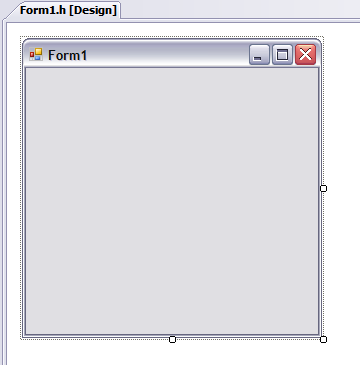


Рис. 5.2. Основное окно приложения в дизайнере интерфейса

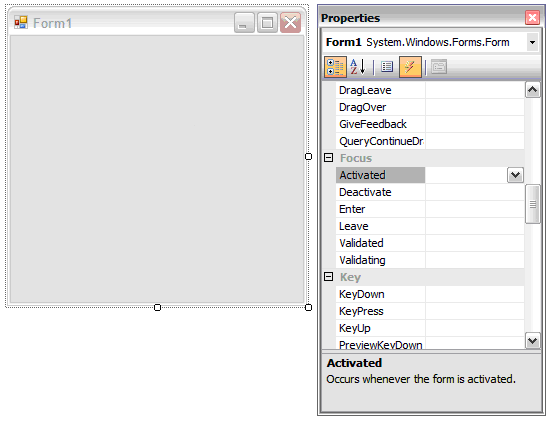


Рис. 5.3. Окна приложения и свойств формы в дизайнере интерфейса

В окне редактора кода появится заготовка обработчика сообщения о со- бытии активации формы.

**private: System::Void Form1\_Activated(System::Object^ sender, System::EventArgs^ e) {**

**}**

В качестве обработчика вставим между фигурными скобками следующий

код.

This->Text = "Form1 - Activated";

Затем следует повторить действия для события деактивации формы

(Deactivated). В полученную заготовку обработчика между фигурными скобка- ми вносится код.

## This->Text = "Form1 - Deactivated";

Для компиляции и запуска приложений необходимо нажать клавишу F5. При появлении формы на экране следует переключить фокус на нее, а затем на другое приложение, чтобы убедиться, что заголовок формы меняет содержание.

## Создание меню

Альтернативным способом управления графическим приложением явля- ется работа с меню. Для создания меню приложения необходимо добавить к форме элемент управления Полоса меню (MenuStrip), расположенный на пане- ли инструментов дизайнера интерфейса. Элемент управления MenuStrip при- соединится к верхней части формы.

Затем следует выбрать элемент управления MenuStrip и ввести пункты меню, начав, например, с пункта Файл (File). После ввода названия пункта ме- ню необходимо нажать клавишу Ввод. Новые поля для ввода дополнительных элементов меню отобразятся снизу и справа от первого элемента меню. В них есть место для ввода названий дополнительных пунктов меню. Можно продол- жить добавлять элементы меню в любом направлении до тех пор, пока меню не будет готово. В поле под полем File следует ввести пункт меню Выход (Exit) и нажать клавишу Ввод.

Чтобы создать обработчик пункта меню, достаточно выполнить двойной щелчок мышью на соответствующем пункте в окне дизайнера интерфейса. Вы- полнив данное действие для пункта Выход (Exit), получим в окне редактора ко- да заготовку обработчика событий. Внутри обработчика событий ExitToolStripMenuItem\_Click необходимо добавить следующий код.

this->Close();

Для компиляции и запуска приложений необходимо нажать клавишу F5. При появлении формы на экране следует выбрать пункт меню Exit и убедиться, что приложение завершается.

Изучите остальные файлы, содержащие код приложения, созданные масте- ром среды Visual Studio.

## Задание на лабораторную работу

По индивидуальному заданию создать прототип графического интерфей- са проектируемой системы. На основе разработанных диаграмм прецедентов разработать дизайн интерфейса, ввести необходимые обработчики событий элементов управления, создать меню приложения. В качестве обработчиков можно ввести как простые заглушки, сообщающие об активации соответст- вующего обработчика, так и имитаторы действий, отображающие активацию действий средствами интерфейса.

