МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Кузбасский государственный технический университет имени Т. Ф. Горбачева»

Кафедра информационных и автоматизированных производственных систем

Составитель
И. С. Сыркин

РАЗРАБОТКА КОДА ИНФОРМАЦИОННЫХ СИСТЕМ

Методические материалы

Рекомендовано цикловой методической комиссией специальности
СПО 09.02.07 Информационные системы и программирование
в качестве электронного издания
для использования в образовательном процессе

Кемерово 2018

Рецензенты

Ванеев О. Н. – кандидат технических наук, доцент кафедры информационных и автоматизированных производственных систем ФГБОУ ВО «Кузбасский государственный технический университет имени Т. Ф. Горбачева»

Чичерин И. В. – кандидат технических наук, доцент, зав. кафедрой информационных и автоматизированных производственных систем ФГБОУ ВО «Кузбасский государственный технический университет имени Т. Ф. Горбачева»

Сыркин Илья Сергеевич

Разработка кода информационных систем: методические указания материалы [Электронный ресурс]: для студентов специальности СПО 09.02.07 Информационные системы и программирование очной формы обучения / сост. И. С. Сыркин; КузГТУ. – Электрон. издан. – Кемерово, 2018.

Методические материалы для дисциплины Разработка кода информационных систем содержат перечень выполняемых практических и лабораторных работ, контрольные вопросы к ним.

Предисловие

Целью освоения дисциплины «Разработка кода информационных систем» является приобретение обучающимися знаний в области проведения тестирования ПО в процессе его создания.

Основными задачами изучения дисциплины «Разработка кода информационных систем», являются:

1. Изучение способов моделирования программных продуктов.
2. Написание кода программных средств.
3. Отладка программных средств.

Содержание дисциплины в соответствии с учебным планом

В соответствии с учебным планом изучение дисциплины «Разработка кода информационных систем» предусматривает проведение лекционных, практических занятий, лабораторных работ и самостоятельной работы обучающимися очной формы обучения.

Общая трудоемкость дисциплины составляет 90 часов.

Промежуточный контроль – экзамен (7 семестр).

Содержание практических занятий и лабораторных работ

При подготовке к практическим занятиям обучающиеся самостоятельно изучают основную и дополнительную литературу, готовят конспекты по темам, предложенным преподавателем.

На практических занятиях преподаватель осуществляет контроль подготовки качества знаний обучающегося, используя: опрос, обсуждение вопросов по темам изучаемой дисциплины, письменный опрос при текущем контроле и предоставление отчетов по практическим занятиям.

1. Лабораторная работа №1. Построение диаграммы вариантов использования и генерация кода

Целью работы является изучение порядка построения диаграмм вариантов использования.

Для выполнения лабораторной работы № 1 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Построение диаграммы вариантов использования и диаграммы последовательности

Визуальное моделирование в UML можно представить, как некоторый процесс поуровневого спуска от наиболее обшей и абстрактной концептуальной модели исходной системы к логической, а затем и к физической модели соответствующей программной системы. Для достижения этих целей вначале строится модель в форме так называемой диаграммы вариантов использования (use case diagram), которая описывает функциональное назначение системы или, другими словами, то, что система будет делать в процессе своего функционирования. Диаграмма вариантов использования является исходным концептуальным представлением или концептуальной моделью системы в процессе ее проектирования и разработки.

Разработка диаграммы вариантов использования преследует цели:

- Определить общие границы и контекст моделируемой предметной области на начальных этапах проектирования системы.
- Сформулировать общие требования к функциональному поведению проектируемой системы.
- Разработать исходную концептуальную модель системы для ее последующей детализации в форме логических и физических моделей.
- Подготовить исходную документацию для взаимодействия разработчиков системы с ее заказчиками и пользователями.

Суть данной диаграммы состоит в следующем: проектируемая система представляется в виде множества сущностей или актеров, взаимодействующих с системой с помощью так называемых вариантов использования. При этом актером (actor) или действующим лицом называется любая сущность, взаимодействующая с системой извне. Это может быть человек, техническое устройство, программа или любая другая система, которая может служить источником воздействия на моделируемую систему так, как определит сам разработчик. В свою очередь, вариант использования (use case) служит для описания сервисов, которые система предоставляет актеру. Другими словами, каждый вариант использования определяет некоторый набор действий, совершаемый системой при диалоге с акте-

ром. При этом ничего не говорится о том, каким образом будет реализовано взаимодействие актеров с системой.

Примечание

Рассматривая диаграмму вариантов использования в качестве модели системы, можно ассоциировать ее с моделью «черного ящика». Действительно, подробная детализация данной диаграммы на начальном этапе проектирования скорее имеет отрицательный характер, поскольку предопределяет способы реализации поведения системы. А согласно рекомендациям, RUP именно эти аспекты должны быть скрыты от разработчика на диаграмме вариантов использования.

В самом общем случае, диаграмма вариантов использования представляет собой граф специального вида, который является графической нотацией для представления конкретных вариантов использования, актеров, возможно, некоторых интерфейсов, и отношений между этими элементами. При этом отдельные компоненты диаграммы могут быть заключены в прямоугольник, который обозначает проектируемую систему в целом. Следует отметить, что отношениями данного графа могут быть только некоторые фиксированные типы взаимосвязей между актерами и вариантами использования, которые в совокупности описывают сервисы или функциональные требования к моделируемой системе.

Рациональный унифицированный процесс разработки модели сложной системы представляет собой разбиение ее на составные части с минимумом взаимных связей на основе выделения пакетов. В самом языке UML пакет Варианты использования является подпакетом пакета Элементы поведения. Последний специфицирует понятия, при помощи которых определяют функциональность моделируемых систем. Элементы пакета вариантов использования являются первичными по отношению к тем, с помощью которых могут быть описаны сущности, такие как системы и подсистемы. Однако внутренняя структура этих сущностей никак не описывается. Базовые элементы этого пакета – вариант использования и актер. С этих понятий мы и приступим к изучению диаграмм вариантов использования.

Вариант использования

Конструкция или стандартный элемент языка UML вариант использования применяется для спецификации общих особенностей

поведения системы или любой другой сущности предметной области без рассмотрения внутренней структуры этой сущности. Каждый вариант использования определяет последовательность действий, которые должны быть выполнены проектируемой системой при взаимодействии ее с соответствующим актером. Диаграмма вариантов может дополняться пояснительным текстом, который раскрывает смысл или семантику составляющих ее компонентов. Такой пояснительный текст получил название примечания или сценария.

Отдельный вариант использования обозначается на диаграмме эллипсом, внутри которого содержится его краткое название или имя в форме глагола с пояснительными словами (рис. 1).

Рис. 1. Графическое обозначение варианта использования

Цель варианта использования заключается в том, чтобы определить законченный аспект или фрагмент поведения некоторой сущности без раскрытия внутренней структуры этой сущности. В качестве такой сущности может выступать исходная система или любой другой элемент модели, который обладает собственным поведением, подобно подсистеме или классу в модели системы.

Каждый вариант использования соответствует отдельному сервису, который предоставляет моделируемую сущность или систему по запросу пользователя (актера), т. е. определяет способ применения этой сущности. Сервис, который инициализируется по запросу пользователя, представляет собой законченную последовательность действий. Это означает, что после того как система закончит обработку запроса пользователя, она должна возвратиться в исходное состояние, в котором готова к выполнению следующих запросов.

Варианты использования описывают не только взаимодействия между пользователями и сущностью, но также реакции сущности на получение отдельных сообщений от пользователей и восприятие этих сообщений за пределами сущности. Варианты использования могут включать в себя описание особенностей способов реализации сервиса и различных исключительных ситуаций, таких как

корректная обработка ошибок системы. Множество вариантов использования в целом должно определять все возможные стороны ожидаемого поведения системы. Для удобства множество вариантов использования может рассматриваться как отдельный пакет.

С системно-аналитической точки зрения варианты использования могут применяться как для спецификации внешних требований к проектируемой системе, так и для спецификации функционального поведения уже существующей системы. Кроме этого, варианты использования неявно устанавливают требования, определяющие, как пользователи должны взаимодействовать с системой, чтобы иметь возможность корректно работать с предоставляемыми данной системой сервисами!

Примечание

Каждый выполняемый вариантом использования метод реализуется как неделимая транзакция, т. е. выполнение сервиса не может быть прервано никаким другим экземпляром варианта использования.

Применение вариантов использования на всех уровнях диаграммы позволяет не только достичь требуемого уровня унификации обозначений для представления функциональности подсистем и системы в целом, но и является мощным средством последовательного уточнения требований к проектируемой системе на основе полууровневого спуска от пакетов системы к операциям классов. С другой стороны, модификация отдельных операций класса может оказать обратное влияние на уточнение сервиса соответствующего варианта использования, т. е. реализовать эффект обратной связи с целью уточнения спецификаций или требований на уровне пакетов системы.

В метамодели UML вариант использования является подклассом классификатора, который описывает последовательности действий, выполняемых отдельным экземпляром варианта использования. Эти действия включают изменения состояния и взаимодействия со средой варианта использования. Эти последовательности могут описываться различными способами, включая такие, как графы деятельности и автоматы.

Примерами вариантов использования могут являться следующие действия: проверка состояния текущего счета клиента, оформление заказа на покупку товара, получение дополнительной инфор-

мации о кредитоспособности клиента, отображение графической формы на экране монитора и другие действия.

Актеры

Актер представляет собой любую внешнюю по отношению к моделируемой системе сущность, которая взаимодействует с системой и использует ее функциональные возможности для достижения определенных целей или решения частных задач. При этом актеры служат для обозначения согласованного множества ролей, которые могут играть пользователи в процессе взаимодействия с проектируемой системой. Каждый актер может рассматриваться как некая отдельная роль относительно конкретного варианта использования. Стандартным графическим обозначением актера на диаграммах является фигурка «человечка», под которой записывается конкретное имя актера (рис. 2).

Рис. 2. Графическое обозначение актера

В некоторых случаях актер может обозначаться в виде прямоугольника класса с ключевым словом «актер» и обычными составляющими элементами класса. Имена актеров должны записываться заглавными буквами и следовать рекомендациям использования имен для типов и классов модели. При этом символ отдельного актера связывает соответствующее описание актера с конкретным именем. Имена абстрактных актеров, как и других абстрактных элементов языка UML, рекомендуется обозначать курсивом.

Примечание

Имя актера должно быть достаточно информативным с точки зрения семантики. Вполне подходят для этой цели наименования должностей в компании (например, продавец, кассир, менеджер, президент). Не рекомендуется давать актерам имена собственные (например, «О.Бендер») или моделей конкретных устройств (например, «маршрутизатор Cisco 3640»), даже если это с очевидностью следует из контекста проекта. Дело в том, что одно и то же лицо может выступать в нескольких ролях и, соответственно, обращаться к различным сервисам системы. Например, посетитель бан-

ка может являться как потенциальным клиентом, и тогда он востребует один из его сервисов, а может быть и налоговым инспектором или следователем прокуратуры. Сервис для последнего может быть совершенно исключительным по своему характеру.

Примерами актеров могут быть: клиент банка, банковский служащий, продавец магазина, менеджер отдела продаж, пассажир авиарейса, водитель автомобиля, администратор гостиницы, сотовый телефон и другие сущности, имеющие отношение к концептуальной модели соответствующей предметной области.

Примечание

В метамодели актер является подклассом классификатора. Актеры могут взаимодействовать с множеством вариантов использования и иметь множество интерфейсов, каждый из которых может представлять особенности взаимодействия других элементов с отдельными актерами.

Актеры используются для моделирования внешних по отношению к проектируемой системе сущностей, которые взаимодействуют с системой и используют ее в качестве отдельных пользователей. В качестве актеров могут выступать другие системы, подсистемы проектируемой системы или отдельные классы. Важно понимать, что каждый актер определяет некоторое согласованное множество ролей, в которых могут выступать пользователи данной системы в процессе взаимодействия с ней. В каждый момент времени с системой взаимодействует вполне определенный пользователь, при этом он играет или выступает в одной из таких ролей. Наиболее наглядный пример актера – конкретный пользователь системы со своими собственными параметрами аутентификации.

Любая сущность, которая согласуется с подобным неформальным определением актера, представляет собой экземпляр или пример актера. Для моделируемой системы актерами могут быть как субъекты-пользователи, так и другие системы. Поскольку пользователи системы всегда являются внешними по отношению к этой системе, то они всегда представляются в виде актеров.

Так как в общем случае актер всегда находится вне системы, его внутренняя структура никак не определяется. Для актера имеет значение только его внешнее представление, т. е. то, как он воспринимается со стороны системы. Актеры взаимодействуют с системой посредством передачи и приема сообщений от вариантов использо-

вания. Сообщение представляет собой запрос актером сервиса от системы и получение этого сервиса. Это взаимодействие может быть выражено посредством ассоциаций между отдельными актерами и вариантами использования или классами. Кроме этого, с актерами могут быть связаны интерфейсы, которые определяют, каким образом другие элементы модели взаимодействуют с этими актерами.

Два и более актера могут иметь общие свойства, т. е. взаимодействовать с одним и тем же множеством вариантов использования одинаковым образом. Такая общность свойств и поведения представляется в виде рассматриваемого ниже отношения обобщения с другим, возможно, абстрактным актером, который моделирует соответствующую общность ролей. Совокупность отношений, которые могут присутствовать на диаграмме вариантов использования, будет рассмотрена ниже в данной главе.

Интерфейсы

Интерфейс (interface) служит для спецификации параметров модели, которые видимы извне без указания их внутренней структуры. В языке UML интерфейс является классификатором и характеризует только ограниченную часть поведения моделируемой сущности. Применительно к диаграммам вариантов использования, интерфейсы определяют совокупность операций, которые обеспечивают необходимый набор сервисов или функциональности для актеров. Интерфейсы не могут содержать ни атрибутов, ни состояний, ни направленных ассоциаций. Они содержат только операции без указания особенностей их реализации. Формально интерфейс эквивалентен абстрактному классу без атрибутов и методов с наличием только абстрактных операций.

На диаграмме вариантов использования интерфейс изображается в виде маленького круга, рядом с которым записывается его имя (рис. 3, а). В качестве имени может быть существительное, которое характеризует соответствующую информацию или сервис (например, «датчик», «сирена», «видеокамера»), но чаще строка текста (например, «запрос к базе данных», «форма ввода», «устройство подачи звукового сигнала»). Если имя записывается на английском, то оно должно начинаться с заглавной буквы I, например, ISecurelnformation, ISensor (рис. 3, б).

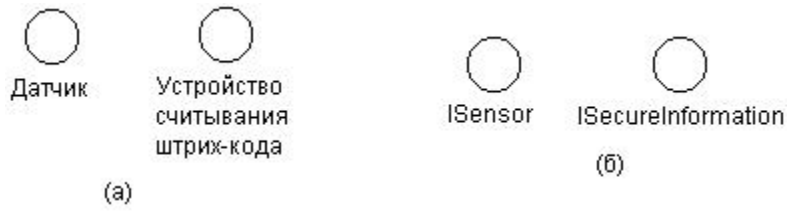


Рис. 3. Графическое изображение интерфейсов на диаграммах вариантов использования

Примечание

Имена интерфейсов подчиняются общим правилам наименования компонентов языка UML, т. е. имя может состоять из любого числа букв, цифр и некоторых знаков препинания, таких как двойное двоеточие «::». Последний символ используется для более сложных имен, включающих в себя не только имя самого интерфейса (после знака), но и имя сущности, которая включает в себя данный интерфейс (перед знаком). Примерами таких имен являются: «Сеть предприятия сервер» для указания на сервер сети предприятия или «Система аутентификации клиентов::форма ввода пароля».

Графический символ отдельного интерфейса может соединяться на диаграмме сплошной линией с тем вариантом использования, который его поддерживает. Сплошная линия в этом случае указывает на тот факт, что связанный с интерфейсом вариант использования должен реализовывать все операции, необходимые для данного интерфейса, а возможно и больше (рис. 4, а). Кроме этого, интерфейсы могут соединяться с вариантами использования пунктирной линией со стрелкой (рис. 4, б), означающей, что вариант использования предназначен для спецификации только того сервиса, который необходим для реализации данного интерфейса.

Рис. 4. Графическое изображение взаимосвязей интерфейсов с вариантами использования

С системно-аналитической точки зрения интерфейс не только отделяет спецификацию операций системы от их реализации, но и определяет общие границы проектируемой системы. В последую-

щем интерфейс может быть уточнен явным указанием тех операций, которые специфицируют отдельный аспект поведения системы. В этом случае он изображается в форме прямоугольника класса с ключевым словом «interface» в секции имени, с пустой секцией атрибутов и с непустой секцией операций. Однако подобное графическое представление используется на диаграммах классов или диаграммах, характеризующих поведение моделируемой системы.

Важность интерфейсов заключается в том, что они определяют стыковочные узлы в проектируемой системе, что совершенно необходимо для организации коллективной работы над проектом. Более того, спецификация интерфейсов способствует «безболезненной» модификации уже существующей системы при переходе на новые технологические решения. В этом случае изменению подвергается только реализация операций, но никак не функциональность самой системы. А это обеспечивает совместимость последующих версий программ с первоначальными при спиральной технологии разработки программных систем.

Примечания

Примечания (notes) в языке UML предназначены для включения в модель произвольной текстовой информации, имеющей непосредственное отношение к контексту разрабатываемого проекта. В качестве такой информации могут быть комментарии разработчика (например, дата и версия разработки диаграммы или ее отдельных компонентов), ограничения (например, на значения отдельных связей или экземпляры сущностей) и помеченные значения. Применительно к диаграммам вариантов использования примечание может носить самую общую информацию, относящуюся к общему контексту системы.

Графически примечания обозначаются прямоугольником с «загнутым» верхним правым уголком (рис. 5). Внутри прямоугольника содержится текст примечания. Примечание может относиться к любому элементу диаграммы, в этом случае их соединяет пунктирная линия. Если примечание относится к нескольким элементам, то от него проводятся, соответственно, несколько линий. Разумеется, примечания могут присутствовать не только на диаграмме вариантов использования, но и на других канонических диаграммах.

Рис. 5. Примеры примечаний в языке UML

Если в примечании указывается ключевое слово «constraint», то данное примечание является ограничением, налагаемым на соответствующий элемент модели, но не на саму диаграмму. При этом запись ограничения заключается в фигурные скобки и должна соответствовать правилам правильного построения выражений языка OCL. Более подробно язык объектных ограничений и примеры его использования будут рассмотрены в приложении. Однако для диаграмм вариантов использования ограничения включать в модели не рекомендуется, поскольку они достаточно жестко регламентируют отдельные аспекты системы. Подобная регламентация противоречит неформальному характеру общей модели системы, в качестве которой выступает диаграмма вариантов использования.

Отношения на диаграмме вариантов использования

Между компонентами диаграммы вариантов использования могут существовать различные отношения, которые описывают взаимодействие экземпляров одних актеров и вариантов использования с экземплярами других актеров и вариантов. Один актер может взаимодействовать с несколькими вариантами использования. В этом случае этот актер обращается к нескольким сервисам данной системы. В свою очередь один вариант использования может взаимодействовать с несколькими актерами, предоставляя для всех них свой сервис. Следует заметить, что два варианта использования, определенные для одной и той же сущности, не могут взаимодействовать друг с другом, поскольку каждый из них самостоятельно описывает законченный вариант использования этой сущности. Более того, варианты использования всегда предусматривают некоторые сигналы или сообщения, когда взаимодействуют с актерами за пре-

делами системы. В то же время могут быть определены другие способы для взаимодействия с элементами внутри системы.

В языке UML имеется несколько стандартных видов отношений между актерами и вариантами использования:

- Отношение ассоциации (association relationship).
- Отношение расширения (extend relationship).
- Отношение обобщения (generalization relationship).
- Отношение включения (include relationship).

При этом общие свойства вариантов использования могут быть представлены тремя различными способами, а именно с помощью отношений расширения, обобщения и включения.

Отношение ассоциации

Отношение ассоциации является одним из фундаментальных понятий в языке UML и в той или иной степени используется при построении всех графических моделей систем в форме канонических диаграмм.

Применительно к диаграммам вариантов использования оно служит для обозначения специфической роли актера в отдельном варианте использования. Другими словами, ассоциация специфицирует семантические особенности взаимодействия актеров и вариантов использования в графической модели системы. Таким образом, это отношение устанавливает, какую конкретную роль играет актер при взаимодействии с экземпляром варианта использования. На диаграмме вариантов использования, так же как и на других диаграммах, отношение ассоциации обозначается сплошной линией между актером и вариантом использования. Эта линия может иметь дополнительные условные обозначения, такие, например, как имя и кратность (рис. 6).

Рис. 6. Пример графического представления отношения ассоциации между актером и вариантом использования

Кратность (multiplicity) ассоциации указывается рядом с обозначением компонента диаграммы, который является участником данной ассоциации. Кратность характеризует общее количество

конкретных экземпляров данного компонента, которые могут выступать в качестве элементов данной ассоциации. Применительно к диаграммам вариантов использования кратность имеет специальное обозначение в форме одной или нескольких цифр и, возможно, специального символа «*» (звездочка).

Примечание

Возвращаясь к общей теории множеств, основы которой были рассмотрены в главе 2, следует заметить, что кратность представляет собой мощность множества экземпляров сущности, участвующей в данной ассоциации. Что касается самого понятия ассоциации, то это одна из наиболее общих форм отношений в языке UML.

Для диаграмм вариантов использования наиболее распространенными являются четыре основные формы записи кратности отношения ассоциации:

- Целое неотрицательное число (включая цифру 0). Предназначено для указания кратности, которая является строго фиксированной для элемента соответствующей ассоциации. В этом случае количество экземпляров актеров или вариантов использования, которые могут выступать в качестве элементов отношения ассоциации, в точности равно указанному числу.

Примером этой формы записи кратности ассоциации является указание кратности «1» для актера «Клиент банка» (рис. 6). Эта запись означает, что каждый экземпляр варианта использования «Оформить кредит для клиента банка» может иметь в качестве своего элемента единственный экземпляр актера «Клиент банка». Другими словами, при оформлении кредита в банке необходимо иметь в виду, что каждый конкретный кредит оформляется на единственного клиента этого банка.

- Два целых неотрицательных числа, разделенные двумя точками и записанные в виде: «первое число .. второе число». Данная запись в языке UML соответствует нотации для множества или интервала целых чисел, которая применяется в некоторых языках программирования для обозначения границ массива элементов. Эту запись следует понимать как множество целых неотрицательных чисел, следующих в последовательно возрастающем порядке:

{первое_число, первое_число+1, первое_число+2, ..., второе_число]. Очевидно, что первое число должно быть строго мень-

ше второго числа в арифметическом смысле, при этом первое число может быть равно 0.

Пример такой формы записи кратности ассоциации – «1..5». Эта запись означает, что количество отдельных экземпляров данного компонента, которые могут выступать в качестве элементов данной ассоциации, равно некоторому заранее неизвестному числу из множества целых чисел {1, 2, 3, 4, 5}. Эта ситуация может иметь место, например, в случае рассмотрения в качестве актера – клиента банка, а в качестве варианта использования – процедуру открытия счета в банке. При этом количество отдельных счетов каждого клиента в данном банке, исходя из некоторых дополнительных соображений, может быть не больше 5. Эти дополнительные соображения как раз и являются внешними требованиями по отношению к проектируемой системе и определяются ее заказчиком на начальных этапах ООАП.

- Два символа, разделенные двумя точками. При этом первый из них является целым неотрицательным числом или 0, а второй – специальным символом «*». Здесь символ «*»обозначает произвольное конечное целое неотрицательное число, значение которого неизвестно на момент задания соответствующего отношения ассоциации.

Пример такой формы записи кратности ассоциации – «2..*». Запись означает, что количество отдельных экземпляров данного компонента, которые могут выступать в качестве элементов данной ассоциации, равно некоторому заранее неизвестному числу из подмножества натуральных чисел: {2, 3, 4}.

- Единственный символ «*», который является сокращением записи интервала «0..*». В этом случае количество отдельных экземпляров данного компонента отношения ассоциации может быть любым целым неотрицательным числом. При этом 0 означает, что для некоторых экземпляров соответствующего компонента данное отношение ассоциации может вовсе не иметь места.

В качестве примера этой записи можно привести кратность отношения ассоциации для варианта использования «Оформить кредит для клиента банка» (рис. 6). Здесь кратность «*» означает, что каждый отдельный клиент банка может оформить для себя несколько кредитов, при этом их общее число заранее неизвестно и ничем

не ограничивается. При этом некоторые клиенты могут совсем не иметь оформленных на свое имя кредитов (вариант значения 0).

Если кратность отношения ассоциации не указана, то по умолчанию принимается ее значение, равное 1.

Более детальное описание семантических особенностей отношения ассоциации будет дано при рассмотрении других диаграмм в последующих главах книги.

Отношение расширения

Отношение расширения определяет взаимосвязь экземпляров отдельного варианта использования с более общим вариантом, свойства которого определяются на основе способа совместного объединения данных экземпляров. В метамодели отношение расширения является направленным и указывает, что применительно к отдельным примерам некоторого варианта использования должны быть выполнены конкретные условия, определенные для расширения данного варианта использования. Так, если имеет место отношение расширения от варианта использования A к варианту использования B, то это означает, что свойства экземпляра варианта использования B могут быть дополнены благодаря наличию свойств у расширенного варианта использования A.

Отношение расширения между вариантами использования обозначается пунктирной линией со стрелкой (вариант отношения зависимости), направленной от того варианта использования, который является расширением для исходного варианта использования. Данная линия со стрелкой помечается ключевым словом «extend» («расширяет»), как показано на рис. 7.

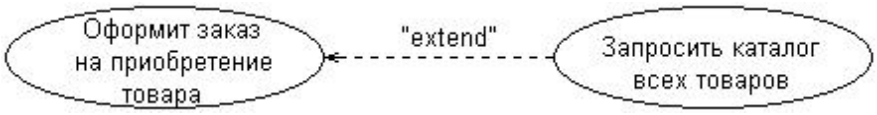


Рис. 7. Пример графического изображения отношения расширения между вариантами использования

Отношение расширения отмечает тот факт, что один из вариантов использования может присоединять к своему поведению некоторое дополнительное поведение, определенное для другого варианта использования. Данное отношение включает в себя некоторое условие и ссылки на точки расширения в базовом варианте использования. Чтобы расширение имело место, должно быть выпол-

нено определенное условие данного отношения. Ссылки на точки расширения определяют те места в базовом варианте использования, в которые должно быть помещено соответствующее расширение при выполнении условия.

Один из вариантов использования может быть расширением для нескольких базовых вариантов, а также иметь в качестве собственных расширений несколько других вариантов. Базовый вариант использования может дополнительно никак не зависеть от своих расширений.

Семантика отношения расширения определяется следующим образом. Если экземпляр варианта использования выполняет некоторую последовательность действий, которая определяет его поведение, и при этом имеется точка расширения на экземпляр другого варианта использования, которая является первой из всех точек расширения у исходного варианта, то проверяется условие данного отношения. Если условие выполняется, исходная последовательность действий расширяется посредством включения действий экземпляра другого варианта использования. Следует заметить, что условие отношения расширения проверяется лишь один раз – при первой ссылке на точку расширения, и если оно выполняется, то все расширяющие варианты использования вставляются в базовый вариант.

В представленном выше примере (рис. 7) при оформлении заказа на приобретение товара только в некоторых случаях может потребоваться предоставление клиенту каталога всех товаров. При этом условием расширения является запрос от клиента на получение каталога товаров. Очевидно, что после получения каталога клиенту необходимо некоторое время на его изучение, в течение которого оформление заказа приостанавливается. После ознакомления с каталогом клиент решает либо в пользу выбора отдельного товара, либо отказа от покупки вообще. Сервис или вариант использования «Оформить заказ на приобретение товара» может отреагировать на выбор клиента уже после того, как клиент получит для ознакомления каталог товаров.

Точка расширения может быть как отдельной точкой в последовательности действий, так и множеством отдельных точек. Важно представлять себе, что если отношение расширения имеет некоторую последовательность точек расширения, только первая из них может

определять множество отдельных точек. Все остальные должны определять в точности одну такую точку. Какая из точек должна быть первой точкой расширения, т. е. определяться единственным расширением. Такие ссылки на расположение точек расширения могут быть представлены различными способами, например, с помощью текста примечания на естественном языке (рис. 8), пред- и постусловий, а также с использованием имен состояний в автомате.

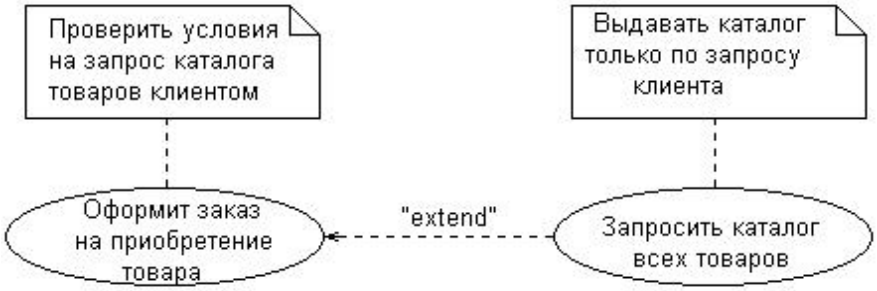


Рис. 8. Графическое изображение отношения расширения с примечаниями условий выполнения вариантов использования

Отношение обобщения

Отношение обобщения служит для указания того факта, что некоторый вариант использования А может быть обобщен до варианта использования В. В этом случае вариант А будет являться специализацией варианта В. При этом В называется предком или родителем по отношению А, а вариант А – потомком по отношению к варианту использования В. Следует подчеркнуть, что потомок наследует все свойства и поведение своего родителя, а также может быть дополнен новыми свойствами и особенностями поведения. Графически данное отношение обозначается сплошной линией со стрелкой в форме незакрашенного треугольника, которая указывает на родительский вариант использования (рис. 9). Эта линия со стрелкой имеет специальное название – стрелка «обобщение».