## Контрольные вопросы

1. Назовите основные элементы графического интерфейса.
2. Что такое обработчик сообщения и для чего он нужен?
3. Какой элемент архитектуры приложения для графического интерфейса реализует обработку событий?
4. Перечислите этапы создания приложения для графического интерфейса.

## Лабораторная работа №6

**АРХИТЕКТУРА «МОДЕЛЬ–ПРЕДСТАВЛЕНИЕ–КОНТРОЛЛЕР»**

*Цель работы*: изучить возможности организации взаимодействия между компонентами архитектуры «Модель-представление-контроллер». Разработать приложение в соответствии с данной архитектурой.

## Общие сведения об архитектуре

**«Модель–представление–контроллер»**

«Модель–представление–контроллер» (Model–view–controller, далее MVC) – архитектура программного обеспечения, в которой модель данных приложения, пользовательский интерфейс и управляющая логика разделены на три отдельных компонента так, что модификация одного из компонентов ока- зывает минимальное воздействие на другие компоненты.

MVC позволяет разделить данные, представление и обработку действий пользователя на три отдельных компонента:

1. Модель – предоставляет данные и позволяет изменять эти данные, а также, возможно, отслеживать их изменение.
2. Представление – отвечает за отображение информации, содержащейся в модели.
3. Контроллер – интерпретирует действия, совершённые пользователем, изменяет некоторым образом модель и информирует представление о необхо- димости соответствующей реакции.

Важно отметить, что как представление, так и поведение зависят от моде- ли. Однако модель не зависит ни от представления, ни от поведения. Это одно из ключевых достоинств подобного разделения. Оно позволяет строить модель

независимо от визуального представления, а также создавать несколько раз- личных представлений для одной модели.

Впервые данный шаблон проектирования был предложен для языка Smalltalk.

## Обобщённое логическое представление архитектуры MVC на UML

В обобщённом логическом представлении архитектуры MVC выделяется три набора компонентов. Первый набор – данные, модель данных или просто модель (model) – соответствует структуре данных предметной области, в кото- рой работает приложение. Обязанности этих компонентов – представлять в системе данные и базовые операции над ними. Компоненты второго набора – представления (view) – соответствуют различным способам представления дан- ных в пользовательском интерфейсе. Для одних и тех же данных может суще- ствовать несколько представлений. Каждому компоненту представления соот- ветствует один компонент из третьего набора, контроллер (controller) – компо- нент, осуществляющий обработку действий пользователя. Такой компонент по- лучает команды, чаще всего – нажатия клавиш и нажатия кнопок мыши в об- ластях, соответствующих визуальным элементам управления – кнопкам, эле- ментам меню и пр. Эти команды он преобразует в действия над данными. В ре- зультате каждого действия требуется обновить все представления всех данных, которые подверглись изменениям (рис. 6.1).



*Subscriber*

*update()*

View

update()

Model

Data

attach(Subscriber) detach(Subscriber) notify()

Controller

handleEvent()

+model

+view

+model

Рис. 6.1. Обобщённая диаграмма классов MVC

На рис. 6.2 изображен сценарий обработки действия пользователя. Взаи- модействие инициируется действием пользователя (action) – нажатие клавиши, кнопки мыши и т. д. Контроллер обрабатывает событие, связанное с действием пользователя (метод handleEvent()) и модифицирует данные в модели (change data). Модель в свою очередь генерирует событие (notify()), которое перехватывается всеми представлениями (update()) и они отображают изме- нённые данные на экране (display).

Возможен вариант, когда событие изменения модели обрабатывается контроллером. В этом случае контроллер отвечает за изменение представления.



Рис. 6.2. Сценарий обработки действия пользователя

## Пример использования шаблона «Модель–представление– контроллер» для реализации простого элемента управления

Рассмотрим реализацию элемента управления «кнопка» с применением шаблона «модель–представление–управление».

В качестве компонента «модель» в элементе управления «кнопка» высту- пает класс ButtonModel с двумя свойствами Text (текст, отображаемый на кнопке) и Pressed (флаг, показывающий нажата кнопка или нет). Класс так- же генерирует события при изменении этих свойств.

Исходный текст класса ButtonModel:

public

gc class ButtonModel {

private:

String

gc \* pText;

bool pressed; public:

ButtonModel(): pText(S""), pressed(false){ }

event EventHandler\* OnTextChanged;

event EventHandler\* OnPressedStatusChanged;

property String

gc \* get\_Text() {return pText;}

property void set\_Text(String

if (pText){

gc \* pText){

if (pText != this->pText &&

pText->Equals(this->pText)){ this->pText = pText;

this->raise\_OnTextChanged(this,

new Even- tArgs());}}}

property bool get\_Pressed() {return pressed; }

property void set\_Pressed(bool pressed){ if (pressed != this->pressed){

this->pressed = pressed;

this->raise\_OnPressedStatusChanged(this,

new Even- tArgs());}}}

Компонент «представление» является стандартным элементом управле- ния WinForms, унаследованным от класса UserControl из пространства имён System.Windows.Forms.

Представление отображает свойство Text модели с помощью стандарт- ного элемента Label, а состояние Pressed – за счёт изменения цвета фона.

Ниже приводится метод updateView(), который вызывается при изме- нении свойств модели (а также при начальной инициализации) и соответст- вующим образом обновляет представление.

void updateView()

{

using System::Drawing::Color; pLabel->Text = pModel->Text;

BackColor = pModel->Pressed ? Color::Gray :

Color::LightGray; pLabel->BackColor = BackColor;

}

Контроллер подписывается на событие нажатия и отпускания кнопки мыши у представления и соответствующим образом меняет свойство Pressed у модели.

public

{

\_\_gc class ButtonController

private:

ButtonView

\_\_gc \* pView;

ButtonModel

gc \* pModel;

ButtonController() {} public:

ButtonController(ButtonView \* pView\_): pModel(pView\_->Model), pView(pView\_) {

pView->add\_MouseDown(new MouseEventHandler(this,

&ButtonController::MousePressed)); pView->add\_MouseUp(new MouseEventHandler(this,

&ButtonController::MouseReleased))

;

}

private:

void MousePressed(Object

{

\_\_gc \* sender, MouseEventArgs

\_\_gc \*)

pModel->Pressed = true;

}

void MouseReleased(Object

{

gc \*, MouseEventArgs

\_\_gc \*)

pModel->Pressed = false;

}

};

Пример создания элемента управления и помещения его на форму приво- дится ниже.

Form1(void)

{

InitializeComponent();

MVC::ButtonModel MVC::ButtonView

gc \* pModel =

new MVC::ButtonModel(); gc \* pView =

new MVC::ButtonView(pModel);

MVC::ButtonController

gc \* pController =

new MVC::ButtonController(pView);

pView->Location = System::Drawing::Point(20, 20); pView->Size = System::Drawing::Size(72, 24); pModel->Text = S"Button";

this->Controls->Add(pView);

}

## Задание на лабораторную работу

В соответствии с индивидуальным заданием разработать несколько эле- ментов управления с архитектурой «модель–представление–контроллер» (дере- во, список, таблица и т. д.). Создать приложение, демонстрирующее их работо- способность.

## Контрольные вопросы

1. Что представляет собой архитектура «модель–представление– контроллер»?
2. Каковы функции компонентов архитектуры «модель–представление– контроллер»?
3. В чём заключаются достоинства и недостатки данного шаблона?
4. Приведите пример разложения функций простого пользовательского интерфейса на компоненты архитектуры «модель–представление–контроллер».

## Лабораторная работа №7 ДИАЛОГОВЫЕ ОКНА

*Цель работы*: изучить возможности по созданию диалоговых окон по шаблонам пользователя для принятия исходных данных в приложении. Озна- комиться с возможностями интернационализации интерфейса приложения для работы в многоязычном операционном окружении. Создать диалоговое окно для ввода информации в приложение, встроить средства интернационализации приложения для русского и английского языков.