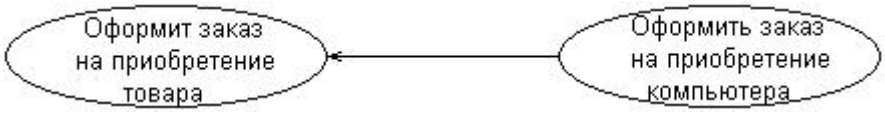


Рис. 9. Пример графического изображения отношения обобщения между вариантами использования

Отношение обобщения между вариантами использования применяется в том случае, когда необходимо отметить, что дочер-

ние варианты использования обладают всеми атрибутами и особенностями поведения родительских вариантов. При этом дочерние варианты использования участвуют во всех отношениях родительских вариантов. В свою очередь, дочерние варианты могут наделяться новыми свойствами поведения, которые отсутствуют у родительских вариантов использования, а также уточнять или модифицировать наследуемые от них свойства поведения.

Применительно к данному отношению, один вариант использования может иметь несколько родительских вариантов. В этом случае реализуется множественное наследование свойств и поведения отношения предков: С другой стороны, один вариант использования может быть предком для нескольких дочерних вариантов, что соответствует таксономическому характеру отношения обобщения.

Между отдельными актерами также может существовать отношение обобщения. Данное отношение является направленным и указывает на факт специализации одних актеров относительно других. Например, отношение обобщения от актера А к актеру В отмечает тот факт, что каждый экземпляр актера А является одновременно экземпляром актера В и обладает всеми его свойствами. В этом случае актер В является родителем по отношению к актеру А, а актер А, соответственно, потомком актера В. При этом актер А обладает способностью играть такое же множество ролей, что и актер В. Графически данное отношение также обозначается стрелкой обобщения, т. е. сплошной линией со стрелкой в форме незакрашенного треугольника, которая указывает на родительского актера (рис. 10).

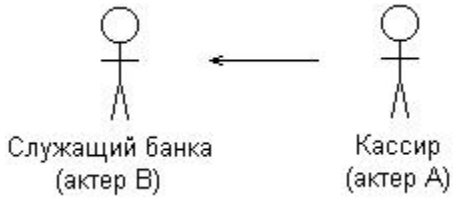


Рис. 10. Пример графического изображения отношения обобщения между актерами

Отношение включения

Отношение включения между двумя вариантами использования указывает, что некоторое заданное поведение для одного варианта использования включается в качестве составного компонента в

последовательность поведения другого варианта использования. Данное отношение является направленным бинарным отношением в том смысле, что пара экземпляров вариантов использования всегда упорядочена в отношении включения.

Семантика этого отношения определяется следующим образом. Когда экземпляр первого варианта использования в процессе своего выполнения достигает точки включения в последовательность поведения экземпляра второго варианта использования, экземпляр первого варианта использования выполняет последовательность действий, определяющую поведение экземпляра второго варианта использования, после чего продолжает выполнение действий своего поведения. При этом предполагается, что даже если экземпляр первого варианта использования может иметь несколько включаемых в себя экземпляров других вариантов, выполняемые ими действия должны закончиться к некоторому моменту, после чего должно быть продолжено выполнение прерванных действий экземпляра первого варианта использования в соответствии с заданным для него поведением.

Один вариант использования может быть включен в несколько других вариантов, а также включать в себя другие варианты. Включаемый вариант использования может быть независимым от базового варианта в том смысле, что он предоставляет последнему некоторое инкапсулированное поведение, детали реализации которого скрыты от последнего и могут быть легко перераспределены между несколькими включаемыми вариантами использования. Более того, базовый вариант может зависеть только от результатов выполнения включаемого в него поведения, но не от структуры включаемых в него вариантов.

Отношение включения, направленное от варианта использования A к варианту использования B, указывает, что каждый экземпляр варианта A включает в себя функциональные свойства, заданные для варианта B. Эти свойства специализируют поведение соответствующего варианта A на данной диаграмме. Графически данное отношение обозначается пунктирной линией со стрелкой (вариант отношения зависимости), направленной от базового варианта использования к включаемому. При этом данная линия со стрелкой помечается ключевым словом «include» («включает»), как показано на рис. 11.

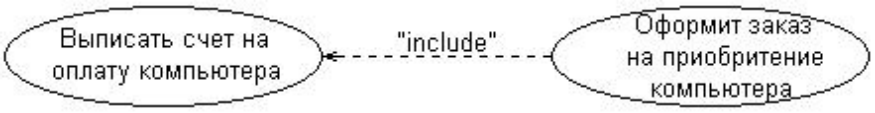


Рис. 11. Пример графического изображения отношения включения между вариантами использования

Примечание

Следует заметить, что рассмотренные три последних отношения могут существовать только между вариантами использования, которые определены для одной и той же сущности. Причина этого заключается в том, что поведение некоторой сущности обусловлено вариантами использования только этой сущности. Другими словами, все экземпляры варианта использования выполняются лишь внутри данной сущности. Если некоторый вариант использования должен иметь отношение обобщения, включения или расширения с вариантом использования другой сущности, получаемые в результате экземпляры вариантов должны быть включены в обе сущности, что противоречит семантическим правилам представления элементов языка UML. Однако эти отношения, определенные в пределах одной сущности, могут быть использованы в пределах другой сущности, если обе сущности связаны между собой отношением обобщения. В этом случае поведение соответствующих вариантов использования подчиняется общим правилам наследования свойств и поведения сущности-предка всеми дочерними сущностями.

Пример построения диаграммы вариантов использования

В качестве примера рассмотрим процесс моделирования системы продажи товаров по каталогу, которая может быть использована при создании соответствующих информационных систем.

В качестве актеров данной системы могут выступать два субъекта, один из которых является продавцом, а другой – покупателем. Каждый из этих актеров взаимодействует с рассматриваемой системой продажи товаров по каталогу и является ее пользователем, т. е. они оба обращаются к соответствующему сервису «Оформить заказ на покупку товара». Как следует из существа выдвигаемых к системе требований, этот сервис выступает в качестве варианта использования разрабатываемой диаграммы, первоначальная структура которой может включать в себя только двух указанных актеров и единственный вариант использования (рис. 12).

Рис. 12. Исходная диаграмма вариантов использования
для примера разработки системы продажи товаров по каталогу

Значения указанных на данной диаграмме кратностей отражают общие правила или логику оформления заказов на покупку товаров. Согласно этим правилам, один продавец может участвовать в оформлении нескольких заказов, в то же время каждый заказ может быть оформлен только одним продавцом, который несет ответственность за корректность его оформления и, в связи с этим, будет иметь агентское вознаграждение за его оформление. С другой стороны, каждый покупатель может оформлять на себя несколько заказов, но, в то же время, каждый заказ должен быть оформлен на единственного покупателя, к которому переходят права собственности на товар после его оплаты.

Примечание

Рассмотренные выше примеры значений для кратности отношения ассоциации могут вызвать невольное восхищение глубиной своей семантики, которая в единственном специальном символе отражает вполне определенные логические условия реализации соответствующих компонентов диаграммы вариантов использования.

На следующем этапе разработки данной диаграммы вариант использования «Оформить заказ на покупку товара» может быть уточнен на основе введения в рассмотрение четырех дополнительных вариантов использования. Это следует из более детального анализа процесса продажи товаров, что позволяет выделить в качестве отдельных сервисов такие действия, как обеспечить покупателя информацией о товаре, согласовать условия оплаты товара и заказать товар со склада. Вполне очевидно, что указанные действия раскрывают поведение исходного варианта использования в смысле его конкретизации, и поэтому между ними будет иметь место отношение включения.

С другой стороны, продажа товаров по каталогу предполагает наличие самостоятельного информационного объекта – каталога товаров, который в некотором смысле не зависит от реализации сер-

виса по обслуживанию покупателей. В нашем случае, каталог товаров может запрашиваться покупателем или продавцом при необходимости выбора товара и уточнения деталей его продажи. Вполне резонно представить сервис «Запросить каталог товаров» в качестве самостоятельного варианта использования.

Полученная в результате последующей детализации уточненная диаграмма вариантов использования будет содержать 5 вариантов использования и 2 актеров (рис. 13), между которыми установлены отношения включения и расширения.

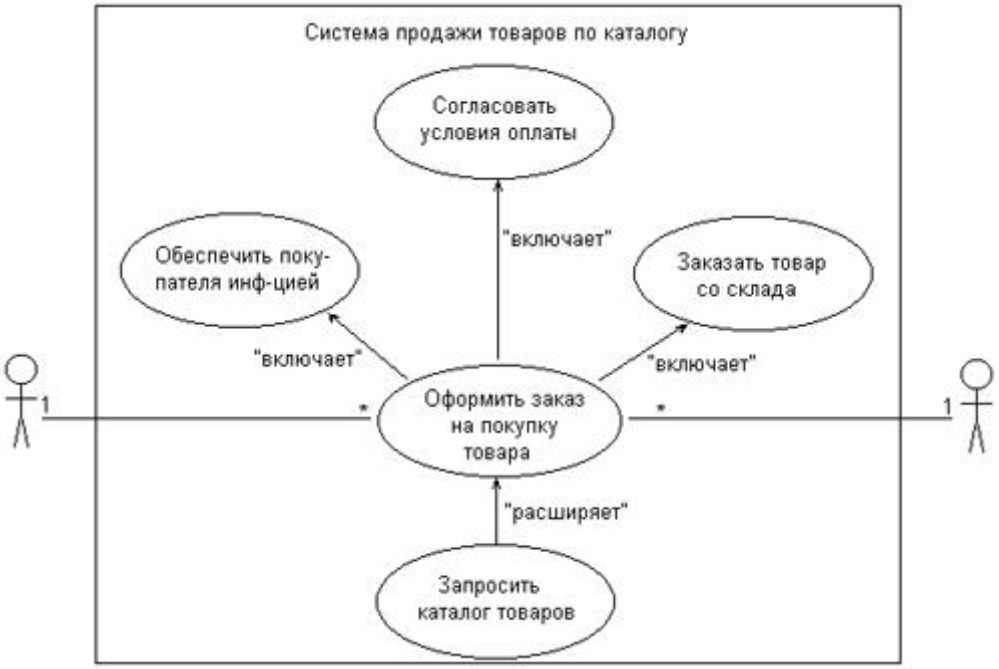


Рис. 13. Уточненный вариант диаграммы вариантов использования для примера системы продажи товаров по каталогу

Приведенная выше диаграмма вариантов использования, в свою очередь, может быть детализирована далее с целью более глубокого уточнения предъявляемых к системе требований и конкретизации деталей ее последующей реализации. В рамках общей парадигмы ООАП подобная детализация может выполняться в двух основных направлениях.

С одной стороны, детализация может быть выполнена на основе установления дополнительных отношений типа отношения «обобщение-специализация» для уже имеющихся компонентов диаграммы вариантов использования. Так, в рамках рассматриваемой системы продажи товаров может иметь самостоятельное значение и специфические особенности отдельная категория товаров – компьютеры. В этом случае диаграмма может быть дополнена вариантом

использования «Оформить заказ на покупку компьютера» и актерами «Покупатель компьютера» и «Продавец компьютеров», которые связаны с соответствующими компонентами диаграммы отношением обобщения (рис. 14).

Уточненный таким способом вариант диаграммы вариантов использования содержит одну важную особенность, которую необходимо отметить. А именно, хотя на данной диаграмме (рис. 14) отсутствуют изображения линий отношения ассоциации между актером «Продавец компьютеров» и вариантом использования «Оформить заказ на покупку компьютера», а также между актером «Покупатель компьютера» и вариантом использования «Оформить заказ на покупку компьютера», наличие отношения обобщения между соответствующими компонентами позволяет им наследовать отношение ассоциации от своих предков. Поскольку принцип наследования является одним из фундаментальных принципов объектно-ориентированного программирования, в нашем примере можно с уверенностью утверждать, что эти линии отношения ассоциации с соответствующими кратностями присутствуют на данной диаграмме в скрытом виде.

Рис. 14. Один из вариантов последующего уточнения диаграммы вариантов использования для примера рассматриваемой системы продажи

Для пояснения изложенного можно привести фрагмент диаграммы вариантов использования для рассмотренного примера, на котором явно указаны отношения ассоциации между дочерними компонентами (рис. 15). Данное изображение фрагмента диаграммы приводится с методической целью, при этом остальные компоненты диаграммы, которые остались без изменений, условно отмечены многоточием.

Рис. 15. Фрагмент диаграммы вариантов использования, который в неявном виде присутствует на уточненной диаграмме с отношением ассоциации между отдельными компонентами

Примечание

Строго говоря, приведенное выше изображение фрагмента диаграммы не является допустимым с точки зрения нотации языка UML Причиной этого следует считать многоточие, которое не может быть использовано в подобной интерпретации. Тем не менее, данное изображение иллюстрирует основные идеи наследования свойств и поведения, которые неявно могут присутствовать в графических моделях сложных систем. С другой стороны, следует всегда помнить, что эта информация является избыточной с точки зрения семантики языка UML, а значит может быть опущена, что и было сделано на предыдущей диаграмме (см. рис. 14).

Второе из основных направлений детализации диаграмм вариантов использования связано с последующей структуризацией ее отдельных компонентов в форме элементов других диаграмм. Например, конкретные особенности реализации вариантов использования в терминах взаимодействующих объектов, определенных в виде классов данной сущности, могут быть заданы на диаграмме кооперации. Указанное направление отражает основные особенности ООАП применительно к их реализации в языке UML. Эти осо-

бенности являются предметом рассмотрения во всех последующих главах книги.

Построение диаграммы вариантов использования является самым первым этапом процесса объектно-ориентированного анализа и проектирования, цель которого – представить совокупность требований к поведению проектируемой системы. Спецификация требований к проектируемой системе в форме диаграммы вариантов использования представляет собой самостоятельную модель, которая в языке UML получила название модели вариантов использования и имеет свое специальное стандартное имя или стереотип «useCaseModel».

В последующем все заданные в этой модели требования представляются в виде общей модели системы, которая состоит из пакета Системы. Последний в свою очередь может представлять собой иерархию пакетов, на самом верхнем уровне которых содержится множество классов модели проектируемой системы. Если же пакет системы со стандартным именем «topLevel Package» является подсистемой, то ее абстрактное поведение в точности такое же, как и у исходной системы.

Рекомендации по разработке диаграмм вариантов использования

Главное назначение диаграммы вариантов использования заключается в формализации функциональных требований к системе с помощью понятий соответствующего пакета и возможности согласования полученной модели с заказчиком на ранней стадии проектирования. Любой из вариантов использования может быть подвергнут дальнейшей декомпозиции на множество подвариантов использования отдельных элементов, которые образуют исходную сущность. Рекомендуемое общее количество актеров в модели – не более 20, а вариантов использования – не более 50. В противном случае модель теряет свою наглядность и, возможно, заменяет собой одну из некоторых других диаграмм.

Семантика построения диаграммы вариантов использования должна определяться следующими особенностями рассмотренных выше элементов модели. Отдельный экземпляр варианта использования по своему содержанию является выполнением последовательности действий, которая инициализируется посредством экземпляра сообщения от экземпляра актера. В качестве отклика или от-

ветной реакции на сообщение актера экземпляр варианта использования выполняет последовательность действий, установленную для данного варианта использования. Экземпляры актеров могут генерировать новые экземпляры сообщений для экземпляров вариантов использования.

Подобное взаимодействие будет продолжаться до тех пор, пока не закончится выполнение требуемой последовательности действий экземпляром варианта использования, и соответствующий экземпляр актера (и никакой другой) не получит требуемый экземпляр сервиса. Окончание взаимодействия означает отсутствие инициализации экземпляров сообщений от экземпляров актеров для соответствующих экземпляров вариантов использования.

Варианты использования могут быть специфицированы в виде текста, а в последующем – с помощью операций и методов вместе с атрибутами, в виде графа деятельности, посредством автомата или любого другого механизма описания поведения, включающего предусловия и постусловия. Взаимодействие между вариантами использования и актерами может уточняться на диаграмме кооперации, когда описываются взаимосвязи между сущностью, содержащей эти варианты использования, и окружением или внешней средой этой сущности.

В случае, когда для представления иерархической структуры проектируемой системы используются подсистемы, система может быть определена в виде вариантов использования на всех уровнях. Отдельные подсистемы или классы могут выступать в роли таких вариантов использования. При этом вариант, соответствующий некоторому из этих элементов, в последующем может уточняться множеством более мелких вариантов использования, каждый из которых будет определять сервис элемента модели, содержащийся в сервисе исходной системы. Вариант использования в целом может рассматриваться как суперсервис для уточняющих его подвариантов, которые, в свою очередь, могут рассматриваться как подсервисы исходного варианта использования.

Функциональность, определенная для более общего варианта использования, полностью наследуется всеми вариантами нижних уровней. Однако следует заметить, что структура элемента-контейнера не может быть представлена вариантами использования, поскольку они могут представлять только функциональность от-

дельных элементов модели. Подчиненные варианты использования кооперируются для совместного выполнения суперсервиса варианта использования верхнего уровня. Эта кооперация также может быть представлена на диаграмме кооперации в виде совместных действий отдельных элементов модели.

Отдельные варианты использования нижнего уровня могут участвовать в нескольких кооперациях, т. е. играть определенную роль при выполнении сервисов нескольких вариантов верхнего уровня. Для отдельных таких кооперациий могут быть определены соответствующие роли актеров, взаимодействующих с конкретными вариантами использования нижнего уровня. Эти роли будут играть актеры нижнего уровня модели системы. Хотя некоторые из таких актеров могут быть актерами верхнего уровня, это не противоречит принятым в языке UML семантическим правилам построения диаграмм вариантов использования. Более того, интерфейсы вариантов использования верхнего уровня могут полностью совпадать по своей структуре с соответствующими интерфейсами вариантов нижнего уровня.

Окружение вариантов использования нижнего уровня является самостоятельным элементом модели, который в свою очередь содержит другие элементы модели, определенные для этих вариантов использования. Таким образом, с точки зрения общего представления верхнего уровня взаимодействие между вариантами использования нижнего уровня определяет результат выполнения сервиса варианта верхнего уровня. Отсюда следует, что в языке UML вариант использования является элементом-контейнером.

Варианты использования классов соответствуют операциям этого класса, поскольку сервис класса является по существу выполнением операций данного класса. Некоторые варианты использования могут соответствовать применению только одной операции, в то время как другие – конечного множества операций, определенных в виде последовательности операций. В то же время одна операция может быть необходима для выполнения нескольких сервисов класса и поэтому будет появляться в нескольких вариантах использования этого класса.

Реализация варианта использования зависит от типа элемента модели, в котором он определен. Например, поскольку варианты использования класса определяются посредством операций этого

класса, они реализуются соответствующими методами. С другой стороны, варианты использования подсистемы реализуются элементами, из которых состоит данная подсистема. Поскольку подсистема не имеет своего собственного поведения, все предлагаемые подсистемой сервисы должны представлять собой композицию сервисов, предлагаемых отдельными элементами этой подсистемы, т. е., в конечном итоге, классами. Эти элементы могут взаимодействовать друг с другом для совместного обеспечения требуемого поведения отдельного варианта использования, посвященной построению диаграмм кооперации. Здесь лишь отметим, что кооперации используются как для уточнения спецификаций в виде вариантов использования нижних уровней диаграммы, так и для описания особенностей их последующей реализации.

Если в качестве моделируемой сущности выступает система или подсистема самого верхнего уровня, то отдельные пользователи вариантов использования этой системы моделируются актерами. Такие актеры, являясь внутренними по отношению к моделируемым подсистемам нижних уровней, часто в явном виде не указываются, хотя и присутствуют неявно в модели подсистемы. Вместо этого варианты использования непосредственно обращаются к тем модельным элементам, которые содержат в себе подобные неявные актеры, т. е. экземпляры которых играют роли таких актеров при взаимодействии с вариантами использования. Эти модельные элементы могут содержаться в других пакетах или подсистемах. В последнем случае роли определяются в том пакете, к которому относится соответствующая подсистема.

С системно-аналитической точки зрения построение диаграммы вариантов использования специфицирует не только функциональные требования к проектируемой системе, но и выполняет исходную структуризацию предметной области. Последняя задача сочетает в себе не только следование техническим рекомендациям, но и является в некотором роде искусством, умением выделять главное в модели системы. Хотя рациональный унифицированный процесс не исключает итеративный возврат в последующем к диаграмме вариантов использования для ее модификации, не вызывает сомнений тот факт, что любая подобная модификация потребует, как по цепочке, изменений во всех других представлениях системы. Поэтому всегда необходимо стремиться к возможно более точному пред-

ставлению модели именно в форме диаграммы вариантов использования.

Если же варианты использования применяются для спецификации части системы, то они будут эквивалентны соответствующим вариантам использования в модели подсистемы для части соответствующего пакета. Важно понимать, что все сервисы системы должны быть явно определены на диаграмме вариантов использования, и никаких других сервисов, которые отсутствуют на данной диаграмме, проектируемая система не может выполнять по определению. Более того, если для моделирования реализации системы используются сразу несколько моделей (например, модель анализа и модель проектирования), то множество вариантов использования всех пакетов системы должно быть эквивалентно множеству вариантов использования модели в целом.

Контрольные вопросы

1. Что такое диаграмма вариантов использования?
2. Какие элементы содержит диаграмма вариантов использования?
3. Что такое диаграмма использования?

2. Лабораторная работа №2. Построение диаграммы последовательности и генерация кода

Целью работы является изучение порядка построения диаграммы последовательности.

Для выполнения лабораторной работы № 2 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Диаграммы последовательности

В ходе проектирования ИС аналитик поэтапно спускается от общей концепции, через понимание ее логической структуры к наиболее детальным моделям, описывающим физическую реализацию.

С помощь диаграммы прецедентов (вариантов использования) выявляются основные пользователи системы и задачи, которые данная система должна решать. С помощью диаграммы деятельности мы описываем последовательность действий для каждого прецедента, необходимая для достижения поставленной цели.

Далее проектируется логическая структура системы с помощью диаграммы классов. На данном этапе выделяются классы, формирующие структуру БД Системы, а также классы реализующие некий набор операций, способствующий достижению целей в рамках выбранного прецедента. Для описания сложного поведения некоторых объектов (экземпляров класса) составляется диаграмма состояний.

Таким образом, аналитиками фиксируются такие поведенческие аспекты как алгоритм действий в рамках одного или нескольких прецедентов, необходимый для достижения определённого результата, а также изменение состояния объектов в ходе выполнения приведенных действий.

Зачастую на этапе спецификации требований необходимо показать не только алгоритм действий или изменение состояния объекта, но и обмен сообщениями между отдельными объектами Системы. Данную задачу решает диаграмма взаимодействия.

Диаграмма взаимодействия предназначена для моделирования отношений между объектами (ролями, классами, компонентами) Системы в рамках одного прецедента.

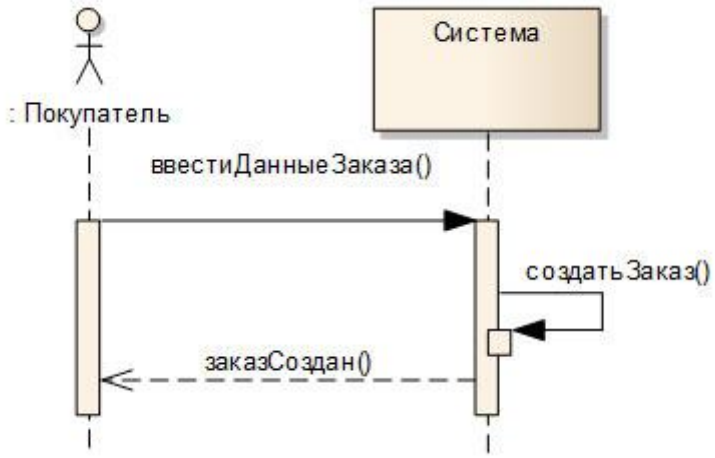
Данный вид диаграмм отражает следующие аспекты проектируемой Системы:

- обмен сообщениями между объектами (в том числе в рамках обмена сообщениями со сторонними Системами);
- ограничения, накладываемые на взаимодействие объектов;
- события, инициирующие взаимодействия объектов.

В отличие от диаграммы деятельности, которая показывает только последовательность (алгоритм) работы Системы, диаграммы взаимодействия акцентируют внимание разработчиков на сообщениях, инициирующих вызов определенных операций объекта (класса) или являющихся результатом выполнения операции.

Диаграмма последовательности является одной из разновидности диаграмм взаимодействия и предназначена для моделирования взаимодействия объектов Системы во времени, а также обмена сообщениями между ними.

Одним из основных принципов ООП является способ информационного обмена между элементами Системы, выражающийся в отправке и получении сообщений друг от друга. Таким образом, основные понятия диаграммы последовательности связаны с понятием Объект и Сообщение.

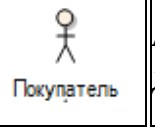


На диаграмме последовательности объекты в основном представляют экземпляры класса или сущности, обладающие поведением. В качестве объектов могут выступать пользователи, инициирующие взаимодействие, классы, обладающие поведением в Системе или программные компоненты, а иногда и Системы в целом.

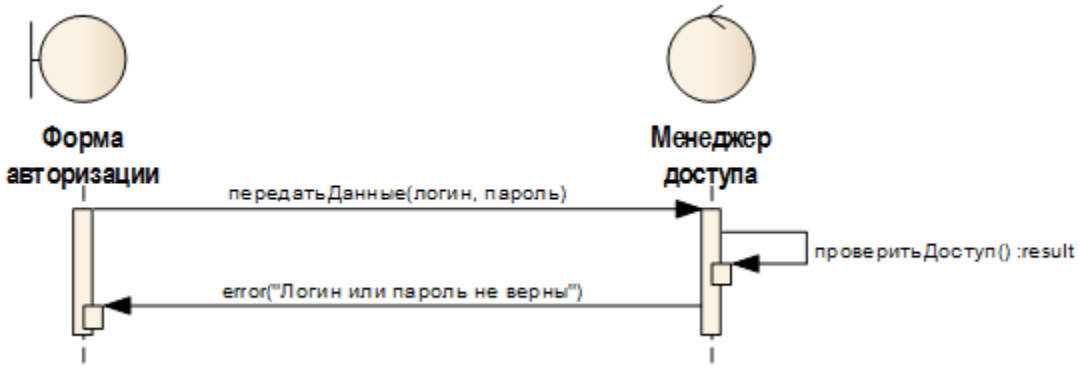
Объекты располагаются слева направо таким образом, чтобы крайним слева был тот объект, который инициирует взаимодействие.

Неотъемлемой частью объекта на диаграмме последовательности является линия жизни объекта. Линия жизни показывает время, в течение которого объект существует в Системе. Периоды активности объекта в момент взаимодействия показываются с помощью фокуса управления. Временная шкала на диаграмме направлена сверху вниз.

На диаграммах последовательности допустимо использование стандартных стереотипов класса:

	**Actor** – экземпляр участника процесса (роль на диаграмме прецедентов)
	**Boundary** – Класс-Разграничитель – используется для классов, отделяющих внутреннюю структуру системы от внешней среды (экранная форма, пользовательский интерфейс, устройство ввода-вывода). Объект со стереотипом <<bondary>> отличается от, привычного нам, класса <<Интерфейс>>, который по большей части предназначен для вызова методов класса, с которым он связан. Объект boundary показывает именно экранную форму, которая принимает и передает данные обработчику.</bondary>>
	**Contro**l – Класс-контроллер – активный элемент, который используются для выполнения некоторых операций над объектами (программный компонент, модуль, обработчик)
	**Entity** – Класс-сущность – обычно применяется для обозначения классов, которые хранят некую информацию о бизнес-объектах (соответствует таблице или элементу БД)

Также одним из основных понятий, связанных с диаграммой последовательности, является *Сообщение*.



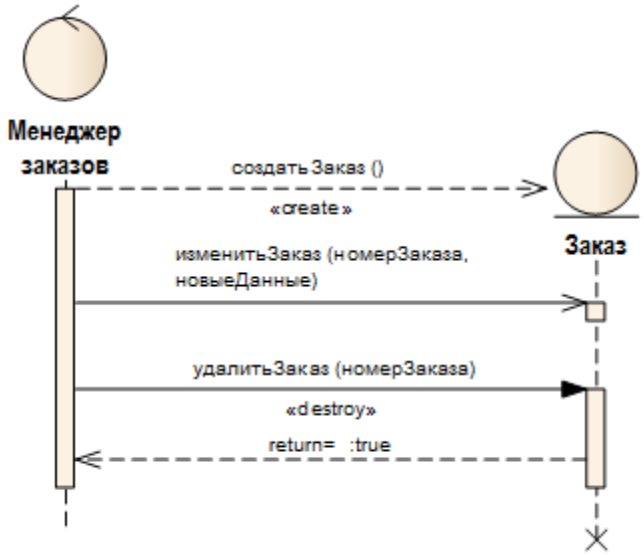
На диаграмме деятельности выделяются сообщения, инициирующие ту или иную деятельность или являющиеся ее следствием. На диаграмме состояний частично показан обмен сообщениями в рамках сообщений инициирующих изменение состояния объекта.