## Назначение и виды диалоговых окон

Диалоговое окно – это разновидность окна приложения, которое содер- жит собственные элементы управления и предназначено для ввода пользова- тельских данных, представления результатов работы приложения или управле- ния приложением.

Существует три основных режима работы диалогового окна:

1. модальный диалог – должен быть обязательно обслужен пользователем (нажата кнопка закрытия диалога), прежде чем приложение сможет продолжить свою работу. Как правило, используется для интерактивного управления при- ложением. Хотя пользователь и не может работать с приложением, пока не за- крыто модальное окно, он может работать с другими приложениями в рамках графического интерфейса;
2. системный модальный диалог – не позволяет продолжить работу ни одному приложению, пока не будет обслужен или закрыт. Этот вид окна ис- пользуется для вывода информации, критичной для продолжения устойчивой работы системы (например, диалог по Ctrl-Alt-Delete);
3. немодальный диалог – позволяет пользователю продолжить работу в приложении без необходимости закрытия диалога. Может использоваться как

для управления приложением, так и для вывода информации о ходе вычисле- ний или о промежуточных результатах работы программы.

Диалоговые окна в основном создаются программистом с учетом назна- чения и интерфейса его приложения. Однако существуют стандартные диало- говые окна, шаблон которых задан в интерфейсе и которые одинаково выглядят во всех приложениях. Эти диалоги используются для управления настройками приложения, которые зависят от системы, например, цветовой гаммой, шриф- тами, а также для выполнения стандартных операций (открытие или сохране- ние файла).

## Создание диалогового окна

Пользовательские диалоговые окна создаются так же, как и окна прило- жения – в визуальном редакторе форм. Для создания диалогового окна следует создать новую форму способом, аналогичным созданию базового окна прило- жения из лабораторной работы №5. Рассмотрим диалоговое окно для измене- ния размеров основной формы приложения. Общий шаблон окна и элементы управления диалогом приведены на рис. 7.1.

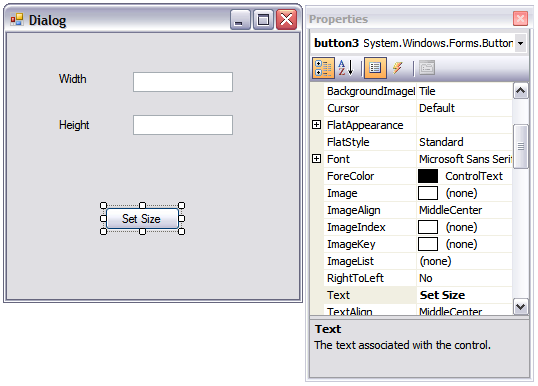


Рис. 7.1. Пример формы диалогового окна

Для хранения и извлечения информации из диалогового окна следует ввести в код формы диалога переменные, связывающие элементы управления и

поля для хранения данных. В нашем случае в код формы вводятся две перемен- ные Width и Height.

При нажатии кнопки Set Size в окне диалога нам необходимо передать информацию из полей ввода в переменные диалога, чтобы затем использовать их в основной программе для управления. Для этого следует переопределить код обработчика нажатия кнопки в окне диалога следующим образом:

**private: System::Void button3\_Click(System::Object^ sender, System::EventArgs^ e)**

**{**

**Width = System::Convert::ToInt32(this->textBox1->Text->); Height = System::Convert::ToInt32(this->textBox2->Text->); this->DialogResult =**

**System::Windows::Forms::DialogResult::OK;**

**}**

Как видно из приведенного фрагмента, вначале в поля Width и Height со-

храняются данные полей ввода из окна диалога, а затем устанавливается ре- зультат выполнения диалога. В данном случае он эквивалентен **DialogResult::OK,** что сигнализирует основному приложению об успешном завершении работы диалогового окна.