Диаграмма последовательности объединяет диаграмму деятельности, диаграмму состояний и диаграмму классов.

Таким образом, на диаграмме последовательности мы можем увидеть следующие аспекты:

- Сообщения, побуждающие объект к действию.
- Действия, которые вызываются сообщениями (методы) – зачастую это передача сообщения следующему объекты или возвращение определенных данных объекта.
- Последовательность обмена сообщениями между объектами.

Итак, прием сообщения инициирует выполнение определенных действий, направленных на решение отдельной задачи тем объектом, которому это сообщение отправлено. Сообщение в большинстве случаев (за исключением диаграмм, описывающих концептуальный уровень Системы) это вызов методов отдельных объектов, поэтому для корректного исполнения метода в сообщении необходимо передать какие-то данные и определить, что мы хотим видеть в ответ. При именовании сообщения на уровне проектирования реализации системы в качестве имени сообщения следует использовать имя метода.



В UML различают следующие виды сообщений:

- синхронное сообщение (**synchCall**) – соответствует синхронному вызову операции и подразумевает ожидание ответа от

объекта получателя. Пока ответ не поступит, никаких действий в Системе не производится;

- асинхронное сообщение (**asynchCall**) – которое соответствует асинхронному вызову операции и подразумевает, что объект может продолжать работу, не ожидая ответа;
- ответное сообщение (**reply**) – ответное сообщение от вызванного метода. Данный вид сообщения показывается на диаграмме по мере необходимости или, когда возвращаемые им данные несут смысловую нагрузку;
- потерянное сообщение (**lost**) – сообщение, не имеющее адресата сообщения, т. е. для него существует событие передачи и отсутствует событие приема;
- найденное сообщение (**found**) – сообщение, не имеющее инициатора сообщения, т. е. для него существует событие приема и отсутствует событие передачи.

Для сообщений также доступен ряд предопределенных стереотипов. Наиболее часто используемые стереотипы это **create** и **destroy.**

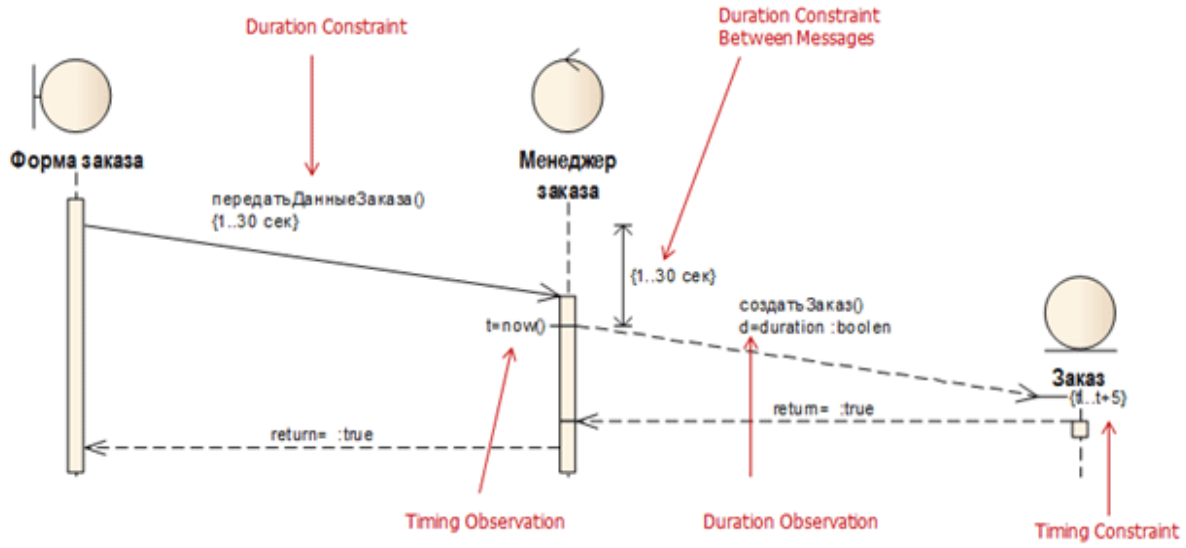
Сообщение со стереотипом **create**, вызывает в классе метод, которые создает экземпляр класса. На диаграмме последовательности не обязательно показывать с самого начала все объекты, участвующие во взаимодействии. При использовании сообщения со стереотипом **create**, создаваемый объект отображается на уровне конца сообщения.

Для уничтожения экземпляра класса используется сообщение со стереотипом **destroy**, при этом в конце линии жизни объекта отображаются две перекрещенные линии.

При отображении работы с сообщениями иногда возникает необходимость указать некоторые временные ограничения. Например, длительность передачи сообщения или ожидание ответа от объекта не должно превышать определенный временной интервал. Можно указать следующие временные параметры:

- ограничение продолжительности (Duration Constraint) – минимальное и максимальное значение продолжительности передачи сообщения
- ограничение продолжительности ожидания между передачей и получением сообщения (Duration Constraint Between Messages)

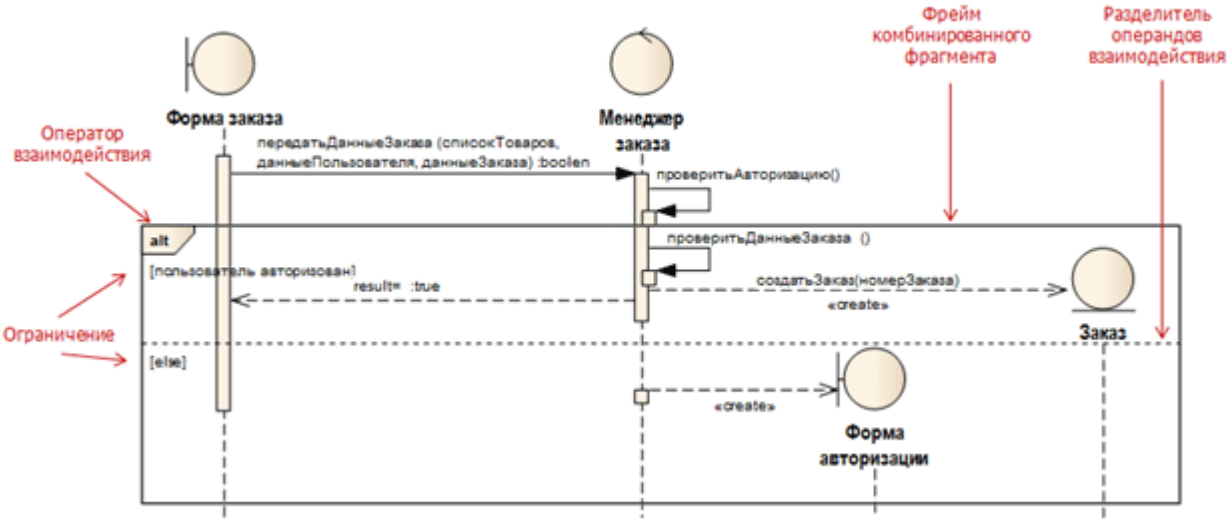
- перехват продолжительности сообщения (Duration Observation)
- временное ограничение (Timing Constraint) – временной интервал, в течение которого сообщение должно прийти к цели (устанавливается на стороне получателя)
- перехват времени, когда сообщение было отправлено (Timing Observation)



Форма заказа передает данные Менеджеру заказа, при этом передача данных не должна длиться больше 30 секунд – данное ограничение может понадобиться при выявлении требований к быстродействию Системы. Далее получение данных с формы инициирует запуск метода для создания экземпляра класса Заказ. Между получением данных от Формы заказа и инициализацией создания объекта должно пройти не более 30 секунд, в противном случае пользователю может быть предоставлено сообщение об ошибке или недоступности сервера. Длительность передачи сообщения о создании объекта может быть зафиксирована в переменной d.

Данное значение может понадобиться при установке временного ограничения на получение ответного сообщения клиентом. В момент передачи сообщения фиксируется временное значение и заносится в переменную t. Таким образом, можно установить ограничение на стороне приемника, указав переменную t в качестве минимального значения и t+ <допустимый интервал> в качестве максимального значения.

До появления UML 2.0 диаграмма последовательности рассматривалась только в рамках моделирования последовательности обмена сообщениями. Расширение сценария отображались с помощью ветвления линий сообщений, что не давало полной картины взаимодействия объектов. Таким образом, для целей моделирования расширения сценария, параллельности процессов или цикличности использовались диаграммы деятельности. Для решения данных задач в UML 2.0 было введено понятие фрейма взаимодействия и операторов взаимодействия.



Отдельные фрагменты диаграммы взаимодействия можно выделить с помощью фрейма. Фрейм должен содержать метку оператора взаимодействия. UML содержит следующие операнды:

- **Alt –** Несколько альтернативных фрагментов (alternative); выполняется только тот фрагмент, условие которого истинно.
- **Opt –** Необязательный (optional) фрагмент; выполняется, только если условие истинно. Эквивалентно **alt** с одной веткой.
- **Par –** Параллельный (parallel); все фрагменты выполняются параллельно.
- **loop –** Цикл (loop); фрагмент может выполняться несколько раз, а защита обозначает тело итерации.
- **region –** Критическая область (critical region); фрагмент может иметь только один поток, выполняющийся за один прием.
- **Neg –** Отрицательный (negative) фрагмент; обозначает неверное взаимодействие.
- **ref –** Ссылка (reference); ссылается на взаимодействие, определенное на другой диаграмме. Фрейм рисуется, чтобы охватить

линии жизни, вовлеченные во взаимодействие. Можно определять параметры и возвращать значение.

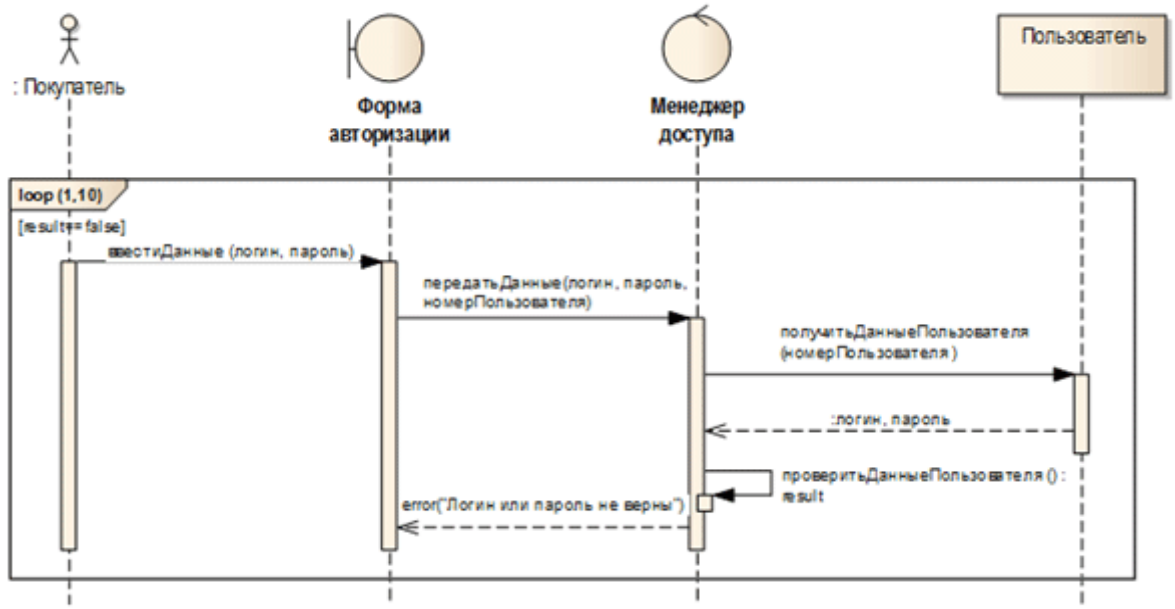
- **Sd –** Диаграмма последовательности (sequence diagram); используется для очерчивания всей диаграммы последовательности, если это необходимо.

При использовании фрагмента условного операнда фрейм должен содержать условие для ограничения взаимодействия. При использовании условного или параллельного операнда фрейм делится на регионы взаимодействия с помощью разделителя операторов взаимодействия.

К условным операндам относятся alt и opt. Операнд alt используется при моделировании расширения сценария, т. е. при наличии альтернативного потока взаимодействия. Оператор opt используется, если сообщение должно быть передано, только при истинности какого-то условия. Данный фрейм используется без разделения на регионы.

Параллельность потоков взаимодействия можно изобразить с помощью операнда par. Внутри фрейма моделируются потоки взаимодействия в отдельных регионах.

Цикличность потока взаимодействия может быть представлена на диаграмме последовательности с помощью операнда loop. При использовании оператора цикла можно указать минимальное и максимальное число итераций. Также фрейм должен содержать условие, при наступлении которого взаимодействие повторяется.



Диаграммы последовательности предназначены для моделирования взаимодействия между несколькими объектами. Зачастую диаграммы последовательности создаются для моделирования взаимодействия в рамках одного прецедента.

На концептуальном уровне можно использовать диаграммы последовательности для моделирования взаимодействия между Бизнес-актерами, но зачастую подобные диаграммы обрастают лишними подробностями и плохо читаются. На данном уровне лучше подойдут диаграммы деятельности, исключение составляют случае, когда необходимо смоделировать обмен сообщениями между двумя независимыми Системами.

Также диаграммы последовательности подойдут для моделирования взаимодействия пользователя и Системы в целом.

На уровне детальной спецификации требований диаграммы последовательности используются для моделирования взаимодействия компонентов Системы и пользовательских классов в рамках выбранного прецедента.

На уровне реализации с помощью диаграммы последовательности моделируется взаимодействие между отдельными компонентами Системы. На данном уровне детализации лучше подойдет диаграмма коммуникации.

Контрольные вопросы

1. Что такое диаграмма последовательности действий?
2. Какие элементы содержит диаграмма последовательности действий?
3. Что такое диаграмма использования?

3. Лабораторная работа №3. Построение диаграммы компонентов и генерация кода

Целью работы является изучение порядка построения диаграммы компонентов

Для выполнения лабораторной работы № 3 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Диаграмма компонентов

Все рассмотренные ранее диаграммы отражали концептуальные и логические аспекты построения модели системы. Особенность логического представления заключается в том, что оно оперирует понятиями, которые не имеют материального воплощения. Другими словами, различные элементы логического представления, такие как классы, ассоциации, состояния, сообщения, не существуют материально или физически. Они лишь отражают понимание статической структуры той или иной системы или динамические аспекты ее поведения.

Для создания конкретной физической системы необходимо реализовать все элементы логического представления в конкретные материальные сущности. Для описания таких реальных сущностей предназначен другой аспект модельного представления, а именно – физическое *представление* модели. В контексте языка *UML* это означает совокупность связанных физических сущностей, включая программное и *аппаратное обеспечение*, а также персонал, которые организованы для выполнения специальных задач.

Физическая система (*physical system*) – реально существующий прототип модели системы.

С тем чтобы пояснить отличие логического и физического представлений, необходимо в общих чертах рассмотреть процесс разработки программной системы. Ее исходным логическим представлением могут служить структурные схемы алгоритмов и процедур, описания интерфейсов и *концептуальные схемы баз* данных. Однако для реализации этой системы необходимо разработать исходный текст программы на языке программирования. При этом уже в тексте программы предполагается организация программного кода, определяемая синтаксисом языка программирования и предполагающая *разбиение* исходного кода на отдельные модули.

Однако исходные тексты программы еще не являются окончательной реализацией проекта, хотя и служат фрагментом его физического представления. Программная система может считаться реализованной в том случае, когда она будет способна выполнять функции своего целевого предназначения. А это возможно, только если программный код системы будет реализован в форме исполняемых модулей, библиотек классов и процедур, стандартных графических интерфейсов, файлов баз данных. Именно эти компоненты являются базовыми элементами физического представления системы в нотации языка *UML*.

Полный проект программной системы представляет собой совокупность моделей логического и физического представлений, которые должны быть согласованы между собой. В языке *UML* для физического представления моделей систем используются так называемые диаграммы реализации, которые включают в себя две отдельные *канонические диаграммы*: диаграмму компонентов и *диаграмму развертывания*.

Диаграмма компонентов, в отличие от ранее рассмотренных диаграмм, описывает особенности физического представления системы. *Диаграмма компонентов* позволяет определить архитектуру разрабатываемой системы, установив зависимости между программными *компонентами*, в роли которых может выступать исходный, бинарный и *исполняемый код*. Во многих средах разработки *модуль* или *компонент* соответствует файлу. Пунктирные стрелки, соединяющие *модули*, показывают отношения взаимозависимости, аналогичные тем, которые имеют *место* при компиляции исходных текстов программ. Основными графическими элементами диаграммы *компонентов* являются *компоненты*, *интерфейсы* и зависимости между ними.

В разработке диаграмм *компонентов* участвуют как системные аналитики и архитекторы, так и программисты. *Диаграмма компонентов* обеспечивает согласованный переход от логического представления к конкретной реализации проекта в форме программного кода. Одни *компоненты* могут существовать только на этапе компиляции программного кода, другие – на этапе его исполнения. *Диаграмма компонентов* отражает общие зависимости между *компонентами*, рассматривая последние в качестве отношений между ними.

- **Компоненты**

Для представления физических сущностей в языке *UML* применяется специальный термин – *компонент*.

Компонент (component) – физически существующая часть системы, которая обеспечивает реализацию классов и отношений, а также функционального поведения моделируемой программной системы.

Компонент предназначен для представления физической организации ассоциированных с ним элементов модели. Дополнительно *компонент* может иметь текстовый стереотип и *помеченные значения*, а некоторые *компоненты* – собственное графическое *представление*. *Компонентом* может быть *исполняемый код* отдельного *модуля*, командные файлы или файлы, содержащие интерпретируемые скрипты.

Компонент служит для общего обозначения элементов физического представления модели и может реализовывать некоторый набор *интерфейсов*. Для графического представления *компонента* используется *специальный символ – прямоугольник* со вставленными слева двумя более мелкими прямоугольниками (рис. 1). Внутри объемлющего прямоугольника записывается имя *компонента* и, возможно, дополнительная *информация*. Этот символ является базовым обозначением *компонента* в языке *UML*.

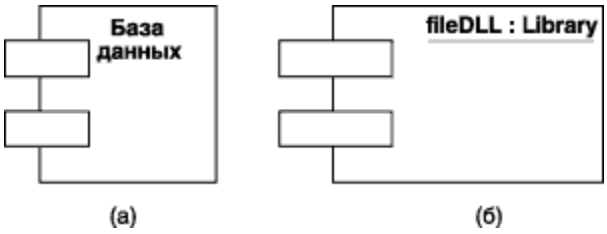


Рис. 1. Графическое изображение компонента

Графическое изображение *компонента* ведет свое происхождение от обозначения *модуля* программы, применявшегося некоторое время для отображения особенностей *инкапсуляции данных* и процедур.

Модуль (module) – часть программной системы, требующая памяти для своего хранения и процессора для исполнения.

В этом случае верхний маленький *прямоугольник* концептуально ассоциировался с данными, которые реализует этот *компонент* (иногда он изображается в форме овала). Нижний маленький

прямоугольник ассоциировался с операциями или методами, реализуемыми *компонентом*. В простых случаях имена данных и методов записывались явно в маленьких прямоугольниках, однако в языке *UML* они не указываются.

Имя *компонента* подчиняется общим правилам именования элементов модели в языке *UML* и может состоять из любого числа букв, цифр и знаков препинания. Отдельный *компонент* может быть представлен на уровне типа или экземпляра. И хотя его графическое изображение в обоих случаях одинаково, правила записи имени *компонента* несколько отличаются.

Если *компонент* представляется на уровне типа, то записывается только имя типа с заглавной буквы в форме: <Имя типа>. Если же *компонент* представляется на уровне экземпляра, то его имя записывается в форме: <имя *компонента* ‘:' Имя типа>. При этом вся строка имени подчеркивается. Так, в первом случае (рис. 1, а) для *компонента* уровня типов указывается имя типа, а во втором (рис. 1, б) для *компонента* уровня экземпляра – собственное имя *компонента* и имя типа.

Правила именования объектов в языке *UML* требуют подчеркивания имени отдельных экземпляров, но применительно к *компонентам* подчеркивание их имени часто опускают. В этом случае *запись* имени *компонента* со строчной буквы характеризует *компонент* уровня примеров.

В качестве собственных имен *компонентов* принято использовать имена исполняемых файлов, динамических библиотек, Web-страниц, текстовых файлов или файлов справки, файлов баз данных или файлов с исходными текстами программ, файлов скриптов и другие.

В отдельных случаях к простому имени *компонента* может быть добавлена *информация* об имени объемлющего пакета и о конкретной версии реализации данного *компонента*. Необходимо заметить, что в этом случае номер версии записывается как помеченное *значение* в фигурных скобках. В других случаях символ *компонента* может быть разделен на секции, чтобы явно указать имена реализованных в нем классов или *интерфейсов*. Такое обозначение *компонента* называется **расширенным.**

Поскольку *компонент* как элемент модели может иметь различную физическую реализацию, иногда его изображают в форме

специального графического символа, иллюстрирующего конкретные особенности реализации. Строго говоря, эти дополнительные обозначения не специфицированы в нотации языка *UML*. Однако, удовлетворяя общим механизмам расширения языка *UML*, они упрощают понимание диаграммы *компонентов*, существенно повышая наглядность графического представления.

Для более наглядного изображения *компонентов* были предложены и стали общепринятыми следующие графические стереотипы:

- Во-первых, стереотипы для *компонентов* развертывания, которые обеспечивают непосредственное выполнение системой своих функций. Такими *компонентами* могут быть динамически подключаемые библиотеки (рис. 2, а), Web-страницы на языке разметки гипертекста (рис. 2, б) и файлы справки (рис. 2, в).
- Во-вторых, стереотипы для *компонентов* в форме *рабочих продуктов*. Как правило – это файлы с исходными текстами программ (рис. 2, г).

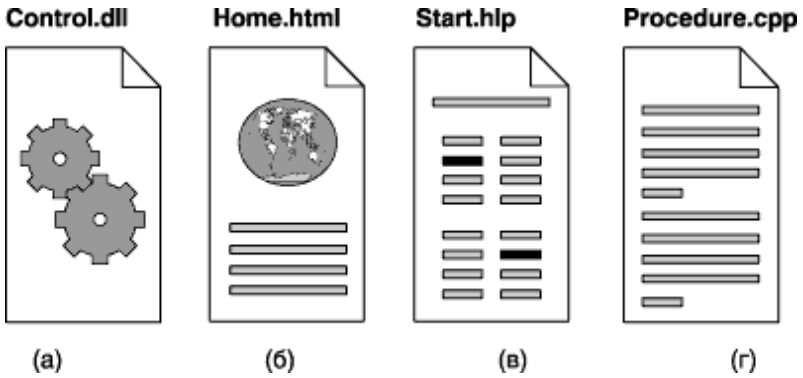


Рис. 2. Варианты графического изображения компонентов на диаграмме компонентов

Эти элементы иногда называют **артефактами**, подчеркивая при этом их законченное информационное содержание, зависящее от конкретной технологии реализации соответствующих *компонентов*. Более того, разработчики могут для этой цели использовать самостоятельные обозначения, поскольку в языке *UML* нет строгой нотации для графического представления артефактов.

Другой способ спецификации различных видов *компонентов* – указание текстового стереотипа *компонента* перед его именем. В языке *UML* для *компонентов* определены следующие стереотипы:

- <<file>> (файл) – определяет наиболее общую разновидность *компонента*, который представляется в виде произвольного физического файла.
- <<executable>> (исполнимый) – определяет разновидность компонента-файла, который является исполнимым файлом и может выполняться на компьютерной платформе.
- <<document>> (документ) – определяет разновидность компонента-файла, который представляется в форме документа произвольного содержания, не являющегося исполнимым файлом или файлом с исходным текстом программы.
- <<library>> (библиотека) – определяет разновидность компонента-файла, который представляется в форме динамической или статической библиотеки.
- <<source>> (источник) – определяет разновидность компонента-файла, представляющего собой файл с исходным текстом программы, который после компиляции может быть преобразован в исполнимый файл.
- <<table>> (таблица) – определяет разновидность *компонента*, который представляется в форме таблицы базы данных.

Отдельными разработчиками предлагались собственные графические стереотипы для изображения тех или иных типов *компонентов*, однако, за небольшим исключением они не нашли широкого применения. В свою *очередь* ряд инструментальных CASE-средств также содержат дополнительный набор графических стереотипов для обозначения *компонентов*.

- **Интерфейсы**

Следующим графическим элементом диаграммы *компонентов* являются *интерфейсы*. В общем случае *интерфейс* графически изображается окружностью, которая соединяется с *компонентом* отрезком линии без стрелок (рис. 3, а). При этом имя *интерфейса*, которое рекомендуется начинать с заглавной буквы «I», записывается рядом с окружностью. Семантически линия означает реализацию *интерфейса*, а наличие *интерфейсов* у *компонента* означает, что данный *компонент* реализует соответствующий набор *интерфейсов*.

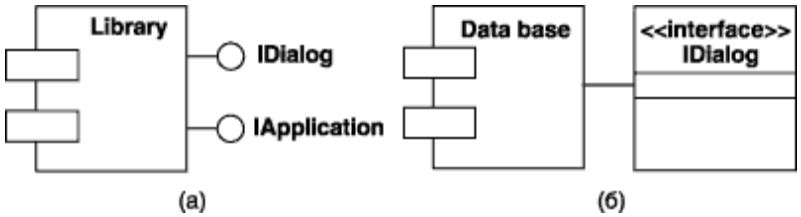


Рис. 3. Графическое изображение интерфейсов
на диаграмме компонентов

Кроме того, *интерфейс* на диаграмме *компонентов* может быть изображен в виде прямоугольника класса со стереотипом <<*interface*>> и секцией поддерживаемых операций (рис. 3, б). Как правило, этот вариант обозначения используется для представления внутренней структуры *интерфейса*.

При разработке программных систем *интерфейсы* обеспечивают не только совместимость различных версий, но и возможность вносить существенные изменения в одни части программы, не изменяя другие. Характер применения *интерфейсов* отдельными *компонентами* может отличаться.

Различают два способа связи *интерфейса* и *компонента*. Если *компонент* реализует некоторый *интерфейс*, то такой *интерфейс* называют *экспортируемым* или **поддерживаемым**, поскольку этот *компонент* предоставляет его в качестве сервиса другим *компонентам*. Если же *компонент* использует некоторый *интерфейс*, который реализуется другим *компонентом*, то такой *интерфейс* для первого *компонента* называется ***импортируемым***. Особенность *импортируемого интерфейса* состоит в том, что на диаграмме *компонентов* это *отношение* изображается с помощью зависимости.

- **Зависимости между компонентами**

В общем случае *отношение* зависимости также было рассмотрено ранее. *Отношение* зависимости служит для представления факта наличия специальной формы связи между двумя элементами модели, когда изменение одного элемента модели оказывает влияние или приводит к изменению другого элемента модели. *Отношение* зависимости на диаграмме *компонентов* изображается пунктирной линией со стрелкой, направленной от клиента или зависимого элемента к источнику или независимому элементу модели.

Зависимости могут отражать связи отдельных файлов программной системы на этапе компиляции и генерации объектного кода. В других случаях зависимость может указывать на наличие в

независимом компоненте описаний классов, которые используются в *зависимом компоненте* для создания соответствующих объектов. Применительно к диаграмме *компонентов* зависимости могут связывать *компоненты* и импортируемые этим *компонентом интерфейсы*, а также различные виды *компонентов* между собой.

В этом случае рисуют стрелку от компонента-клиента к *импортируемому интерфейсу* (рис. 4). Наличие такой стрелки означает, что *компонент* не реализует соответствующий *интерфейс*, а использует его в процессе своего выполнения. При этом на этой же диаграмме может присутствовать и другой *компонент*, который реализует этот *интерфейс*. *Отношение* реализации *интерфейса* обозначается на диаграмме *компонентов* обычной линией без стрелки.

Так, например, изображенный ниже фрагмент *диаграммы компонентов* представляет информацию о том, что *компонент* с именем *Control* зависит от *импортируемого интерфейса* IDialog, который, в свою *очередь*, реализуется *компонентом* с именем DataBase . При этом для второго *компонента* этот *интерфейс* является *экспортируемым*. Изобразить *связь* второго *компонента* DataBase с этим *интерфейсом* в форме зависимости нельзя, поскольку этот *компонент* реализует указанный *интерфейс*.