Создайте в меню приложения метод вызова диалогового окна, например, пункт Dialog. Код работы этого метода выглядит следующим образом:

**private: System::Void dialogToolStripMenuItem\_Click(System::Object^ sender, System::EventArgs^ e)**

**{**

**Test::Dialog^ dlg = gcnew Test::Dialog; if (dlg->ShowDialog() == System::Windows::Forms::DialogResult::OK)**

**{**

**this->Height = dlg->Height; this->Width = dlg->Width;**

**}**

**}**

Вначале создается экземпляр диалогового окна dlg. Затем вызывается ме- тод **ShowDialog()**, указывающий, что данная форма будет модальным диалогом. Если необходимо вызвать немодальный диалог, то используется обычный метод **Show()**. Далее приложение ожидает завершения диалога и присваивает значения его переменных, введенные пользователем, атрибутам формы. После этого форма автоматически меняет свои размеры на экране.

## Контроль пользовательского ввода

Поскольку диалоговые окна предназначены для приема информации от пользователя, важным моментом является проверка входящей информации на правильность и соответствие форматам, которые предполагает использовать приложение. Ввод данных в неправильной форме может нарушить работу при- ложения или привести к его аварийному завершению.

Для проверки введенных данных используется механизм валидации. Ва- лидация заключается в контроле за значениями элемента пользовательского ин- терфейса до его передачи во внутренние атрибуты диалогового окна. Если кон- троль пройден, данные передаются обычным образом. Если пользователь ошибся при вводе данных, программа может сообщить ему об этом или попы- таться самостоятельно исправить введенные данные.

Валидация информации происходит при получении сообщения о необхо- димости выполнения проверки данных в элементе управления. Каждый элемент управления имеет свойство *CausesValidation,* которое определяет необходи- мость проверки значения в момент, когда данный объект управления теряет фокус. Для того чтобы обеспечить валидацию для объекта управления, следует установить это свойство в значение Истина (True).

Кроме установки свойства, необходимо добавить в окно диалога специ- альный компонент для выдачи пользователю подсказок в процессе валидации. Этот компонент называется ErrorProvider и активируется при обнаружении ошибки ввода пользовательских данных. Его назначение – выдача предупреж- дения пользователю, содержащего подсказку относительно того, в какой форме следует вводить информацию с помощью определенного элемента управления данного диалога.

Каждый элемент управления, способный выполнять валидацию, имеет обработчик события валидации. Рассмотрим пример обработчика для валида- ции поля ввода ширины окна. Предположим, что в это поле можно вводить только целочисленные значения больше 0 и меньше 1000. После ввода данных программа должна преобразовать введенное значение в число, чтобы исклю- чить случай, когда пользователь вводит строки. Затем введенное число прове- ряется на соответствие диапазону значений. Если все значения диалога введены верно, он закроется и выполнит действия, предписанные кодом пользователя.

Обработчик валидации для значения ширины окна будет выглядеть сле- дующим образом:

**private: System::Void textBox1\_Validating(System::Object^ sender, System::ComponentModel::CancelEventArgs^ e)**

**{**

**int value = 0; try**

**{**

**value = System::Convert::ToInt32(textBox1->Text);**

**}**

**catch(...)**

**{**

**e->Cancel = true; errorProvider1->SetError(**

**textBox1,"You must enter a number");**

**}**

**if((value <= 0)||(value > 1000))**

**{**

**e->Cancel = true;**

**errorProvider1->SetError(**

**textBox1,"Value between 0 and 1000");**

**}**

**}**

Если валидация одного из элементов управления не произойдет, пользо-

ватель получит предупреждение об ошибке, как показано на рис. 7.2.

Валидация является удобным средством введения элемента защитного программирования в пользовательский интерфейс. С ее помощью можно ис- ключить ошибки работы программы, связанные с неверной интерпретацией входных данных, а также корректировать информацию в случае, когда она вы- ходит за пределы диапазона возможных значений. Встроенные средства вали- дации облегчают программирование интерактивной обработки ошибок, предос- тавляя пользователю удобный интерфейс.