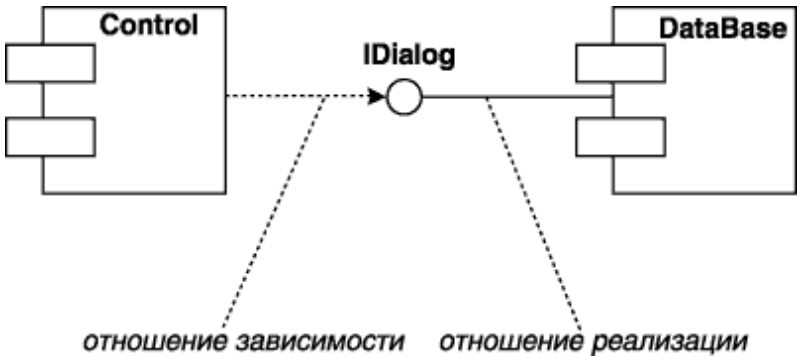


Рис. 4. Фрагмент диаграммы компонентов с отношениями зависимости и реализации

Другим случаем отношения зависимости на диаграмме компонентов является *отношение* программного вызова и компиляции между различными видами *компонентов*. Для рассмотренного фрагмента *диаграммы компонентов* (рис. 5) наличие подобной зависимости означает, что исполнимый *компонент Control*.exe использует или импортирует некоторую функциональность

компонента Library.dll, вызывает страницу гипертекста *Home*.html и *файл* помощи *Search*.hlp, а исходный текст этого исполнимого *компонента* хранится в файле *Control*.cpp. При этом характер отдельных видов зависимостей может быть отмечен дополнительно с помощью текстовых стереотипов.

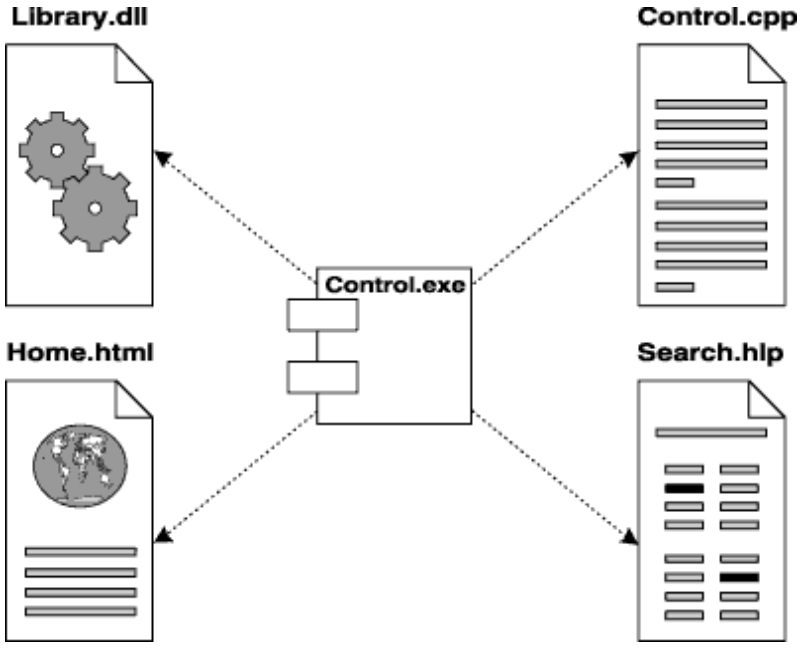


Рис. 5. Графическое изображение отношения зависимости между компонентами

На диаграмме компонентов могут быть также представлены отношения зависимости между *компонентами* и реализованными в них классами. Эта *информация* имеет *значение* для обеспечения согласования логического и физического представлений модели системы. Разумеется, изменения в структуре описаний классов могут привести к изменению этой зависимости. Ниже приводится фрагмент зависимости подобного рода, когда исполнимый *компонент Control*.exe зависит от соответствующих классов (рис. 6).

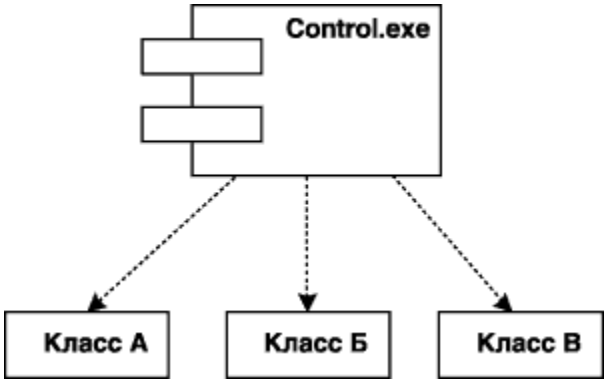


Рис. 6. Графическое изображение зависимости между компонентом и классами

В этом случае из *диаграммы компонентов* не следует, что классы реализованы данным *компонентом*. Если требуется подчеркнуть, что некоторый *компонент* реализует отдельные классы, то для обозначения *компонента* используется расширенный символ прямоугольника. При этом *прямоугольник компонента* делится на две секции горизонтальной линией. Верхняя секция служит для записи имени *компонента* и, возможно, дополнительной информации, а нижняя секция – для указания реализуемых данным *компонентом* классов (рис. 7).

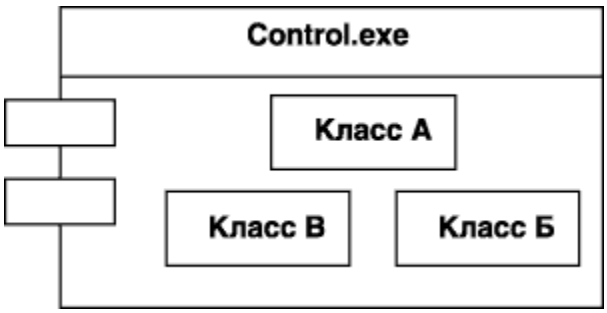


Рис. 7. Графическое изображение компонента с информацией о реализуемых им классах

В случае если *компонент* является экземпляром и реализует три отдельных объекта, он изображается в форме *компонента* уровня экземпляров (рис. 8). Объекты, которые находятся в отдельном компоненте-экземпляре, изображаются вложенными в символ данного *компонента*. Подобная вложенность означает, что выполнение *компонента* влечет за собой выполнение операций соответствующих объектов. При этом существование *компонента* в течение времени исполнения программы обеспечивает функциональ-

ность всех вложенных в него объектов. Что касается доступа к этим объектам, то он может быть дополнительно специфицирован с помощью видимости, подобно видимости пакетов.

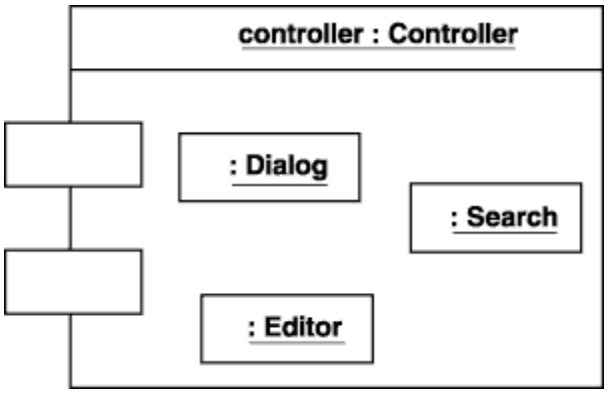


Рис. 8. Графическое изображение компонента-экземпляра, реализующего отдельные объекты

Для *компонентов* с исходным текстом программы видимость может означать возможность внесения изменений в соответствующие тексты программ с их последующей перекомпиляцией. Для *компонентов* с исполняемым кодом программы видимость может характеризовать возможность запуска на *исполнение* соответствующего *компонента* или вызова реализованных в нем операций или методов.

- **Рекомендации по построению диаграммы компонентов**

Разработка *диаграммы компонентов* предполагает использование информации не только о логическом представлении модели системы, но и об особенностях ее физической реализации. В первую *очередь*, необходимо решить, из каких физических частей или файлов будет состоять программная система. На этом этапе следует обратить внимание на такую реализацию системы, которая обеспечивала бы возможность повторного использования кода за счет рациональной декомпозиции *компонентов*, а также создание объектов только при их необходимости.

Общая *производительность* программной системы существенно зависит от рационального использования вычислительных ресурсов. Для этой цели необходимо большую часть описаний классов, их операций и методов вынести в динамические библиотеки, оставив в исполняемых *компонентах* только самые необходимые для инициализации программы фрагменты программного кода.

После общей структуризации физического представления системы необходимо дополнить модель интерфейсами и схемами *базы данных*. При разработке *интерфейсов* следует обращать внимание на согласование различных частей программной системы. Включение в модель схемы *базы данных* предполагает спецификацию отдельных таблиц и установление информационных связей между ними.

Завершающий этап построения *диаграммы компонентов* связан с установлением и нанесением на диаграмму взаимосвязей между *компонентами*, а также отношений реализации. Эти отношения должны иллюстрировать все важнейшие аспекты физической реализации системы, начиная с особенностей компиляции исходных текстов программ и заканчивая исполнением отдельных частей программы на этапе ее выполнения. Для этой цели можно использовать различные графические стереотипы *компонентов*.

При разработке диаграммы *компонентов* следует придерживаться общих принципов создания моделей на языке *UML*. В частности, в первую *очередь* необходимо использовать уже имеющиеся в языке *UML* и общепринятые графические и текстовые стереотипы. В большинстве типовых проектов этого набора достаточно для представления *компонентов* и зависимостей между ними.

Если же проект содержит физические элементы, описание которых отсутствует в языке *UML*, то следует воспользоваться механизмом расширения. В частности, можно применить дополнительные стереотипы для отдельных нетиповых *компонентов* или *помеченные значения* для уточнения отдельных характеристик *компонентов*.

Контрольные вопросы

1. Какие компоненты изображаются на диаграмме компонентов?
2. Каким символом изображается библиотека?
3. Как изображаются зависимости между компонентами?

4. Практическая работа №1.
Обоснование выбора технических средств

Целью работы является изучение порядка выбора технических средств ИС

Для выполнения практической работы № 1 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Основные технические средства

Орудия и средства (инструменты) производства являются необходимым компонентом любой технологии. Не являются исключениями информационные технологии, инструментальную базу которых образуют технические, программные и лингвистические средства. Инструментальные средства информационных технологий – совокупность технических программных и лингвистических средств, обеспечивающих реализацию информационных процессов. В составе технического обеспечения информационных технологий (с некоторой долей условности) различают следующие группы средств:

- компьютерная техника (ЭВМ и периферийные устройства), обеспечивающая электронное представление информации и автоматизацию всех информационных процессов;
- телекоммуникационные средства и системы, обеспечивающие передачу информации на расстояние;
- полиграфическая, копировальная и множительная техника, предназначенная для копирования и тиражирования информации; средства записи и воспроизведения аудиовизуальной информации (фото-, теле- видео-, киноизображения и звука);
- оргтехника (офисная техника), предназначенная для механизации и автоматизации конторского труда и управленческой деятельности.

Условность подобной классификации связана с нарушением единства основания и принципа непересекаемости делений: одни и те же средства (например, компьютерные) представлены во всех пяти группах; а копировально-множительная техника и средства связи ши- роко используются в офисе. Имеет смысл классифицировать технические средства в разрезе информационных процессов, для реализации которых они предназначены.

1. Средства сбора (регистрации) и ввода (записи) информации:

- персональные компьютеры – средства ввода текстовой, табличной, графической, аудиовизуальной и иной информации и записи ее на машиночитаемые носители;

- сканеры – средства оптического ввода – автоматического счи- тывания текста или изображения на бумажном носителе с последующим преобразованием его в формат, доступный для обработки и хранения в ЭВМ;

- дигизайтеры – средства бесклавиатурного ввода текста и графических изображений в ЭВМ;

- оргавтоматы – комплекс электромеханических и электронных средств автоматизации процесса составления, редактирования и изготовления текстовых и табличных документов;

- диктофоны – средства записи звуковой (преимущественно речевой) информации на различные носители (пленочные, магнитные, оптические) часто с целью преобразования ее в текстовую информацию;

- магнитофоны – средства записи аудиальной информации;

- фото-, кино-, теле-, видеокамеры – средства записи статичных и движущихся изображений и аудиовизуальной информации;

- измерительная техника (датчики, приборы, установки) – средства фиксации и измерения сигнала, извещающего о наступлении контролируемых событий и др.

2. Средства семантической и технической обработки информации:

- компьютеры (микрокомпьютеры, персональные, портативные, карманные, большие, сверхбольшие) – средства автоматизированной обработки цифровой информации;

- монтажное оборудование – средства обработки (монтажа) аудиальной, визуальной, аудиовизуальной, мультимедийной информации (цифровые и аналоговые устройства монтажа звука и изображения, монтажные столы);

- средства репрографии и оперативной полиграфии – оборудование для копирования и тиражирования документов (средства фотокопирования, диазокопирования, электрофотографии, термографии, электронно-искрового копирования, ризографического копирования, микрофильмирования; оборудование для гектографической, трафаретной, офсетной печати);

- средства технической обработки носителей информации (фальцевальные, перфорирующие и резательные машины, машины для уничтожения бумаг и др.);

- средства технической обработки документов (скрепляющее, склеивающее и переплетное оборудование, машины для нанесения защитных покрытий на документы);

- средства технической обработки корреспонденции (конвертовскрывающие, адресовальные, штемпелевальные, маркировальные машины и устройства, машины для уничтожения бумаг и т. п.) и др.

3. Средства хранения информации:

- компьютеры – средства хранения электронных документов и данных (серверы БД, файловые серверы, серверы приложений и др., локальные компьютеры);

- носители информации (бумажные, пленочные, магнитные, оптические, голографические, микроносители, перфоносители);

- канцелярские средства хранения документов (мультифоры, папки, планшеты, контейнеры и др.);

- картотеки (плоские, вертикальные, элеваторные, вращающиеся и др.) и картотечное оборудование;

- офисная мебель (шкафы, столы, стеллажи, сейфы и др.).

4. Средства поиска информации:

- автоматизированные ИПС (электронные каталоги, банки данных, электронные библиотеки, Web-ресурсы Интернет и др.);

- механизированные ИПС – ИПС, основанные на использования перфо- и микроносителей информации, осуществляющие поиск методом механической сортировки записей и кодов специальными устройствами (счетно-перфорационные машины, считывающие устрой-

ства, селекторы);

- ручные ИПС (карточные каталоги и картотеки, справочно-поисковый аппарат печатных изданий и др.).

5. Средства передачи информации:

- локальные, региональные, глобальные, корпоративные вычислительные сети – средства электронной связи, передачи на расстояние компьютерной информации;

- средства (аппаратура) электрической, радио-, телевизионной связи (телефонные, телеграфные, факсимильные аппараты, радио, телевизионные передатчики и приемники и др.).

- каналы связи – средства передачи акустических, оптических и электрических сигналов – делятся на беспроводные (радиосвязь, спутниковая связь) и проводные (кабельная связь: коаксиальны й кабель, незащищенная витая пара, оптоволоконный кабель);

- транспортные средства – средства механической доставки документов (тележки для перевозки документов внутри помещений, лифтовое оборудование, транспортеры, конвейеры, пневматическая почта, автомобильный и иной транспорт и др.).

6. Средства вывода информации: видеомониторы, мультимедийные проекторы, плазменные панели – средства отображения электронной информации;

- принтеры (матричные, струйные, лазерные) – печатающие устройства, обеспечивающие перенос машиночитаемой текстовой, числовой и графической информации на бумажный носитель;

- плоттеры (графопостроители) – устройства, обеспечивающие перенос машиночитаемой графической информации на бумажный носитель;

- аудиотехника – средства вывода звуковой информации (радиоприемники, проигрыватели, магнитофоны, аудиоплееры, музыкальные центры и др.);

- видеотехника – средства вывода аудиовизульной информации (телевизоры, домашние кинотеатры, кинопроекционная аппаратура, видеосистемы, DVD-плееры и др.).

Контрольные вопросы

1. Какие основные технические средства ИС вы знаете?
2. Какие средства используются для вывода информации?
3. Какие средства используются для ввода информации?

5. Практическая работа №2. Стоимостная оценка проекта

Целью работы является изучение порядка выбора технических средств ИС

Для выполнения практической работы № 2 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Стоимостная оценка проекта

Реальная оценка проектирования особенно важна уже в начале выполнения темы на этапе технического задания. При проведении предварительных расчетов следует иметь в виду, что центральное место в теме занимает программа (комплекс программ; математическая модель, реализованная на ЭВМ, и т. д.) и поэтому стоимостной анализ имеет свои особенности. Тема, организационно-экономическое обоснование которой является содержанием курсового или разделом дипломного проектов, может быть выполнена в различных организациях, различным составом исполнителей. Но в любом случае перечень исходных данных для расчета будет следующим (все цены и расценки берутся на один период времени):

1. Состав разработчиков проекта, чел., с указанием должностей и окладов, руб./мес.

2. Стоимость приобретения ЭВМ (указать тип ЭВМ), на которой выполнялась тема, руб.

3. Мощность ЭВМ, кВт.

4. Стоимость 1 кВт-ч электроэнергии для организации, где выполнялся проект, руб.

5. Тарифы коммунальных платежей для расчета затрат, связанных с содержанием помещения, или стоимость аренды 1 $м^2$ помещения под размещение вычислительной техники, руб.

Стоимостная оценка проекта $C_{ПР}$ включает следующие составляющие:

$$C_{ПР} = C_{ТР} + C_{ОТЛ} + C_{ЭВМ} + C_{П},$$

где $C_{ТР}$ – оценка труда разработчиков темы, руб.; $C_{ОТЛ}$ – затраты на отладку программного обеспечения по задаче, руб.; $C_{ЭВМ}$ – стоимость ЭВМ конкретного типа и других технических средств, если они приобретались специально для выполнения этой темы, руб.; $C_{П}$ – прочие затраты (их состав уточняется с преподавателем-

консультантом), руб. В конкретном расчете это могут быть фактические выплаты по командировкам, стоимость приобретения канцелярских принадлежностей для выполнения проекта, тиражирования документации, оплата услуг сторонних организаций и др.

Оценка труда разработчиков задачи может быть определена исходя из их должностных окладов (O) и периода проектирования $T^{ПР}$, Так как период проектирования или трудоемкость темы по различным методикам определяются или в человеко-днях, или в человеко-часах, то месячный должностной оклад соответственно переводится в дневной или часовой с учетом продолжительности рабочего дня и количества рабочих дней в месяце (в среднем в расчетах принимается 22 рабочих дня). Тогда

$$C_{ТР} = O \cdot T_{ПР} \cdot \left[(1 + \frac{П_Д}{100}) \cdot (1 + \frac{П_{ЕСН}}{100}) + \frac{П_{НК}}{100} \right],$$

где $П_Д$ – процент дополнительной заработной платы; $П_{ЕСН}$ – ставка единого социального налога, %; $П_{НК}$ – процент накладных расходов.

Конкретные значения $П_Д$, $П_Н$ и $П_{НК}$ на момент выполнения проекта уточняются с экономистом базовой организации.

В студенческих работах часто представляется обоснование темы, которую студент разрабатывал самостоятельно. Тогда при расчете $C_{ТР}$ принимается оклад младшего программиста, инженера-системотехника и т. д. той организации, где выполнялся проект.

Стоимостная оценка использования ЭВМ при проектировании проводится по следующей формуле:

$$C_{ОТЛ} = T_{ОТЛ} \cdot C_{МЧ},$$

где $T_{ОТЛ}$ – время отладки ЭВМ, ч. $T_{ОТЛ}$ определяется как сумма продолжительности работ календарного (сетевого) графика, которые предполагают работу на ЭВМ. Если работы календарного (сетевого) графика оценивались в человеко-днях, то с учетом продолжительности рабочего дня на ЭВМ (4 часа по нормам охраны труда), период отладки переводится в человеко-часах. $T_{ОТЛ}$ может определяться и по одной из методик; $C_{МЧ}$ – стоимость машино-часа работы ЭВМ, руб. Ее можно определить исходя из эксплуатационных расходов,

связанных с использованием вычислительной техники в течение определенного периода, по формуле

$$C_{МЧ} = \frac{З_{ЭКСПЛ}}{T_{Д} \cdot},$$

где $З_{ЭКСПЛ}$ – суммарные затраты за определенный период работы ЭВМ, которая использовалась при разработке проекта (месяц, квартал, год), руб.; $T_{Д}$ – действительный фонд времени работы ЭВМ за тот же период, ч.

В условиях изменения тарифов эксплуатационные затраты предпочтительнее рассматривать за более короткий период. Но так как отдельный учет затрат по работе ЭВМ ведется во всех организациях, даже продающих машинное время посторонним пользователям (например, электронные библиотеки), можно предложить укрупненный годовой расчет расходов:

$$З_{ЭКСПЛ} = З_{ТР} + З_{ПОМ} + З_{ЭН} + З_{АМ} + З_{М} + З_{Р},$$

где $З_{ТР}$ – затраты на оплату труда обслуживающего ЭВМ персонала, руб./год; $З_{ПОМ}$ – затраты на содержание (в частности, на аренду) помещения под размещение вычислительной техники, руб./год; $З_{ЭН}$ – затраты на силовую электроэнергию, руб./год; $З_{АМ}$ – затраты на амортизацию (амортизационные отчисления), руб./год; $З_{М}$ – затраты на материалы (носители информации), руб./год; $З_{Р}$ – затраты на ремонт, руб./год.

В расчете $З_{ТР}$, руб./год, следует иметь в виду, что по штатному расписанию в качестве обслуживающего персонала ЭВМ можно принимать инженеров-электроников, техников и операторов различных категорий, которые поддерживают работоспособность ЭВМ и по должностным обязанностям обслуживают несколько ЭВМ, а не заняты разработкой конкретных задач:

$$З_{ТР} = \sum_{i=1}^{Ч} 12 \cdot O_i \cdot \left[(1 + \frac{П_{Д}}{100})(1 + \frac{П_{ЕСН}}{100}) + \frac{П_{НК}}{100} \right],$$

где O_i – доля месячного оклада i-го работника за обслуживание одной ЭВМ, руб./мес.; $П_Д$ – процент дополнительной заработной платы; $П_{ЕСН}$ – ставка единого социального налога, %; $П_{НК}$ – процент накладных расходов.

$П_Д$, $П_{ЕСН}$ и $П_{НК}$ уточняются с преподавателем-консультантом.

Вычислительная техника размещается на определенных площадях. Если нет фактических данных по затратам, связанным с содержанием помещений, то этот вид расходов может определяться через среднюю ставку арендных платежей (ее значение уточняется с преподавателем-консультантом на момент выполнения проекта), руб./год,

$$З_{АР} = S_П \cdot C_А,$$

где $S_П$ – площадь помещения, м^2; $C_А$ – средняя ставка арендных платежей, руб./ м^2.

Подобный расчет $З_{ПОМ}$ может выполняться на основе тарифов коммунальных платежей за использование площадей, расчет приводится далее в примере организационно-экономического обоснования.

Если отладка программного обеспечения проводилась на ЭВМ в дисплейном классе, лаборатории, отделе предприятия и т. д., где находятся несколько машин, то $З_{АР}$ и $З_{ПОМ}$ пересчитываются на одну машину.

Для расчета затрат, связанных с потреблением электроэнергии, как и для расчета себестоимости машино-часа, необходимо определить действительный фонд времени работы ЭВМ за рассматриваемый период $Т_Д$. При расчете годовых затрат $Т_Д$ определяется исходя из номинального фонда $Т_Н$. Номинальный фонд времени рассчитывается конкретно для каждого календарного года в соответствии с приказом Министерства труда России. Например, для 2004 года номинальный фонд времени в днях:

$$Тн=К_Д-К_В-К_П,$$

где $К_Д$ – количество дней в году; $К_В$ – количество выходных дней в году; $К_П$ – количество праздничных дней в году, не совпадающих с выходными.

$$Тн = 366-52\cdot2-11=251 \text{ дн.}$$

В частности, номинальный фонд времени на 2004 год, выраженный в часах с учетом сокращения на один час четырех предпраздничных дней,

$$T_н = 247 \cdot 8 + 4 \cdot 7 = 2004 \text{ ч/год.}$$

Тогда

$$T_Д = T_Н \cdot (1 - \frac{\alpha}{100}),$$

где α – процент потерь рабочего времени, связанных с профилактикой и ремонтом ЭВМ, выбираемый из интервала 2–7 %.

Затраты на силовую электроэнергию, руб./год,

$$З_{ЭН} = M \cdot T_Д \cdot Ц_Э \cdot K_И,$$

где M – паспортная мощность ЭВМ, кВт; $T_Д$ – действительный годовой фонд времени, ч; $Ц_{ЭН}$ – цена одного кВт-ч энергии на момент выполнения расчета, руб.; $K_И$ – коэффициент интенсивного использования мощности, $K_И = 0{,}9$.

Остальные затраты определяются следующим образом исходя из балансовой стоимости оборудования:

$$S_{БАЛ} = Ц_{ПР} \cdot (1 + \frac{\eta}{100}),$$

где $Ц_{ПР}$ – цена приобретения ЭВМ, руб.; η – коэффициент, характеризующий дополнительные затраты, связанные с доставкой, монтажом и наладкой оборудования. В настоящее время эти затраты могут составлять 2–7 % стоимости ЭВМ.

Общая схема расчета $З_{АМ}$, $З_М$, $З_Р$ (т.е. $З_i$) следующая:

$$З_i = S_{БАЛ} \cdot \frac{H_i}{100},$$

где $S_{БАЛ}$ – балансовая стоимость ЭВМ, руб.; H_i – соответствующий определенному виду затрат норматив (конкретно – $H_{АМ}$, $H_Н$, $H_Р$), % (прил. 4).

Результаты расчета эксплуатационных расходов заносятся в табл. 5.1.

Таблица 5.1

Смета эксплуатационных расходов по работе ЭВМ

№ п/п	Наименование статей затрат	Значение, руб.
1.	**Затраты на оплату труда обслуживающего персонала, в том числе:** основная заработная плата дополнительная заработная плата единый социальный налог накладные расходы	
2.	**Затраты на содержание (на аренду) помещения**	
…	**. . . . . .**	
6.	**Затраты на ремонт**	
	Итого эксплуатационных расходов (*З* $_{ЭКСПЛ}$)	

После определения себестоимости машино-часа работы ЭВМ, стоимости отладки и остальных затрат себестоимость проектирования оформляется в табл. 5.2.

Если при проектировании производился выбор определенного типа ЭВМ из нескольких, то может быть выполнен дополнительный раздел «Экономическое обоснование выбора ЭВМ». Для этого по вышеприведенной схеме просчитываются эксплуатационные расходы по работе всех рассматриваемых типов ЭВМ за одинаковый период времени, определяются стоимости часа работы каждой машины и делается вывод об экономичности использования той или иной вычислительной техники на основе минимального значения этих показателей.

Таблица 5.2

Себестоимость проектирования

№ п/п	Наименование статей затрат	Значение, руб.
1.	**Расходы по оплате труда разработчиков темы, в том числе:** основная заработная плата дополнительная заработная плата единый социальный налог накладные расходы	
...	**. . . . . .**	
4.	**Прочие проектные расходы**	
	Итого себестоимость проекта (*С*$_{ПР}$)	

Контрольные вопросы

1. Из каких основных статей состоит расчет проектирования ИС?
2. Какие основные расходы существуют при эксплуатации ИС?

6. Практическая работа №3. Построение и обоснование модели проекта

Целью работы является изучение порядка построения и обоснования модели проекта

Для выполнения практической работы № 3 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Разработка архитектуры ПО

1 Влияние стандартов на процессы архитектурного проектирования

Осмысленная разработка программных продуктов – комплексное понятие, состоящее из *множества* высокоинтеллектуальных и взаимосвязанных процессов, таких как:

- анализ требований;
- проектирование;
- кодирование;
- тестирование;
- и т. д.

Каждая из обозначенных активностей должна основываться на базе соответствующих стандартов и лучших практик. Дело в том, что подобный тип документов создается на основе наиболее успешного опыта применения специфического вида деятельности, рабочей технологии или артефактов, эталонное описание которых он содержит. Стандарт концентрирует все лучшее и наиболее эффективное, с точки зрения практического достижения конечного результата, подтвержденного массивом статистических данных.

Перед тем как внедрять стандарты в процессы конкретной организации, следует соответствующим образом адаптировать их под реалии конкретной организации.

Должны быть учтены такие аспекты, как:

- ресурсная база организации;
- количество выделенных для деятельности ресурсов, их доступность, квалификация (если речь идет о специалистах), надежность и т. д.;
- сегмент/домен/направление деятельности компании;

- сфера деятельности, с учетом специфических требований к ним со стороны отраслевых/государственных/международных регуляторных органов;
- степень влияния информационных технологий на поддержку и развитие бизнеса;
- инновационность компании и степень участия в инновациях информационных технологий;
- и т. д.

Если тема применения и использования стандартов в целях повышения эффективности деятельности Вас заинтересовала, то мы рекомендуем обратиться к моделям зрелости *CMMI*, методологии *SPICE* и другим подобным документам. В настоящей лекции мы больше к данному аспекту, влияющему на качество архитектуры и архитектурного проектирования, возвращаться не будем.

Перечень стандартов и их содержание на конкретном предприятии должно определяться на стадии планирования.