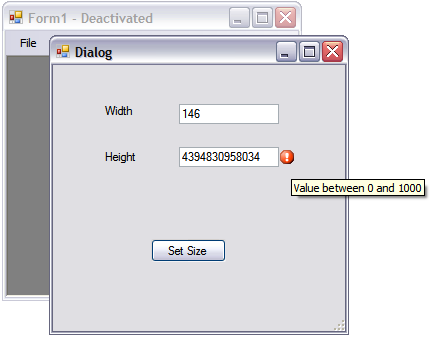


Рис. 7.2. Пример валидации значения элемента управления

## Интернационализация приложений

Интернационализация – это процесс проектирования и разработки про- граммных продуктов, которые работают в средах с различными языками и регио- нальными параметрами. В современном мире разработка программного обеспече- ния для различных групп пользователей, в том числе с учетом критерия языка,

на котором они общаются, позволяет существенно расширить рынок сбыта программного обеспечения.

Процесс создания международных приложений состоит из двух этапов: *глобализации*, которая представляет собой процесс разработки приложений, способных адаптироваться к различным языковым и региональным парамет- рам, и *локализации* – процесса перевода на нужный язык ресурсов для работы в определенных языковых и региональных параметрах.

При выполнении локализованного приложения его внешний вид опреде- ляется двумя значениями – Язык и Региональные параметры (они представляют собой набор сведений о настройках пользователя в таких аспектах, как язык, окружение и обычаи). Языковые и региональные параметры пользовательского интерфейса определяют, какие ресурсы для выполнения интернационализации будут загружаться. Языковые и региональные параметры интерфейса пользова- теля задаются с помощью параметра CurrentUICulture в программном коде. Они определяют формат таких значений, как даты, числа, денежные единицы и пр.

Среда разработки Visual Studio предоставляет значительную поддержку для локализации приложений, работающих с формами Windows Forms. Суще- ствуют два способа создания файлов ресурсов при помощи Visual Studio.

1. Файлы ресурсов могут создаваться системой проектов для локализуе- мых элементов пользовательского интерфейса, таких как текст и изображения в форме. Затем файлы ресурсов будут встраиваться в сопутствующие сборки. Та- кие ресурсы называются ресурсами форм.
2. Добавление шаблона файла ресурсов и затем редактирование шаблона с помощью конструктора XML. Второй способ применяется для создания лока- лизуемых строк в диалоговых окнах и сообщениях об ошибках. Затем необхо- димо написать код для доступа к этим ресурсам. Такие ресурсы называются ре- сурсами проекта.

В общем случае ресурсы форм следует использовать для всех ресурсов, от- носящихся к форме в приложении Windows Forms. Ресурсы проекта следует использовать для всех строк и изображений, которые не основаны на форме, таких как сообщения об ошибках.

При создании локализуемого приложения следует указать для формы свойство Локализация (Localizable). В свойстве Язык (Language) следует вы- брать требуемый язык локализации. Все элементы управления, размещенные в форме, должны иметь соответствующие атрибуты на указанном языке (надпи- си, комментарии и др.). Пример локализованной формы в дизайнере приведен на рис. 7.3.

Меняя язык локализации в свойствах формы и заполняя поля для элемен- тов управления, пользователь создает соответствующие файлы ресурсов, вклю- чаемые в проект. Эти файлы имеют расширение .resx и доступны для редакти- рования при вызове команды меню **Показать все файлы**.

Для изменения интерфейса с использованием глобализации необходимо выполнить код установки культуры, который имеет следующий вид:

**System::Globalization::CultureInfo^ culture =**

**gcnew System::Globalization::CultureInfo("ru-RU"); System::Threading::Thread::CurrentThread->CurrentCulture =**

**culture; apply\_Culture(culture);**

Функция apply\_Culture является пользовательской функцией, которая ус- танавливает значения локализации для каждого элемента управления. В общих чертах она выглядит следующим образом:

**private: System::Void apply\_Culture(System::Globalization::CultureInfo^ culture)**

**{**

**System::ComponentModel::ComponentResourceManager^ resources = (gcnew System::ComponentModel::ComponentResourceManager(Form1::typeid**

**));**

**resources->ApplyResources(this->toolStripButton1, L"toolStripButton1", culture);**

**resources->ApplyResources(this->toolStripButton2, L"toolStripButton2", culture);**

**resources->ApplyResources(this->toolStripButton3, L"toolStripButton3", culture);**

**}**

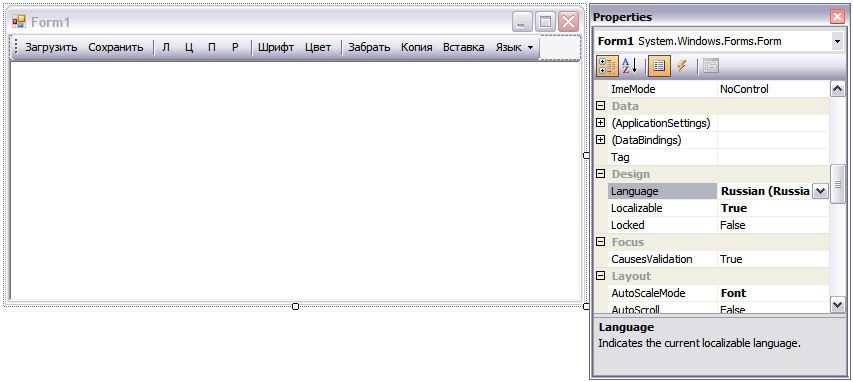


Рис. 7.3. Пример локализованной формы

## Задание на лабораторную работу

42

По индивидуальному заданию создать прототип графического интерфей- са проектируемой системы. Для ввода пользовательских данных разработать дизайн диалоговых окон, реализовать ввод данных в программу с валидацией. Приложение должно быть локализовано как минимум для двух языков, вклю- чая все формы и элементы управления.

## Контрольные вопросы

1. Укажите основные режимы работы диалоговых окон, их отличия и об- ласть применения.
2. Что такое валидация? Как она реализуется в приложениях?
3. Опишите процесс создания международных приложений.
4. Поясните механизм работы многоязычного интерфейса локализованного приложения.

## Литература

43

1. Брауде, Э. Технология разработки программного обеспечения / Э. Брауде. – СПб. : Питер, 2004. – 655 с.
2. Константайн, Л. Разработка программного обеспечения / Л. Констан- тайн. – СПб. : Питер, 2004. – 592 с.
3. Соммервиль, И. Инженерия программного обеспечения / И. Соммер- виль. – 6-е изд. – М. : Вильямс, 2002. – 624 с.
4. Хортон, А. Microsoft Visual C++ 2005: базовый курс / А. Хортон. – М. : Диалектика, 2007. – 1152 с.
5. Торрес, Р. Дж. Практическое руководство по проектированию и разработке пользовательского интерфейса / Дж. Р. Торрес. – М. : Вильямс, 2002. – 400 с.

Св. план 2010, поз. 55

*Учебное издание*

**Отвагин** Алексей Владимирович

**Павлёнок** Наталья Александровна

# ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ И ПРОГРАММИРОВАНИЕ

Лабораторный практикум

для студентов специальности 1-40 02 01

«Вычислительные машины, системы и сети» всех форм обучения

Редактор Г. С. Корбут Корректор Е. Н. Батурчик

Компьютерная верстка М. В. Гуртатовская

|  |  |  |
| --- | --- | --- |
| Подписано в печать 02.12.2010. | Формат 60х84 1/16. | Бумага офсетная. |
| Гарнитура «Таймс». Уч.-изд. л. 2,7. | Отпечатано на ризографе. Тираж 100 экз. | Усл. печ. л. Заказ 305. |

Издатель и полиграфическое исполнение: учреждение образования

«Белорусский государственный университет информатики и радиоэлектроники» ЛИ №02330/0494371 от 16.03.2009. ЛП №02330/0494175 от 03.04.2009.

220013, Минск, П. Бровки, 6

44