В момент планирования перечня стандартов, который действительно необходим, целесообразно руководствоваться принципом «золотой середины». Чрезмерное количество внедряемых стандартов требует достаточно большого количества ресурсов. Следует осознавать – внедрение стандарта делает процесс более унифицированным и качественным за счет его регламентации, но при этом, в отдельных аспектах, понижает степень его гибкости. Если компания пришла к пониманию необходимости внедрения определенной практики или стандарта эволюционным путем, то это облегчает и упрощает процесс внедрения и использования соответствующего регламента. Если изменения носят революционный характер, то попытка привнести в организацию избыточное количество стандартов может носить катастрофический характер и остаться «на бумаге», что не является целью их внедрения.

Стандартизация должна обеспечить постепенное повышение качества конкретных процессов (*анализ*, *документирование*, *кодирование* и т. д.), обеспечить прозрачность, однозначность и не противоречивость процессов, минимизацию или устранение появления возможных ошибок при разработке архитектур и программных продуктов.

В качестве отступления мы приведем следующий пример.

Требования к процедуре сертификации авиационных продуктов предусматривают, что в процессе разработки специфического (для авиационных нужд) программного обеспечения, должны использоваться как *минимум* три стандарта:

- Стандарт на работу с требованиями;

Должен описывать методы, правила и инструменты, применяемые для сбора, разработки и управления требованиями, их возможные форматы и нотации.

- Стандарт на разработку архитектуры программного продукта;

Содержит правила, формирующие архитектуру программного продукта, приемлемые и неприемлемые методы её разработки, описывает возможные функциональные и не функциональные ограничения (запрет на использование рекурсии или максимальная вложенность вызовов).

- Стандарт процесса кодирования;

Регламентирует исходный код программы, ссылки на описания используемого языка программирования, его синтаксис, ограничение, желательные и нежелательные конструкции языка и прочие правила, касающиеся разработки кода программы;

Стандарты определяют рабочие принципы, которые должны действовать на всем протяжении жизненного *цикла* программного продукта. Например, если в стандарте зафиксировано, что нельзя вставлять комментарии в программный код (комментарии следует располагать в строго определенном месте) то, все комментарии, расположенные не в соответствии с этим правилом, должны быть перенесены или удалены. Несоответствие принятым стандартам грозит возникновением потенциальных ошибок. Если стандарт *по* каким-то причинам не соблюдается, то все «нарушения» должны устраняться. Если стандарт требует доработок в связи с изменившимися первоначальными предпосылками его создания, то его необходимо либо дорабатывать, либо менять. Слепое следование стандартам губительно, так же, как и постоянные попытки обойти их *по* необоснованным причинам.

Проектирование архитектуры программных продуктов – это одна из активностей, для которых должны быть приняты определенные стандарты деятельности. При этом, современные требова-

ния к архитектурам предусматривают создание достаточно гибких и адаптивных архитектур.

Первое, «верхнеуровневое» проектирование, должно удовлетворять цели достаточно свободной и абстрактной реализации архитектуры, но при этом учитывать жесткие требования стандартов. Поэтому требования к программным продуктам принято делить на 2 типа критичности – высокоуровневые (функциональные) и низкоуровневые (нефункциональные) требования. Гибкость архитектур реализуется за счет разной степени принципиальности учета требований и их последующей реализации в программном коде будущих информационных систем. Требования к нефункциональным характеристикам должны быть более четкими и однозначными.

Типы требований мы уже немного обсуждали ранее и будем продолжать это делать *по* ходу нашего курса, изучая различные аспекты архитектур и архитектурного проектирования. Сейчас отметим, что подход к работе с ним должен быть также соответствующим образом стандартизован. За счет стандартов важно достичь согласованности между типами требований в момент их согласования друг с другом в виде отдельных компонентов архитектуры или программного продукта. Это позволит добиться непротиворечивого результата, в виде оптимальной архитектуры и эффективного программного обеспечения.

Большинство современных стандартов, описывающих работу с требованиями, выдвигают следующие характеристики, которым должен отвечать каждый из них:

- конкретность/измеримость;
- корректность/обоснованность;
- соответствие функциональности программного продукта;
- возможность последующей верификации;
- уникальность;
- последовательность;
- непротиворечивость;
- возможность декомпозиции;
- изменяемость с минимальными затратами;
- трассируемость.

На сегодняшний день существует множество программных продуктов, которые позволяют автоматизировать и унифицировать процесс разработки и последующего отслеживания требований.

Применение специализированного инструментария, поддерживающего принятые стандарты, в большинстве случаев, положительно сказывается на качестве процессов архитектурного проектирования, разрабатываемых архитектурах и программных продуктов в целом, за счет того, что их использование позволяет выявить те программные компоненты и детали, которые подвержены изменениям меньше остальных. Именно они и составят основу разрабатываемых архитектур, которая будет наименее гибкой, а те требования, которые чаще других изменяются, будут лежать в основе достаточно гибкой функциональности.

Но изменения, иногда вносятся и в «скелет».

Принципиальное следование всем принятым стандартам возможно только тогда, когда речь идет об устоявшихся отлаженных процессах архитектурного проектирования и разработки информационных систем, которые достигли своей наивысшей точки качественного развития. Утопия.

Принципы и темп развития современного мира определяют его черты.

Программные продукты, как часть этих черт должны быть в состоянии поддерживать соответствующую динамику и темпы. Элементы информационных систем и их *архитектура* предназначены для удовлетворения требований к разрабатываемым продуктам. Стоит понимать, что требования не только могут изменяться, но и качественно трансформироваться, переходя из категории нефункциональных в функциональные и наоборот.

Можно привести следующий пример:

В случае, если речь идет о стадии внедрения *ПО*, то такая характеристика как сопровождаемость воспринимается пользователями, как некритичная и не значимая с их точки зрения, но как только начинается стадия промышленной эксплуатации программного продукта, *затраты* на его поддержку и развития сразу дают о себе знать. Хорошо, если компания имеет достаточный для этого *ресурс*, но если это не так, то *поддержка* процессов и соответствующего для их выполнения уровня данных «кочует» на плечи бизнес пользователей и сразу переходит в разряд функциональных.

Все вышесказанное явно свидетельствует о значимости и необходимости стандартизированного подхода к созданию новых, инновационных, прорывных информационных продуктов, которые в

своей основе будут иметь надежные, качественные, производительные и масштабируемые архитектуры, которые позволят создавать функционально гибкие продукты.

Без четких, ясных требований к выполняемым функциям создать *программное обеспечение* невозможно. Требования – фундамент каждой разработки.

Процессы разработки должны начинаться с процесса разработки требований и вестись в соответствии с планом, корректируемым *по* »ходу».

2 Инструментарий разработки и моделирования требований к процессам и архитектуре программных продуктов

Для перехода от набора требований, порой разрозненных, к стройной и логичной архитектуре программного продукта, нужно использовать специализированный *инструментарий*. В нашем случае это специальные технологические *средства разработки* и моделирования процессов, на основе выявленных требований, которые позволят создать единое видение разрабатываемой архитектуры, с учетом определенных правил представления информации, принятой для конкретного проекта *по* созданию программного продукта или процесса архитектурного проектирования.

Тему правил, методов, методологий, в соответствии с которыми разрабатывается *архитектура* программных продуктов, мы будем подробно рассматривать в лекции №4.

Как было отмечено ранее, архитектурное проектирование информационных систем – это область деятельности, которая имеет много общего с архитектурным проектированием в строительстве.

«Строительство» – старший собрат информационных технологий. Поэтому было бы оптимально, для процессов создания программных продуктов многое необходимое для развития данной профессиональной области, почерпнуть, поинтересовавшись тем, как в строительстве успешно решались схожие задачи. В гражданском и промышленном строительстве языком описания архитектуры являются архитектурно-строительные чертежи, объемные прототипы и модели, текстовые описания возводимых объектов. Младший собрат поступил правильно и перенял необходимый *опыт* у старшего. В качестве средств разработки и моделирования характеристик информационных продуктов, в архитектурном проектировании используются различные языки, которые принято на-

зывать нотациями, а инструментами, которые поддерживают работу с нотациями – *CASE* средствами.

Из современных нотаций можно выделить следующие:

- Блок-схемы (схемы алгоритмов)

Данный тип схем (графических моделей) визуализирует разрабатываемые алгоритмы или процессы. В описании блок-схем принято отдельные стадии процессов изображать в виде блоков (отсюда и соответствующее название данной нотации). Блоки различаются в зависимости от семантики, содержащейся в представляемой ими форме. Отображаемые блоки соединены между собой линиями, имеющими направление и определенную алгоритмами последовательность.

В широком значении термина «блок-схемы» он является метанотацией. Различные спецификации «блок-схем» получили свое определенное назначение и форму представления необходимой информации, в зависимости от специфики каждой конкретной нотации, но при этом общие правила построения моделей наследуются от правил построения «блок-схем».

- DFD-диаграмма (диаграмма потоков данных)

DFD – это нотация структурного анализа.

В данной нотации принято описывать внешние, по отношению к разрабатываемому продукту:

- источники данных;
- адресаты данных;
- логические функции;
- потоки данных;
- хранилища данных.

Диаграмма потоков данных – это один из первых и основных инструментов структурного анализа и проектирования верхнеуровневой архитектуры программных продуктов, существовавших до последующего широкого распространения UML.

В основе данной нотации находится методология проектирования и метод построения модели потоков данных проектируемой информационной системы. В соответствии с соответствующей методологией проектирования модель системы идентифицируется как иерархия диаграмм потоков данных, которые представляют собой последовательный процесс преобразования данных, с момента их

поступления в систему, до момента представления информации конечному (или псевдоконечному) пользователю.

Диаграммы потоков данных разработаны для определения основных процессов или модулей программного продукта. Далее, в целях комплексного представления разрабатываемой архитектуры, они детализируются диаграммами нижних уровней представления информации (DFD) и процессов (IDEF). Подобный принцип отображения данных продолжается, реализуя многоуровневую иерархию подчиненных и взаимосвязанных диаграмм. В результате будет достигнут нужный уровень декомпозиции, на котором процессы становятся абсолютно понятными и прозрачными.

В современных процессах разработки программного обеспечения подобный подход к проектированию программных продуктов практически не применяется, по причине смещения акцентов создания информационных систем от структурного к объектно-ориентированному подходу.

Данная методология анализа и проектирования на сегодняшний день эффективно используется в бизнес-анализе и соответствующих дисциплинах, по причине ее наглядности и понятности для стэйкхолдеров.

- ER-диаграмма (диаграмма сущность-связь)

ER – это тип нотации, задача которой состоит в создании формальной конструкции, описывающей и визуализирующей верхнеуровневое представление бизнес объектов будущей системы, их атрибуты и связь между ними, в виде основных элементов, которые будут использоваться в качестве фундамента для проектирования таблиц будущей базы данных программного продукта. Таким образом, ER-диаграмма представляется как концептуальная модель создаваемой информационной системы, в терминах конкретной предметной области. Данный тип диаграмм помогает выделить ключевые сущности и определить связи между ними. На сегодняшний день существует достаточно мощный инструментарий, который может позволить выполнить преобразование созданной ER-диаграммы в конкретную схему базы данных на основе выбранной модели данных.

- EPC-диаграмма (диаграмма событийной цепочки процессов)

Событийная цепочка процессов – это тип нотации, задача которой заключается в создании моделей бизнес процессов. Она используется для моделирования, анализа и реорганизации бизнес-процессов. Ее отличие, от ранее рассмотренных, состоит в том, что EPC применяют для создания функциональных моделей, имитирующих конкретные процессы, реализуемые в определенном программном продукте.

История развития данного типа диаграмм связана с разработкой одноименного метода в рамках работ по созданию методологии ARIS (Architecture of Integrated Information Systems – Архитектура интегрированных информационных систем).

Диаграмма EPC оформляется в виде упорядоченной последовательности событий и функций. Каждая функция при этом должна определяться начальными и конечными событиями, могут быть указаны участники, исполнители, материальные и информационные потоки, сопровождающие отдельные функции. Дополнительную ценность диаграммам EPC придает тот факт, что она может использоваться для настройки программных продуктов типа ERP, в части создания или улучшения бизнес-процессов.

- BPMN-диаграммы

Один из самых распространенных типов нотаций на сегодняшний день – это BPMN (Business Process Model and Notation).

EPC, ER, BPMN – это не просто нотации, а методы реализации конкретных моделей процессов. BPMN по своему назначению более уместно сравнивать с диаграммами EPC. Причина этого заключается в том, что они имеют схожую цель – функциональное моделирование бизнес процессов. Рассматриваемая диаграмма самый новый, из приведенных в обзоре, принятый стандарт моделирования бизнес процессов. Главное позиционируемое преимущество диаграммы BPMN – понятная всем стэйкхолдерам разрабатываемой архитектуры и программного продукта визуализация моделируемых процессов.

Нотация имеет достаточно ясный инструментарий, позволяющий моделировать как относительно простые, так и сложные бизнес-процессы. Это достигается за счет применения двух групп элементов:

- Первая группа (simple notation) состоит из основных элементов BPMN, которые удовлетворяют требованиям создания простой

графической нотации. Подавляющее большинство бизнес-процессов моделируются с использованием элементов данной группы.

- Вторая группа (powerful notation) содержит полный «пакет» элементов BPMN. В него кроме основных блоков входят специфические комплексные модули, представляющие различные типы и виды элементов, необходимых для моделирования самых сложных процессов.

Подобная преемственность инструментария групп BPMN позволяет удовлетворять требованиям комплексной нотации и управлять более сложными ситуациями моделирования.

Оптимально смоделированные диаграммы BPMN могут применяться для настройки, рефакторинга и реинжиниринга бизнес процессов во многих современных информационных системах класса ECM.

- UML-диаграммы

UML (Unified Modeling Language) – не просто нотация или группа нотаций, а язык графического описания для объектного моделирования.

UML – это язык широкого профиля, который создан для формирования серии нотаций, поддерживающих полный цикл разработки, как архитектуры, так и функциональности программных продуктов, реализуемых по принципам объектно-ориентируемого программирования.

Широкая распространенность и повсеместное применение данного языка привели к тому, что он стал открытым стандартом, использующим графические обозначения для создания моделей систем, которые принято называть UML-моделями. Специфика UML не предполагает ориентацию на описание архитектуры, так как основное его назначение заключается в определении, визуализации, проектировании, документировании, преимущественно программных продуктов, бизнес процессов и структур данных.

Язык UML получил популярность за счет жесткой связки с популярной и распространенной методологией разработки и внедрения программных продуктов RUP, в соответствии с которой организована работа многих консалтинговых и производственных компаний отрасли информационных технологий. Не смотря на множество преимуществ, UML не является панацеей и абсолютным универсу-

мом в процессах проектирования архитектур и программных продуктов.

Применять и разбираться в диаграммах UML могут только узкоспециализированные специалисты области проектирования программных систем. Вовлечь в обсуждение и согласование разрабатываемой архитектуры информационного продукта, задокументированного на UML, стэйкхолдеров, представителей не направления информационных технологий, будет достаточно сложно.

UML, в отличие от ER и BPMN диаграмм, поддерживает генерацию не просто отдельной функциональности, привязанной к специфичным типам информационных систем, а целостного программного кода на основании сформированных UML-моделей.

- EIP-диаграммы

Самая специфичная и наименее распространенная нотация из всех рассматриваемых в данной лекции.

EIP (Enterprise Integration Patterns) диаграммы – это набор из 65 шаблонов диаграмм, которые предназначены для описания специализированных процессов взаимодействия между программными продуктами. Специфика данной нотации состоит в том, что она разработана для описания конкретных процессов, обеспечивающих интеграционное взаимодействие между приложениями и функциональность программных продуктов, ориентированных на обмен сообщениями между информационными системами.

Шаблоны интеграции приложений и необходимая функциональность, описание которых поддерживают EIP диаграммы, встроены в множество современных специализированных открытых программных продуктов и доступны для применения в специализированных интеграционных процессах.

На сегодняшний день, в силу своей специфичности, данный вид нотации не имеет широкого распространения, но, учитывая темпы роста популярности интеграционных процессов, можно спрогнозировать, что в ближайшее время интерес к EIP-диаграммам вырастет.

- Модели компонентов, программных продуктов и функциональных процессов

Модели, как таковые, не являются нотациями, а представляют собой набор необходимых для реализаций процессов/подпроцессов, представленных в виде нескольких взаимодополняющих друг друга

нотаций, с целью моделирования конечного результата. Говоря о модели, со структурной точки зрения, корректно будет сказать, что это «метанотация», включающая в себя несколько основных нотаций, реализация диаграмм которых запланирована для качественной разработки конкретной архитектуры.

Таким образом, модель – это абстракция, которая фокусирует субъект, работающий с ней на основных, критичных для реализации, характеристиках конечного продукта. Из примеров CASE средств, способных реализовать и представить нотации в виде модели можно выделить ARIS, UML.

Так же возможна схема, когда исполнитель разработает необходимые диаграммы в разных CASE средствах, после чего компилирует модель самостоятельно. В таких случаях дальнейшее сопровождение и развитие модели затруднительно (сложность сопровождения, развития, управления моделью) и целиком и полностью лежит на «плечах» ответственных за это специалистов.

- Прототипы

Когда мы говорили о моделях, то отметили, что модель – это следующий, качественный шаг вперед к реализации программного обеспечения, по сравнению с диаграммами, реализуемыми нотациями.

После того, как реализованы несколько моделей требований, можно приступать к реализации прототипов, если это предусмотрено и спланировано в проекте разработки программного продукта.

Прототип представляет собой «черновой» вариант конечной реализации программного продукта или его компонента, разработанного с учетом определенных ограничений для демонстрации заинтересованным сторонам на определенной стадии разработки только архитектуры или программного продукта в целом.

Цель разработки прототипа – подтвердить ожидания стэйкхолдеров от реализации конечного продукта и согласовать её функциональность. Прототипы принято реализовывать в специализированном программном коде, поэтому их следует воспринимать как часть процесса разработки, но при этом не рассчитывать на то, что прототип будет являться частью будущей архитектуры или информационной системы.

Прототипы, как правило, разрабатываются достаточно быстро с помощью специализированного инструментария, освоение кото-

рого требует гораздо меньше времени и сил, чем освоение определенного языка программирования.

Жизненный цикл прототипа заканчивается в тот момент, когда ожидания стэйкхолдеров подтверждены и разработанный облик (не более) будущего продукта запланирован к реализации.

- Интерфейсы:

Работа по созданию интерфейсов – это отдельная область разработки программных продуктов, которая не определяет архитектуры, но в большинстве случаев зависит от неё. Именно поэтому мы решили упомянуть её в конце списка, факультативно.

Эта область требует к себе довольно много внимания и сил. Она определяет многие функциональные характеристики программных продуктов (эргономика, дизайн интерфейсов, воспринимаемость и удобство работы с информацией и т. д.).

Значимость корректных и эффективных интерфейсов все больше и больше растет для разнообразных программных продуктов. Интерфейс представляет собой «верхушку», с помощью которой пользователь будет взаимодействовать с «нутром» программного продукта.

Важно, чтобы уже на ранних стадиях разработки интерфейсов пользователь имел представление о том, с чем и каким образом ему придется работать и постепенно привыкал, при возможности корректируя его или свои ожидания от него.

На текущий момент есть множество инструментов для создания интерфейсов разной степени сложности и детальности. Мы выделим следующие Balsamiq mockup, Axure RP и пр.

Каждый крупный *программный продукт* или его *модуль*, который состоит из совокупности компонентов и связей между ними, должен быть детально описан в соответствующий момент процесса архитектурного проектирования. Описание должно быть выполнено *по* стандартизированным правилам организации или проекта, с использованием необходимых нотаций, для данного конкретного случая.

По принципам программной инженерии и здравой логики, которая находится в основе процессов разработки информационных систем, любая подсистема или *компонент* в дальнейшем может выступать в качестве составного элемента более масштабной системы. Поэтому, в архитектурном проектировании должно обязательно содержаться подробное описание укрупнённых частей системы, вы-

полненных с помощью *CASE* средств и нотаций, согласованных друг с другом и обеспечивающих последующую преемственность.

Важно, чтобы выбранная *нотация* или серия нотаций поддерживали существующий процесс архитектурного проектирования, способствовали его развитию и совершенствованию и были «в силах» поддержать соответствующую фиксацию функциональных, нефункциональных и прочих требований к разрабатываемой архитектуре и программному продукту в целом.

Необходимо осознавать, что нет единой нотации или языка проектирования, который мог бы быть решением всех поставленных задач процесса проектирования. Сегодня существует достаточное количество универсальных инструментов и средств, но ни один из них не может обеспечить весь спектр возникающих задач проектирования.

Для того, чтобы создать достаточно комплексную и взаимосвязанную среду проектирования архитектур, моделей и функциональности программных продуктов нужно использовать инструменты и диаграммы, которые смогут поддержать процессы разработки, в зависимости от специфики конкретной задачи, но при этом должен существовать минимальный набор необходимых диаграмм, который должен присутствовать в каждом проекте *по* разработке программного продукта не смотря на его масштабы и *значимость*.

3 О функциональных и нефункциональных требованиях к архитектуре и функциональности программного обеспечения

Обсуждение различных типов и видов требований мы начали в предыдущей лекции.

В этой части мы уделим больше внимания рассмотрению функциональных и нефункциональных требований к архитектуре программного обеспечения.

Принято думать, что *архитектура* формируется под воздействием, прежде всего, нефункциональных требований, которые, в большинстве случаев, отсутствуют в проектируемых функциональных и бизнес моделях систем, но начинают свой *жизненный цикл* в процессе реализации программных продуктов. Попробуем обосновать, что это не совсем верно.

Многие современные ученые области инженерии требований высказывают мнение о том, что не бывает нефункциональных требований.

Нефункциональные требования – это абсолютно *функциональные требования*, которые описывают функции системы с точки зрения «каких-то» стэйкхолдеров, о которых забыли в процессе сбора и анализа требований. Вы знаете какое-нибудь *программное обеспечение*, в процессе разработки которого активно участвовали будущие специалисты групп сопровождения, тестирования, развития его архитектуры и функциональности? Как часто проектируя программные продукты о многих его характеристиках, не значимых на первый взгляд для бизнес стейкхолдеров, но критичных для других пользователей, просто забывают или намеренно игнорируют, считая их не значимыми и второстепенными.

Ранее мы показали, что члены группы поддержки, тестирования, системные администраторы и некоторые другие технические специалисты являются такими же стэйкхолдерами, как и будущие основные бизнес пользователи системы. Игнорирование их требований может привести к тому, что фаза внедрения пройдет достаточно успешно, но вот последующее сопровождение и развитие системы может быть достаточно проблематичным.

Основная задача заключается в том, что для современного бизнес мира «последующее» значит то, о чем можно забыть сегодня и не вспоминать до тех пор, пока забытое не напомнит о себе. В случае с нефункциональными требованиями такой подход к работе может оказаться слишком дорогостоящим. Как только информационный продукт перейдет в стадию промышленной эксплуатации и над ним начнут работать группы специалистов, которые должны будут поддерживать достаточно высокий уровень сервисной поддержки и развития системы, может оказаться, что его реализация не включает в себя множество разнообразных технических аспектов. Ведь именно от таких аспектов зависит качество продукта, обеспечивающее *оптимальность* бизнес процессов, моделей и данных. Довольно распространенная ситуация, когда на ранних фазах создания программных продуктов о таких характеристиках как *надежность*, *быстродействие*, *безопасность* многие специалисты и стэйкхолдеры, вовлеченные в процесс проектирования архитектуры и функциональности просто не задумываются, отдавая их на дальнейшую проработку и реализацию разработчикам. Все бы хорошо, но для разработчиков существует множество факторов, в соответствии с которыми, модели программных компонентов, которые не

имеют четких требований будут реализованы не так как задумывалось при составлении специализированных документов, а так, как разработчик «сможет». Это «сможет» будет определяться квалификацией, инструментарием и, конечно же тем количеством времени, которое у него/них будет в наличии, а его, как известно, практически всегда не хватает. Таким образом, получается следующий парадокс – качество продукта будет напрямую зависеть не от стэйкхолдеров, а от специалистов, для которых важность продукта, который они разрабатывают, не очевидна. Стэйкхолдерам важно получить законченную функциональность в сроки согласованных *работ*, а иногда даже намного, намного ранее и получить дополнительное время на реализацию качественных атрибутов, *значимость* которых определяется только в процессе разработки. Именно поэтому многие характеристики разрабатываемого программного продукта и его архитектуры, скрытые от внимательного взора руководства, необходимо обозначать в активностях, предваряющих стадии проектирования и разработки. Многое на этой стадии зависит от опыта и мастерства системного архитектора и его команды, от профессионализма которых будет зависеть стратегический успех информационной системы, разрабатываемой для бизнес-необходимостей конкретной организации.

Этим вступлением мы постарались показать то, что требования не бывают второстепенными или незначимыми. Незначимое сегодня окажется жизненно необходимым завтра. Рамки *работ* над требованиями должны быть спланированы и ясно понятны всем стэйкхолдерам, участвующим в проекте.

После того, как мы определились с тем, что для создания качественной архитектуры не должно быть разного отношения к функциональным и не функциональным требованиям. Все требования необходимо рассматривать с точки зрения их критичности *по* отношению к архитектуре программного продукта.

Высокоуровневое назначение программного продукта – приносить выгоду его владельцу за счет автоматизации труда (интеллектуального, физического, монотонного и т. д.) человека, а в идеале за счет полной замены человеческого ресурса и фактора, если это возможно.

Здесь будет уместно сравнение со строением человека, в котором есть множество органов, каждый из которых отвечает за опре-

деленную функцию и каждый развивается не только как самостоятельная *единица*, но и как часть целого. В тот момент, когда определенный орган дает сбой и не может поддерживать свою целевую функциональность, при условии что это не отражается на деятельности всего организма, то можно считать, что это не критично, но если весь организм выходит из строя на определенный период, то важно понять в чем состоит источник проблемы и устранить его, а если подойти системно, то не допустить подобной возможности. Таким образом, те элементы органов, которые влияют на их полную функциональность и взаимосвязаны с другими, являются наиболее приоритетными для исследования и оптимального содержания и развития. Подобными компонентами в программном продукте являются компоненты, наиболее критичные для рассмотрения с точки зрения архитектуры и сбора требования.

Но при разработке определенной информационной системы, классификация требований необходима для целей дальнейшей их трассировки и управления ими. Традиционно выделяют 2 основные группы описание которых и более подробную литературу *по* работе с которыми наши коллеги найдут в соответствующей литературе.

Сначала мы рассмотрим *функциональные требования*.

Функциональные требования описывают «поведение» системы и информацию, с которой система будет работать. Они описывают возможности системы в терминах поведения или выполняемых операций

Цель использования данной группы требований (как следует из названия) заключается в регламентации возможностей и соответствующем им поведении разрабатываемого программного продукта.

Функциональные требования должны отвечать на вопрос – каким образом должны быть алгоритмизированы процессы информационного продукта, чтобы взаимодействие между пользователем и системой удовлетворяло потребности стэйкхолдеров?

Именно с помощью функциональных требований, в большинстве случаев, определяются рамки *работ по* процессам, сопровождающим цикл создания программных продуктов. Данный тип требований устанавливает:

- цели разрабатываемого функционала;
- задачи, которые должны быть выполнены для достижения поставленной цели;

- сервисы, поддерживающие выполнение задач;
- … (и многое другое, касающееся непосредственно функциональных возможностей и особенностей программного продукта).

На сегодняшний день существует множество подходов к разработке и фиксации функциональных требований. Кратко рассмотрим наиболее популярные из них:

- Классический подход

Суть классического подхода состоит в разработке требований с помощью итеративной работы с верхнеуровневыми требованиями стэйкхолдеров, и постепенной детализации до уровня понятного разработчикам. Это наиболее изученный и популярный подход, который подробно описан в современной литературе. Мы можем порекомендовать для изучения книги К. Вигерса, которые уже упоминались нами ранее.

Дополнительно отметим, что в подобном подходе основная тяжесть создания полноценного документа, охватывающего весь *программный продукт*, ложится на сотрудника, ответственного за сбор, *анализ* и синтез информации. При современной динамике изменений бизнес процессов и данных, поступающих от внешних и внутренних аспектов бизнеса, непосредственно влияющих на требования стэйкхолдеров, классический подход становится наименее эффективным. Именно поэтому в последнее время водопадные модели разработки программного обеспечения заменяются «гибкими» подходами (agile), цель которых в быстром и эффективном решении возникающих задач, даже если следующая будет противоречить предыдущей.

- Use cases (Варианты использования);

В этом подходе *функциональные требования* записываются с помощью системы специализированных правил, которые, к примеру, должны фиксироваться следующим образом: «программный продукт должен обеспечить учетчику возможность формирования акта выдачи бланков строгой отчетности».

Если определить максимальное количество возможных вариантов использования разрабатываемого программного продукта предполагаемыми целевыми пользователями, то мы получим достаточно полные *функциональные требования*.

Но, при попытке тотального применения к работе над функциональными требованиями этого подхода существует *вероят-*

ность того, что отдельные, незначимые для определенных стэйкхолдеров, потребности будут не учтены. В этом случае существует *вероятность* упустить суть конкретных функциональных требований, которые, как правило, скрыты от постороннего взгляда. Чтобы этого не произошло, важно использовать в обработке зафиксированных use cases классический подход, в рамках которого должен быть проведен *анализ*, систематизация и синтез информации. Это поможет выявить истинное назначение предполагаемого к реализации функционала и не разрешить отдельных деталей.

Нефункциональные требования, в *дополнение* к функциональным, направленны на обеспечение технической целостности разрабатываемого функционала и поддержку характеристик реализуемого программного обеспечения, которые необходимы для создания оптимальной архитектуры.

Они регламентируют внутренние и внешние условия функционирования программного продукта. Выделяют следующие основные группы нефункциональных требований:

- атрибуты качества;
  - безопасность;
  - надежность;
  - производительность;
    - скорость и время отклика приложения;
    - пропускная способность workflow;
    - количество необходимой оперативной памяти;
- ограничения;
  - платформа реализации архитектуры и программного продукта;
  - тип используемого сервера приложений.

Нефункциональные требования описывают условия, которые не относятся к поведению и функциональности системы, но обеспечивают их на уровне компонентов архитектуры программного продукта.

Ранее описанная методология use cases может быть применена для сбора и анализа нефункциональных требований. При взаимодействии со стэйкхолдерами необходимо фиксацию каждого правила подытоживать фиксацией нефункциональных требований, выраженную в виде количественной характеристики определенного параметра: «программный продукт должен обеспечить учетчику воз-

можность формирования акта выдачи бланков строгой отчетности за время не более 0,5 с после того, как поступила соответствующая команда.

В этой части нашего курса мы постарались достаточно подробно рассказать о требованиях, которые оказывают свое влияние на архитектуру и функциональность программных продуктов, в том числе достаточно подробно рассмотрели функциональные и нефункциональные требования. Но предложенный объем является необходимым и достаточным только для того, чтобы у наших коллег сложилось понимание того, какую роль играют требования в процессах проектирования архитектуры и реализации программных продуктов. Более основательное изучение темы инженерии требований изложены в специализированных курсах и литературе, к которым мы и рекомендуем Вам обратиться.

4 Прочие виды требований

Любое предприятие, которое осознанно пришло к необходимости создания или применения программного продукта, обладающего оптимальной архитектурой, для определенных условий и аспектов деятельности, можно считать сложной организационно-технической системой, которая имеет определенные цели и реализует конкретные функциональные задачи. При предпосылках использования программных продуктов в условиях организационного управления, *процесс исполнения* каждой задачи нуждается в конкретных ресурсах для реализации цели функционирования предприятия.

Сложная организационно-программная система состоит из иерархических уровней и поддерживающих их программных структур (исполнители бизнес процессов и необходимые программные продукты), каждый из которых постоянно расширяется и развивается. Из этого следует, что развитие каждого отдельного элемента (новая версия информационной системы с исправленными ошибками и дополнительными функциональными возможностями или развитие конкретного исполнителя) будет приводить к улучшению характеристик программного продукта, используемого на конкретном предприятии. Но целенаправленное развитие можно будет обеспечить только при условии соблюдения требования взаимодействия всех составных компонентов организации.

Запланированное функционирование бизнес процессов, поддерживаемых архитектурой программных продуктов, необходимо обеспечить требованиями к ресурсам, которые могут быть кратко выражены следующими аспектами:

- Время

Время на обучение пользователей, стабилизацию реализованных и внедренных принципов работы, поддерживающих достижение целей бизнес процессов, время на осознание организацией «хозяйнических» инстинктов по отношению к программному продукту

- Финансы

Финансовые ресурсы должны обеспечить необходимый уровень поддержки, выраженные в оптимальном количестве квалифицированных исполнителей их профессиональной мотивации, требуемом уровне технической оснащенности и пр.

- Данные и информация

Для того, чтобы результат деятельности соответствовал ожиданиям стейкхолдеров требуются данные надлежащего качества, на основе которых стало бы возможным достижение поставленных целей и качественной трансформации данных в значимую для бизнеса информацию. В дальнейшем повторное использование подобной информации позволит повысить уровень ценности тактических и стратегических процессов и способствовать повышению уровня экспертности и зрелости компании в целом.

Если перейти от организационных требований к реализации архитектуры программных продуктов, общая специфика большинства современных практик процессов создания архитектур, то можно выявить ряд общих особенностей, которые заключаются в:

1. Преимущественной ориентированности на водопадные модели процессов реализации архитектур и программных продуктов.
2. Подавляющем фокусировании на архитектуре программного продукта и практически полном игнорировании того факта, что система включает в себя не только информационно-программные компоненты, но и другие аспекты (технические средства, персонал), которые также должны быть рассмотрены и учтены при проектировании решения.
3. Отделение активности технико-экономического обоснования реализации программного продукта от разработки бизнес–процессов и разработки архитектуры системы.

После детального и компетентного изучения специализированных методик (с которыми мы познакомимся чуть позднее) результатом их осмысления можно сформулировать следующие требования к процессам проектирования и реализации архитектуры, в частности, и программных продуктов в целом:

- проектирование и создание архитектуры, бизнес–анализ, технико-экономическое обоснование создания продукта, моделирование процессов должны быть неразрывно связаны друг с другом и изменение одного из составляющих должно запускать процессы анализа влияния и управления изменениями;
- сущность процессов проектирования и разработки архитектуры, функциональности программных продуктов должна быть преимущественно итерационной.

В практике создания программных продуктов, каждая реализуемая информационная система имеет собственную специфику, выраженную определенными условиями и факторами, которые имеют различную природу возникновения и силу влияния на архитектуру и функциональность. Часть из них мы рассмотрели в предыдущей лекции, часть из них мы будем рассматривать в следующей.

Активность инженерии требований, которая ставит своей целью работу с требованиями – это отдельная область, которая нуждается в подробном рассмотрении. Если Вас заинтересовало данное направление отрасли информационных технологий, вы сможете самостоятельно приступить к ее изучению, используя общедоступные источники информации.

Мы же далее будем рассматривать только те аспекты, которые оказывают достаточно сильное влияние на предмет нашего курса.

5 Зависимости и связи между различными видами требований, функциональности и архитектуры программного обеспечения

После того, как мы определились с основными видами требований, которые необходимо фиксировать, формализовывать и реализовывать в процессах проектирования и разработки для целей реализации архитектуры и функциональности программных продуктов, стоит задуматься о соблюдении целостности архитектуры, необходимой для:

- создания оптимального функционала;

- реализации значимых аспектов разрабатываемого программного обеспечения;
- разработки кроссбизнес-процессов, поддерживаемых архитектурой и функциональностью программного продукта;
- взаимосвязи между различными стадиями процессов разработки и внедрения программных продуктов (интеграция, миграция данных и пр.);
- и др.

Каждое требование, зафиксированное в целях создания информационной системы, *по* ходу стадий проектирования, разработки, тестирования и внедрения программного продукта должно быть трансформировано в определенный программный *модуль*, тестовую процедуру, *пункт* инструкции пользователя и т.д. – это один из основных постулатов создания качественного и адекватного программного продукта, который называется трассирование (трассировка).

Трассирование представляет собой процесс или *атрибут* в рамках реализации информационный системы, который обеспечивает *связь* между его элементами и функциональными процессами. Трассировка должна способствовать установлению связи между:

- всеми видами требований;
- функциональными и нефункциональными процессами;
- результатами;
- необходимой отчетностью.

Оптимально выстроенный процесс трассирования должен ясно и однозначно позволять понять что (?), откуда (?), каким образом (?) вышло, и куда (?), и в каком виде (?) поступило.

К примеру можно привести стандарты создания программных продуктов для авиационной промышленности. В них зафиксировано, что *команда*, задействованная в разработке программного продукта должна в любой момент времени жизненного *цикла* программного продукта уметь отследить цепочку от результатов тестирования, к самим тестам, от тестов к бинарному коду, от бинарного кода к исходному коду, от исходного кода к низкоуровневым требованиям, от низкоуровневых требований к архитектуре, от архитектуры к высокоуровневым требованиям, от высокоуровневых требований к системным требованиям и в обратную сторону. Но, как правило, в стандартах не определяется способ трассирования. Это

оставляет системным архитекторам или другим специалистам, ответственным за создание информационных систем, определенную свободу действий, регламентированную принципами здравого смысла и необходимостью получения оптимального результата для конкретных заданных условий.

Когда мы говорим о трассируемости документов, содержащих требования к разрабатываемому продукту, то вполне достаточно обеспечить начальную *связность* на первых этапах разработки за счет ссылочности создаваемых документов и уникальной идентификации каждого конкретного требования. Если же мы начнем обсуждать непосредственно процессы проектирования, разработки требований и кода программного обеспечения, то необходимо осветить процессы валидации, верификации и тестирования.

Под валидацией понимается процесс, направленный на *доказательство* того, что верхнеуровневые требования стэйкхолдеров будут полностью удовлетворены и покрыты в разработанной функциональности программного обеспечения.

Верификация является активностью процесса валидации, цель которой проверка и последующее достижение соответствия между требованием и реализованными архитектурой и функциональность программного обеспечения.

Тестирование представляет собой «сугуботехническую» часть активности верификации, направленную на *испытание* программного продукта.

При выполнении тестирования должны быть решены 2 основные задачи:

1. Демонстрация соответствия требований реализации программного продукта.

2. Выявление ситуаций и аспектов, в которых функциональность и архитектура является несоответствующим зафиксированных в документах требованиям с последующим выполнением п. 1.

Необходимо четко представлять, что ни один из представленных процессов, сам *по* себе, не обеспечит качественной трассируемости требований с момента их фиксации до момента промышленной эксплуатации программного продукта. Даже наиболее развитый и формализованный процесс тестирования программного обеспечения не позволяет однозначно, точно и полностью выявить все несоответствия и установить адекватную функциональность, необходи-

мую для достижения поставленных результатов. Причины этого заключаются в множестве разнообразных факторов, основной из которых это пресловутый *человеческий фактор*, к более подробному рассмотрению которого мы периодически возвращаемся в процессе изучения нашего курса.

Только в совокупности, используя свойства, порожденные эффектом эмерджентности, можно добиться достижения системных результатов от процессов трассировки.

При выполнении каждой стадии работы над требованиями и разрабатываемой архитектуры и функциональности информационной системы, в процессы проектирования и реализации должны быть вовлечены все стэйкхолдеры, которые при необходимости смогут прояснить проблемную ситуацию, в рамках зафиксированных требований. Относительно частая связь между разработчиками и стэйкхолдерами будет способствовать:

- повышению прозрачности процессов проектирования и разработки для влиятельных стэйкхолдеров;
- эффективности процессов взаимодействия и оптимизации связей между различными членами команды, заинтересованными сторонами и другими вовлеченными в общие процессы взаимодействия пользователей.

Эти факторы в совокупности приведут к повышению степени доверия между исполнителями и заказчиками и, как следствие, будут позитивно влиять на степень удовлетворенности пользователей и эффективность разрабатываемого программного продукта.

Тема трассирования, как многие рассматриваемые в нашем курсе, является полноценной областью сферы информационных технологий, для успешного изучения которых требуется затратить определенное время, при этом уже обладая определенным багажом знаний и опыта, полученного во время практической работы над созданием информационных продуктов.

Те аспекты, которые мы осветили, являются основными и позволят нашим коллегам быть готовым к необходимости создания и участия в процессах трассировки.

6 Риски реализации архитектуры и методы управления ими

На текущий момент, когда в процессе изучения нашего курса мы подошли к стадии обсуждения рисков, важно немного более

подробно рассказать о том, что именно мы будем понимать под рисками реализации архитектуры.

Риск – это потенциальная возможность наступления вероятного события/явления или их совокупности, которые могут вызывать определенное влияние (негативное или позитивное) на осуществляемую *деятельность* или реализуемый продукт.

Под рисками реализации архитектуры мы будем понимать совокупность факторов, управленческих решений и других аспектов, которые могут оказать влияние на конечный результат разработки архитектуры и функциональности программных продуктов. Мы не ставим себе целью предложить Вам способ или инструмент, который позволит справиться с рисковой составляющей, но опишем то, как стоит воспринимать конкретный риск и что надо сделать, чтобы быть готовым к работе с ним.

Как правило, риски реализации архитектуры можно ассоциировать со следующими причинами их возникновения:

- Попытка создания оптимальной архитектуры на основе уже существующей, но не удовлетворяющей ожидания заказчиков.

Довольно популярная ситуация, складывается в процессе вынужденного реинжиниринга, когда без обоснованного анализа причинно-следственных связей, разработчик или архитектор спонтанно принимают решение о переделке какого-то компонента. В данном случае риск заключается в том, что идея не перешла на стадию «выдержанного» предложения, а сразу начала претворяться в код. Через какое-то время становится понятно, что какой-то элемент остался недостаточно проработан. В таком случае, достаточно «благостном», приходится отказываться от реализации, но если какие-то части уже внедрены в функционирование и ждут своих доработок, то приходится в режиме пожарника искать «обходные» варианты и усугублять имеющуюся информационную систему в целом

- В процессе реализации архитектуры программного продукта меняются ключевые участники команды, задействованной в работе над ней.

В том случае, когда новые члены команды, приступающие к работе над архитектурой и функциональностью программного продукта первым делом, не изучив причин реализации и их следствия, начинают недоумевать по поводу выбранного способа реализации. Им кажется, что все сделано не оптимально, не очевидно, слишком

сложно. После того, как выполнены определенные доработки, в процессе тестирования начинают появляться дыры в уже разработанном решении и архитектура из целостного монолита становится «костыльной». В итоге систему требуется переделать почти полностью. Чем старше система, тем таких ситуаций больше

Описанные ситуации, как и множество других, особенно выделяются в длительных проектах.

На текущий день, в области создания информационных систем накоплен достаточно большой *опыт*, на основе которого возможно снижение, *локализация* или устранение вероятности наступления подобных ситуаций. Среди рисков реализации архитектуры следует выделить следующие, на которых мы сконцентрируемся в дальнейшем:

- Риск смены разработчика.
- Риск ошибочно принятого архитектурного решения.

Эти риски сильно влияют на *предсказуемость* процессов перепроектирования и разработки программного продукта. Что характерно, наступление первого усиливает *вероятность* наступления второго.

Риск смены разработчика связан с потерей ключевой информации об архитектуре и функциональности *по* причинам изменения ключевого лица. Многие мысли, воплощенные в коде, которые не нашли отражение на бумаге, не всегда понятны тому, кто не является их автором. Подобных рисков можно избежать полным документированием выбранных решений до того момента, как они будут применены. Это позволит относительно безболезненно единовременно выдержать смену любых участников команды. Ключевым фактором успеха становится *поддержка* системности процессов документирования.

Причины рисков ошибочно принятых архитектурных решений заключаются в недостаточной трассируемости требований от стадии сбора информации, к стадии проектирования или же в недостаточной компетенции лиц, принимающих важные технические решения.

Для того чтобы максимально обезопасить *программный продукт* от подобного рода рисков, необходимо как можно более комплексно подойти к решению некоторых вопросов еще на уровне формирования идеи. Все важные решения, связанные с проектированием, разработкой и последующим изменением архитектуры и

функциональности программного продукта необходимо мысленно приостанавливать и стремиться задокументировать в виде концептуально проработанного предложения. Подобный подход помогает не только в принятии обоснованного и согласованного, с уже созданными компонентами информационно системы, решения, но и является своего рода дополнительной тренировкой аналитических способностей.

После того как идея получает свою материализацию на бумаге, в большинстве случаев, она может немного трансформироваться и становится более обдуманной и обоснованной.

Этот метод позволяет подойти к необходимым доработкам с позиции необходимых и достаточных трудозатрат и не тратить ценные ресурсы на «холостые ходы».

Каждое зафиксированное изменение в последующей стадии анализа и проектирования должно быть соотнесено с уже существующей архитектурой с помощью активности анализа влияния. Для этого есть множество разнообразных инструментов от общения с ответственными разработчиками, до построения диаграммы причинно следственных связей, на которых должно быть продемонстрировано то, как предлагаемое изменение будет влиять на *программный продукт* в целом. Если конструктивных моментов больше и изменение будет влиять положительно, то незамедлительно стоит переходить к его реализации, в противном случае имеет смысл отложить её или поискать более выигрышные варианты изменений.

В процессе работы над рисками важно помнить первоначальные цели, достижение которых является главным результатом нашей деятельности. Иногда бывает, что увлекшись поиском обходных решений и пытаясь минимизировать величину возможного ущерба, проектная *команда*, занятая поиском наиболее выигрышных ответов, отклоняется в сторону, и *архитектура* становится расплывчатой и слишком гибкой, что впоследствии отражается на её характеристиках и функциональности программного продукта. Кроме того, необходимо учитывать имеющиеся ресурсы, технологии и возможности бизнеса. *Программный продукт* в процессе своего функционирования должен поддерживаться и обеспечиваться в соответствии с тем количеством ресурсов, которое предусматривается для конкретной стадии его использования.

В том случае, если *по* ходу разработки программного продукта первоначальные предпосылки его создания изменились, подобная ситуация является возможностью для появления или увеличения количества и степени проявления всевозможных рисков (не только архитектурных, но и бизнес, операционных и т. д.).

Поэтому важно контролировать:

- Детали реализации архитектуры и функциональности информационной системы.
- Влияние принятых изменений:
  - не только на сам продукт, но и на величину ресурсов, необходимых для разработки и сопровождения (как бизнес, так и технических) принятых изменений.
- Ограничения, которые сопровождают:
  - разработку программного продукта;
  - жизненный цикл программного продукта с момента его выхода на рынок.

Контрольные вопросы

1. Какие технологии для проектирования архитектуры ПО существуют?
2. По каким принципам начинают разрабатывать архитектуру ПО?

7. Лабораторная работа №4. Установка и настройка системы контроля версий с разграничением ролей

Целью работы является изучение порядка работы с системой контроля версий GIT. Результатом практической работы является отчет, в котором должны быть приведено описание созданного репозитария, демонстрация приемов работы с ним.

Для выполнения лабораторной работы № 4 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Изучение работы в системе контроля версий

Git – это набор консольных утилит, которые отслеживают и фиксируют изменения в файлах (чаще всего речь идет об исходном коде программ, но вы можете использовать его для любых файлов на ваш вкус). С его помощью вы можете откатиться на более старую версию вашего проекта, сравнивать, анализировать, сливать изменения и многое другое. Этот процесс называется контролем версий. Существуют различные системы для контроля версий. Вы, возможно, о них слышали: SVN, Mercurial, Perforce, CVS, Bitkeeper и другие.

Git является распределенным, то есть не зависит от одного центрального сервера, на котором хранятся файлы. Вместо этого он работает полностью локально, сохраняя данные в папках на жестком диске, которые называются репозиторием. Тем не менее, вы можете хранить копию репозитория онлайн, это сильно облегчает работу над одним проектом для нескольких людей. Для этого используются сайты вроде github и bitbucket.

- Установка

Установить git на свою машину очень просто:

Windows – мы рекомендуем git for windows, так как он содержит и клиент с графическим интерфейсом, и эмулятор bash.

- Настройка

Итак, мы установили git, теперь нужно добавить немного настроек. Есть довольно много опций, с которыми можно играть, но мы настроим самые важные: наше имя пользователя и адрес электронной почты. Откройте терминал и запустите команды:

```
git config --global user.name «My Name»
git config --global user.email myEmail@example.com
```

Теперь каждое наше действие будет отмечено именем и почтой. Таким образом, пользователи всегда будут в курсе, кто отвечает за какие изменения – это вносит порядок.

- Создание нового репозитория

Как мы отметили ранее, git хранит свои файлы и историю прямо в папке проекта. Чтобы создать новый репозиторий, нам нужно открыть терминал, зайти в папку нашего проекта и выполнить команду init. Это включит приложение в этой конкретной папке и создаст скрытую директорию .git, где будет храниться история репозитория и настройки.

Создайте на рабочем столе папку под названием git_exercise. Для этого в окне терминала введите:

```
$ mkdir Desktop/git_exercise/
$ cd Desktop/git_exercise/
$ git init
```

Командная строка должна вернуть что-то вроде:

```
Initialized empty Git repository in
/home/user/Desktop/git_exercise/.git/
```

Это значит, что наш репозиторий был успешно создан, но пока что пуст. Теперь создайте текстовый файл под названием hello.txt и сохраните его в директории git_exercise.

- Определение состояния

status – это еще одна важнейшая команда, которая показывает информацию о текущем состоянии репозитория: актуальна ли информация на нём, нет ли чего-то нового, что поменялось, и так далее. Запуск git status на нашем свежесозданном репозитории должен выдать:

```
$ git status
On branch master
Initial commit
Untracked files:
(use «git add ...» to include in what will be committed)
hello.txt
```

Сообщение говорит о том, что файл hello.txt неотслеживаемый. Это значит, что файл новый и система еще не знает, нужно ли следить за изменениями в файле или его можно просто игнорировать. Для того, чтобы начать отслеживать новый файл, нужно его специальным образом объявить.

- Подготовка файлов

В git есть концепция области подготовленных файлов. Можно представить ее как холст, на который наносят изменения, которые нужны в коммите. Сперва он пустой, но затем мы добавляем на него файлы (или части файлов, или даже одиночные строчки) командой add и, наконец, коммитим все нужное в репозиторий (создаем слепок нужного нам состояния) командой commit.

В нашем случае у нас только один файл, так что добавим его:

```
$ git add hello.txt
```

Если нам нужно добавить все, что находится в директории, мы можем использовать

```
$ git add -A
```

Проверим статус снова, на этот раз мы должны получить другой ответ:

```
$ git status
On branch master
Initial commit
Changes to be committed:
(use «git rm --cached ...» to unstage)
new file: hello.txt
```

Файл готов к коммиту. Сообщение о состоянии также говорит нам о том, какие изменения относительно файла были проведены в области подготовки – в данном случае это новый файл, но файлы могут быть модифицированы или удалены.

- Коммит (фиксация изменений)

Коммит представляет собой состояние репозитория в определенный момент времени. Это похоже на снапшот, к которому мы можем вернуться и увидеть состояние объектов на определенный момент времени.

Чтобы зафиксировать изменения, нам нужно хотя бы одно изменение в области подготовки (мы только что создали его при помощи git add), после которого мы может коммитить:

```
$ git commit -m «Initial commit.»
```

Эта команда создаст новый коммит со всеми изменениями из области подготовки (добавление файла hello.txt). Ключ -m и сообщение «Initial commit.» – это созданное пользователем описание всех изменений, включенных в коммит. Считается хорошей практикой делать коммиты часто и всегда писать содержательные комментарии.

- Удаленные репозитории

Сейчас наш коммит является локальным – существует только в директории .git на нашей файловой системе. Несмотря на то, что сам по себе локальный репозиторий полезен, в большинстве случаев мы хотим поделиться нашей работой или доставить код на сервер, где он будет выполняться.

- 1. Подключение к удаленному репозиторию

Чтобы загрузить что-нибудь в удаленный репозиторий, сначала нужно к нему подключиться. В нашем руководстве мы будем использовать адрес https://github.com/tutorialzine/awesome-project, но вам посоветуем попробовать создать свой репозиторий в GitHub, BitBucket или любом другом сервисе. Регистрация и установка может занять время, но все подобные сервисы предоставляют хорошую документацию.

Чтобы связать наш локальный репозиторий с репозиторием на GitHub, выполним следующую команду в терминале. Обратите внимание, что нужно обязательно изменить URI репозитория на свой.

```
# This is only an example. Replace the URI with your own repository address
$ git remote add origin https://github.com/tutorialzine/awesome-project.git
```

Проект может иметь несколько удаленных репозиториев одновременно. Чтобы их различать, мы дадим им разные имена. Обычно главный репозиторий называется origin.

- 2. Отправка изменений на сервер

Сейчас самое время переслать наш локальный коммит на сервер. Этот процесс происходит каждый раз, когда мы хотим обновить данные в удаленном репозитории. Команда, предназначенная для этого – push. Она принимает два параметра: имя удаленного репозитория (мы назвали наш origin) и ветку, в которую необходимо

внести изменения (master – это ветка по умолчанию для всех репозиториев).

```
$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 212 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/tutorialzine/awesome-project.git
* [new branch] master -> master
```

В зависимости от сервиса, который вы используете, вам может потребоваться аутентифицироваться, чтобы изменения отправились. Если все сделано правильно, то когда вы посмотрите в удаленный репозиторий при помощи браузера, вы увидете файл hello.txt

- 3. Клонирование репозитория

Сейчас другие пользователи GitHub могут просматривать ваш репозиторий. Они могут скачать из него данные и получить полностью работоспособную копию вашего проекта при помощи команды clone.

```
$ git clone https://github.com/tutorialzine/awesome-project.git
```

Новый локальный репозиторий создается автоматически с GitHub в качестве удаленного репозитория.

- 4. Запрос изменений с сервера

Если вы сделали изменения в вашем репозитории, другие пользователи могут скачать изменения при помощи команды pull.

```
$ git pull origin master
From https://github.com/tutorialzine/awesome-project
* branch master -> FETCH_HEAD
Already up-to-date.
```

Так как новых коммитов с тех пор, как мы склонировали себе проект, не было, никаких изменений доступных для скачивания нет.

- Ветвление

Во время разработки новой функциональности считается хорошей практикой работать с копией оригинального проекта, которую называют веткой. Ветви имеют свою собственную историю и изолированные друг от друга изменения до тех пор, пока вы не решаете слить изменения вместе. Это происходит по набору причин:

- Уже рабочая, стабильная версия кода сохраняется.

- Различные новые функции могут разрабатываться параллельно разными программистами.
- Разработчики могут работать с собственными ветками без риска, что кодовая база поменяется из-за чужих изменений.
- В случае сомнений, различные реализации одной и той же идеи могут быть разработаны в разных ветках и затем сравниваться.
- 1. Создание новой ветки

Основная ветка в каждом репозитории называется master. Чтобы создать еще одну ветку, используем команду branch <name>

```
$ git branch amazing_new_feature
```

Это создаст новую ветку, пока что точную копию ветки master.

- 2. Переключение между ветками

Сейчас, если мы запустим branch, мы увидим две доступные опции:

```
$ git branch
amazing_new_feature
* master
```

master – это активная ветка, она помечена звездочкой. Но мы хотим работать с нашей «новой потрясающей фичей», так что нам понадобится переключиться на другую ветку. Для этого воспользуемся командой checkout, она принимает один параметр – имя ветки, на которую необходимо переключиться.

```
$ git checkout amazing_new_feature
```

- 3. Слияние веток

Наша «потрясающая новая фича» будет еще одним текстовым файлом под названием feature.txt. Мы создадим его, добавим и закоммитим:

```
$ git add feature.txt
$ git commit -m «New feature complete.»
```

Изменения завершены, теперь мы можем переключиться обратно на ветку master.

```
$ git checkout master
```

Теперь, если мы откроем наш проект в файловом менеджере, мы не увидим файла feature.txt, потому что мы переключились обратно на ветку master, в которой такого файла не существует. Чтобы он появился, нужно воспользоваться merge для объединения веток

(применения изменений из ветки amazing_new_feature к основной версии проекта).

```
$ git merge amazing_new_feature
```

Теперь ветка master актуальна. Ветка amazing_new_feature больше не нужна, и ее можно удалить.

```
$ git branch -d awesome_new_feature
```

- Дополнительно

В последней части этого руководства мы расскажем о некоторых дополнительных трюках, которые могут вам помочь.

- 1. Отслеживание изменений, сделанных в коммитах

У каждого коммита есть свой уникальный идентификатор в виде строки цифр и букв. Чтобы просмотреть список всех коммитов и их идентификаторов, можно использовать команду log:

Вывод git log

Как вы можете заметить, идентификаторы довольно длинные, но для работы с ними не обязательно копировать их целиком – первых нескольких символов будет вполне достаточно. Чтобы посмотреть, что нового появилось в коммите, мы можем воспользоваться командой show [commit]

Вывод git show

Чтобы увидеть разницу между двумя коммитами, используется команда diff (с указанием промежутка между коммитами):

Вывод git diff

Мы сравнили первый коммит с последним, чтобы увидеть все изменения, которые были когда-либо сделаны. Обычно проще использовать git difftool, так как эта команда запускает графический клиент, в котором наглядно сопоставляет все изменения.

- 2. Возвращение файла к предыдущему состоянию

Гит позволяет вернуть выбранный файл к состоянию на момент определенного коммита. Это делается уже знакомой нам командой checkout, которую мы ранее использовали для переключения между ветками. Но она также может быть использована для переключения между коммитами (это довольно распространенная ситуация для Гита – использование одной команды для различных, на первый взгляд, слабо связанных задач).

В следующем примере мы возьмем файл hello.txt и откатим все изменения, совершенные над ним к первому коммиту. Чтобы сде-

лать это, мы подставим в команду идентификатор нужного коммита, а также путь до файла:

```
$ git checkout 09bd8cc1 hello.txt
```

- 3. Исправление коммита

Если вы опечатались в комментарии или забыли добавить файл и заметили это сразу после того, как закоммитили изменения, вы легко можете это поправить при помощи commit –amend. Эта команда добавит все из последнего коммита в область подготовленных файлов и попытается сделать новый коммит. Это дает вам возможность поправить комментарий или добавить недостающие файлы в область подготовленных файлов.

Для более сложных исправлений, например, не в последнем коммите или если вы успели отправить изменения на сервер, нужно использовать revert. Эта команда создаст коммит, отменяющий изменения, совершенные в коммите с заданным идентификатором. Самый последний коммит может быть доступен по алиасу HEAD:

```
$ git revert HEAD
```

Для остальных будем использовать идентификаторы:

```
$ git revert b10cc123
```

При отмене старых коммитов нужно быть готовым к тому, что возникнут конфликты. Такое случается, если файл был изменен еще одним, более новым коммитом. И теперь git не может найти строчки, состояние которых нужно откатить, так как они больше не существуют.

- 4. Разрешение конфликтов при слиянии

Помимо сценария, описанного в предыдущем пункте, конфликты регулярно возникают при слиянии ветвей или при отправке чужого кода. Иногда конфликты исправляются автоматически, но обычно с этим приходится разбираться вручную – решать, какой код остается, а какой нужно удалить.

Давайте посмотрим на примеры, где мы попытаемся слить две ветки под названием john_branch и tim_branch. И Тим, и Джон правят один и тот же файл: функцию, которая отображает элементы массива.

Джон использует цикл:

```
// Use a for loop to console.log contents.
for(var i=0; i<arr.length; i++) {
```

```
console.log(arr[i]);
}
```

Тим предпочитает forEach:

```
// Use forEach to console.log contents.
arr.forEach(function(item) {
console.log(item);
});
```

Они оба коммитят свой код в соответствующую ветку. Теперь, если они попытаются слить две ветки, они получат сообщение об ошибке:

```
$ git merge tim_branch
Auto-merging print_array.js
CONFLICT (content): Merge conflict in print_array.js
Automatic merge failed; fix conflicts and then commit the result.
```

Система не смогла разрешить конфликт автоматически, значит, это придется сделать разработчикам. Приложение отметило строки, содержащие конфликт:

Вывод

Над разделителем ======= мы видим последний (HEAD) коммит, а под ним – конфликтующий. Таким образом, мы можем увидеть, чем они отличаются и решать, какая версия лучше. Или вовсе написать новую. В этой ситуации мы так и поступим, перепишем все, удалив разделители, и дадим git понять, что закончили.

```
// Not using for loop or forEach.
// Use Array.toString() to console.log contents.
console.log(arr.toString());
```

Когда все готово, нужно закоммитить изменения, чтобы закончить процесс:

```
$ git add -A
$ git commit -m «Array printing conflict resolved.»
```

Как вы можете заметить, процесс довольно утомительный и может быть очень сложным в больших проектах. Многие разработчики предпочитают использовать для разрешения конфликтов клиенты с графическим интерфейсом. (Для запуска нужно набрать git mergetool).

- 5. Настройка .gitignore

В большинстве проектов есть файлы или целые директории, в которые мы не хотим (и, скорее всего, не захотим) коммитить. Мы можем удостовериться, что они случайно не попадут в git add -A при помощи файла .gitignore

1. Создайте вручную файл под названием .gitignore и сохраните его в директорию проекта.
2. Внутри файла перечислите названия файлов/папок, которые нужно игнорировать, каждый с новой строки.
3. Файл .gitignore должен быть добавлен, закоммичен и отправлен на сервер, как любой другой файл в проекте.

Вот хорошие примеры файлов, которые нужно игнорировать:

- Логи
- Артефакты систем сборки
- Папки node_modules в проектах node.js
- Папки, созданные IDE, например, Netbeans или IntelliJ
- Разнообразные заметки разработчика.

Файл .gitignore, исключающий все перечисленное выше, будет выглядеть так:

```
*.log
build/
node_modules/
.idea/
my_notes.txt
```

Символ слэша в конце некоторых линий означает директорию (и тот факт, что мы рекурсивно игнорируем все ее содержимое). Звездочка, как обычно, означает шаблон.

Контрольные вопросы

1. Приведите основные команды git.
2. Как создать новую ветку в git?
3. Как переключиться в существующую ветку?
4. Как отправить изменения на сервер?

8. Лабораторная работа №5. Проектирование и разработка интерфейса пользователя

Целью работы является изучение порядка работы при проектировании интерфейса пользователя. Результатом практической работы является отчет, в котором должны быть приведено описание разработанного интерфейса пользователя.

Для выполнения лабораторной работы № 5 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

1.1. Теоретические сведения

Графический интерфейс пользователя (GUI) – разновидность пользовательского интерфейса, в котором элементы интерфейса (меню, кнопки, значки, списки и т. п.), представленные пользователю на дисплее, исполнены в виде графических изображений.

В GUI пользователь имеет произвольный доступ (с помощью устройств ввода – клавиатуры, мыши, джойстика и т. п.) ко всем видимым экранным объектам (элементам интерфейса) и осуществляет непосредственное манипулирование ими.

Графический интерфейс пользователя является частью пользовательского интерфейса и определяет взаимодействие с пользователем на уровне визуализированной информации.

Можно выделить следующие виды графического интерфейса пользователя:

– простой: типовые экранные формы и стандартные элементы интерфейса, обеспечиваемые самой подсистемой GUI;

– истинно-графический, двухмерный: нестандартные элементы интерфейса и оригинальные метафоры, реализованные собственными средствами приложения или сторонней библиотекой;

– трёхмерный.

Проектирование графического интерфейса пользователя представляет собой междисциплинарную деятельность. Оно требует усилий многофункциональной бригады – один человек, как правило, не обладает знаниями, необходимыми для реализации многоаспектного подхода к проектированию GUI-интерфейса. Надлежащее проектирование GUI-интерфейса требует объединения навыков художника-графика, специалиста по анализу требований, системного проектировщика, программиста, эксперта по технологии, специали-

ста в области социальной психологии, а также, возможно, некоторых других специалистов, в зависимости от характера системы.

В современном мире миллиарды вычислительных устройств. Еще больше программ для них. И у каждой свой интерфейс, являющийся «рычагами» взаимодействия между пользователем и машинным кодом. Не удивительно, что чем лучше интерфейс, тем эффективнее взаимодействие.

Однако далеко не все разработчики и даже дизайнеры, задумываются о создании удобного и понятного графического интерфейса пользователя.

Какие элементы интерфейса (ЭИ) создавать?

1. Разработка интерфейса обычно начинается с определения задачи или набора задач, для которых продукт предназначен.

2. Простое должно оставаться простым. Не стоит усложнять интерфейсы. Нужно постоянно думать о том, как сделать интерфейс проще и понятнее.

3. Пользователи не задумываются над тем, как устроена программа. Все, что они видят – это интерфейс. Поэтому с точки зрения потребителя именно интерфейс является конечным продуктом.

4. Интерфейс должен быть ориентированным на человека, т. е. отвечать нуждам человека и учитывать его слабости. Нужно постоянно думать о том, с какими трудностями может столкнуться пользователь.

Необходимо думать о поведении и привычках пользователей. Не менять хорошо известные всем ЭИ на неожиданные, а новые делать интуитивно понятными.

5. Разрабатывать интерфейс необходимо исходя из принципа наименьшего возможного количества действий со стороны пользователя.

Какой должен быть дизайн элементов интерфейса?

В дизайне ЭИ нужно учитывать все: начиная от цвета, формы, пропорций, заканчивая когнитивной психологией. Однако, несколько принципов все же стоит отметить:

1. Цвет. Цвета делятся на теплые (желтый, оранжевый, красный), холодные (синий, зеленый), нейтральные (серый). Обычно для ЭИ используют теплые цвета. Это как раз связано с психологией восприятия. Стоит отметить, что мнение о цвете – очень субъективно и может меняться даже от настроения пользователя.

2. Форма. В большинстве случаев – прямоугольник со скругленными углами. Или круг. Опять же, форма как и цвет достаточно субъективна.

3. Основные ЭИ (часто используемые) должны быть выделены. Например, размером или цветом.

4. Иконки в программе должны быть очевидными. Или подписанными. Ведь, по сути дела, вместо того чтобы объяснять, пиктограммы зачастую сами требуют для себя объяснений.

Как правильно расположить элементы интерфейса на экране?

1. Есть утверждение, что визуальная привлекательность основана на пропорциях. Помните известное число 1.62? Это так называемый принцип Золотого сечения. Суть в том, что весь отрезок относится к большей его части так, как большая часть, относится к меньшей. Например, общая ширина сайта 900px, делим 900 на 1.62, получаем ~555px, это ширина блока с контентом. Теперь от 900 отнимаем 555 и получаем 345px. Это ширина меньшей части.

2. Перед расположением, ЭИ следует упорядочить (сгруппировать) по значимости. То есть определить, какие наиболее важны, а какие – менее.

3. Обычно (но не обязательно), элементы размещаются в следующей градации: слева направо, сверху вниз. Слева вверху самые значимые элементы, справа внизу – менее. Это связано с порядком чтения текста. В случае с сенсорными экранами, самые важные элементы, располагаются в области действия больших пальцев рук.

4. Необходимо учитывать привычки пользователя. Например, если в Windows кнопка закрыть находится в правом верхнем углу, то программе аналогичную кнопку необходимо расположить там же. То есть интерфейс должен иметь как можно больше аналогий, с известными пользователю вещами.

5. Размещать ЭИ стоит поближе там, где большую часть времени находится курсор пользователя. Чтобы ему не пришлось перемещать курсор, например, от одного конца экрана к другому.

6. Элемент интерфейса можно считать видимым, если он либо в данный момент доступен для органов восприятия человека, либо он был настолько недавно воспринят, что еще не успел выйти из кратковременной памяти. Для нормальной работы интерфейса, должны быть видимы только необходимые вещи – те, что иденти-

фицируют части работающих систем, и те, что отображают способ, которым пользователь может взаимодействовать с устройством.

7. Отступы между ЭИ лучше делать равными или кратными друг другу.

Как элементы интерфейса должны себя вести?

1. Пользователи привыкают. Например, при удалении файла, появляется окно с подтверждением: «Да» или «Нет». Со временем, пользователь перестает читать предупреждение и по привычке нажимает «Да». Поэтому диалоговое окно, которое было призвано обеспечить безопасность, абсолютно не выполняет своей роли. Следовательно, необходимо дать пользователю возможность отменять, сделанные им действия.

2. Если пользователю дают информацию, которую он должен куда-то ввести или как-то обработать, то информация должна оставаться на экране до того момента, пока человек ее не обработает. Иначе он может просто забыть.

3. Нужно избегать двусмысленности. Например, на фонарике есть одна кнопка. По нажатию фонарик включается, нажали еще раз – выключился. Если в фонарике перегорела лампочка, то при нажатии на кнопку не понятно, включаем мы его или нет. Поэтому, вместо одной кнопки выключателя, лучше использовать переключатель (например, checkbox с двумя позициями: «вкл.» и «выкл.»). За исключением случаев, когда состояние задачи, очевидно.

4. Имеет смысл делать монотонные интерфейсы. Монотонный интерфейс – это интерфейс, в котором какое-то действие, можно сделать только одним способом. Такой подход обеспечит быструю привыкаемость к программе и автоматизацию действий.

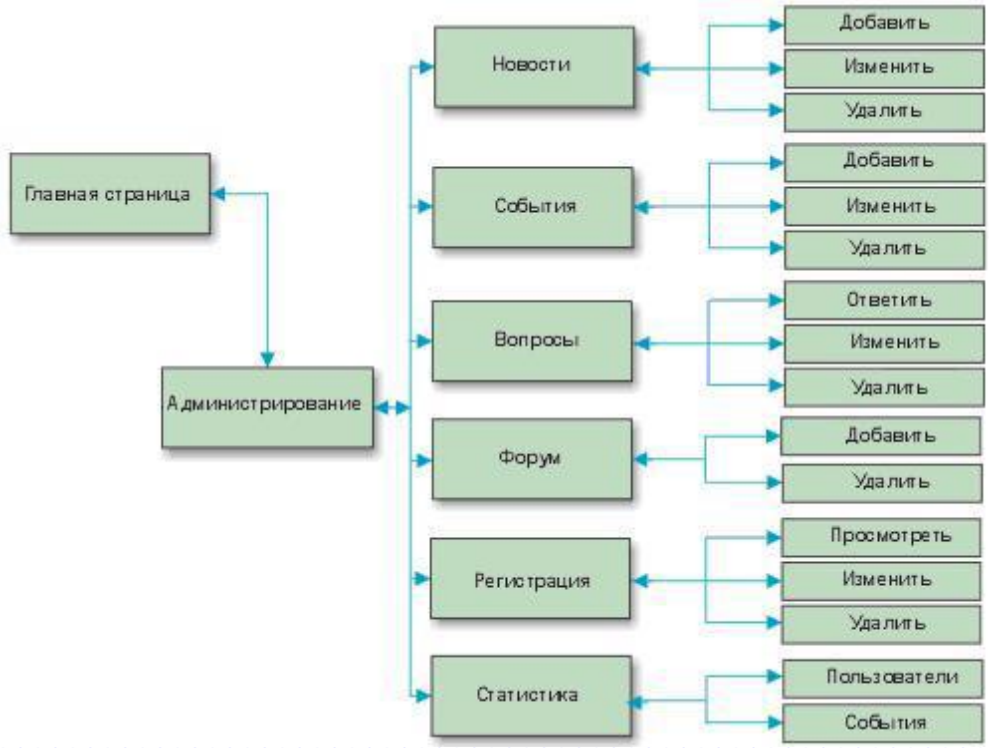
5. Не стоит делать адаптивные интерфейсы, которые изменяются со временем. Так как для выполнения какой-то задачи, лучше изучать только один интерфейс, а не несколько. Пример – стартовая страница браузера Chrome.

6. Если задержки в процессе выполнения программы неизбежны или действие, производимое пользователем очень значимо, важно, чтобы в интерфейсе была предусмотрена сообщающая о них обратная связь. Например, можно использовать индикатор хода выполнения задачи (status bar).

7. ЭИ должны отвечать. Если пользователь произвел клик, то ЭИ должен как-то отозваться, чтобы человек понял, что клик произошел.

Карта навигации – информация на карте навигации аналогична разделу «Содержание» обычной книги. В карте представлен полный перечень разделов и/или всех страниц, имеющихся на сайте. Нередко, заголовки страниц в списке служат ссылками на эти страницы.

Примеры:



1.2. Описание работы

1. Создайте карту навигации для выбранной системы. На карте в зависимости от специфики системы выделите разделы, доступные различным пользователям в зависимости от роли, опишите условия перехода из различных разделов (при необходимости)

Контрольные вопросы

1. Какие базовые принципы создания GUI?
2. Какие виды GUI существуют?
3. Опишите стандартные виды элементов GUI.

9. Лабораторная работа №6. Реализация алгоритмов обработки числовых данных. Отладка приложения

Целью работы является изучение порядка работы при обработке числовых данных. Результатом практической работы является отчет, в котором должны быть приведено описание разработанных алгоритмов и результаты отладки ПО.

Для выполнения лабораторной работы № 6 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Работа с массивами данных в C# с использованием Windows Forms

1. Цель

ознакомиться с компонентами и методами работы с массивами в C# с использованием Windows Form в простых Windows приложениях.

2. Методические указания

1. При изучении языка программирования C# будет использоваться интегрированная среда разработки программного обеспечения Microsoft Visual Studio Express 2013 для Windows Desktop. Будут использованы основные элементы .NET Framework и связь C# с элементами платформы .NET.

2. По окончанию работы сохраните все созданные файлы на своих носителях.

3. Защита лабораторной работы производится только при наличии электронного варианта работающего скрипта.

3. Теоретические сведения

Алгоритмы и задачи, рассматриваемые в этой лабораторной работе, являются частью фундамента, на котором строится образование программиста. Нет ни одной проблемной области, в задачах которой не требовались бы массивы.

Последовательность элементов – a1, a2, …. an – одна из любимых структур в математике. Последовательность можно рассматривать как функцию a(i), которая по заданному значению индекса элемента возвращает его значение. Эта функция задает отображение integer -> T, где T – это тип элементов последовательности.

В программировании последовательности называются массивами. Массив – это упорядоченная последовательность элементов одного типа. Порядок элементов задается с помощью индексов.

Для программистов важно то, как массивы хранятся в памяти. Массивы занимают непрерывную область памяти, поэтому, зная адрес начального элемента массива, зная, сколько байтов памяти требуется для хранения одного элемента, и, зная индекс (индексы) некоторого элемента, нетрудно вычислить его адрес, а значит и хранимое по этому адресу значение элемента. На этом основана адресная арифметика в языках C, C++, где адрес элемента a(i) задается адресным выражением a+i, в котором имя массива a воспринимается как адрес первого элемента. При вычислении адреса i-го элемента индекс i умножается на длину слова, требуемого для хранения элементов типа T. Адресная арифметика приводит к 0-базируемости элементов массива, когда индекс первого элемента равен нулю, поскольку первому элементу соответствует адресное выражение a+0.

Язык C# сохранил 0-базируемость массивов. Индексы элементов массива в языке C# изменяются в плотном интервале значений от нижней границы, всегда равной 0, до верхней границы, заданной динамически вычисляемым выражением, возможно зависящим от переменных. Массивы C# являются 0-базируемыми динамическими массивами. Это важно понимать с самого начала.

Не менее важно понимать и то, что массивы C# относятся к ссылочным типам. Рассмотрим следующий фрагмент программного кода:

```
int[] x, y = {1, 2, 3};
double[] z;
int[,] u, v = {{1,3,5},{2,4,6}};
```

Здесь объявлены пять переменных – x, y, z, u, v. Все они являются массивами, но разных типов. Переменные x и y принадлежат типу T = int*+, задающему одномерные массивы с элементами целого типа int. Переменные u и v принадлежат другому типу T1 = int[,]- двумерных массивов с элементами целого типа. Пере-

менная z принадлежит типу T3 –одномерных массивов с элементами вещественного типа double. Все три типа массивов – T1, T2, T3 являются наследниками общего родителя – типа (класса) Array, наследуя от родителя многие полезные свойства и методы. Все пять переменных являются типизированными ссылками. В момент объявления три переменные – x, z и u не инициализированы и потому являются «висячими» ссылками со значением null. Переменные y и v объявлены с инициализацией. При инициализации в динамической памяти создаются два объекта соответственно типов T1 и T3, фактически и задающие реальные массивы. У первого из этих объектов 3 элемента, у второго – 6. Ссылки y и v связываются с этими объектами. При связывании тип ссылки и тип объекта должны быть согласованными. Заметьте, число элементов в массиве является характеристикой объекта, а не типа. Ссылка может быть связана с объектами, содержащими различное число элементов, необходимо лишь выполнение условия согласования типов.

Дополним код следующими строчками:

x = new int[10];

z = new double[20];

u = new int[3, 5];

Здесь последовательно вызываются три конструктора типов T1, T2, T3, создающие новые три объекта в памяти и ссылки x, y, z связываются с этими объектами, так что у массива x теперь 10 элементов, z – 20, u -15.

Рассмотрим, что произойдет в результате присваивания:

x = y;

u = v;

Присваивание законно, поскольку переменные в левой и правой части имеют один и тот же (следовательно, согласованный) тип. В результате присваивания переменные порвут связь с теми объектами, с которыми они были связаны, и будут связаны с новыми объектами, так что x теперь имеет 3 элемента, u – 6.

Ввод – вывод массивов

Как у массивов появляются значения, как они изменяются? Возможны три основных способа:

– вычисление значений в программе;

– значения вводит пользователь;

– связывание с источником данных.

В задачах этой лабораторной работы ограничимся пока рассмотрением первых двух способов. Первый способ более или менее понятен. Простые примеры его применения

приводились чуть выше. Приведу некоторые рекомендации по вводу и выводу массивов, ориентированные на работу с конечным пользователем.

Для консольных приложений ввод массива обычно проходит несколько этапов:

1) ввод размеров массива;

2) создание массива;

3) организация цикла по числу элементов массива, в теле которого выполняется:

– приглашение к вводу очередного элемента;

– ввод элемента;

– проверка корректности введенного значения.

Вначале у пользователя запрашиваются размеры массива, затем создается массив заданного размера. В цикле по числу элементов организуется ввод значений. Вводу каждого значения предшествует приглашение к вводу с указанием типа вводимого значения, а при необходимости и диапазона, в котором должно находиться требуемое значение. Поскольку ввод значений – это ответственная операция, а на пользователя никогда нельзя положиться, то после ввода часто организуется проверка корректности введенного значения. При некорректном задании значения элемента ввод повторяется, пока не будет, достигнут желаемый результат.

При выводе массива на консоль, обычно вначале выводится имя массива, а затем его элементы в виде пары: <имя> = <значение> (например, f*5+ = 77,7). Задача осложняется для многомерных массивов, когда для пользователя важно видеть не только значения, но и структуру массива, например располагая одну строку массива в одной строке экрана. Приведу пример того, как может выглядеть ввод – вывод массива в консольном приложении. На рис. 6.1. показан экран в процессе работы консольной программы, организующей ввод массива.

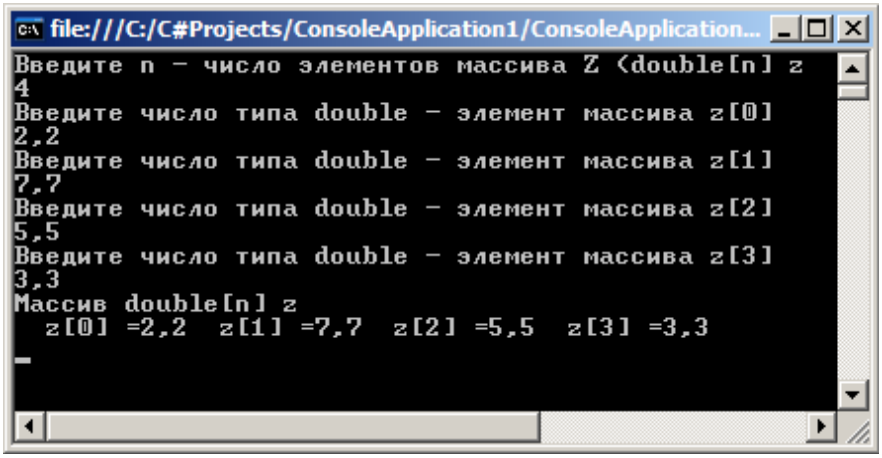


Рис. 6.1. Ввод – вывод массива на консоль

На рис. 6.2. показана работа программы с контролем ввода, где пользователь повторяет ввод, пока не введет значение, удовлетворяющее типу элемента массива.

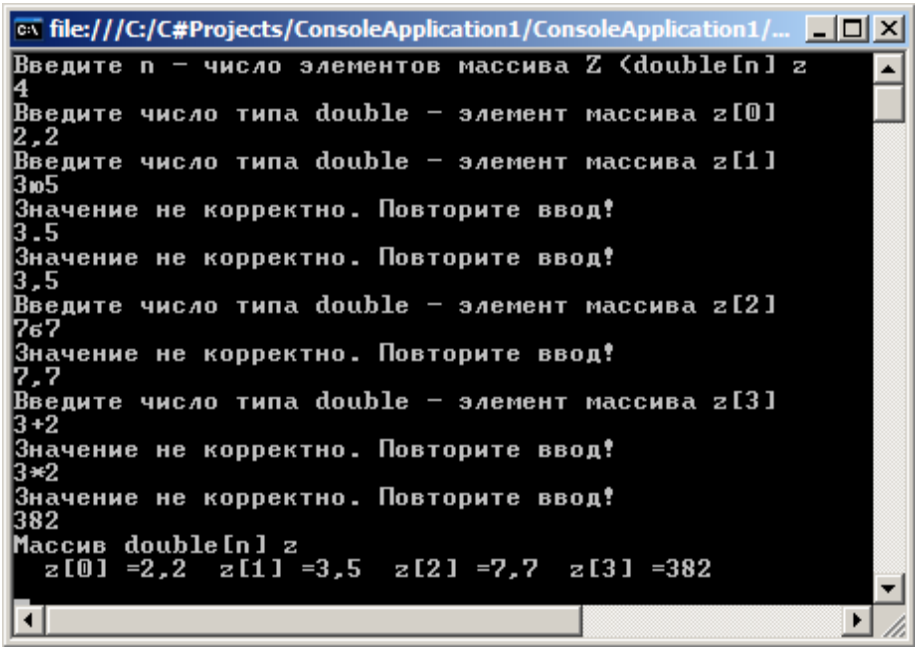


Рис. 6.2. Ввод – вывод массива на консоль с контролем значений

Как организовать контроль ввода? Наиболее разумно использовать для этих целей конструкцию охраняемых блоков – **try – catch** блоков. Это общий подход, когда все опасные действия, обычно связанные с работой пользователя, внешних устройств, внешних источников данных, размещаются в охраняемых блоках. Вот как может выглядеть ввод элемента массива **z[i]** типа **double**, помещенный в охраняемый блок:

bool correct;

```
do
{
correct = true;
try
{
z[i] = Convert.ToDouble(Console.ReadLine());
}
catch (Exception e)
{
Console.WriteLine
(«Значение не корректно. Повторите ввод!»);
correct = false;
}
} while (!correct);
```

Как правило, для ввода-вывода массивов пишутся специальные процедуры, вызываемые в нужный момент.

Ввод – вывод массивов в Windows приложениях

Приложения Windows позволяют построить пользовательский интерфейс, облегчающий работу по вводу и выводу массивов. И здесь, когда данные задаются пользователем, заполнение массива проходит через те же этапы, что рассматривались для консольных приложений. Но выглядит все это более наглядно и понятно. Пример подобного интерфейса, обеспечивающего работу по вводу и выводу одномерного массива, показан на рис. 6.3.

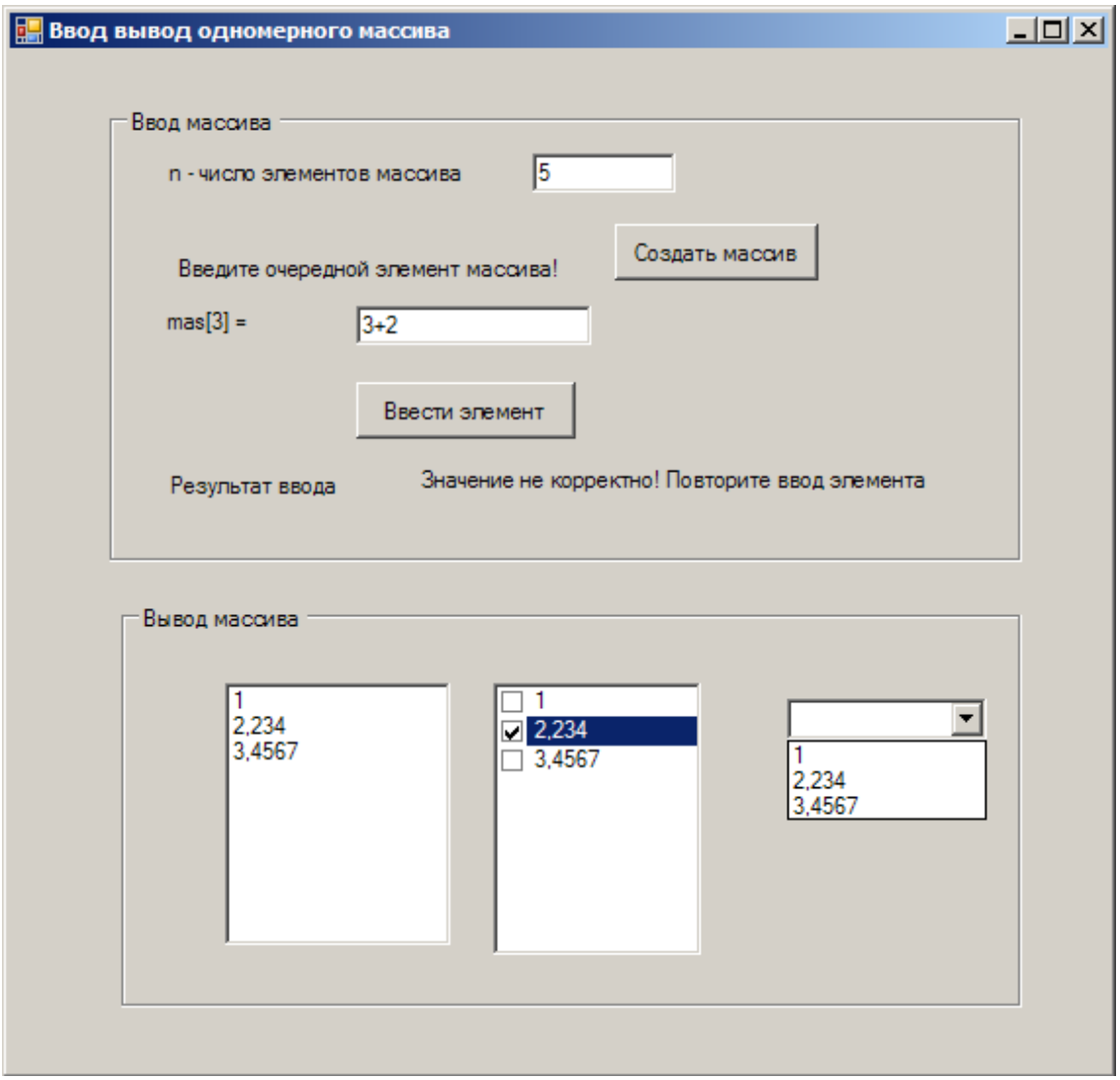


Рис. 6.3. Форма для ввода – вывода одномерного массива

Пользователь вводит в текстовое окно число элементов массива и нажимает командную кнопку «Создать массив», обработчик которой создает массив заданной размерности. Затем он может переходить к следующему этапу – вводу элементов массива. Очередной элемент массива вводится в текстовое окно, а обработчик командной кнопки «Ввести элемент» обеспечивает передачу значения в массив. Корректность ввода можно контролировать и здесь, проверяя значение введенного элемента и выводя в специальное окно сообщение в случае его некорректности, добиваясь, в конечном итоге, получения от пользователя корректного ввода.

Для облегчения работы пользователя текстовое окно для ввода элемента сопровождается специальным окном, содержащим информацию, какой именно элемент должен вводить пользователь. В примере возможного интерфейса пользователя, показанного на

рис. 6.3., после того, как все элементы массива введены, окно ввода становится недоступным для ввода элементов. Интерфейс формы позволяет многократно создавать новый массив, повторяя весь процесс.

На рис. 6.3. форма разделена на две части – для ввода и вывода массива. Крайне важно уметь организовать ввод массива, принимая данные от пользователя. Не менее важно уметь отображать существующий массив в форме, удобной для восприятия пользователя. На рисунке показаны три различных элемента управления, пригодные для этих целей – **ListBox, CheckedListBox и ComboBox.** Программа, форма которой показана на рис. 6.3, сделана так, что как только вводится очередной элемент, он немедленно отображается во всех трех списках.

Отображать массив в трех списках конечно не нужно, это сделано только в целях демонстрации возможностей различных элементов управления. Для целей вывода подходит любой из них, выбор зависит от контекста и предпочтений пользователя. Элемент **CheckedListBox** обладает дополнительными свойствами в сравнении с элементом **ListBox**, позволяя отмечать некоторые элементы списка (массива). Отмеченные пользователем элементы составляют специальную коллекцию. Эта коллекция доступна, с ней можно работать, что иногда весьма полезно.

Ввод двумерного массива немногим отличается от ввода одномерного массива. Сложнее обстоит дело с выводом двумерного массива, если при выводе пытаться отобразить структуру массива. К сожалению все три элемента управления, хорошо справляющиеся с отображением одномерного массива, плохо приспособлены для показа структуры двумерного массива. Хотя у того же элемента **ListBox** есть свойство **MultiColumn,** включение которого позволяет показывать массив в виде строк и столбцов, но это не вполне то, что нужно для наших целей – отображения структуры двумерного массива. Хотелось бы, чтобы элемент имел такие свойства, как **Rows** и **Columns,** а их у элемента **ListBox** нет. Нет их и у элементов **ComboBox** и **CheckedListBox**. Приходится обходиться тем, что есть. На рис. 6.4. показан пример формы, поддерживающей работу по вводу и выводу двумерного массива.

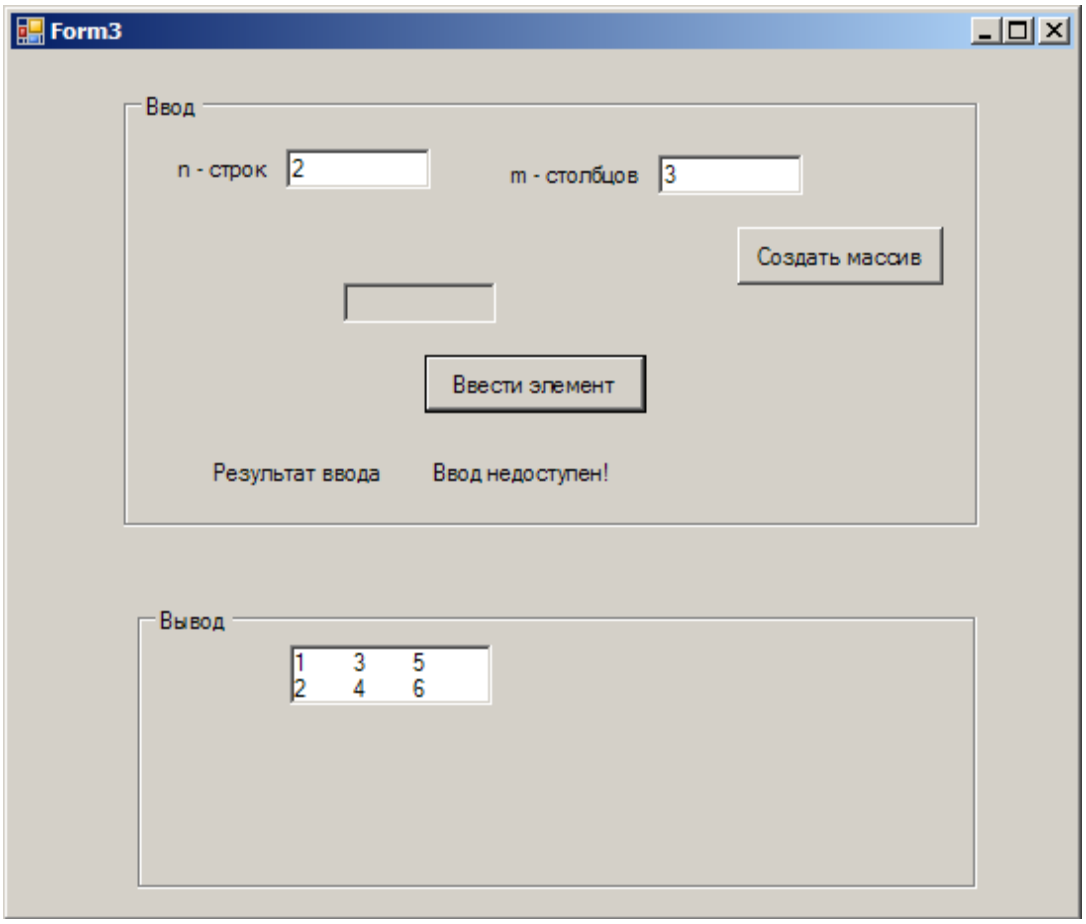


Рис. 6.4. Форма, поддерживающая ввод
и вывод двумерного массива

Интерфейс формы схож с тем, что использовался для организации работы с одномерным массивом. Программная настройка размеров элемента управления **ListBox** позволила даже отобразить структуру массива. Но в общей ситуации, когда значения, вводимые пользователем, могут колебаться в широком диапазоне, трудно гарантировать отображение структуры двумерного массива. Однако ситуация не безнадежна. Есть и другие, более мощные и более подходящие для наших целей элементы управления.

Элемент управления DataGridView и отображение массивов

Элемент управления **DataGridView** является последней новинкой в серии табличных элементов **DataGrid**, позволяющих отображать таблицы. Главное назначение этих элементов – связывание с таблицами внешних источников данных, прежде всего с таблицами баз данных. Мы же сейчас рассмотрим другое его применение – в интерфейсе, позволяющем пользователю вводить и отображать матрицы – двумерные массивы.

Рассмотрим классическую задачу умножения прямоугольных матриц **C=A*B.** Построим интерфейс, позволяющий пользователю задавать размеры перемножаемых матриц, вводить данные для исходных матриц **A** и **B,** перемножать матрицы и видеть результаты этой операции. На рис. 6.5 показан возможный вид формы, поддерживающей работу пользователя. Форма показана в тот момент, когда пользователь уже задал размеры и значения исходных матриц, выполнил умножение матриц и получил результат.

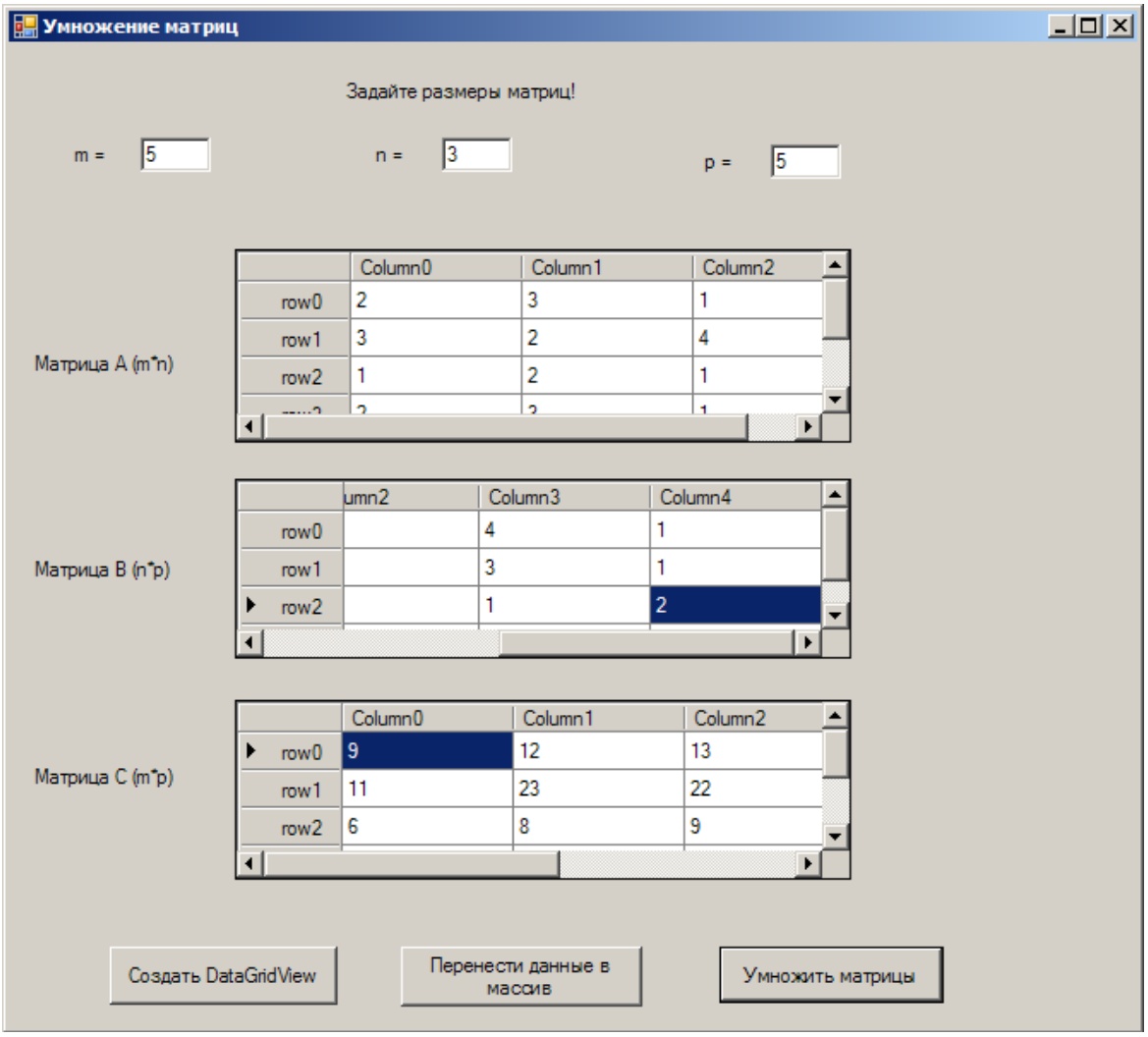


Рис. 6.5. Форма с элементами DataGridView, поддерживающая работу с матрицами

На форме расположены три текстовых окна для задания размеров матриц, три элемента **DataGridView** для отображения матриц, три командные кнопки для выполнения операций, доступных пользователю. Кроме того, на форме присутствуют 9 меток (элементов управления **label**), семь из которых видимы на рис. 6.5. В них ото-

бражается информация, связанная с формой и отдельными элементами управления. Текст у невидимых на рисунке меток появляется тогда, когда обнаруживается, что пользователь некорректно задал значение какого-либо элемента исходных матриц.

А теперь перейдем к описанию того, как этот интерфейс реализован. В классе Form2, которому принадлежит наша форма, зададим поля, определяющие размеры матриц и сами матрицы:

```
//поля класса Form
int m, n, p; //размеры матриц
double[,] A, B, C; //сами матрицы
```

Рассмотрим теперь, как выглядит обработчик события «**Click**» командной кнопки «Создать DataGridView». Предполагается, что пользователь разумен и, прежде чем нажать эту кнопку, задает размеры матриц в соответствующих текстовых окнах. Напомню, что при перемножении матриц размеры матриц должны быть согласованы – число столбцов первого сомножителя должно совпадать с числом строк второго сомножителя, а размеры результирующей матрицы определяются размерами сомножителей. Поэтому для трех матриц в данном случае достаточно задать не шесть, а три параметра, определяющих размеры.

Обработчик события выполняет три задачи – создает сами матрицы, осуществляет чистку элементов управления **DataGridView,** удаляя предыдущее состояние, затем добавляет столбцы и строки в эти элементы в полном соответствии с заданными размерами матриц. Вот текст обработчика:

```
private void button1_Click(object sender, EventArgs e)
{
//создание матриц
m = Convert.ToInt32(textBox1.Text);
n = Convert.ToInt32(textBox2.Text);
p = Convert.ToInt32(textBox3.Text);
A = new double[m, n];
B = new double[n, p];
C = new double[m, p];
//Чистка DGView, если они не пусты
int k =0;
k = dataGridView1.ColumnCount;
if (k != 0)
```

```
for (int i = 0; i < k; i++)
dataGridView1.Columns.RemoveAt(0);
dataGridView2.Columns.Clear();
dataGridView3.Columns.Clear();
//Заполнение DGView столбцами
AddColumns(n, dataGridView1);
AddColumns(p, dataGridView2);
AddColumns(p, dataGridView3);
//Заполнение DGView строками
AddRows(m, dataGridView1);
AddRows(n, dataGridView2);
AddRows(m, dataGridView3);
}
```

Комментарий. Чистка предыдущего состояния элементов **DataGridView** сводится к удалению столбцов. Продемонстрированы два возможных способа выполнения этой операции. Для первого элемента показано, как можно работать с коллекцией столбцов. Организуется цикл по числу столбцов коллекции и в цикле выполняется метод **RemoveAt,** аргументом которого является индекс удаляемого столбца. Поскольку после удаления столбца происходит перенумерация столбцов, то на каждом шаге цикла удаляется первый столбец, индекс которого всегда равен нулю. Удаление столбцов коллекции можно выполнить одним махом, – вызывая метод **Clear()** коллекции, что и делается для остальных двух элементов **DataGridView**;

После чистки предыдущего состояния, можно задать новую конфигурацию элемента, добавив в него вначале нужное количество столбцов, а затем и строк. Эти задачи выполняют специально написанные процедуры **AddColumns** и **AddRows**. Вот их текст:

```
private void AddColumns(int n, DataGridView dgw)
{
//добавляет n столбцов в элемент управления dgw
//Заполнение DGView столбцами
DataGridViewColumn column;
for (int i = 0; i < n; i++)
{
column = new DataGridViewTextBoxColumn();
column.DataPropertyName = «Column» + i.ToString();
```

```
column.Name = «Column» + i.ToString();
dgw.Columns.Add(column);
}
}
private void AddRows(int m, DataGridView dgw)
{
//добавляет m строк в элемент управления dgw
//Заполнение DGView строками
for (int i = 0; i < m; i++)
{
dgw.Rows.Add();
dgw.Rows[i].HeaderCell.Value
= «row» + i.ToString();
}
}
```

Создаются столбцы в коллекции **Columns** по одному. В цикле по числу столбцов матрицы, которую должен отображать элемент управления **DataGridView,** вызывается метод **Add** этой коллекции, создающий очередной столбец. Одновременно в этом же цикле создается и имя столбца (свойство **Name**), отображаемое в форме. Показана возможность формирования еще одного имени (**DataPropertyName),** используемого при связывании со столбцом таблицы внешнего источника данных. В нашем примере это имя не используется.

Создав столбцы, нужно создать еще и нужное количество строк у каждого из элементов **DataGridView**. Делается это аналогичным образом, вызывая метод **Add** коллекции **Rows.** Чуть по-другому задаются имена строк, – для этого используется специальный объект **HeaderCell,** имеющийся у каждой строки и задающий ячейку заголовка.

После того как сформированы строки и столбцы, элемент **DataGridView** готов к тому, чтобы пользователь или программа вводила значения в ячейки сформированной таблицы.

Рассмотрим теперь, как выглядит обработчик события «Click» следующей командной кнопки «Перенести данные в массив». Предполагается, что пользователь разумен и, прежде чем нажать эту кнопку, задает значения элементов перемножаемых матриц в соответствующих ячейках подготовленных таблиц первых двух

элементов **DataGridView**. Обработчик события выполняет следующие задачи – в цикле читает элементы, записанные пользователем в таблицы **DataGridView**, проверяет их корректность и в случае успеха переписывает их в матрицы. Текст обработчика:

```
private void button2_Click(object sender, EventArgs e)
{
string elem = «»;
bool correct = true;
for (int i = 0; i < m; i++)
for (int j = 0; j < n; j++)
{
try
{
elem=dataGridView1.Rows[i].Cells[j].Value.ToString();
A[i, j] = Convert.ToDouble(elem);
label8.Text = «»;
}
catch (Exception any)
{
label8.Text = «Значение элемента» +
«A[« + i.ToString() +», « + j.ToString() + « ]»
+ « не корректно. Повторите ввод!»;
dataGridView1.Rows[i].Cells[j].Selected= true;
return;
}
}
for (int i = 0; i < n; i++)
for (int j = 0; j < p; j++)
{
do
{
correct = true;
try
{
elem =
dataGridView2.Rows[i].Cells[j].Value.ToString();
B[i, j] = Convert.ToDouble(elem);
label9.Text = «»;
```

```
}
catch (Exception any)
{
label9.Text = «Значение элемента» +
«B[« + i.ToString() + «, « + j.ToString() + «]»
+ « не корректно. Повторите ввод!»;
dataGridView2.Rows[i].Cells[j].Selected=true;
Form3 frm = new Form3();
frm.label1.Text =
«B[« + i.ToString() + «,» + j.ToString() + «]= «;
frm.ShowDialog();
dataGridView2.Rows[i].Cells[j].Value = frm.textBox1.Text;
correct = false;
}
} while (!correct);
}
}
```

Основная задача переноса данных из таблицы элемента **DataGridView** в соответствующий массив не вызывает проблем. Конструкция **Rows[i].Cells[j]** позволяет добраться до нужного элемента таблицы, после чего остается присвоить его значение элементу массива.

Как всегда при вводе основной проблемой является обеспечение корректности вводимых данных. Схема, рассматриваемая нами ранее, нуждается в корректировке. Дело в том, что ранее проверка корректности осуществлялась сразу же после ввода пользователем значения элемента. Теперь проверка корректности выполняется, после того как пользователь полностью заполнил таблицы, при этом некоторые элементы он мог задать некорректно. Просматривая таблицу, необходимо обнаружить некорректно заданные значения и предоставить возможность их исправления. В программе предлагаются два различных подхода к решению этой проблемы.

Первый подход демонстрируется на примере ввода элементов матрицы **A.** Как обычно, преобразование данных, введенных пользователем, в значение, допустимое для элементов матрицы **A,** помещается в охраняемый блок. Если данные некорректны и возникает исключительная ситуация, то она перехватывается универсальным обработчиком **catch(Exception).** Заметьте, в данном варианте

нет цикла, работающего до тех пор, пока не будет введено корректное значение. Обработчик исключения просто прерывает работу по переносу данных, вызывая оператор return. Но предварительно он формирует информационное сообщение об ошибке и выводит его в форму. (Помните, специально для этих целей у формы были заготовлены две метки). В сообщении пользователю предлагается исправить некорректно заданный элемент и повторить ввод – повторно нажать командную кнопку «перенести данные в массив». Этот подход понятен и легко реализуем. Недостатком является его неэффективность, поскольку повторно будут переноситься в массив все элементы, в том числе и те, что были введены вполне корректно. У программиста такая ситуация может вызывать чувство неудовлетворенности своей работой.

На примере ввода элементов матрицы **B** продемонстрируем другой подход, когда исправляется только некорректно заданное значение. Прежде, чем читать дальше, попробуйте найти собственное решение этой задачи. Это оказывается не так просто, как может показаться с первого взгляда. Для организации диалога с пользователем пришлось организовать специальное диалоговое окно, представляющее обычную форму с двумя элементами управления – меткой для выдачи информационного сообщения и текстовым окном для ввода пользователем корректного значения. При обнаружении ошибки ввода открывается диалоговое окно, в которое пользователь вводит корректное значение элемента и закрывает окно диалога. Введенное пользователем значение переносится в нужную ячейку таблицы **DataGridView,** а оттуда в матрицу.

При проектировании диалогового окна значение свойства формы **FormBorderStyle,** установленное по умолчанию как «sizeable» следует заменить значением **«FixedDialog»,** что влияет на внешний вид и поведение формы. Важно отметить, что форма, представляющее диалоговое окно, должна вызываться не методом **Show,** а методом **ShowDialog.** Иначе произойдет зацикливание, начнут порождаться десятки диалоговых окон, прежде чем успеете нажать спасительную в таких случаях комбинацию Ctrl+ Alt + Del.

Приведем снимки экранов, демонстрирующие ситуации, в которых пользователь ввел некорректные значения. На рис. 6.6. показано информационное сообщение, появляющееся при обнаружении некорректного ввода значений элементов матрицы **A.**

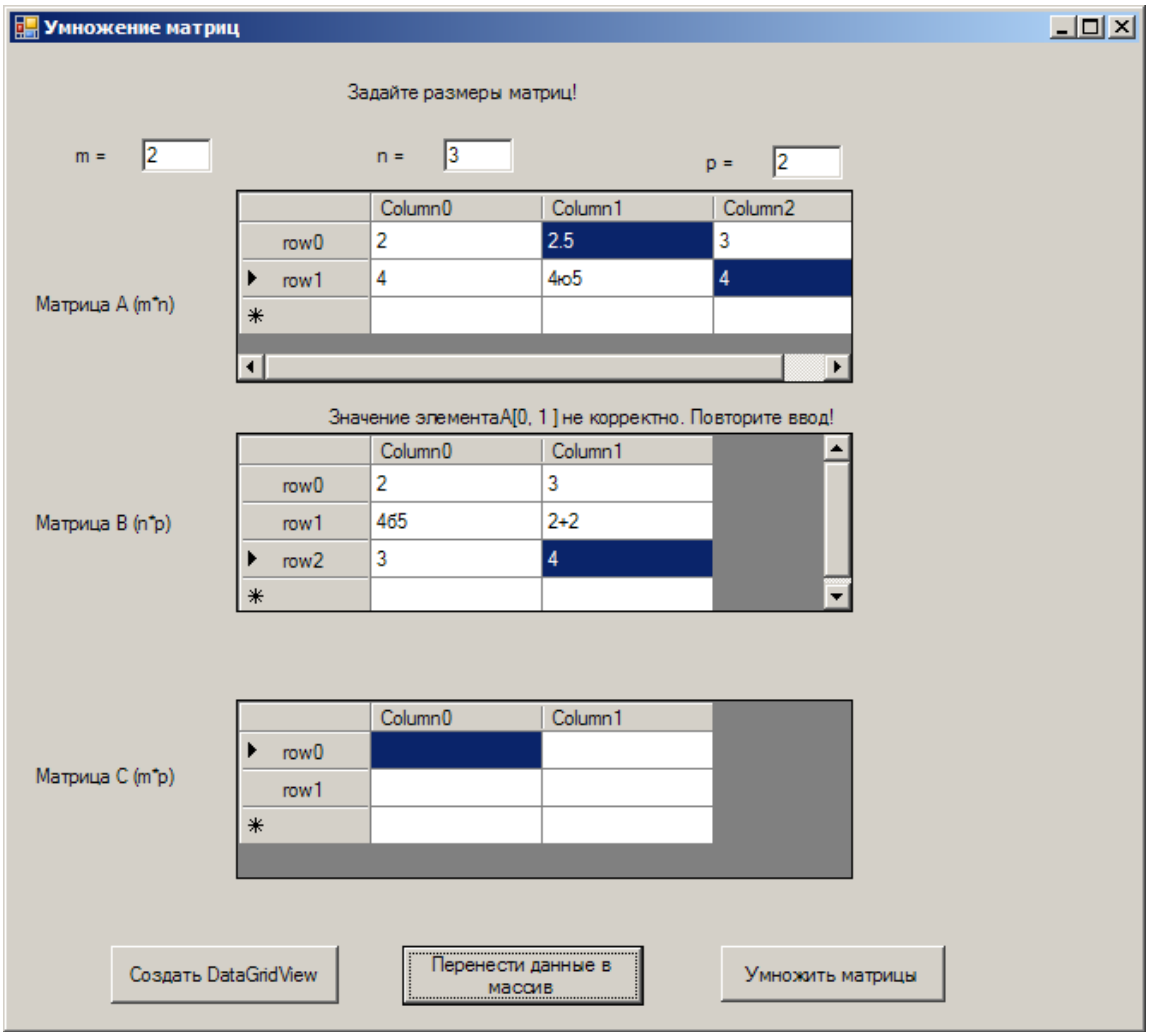


Рис. 6.6. Информационное сообщение
о некорректных значениях матрицы A

После появления подобного сообщения пользователь должен исправить некорректное значение (одно или несколько) и повторить процесс переноса данных. На рис. 6.7. показана ситуация, когда некорректно заданное значение исправляется в открывшемся окне диалога.

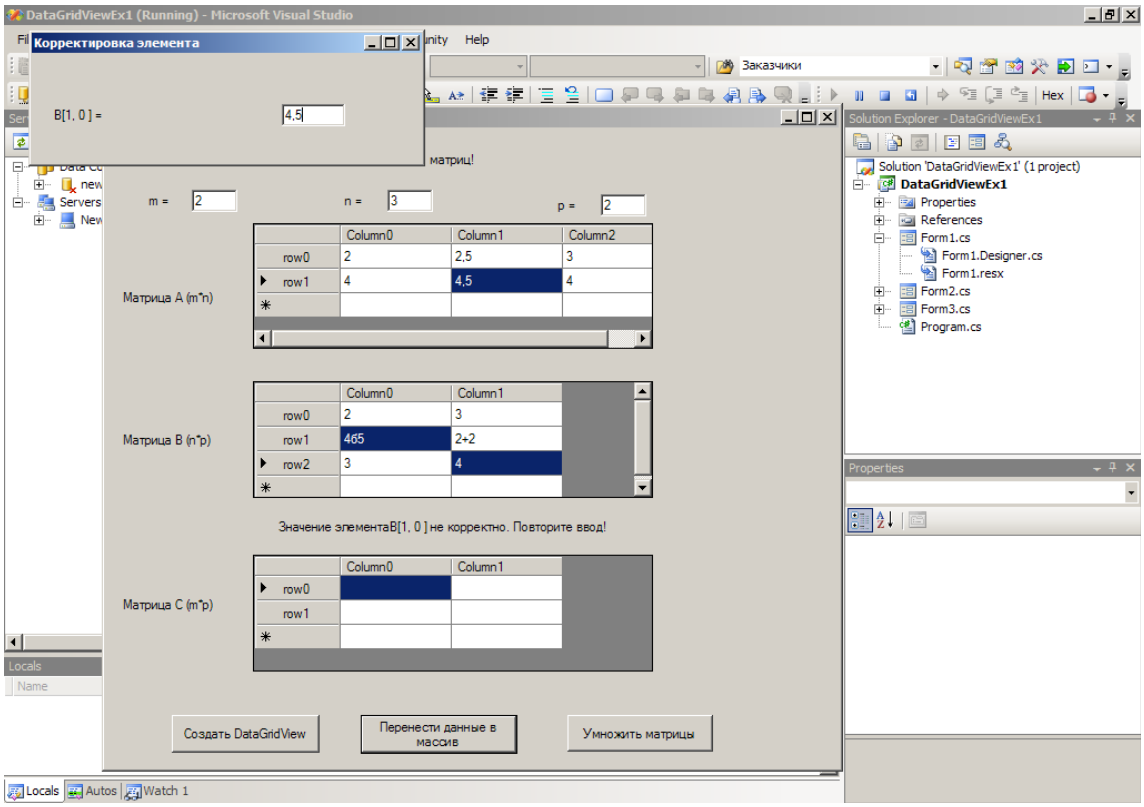


Рис. 6.7. Диалоговое окно
для корректировки значений элементов матрицы B

Обработчик события **«Click»** командной кнопки «Умножить матрицы» выполняет ответственные задачи – реализует умножение матриц и отображает полученный результат в таблице соответствующего элемента **DataGridView**. Но оба эти действия выполняются естественным образом, не требуя кроме циклов никаких специальных средств и программистских ухищрений. Программный код без дополнительных комментариев:

```
private void button3_Click(object sender, EventArgs e)
{
MultMatr(A, B, C);
FillDG();
}
void MultMatr(double[,] A, double[,] B, double[,] C)
{
int m = A.GetLength(0);
int n = A.GetLength(1);
int p = B.GetLength(1);
double S =0;
for(int i=0; i < m; i++)
```

```
for (int j = 0; j < p; j++)
{
S = 0;
for (int k = 0; k < n; k++)
S += A[i, k] * B[k, j];
C[i, j] = S;
}
}
void FillDG()
{
for (int i = 0; i < m; i++)
for (int j = 0; j < p; j++)
dataGridView3.Rows[i].Cells[j].Value
= C[i, j].ToString();}
```

4. Содержание отчёта

Отчёт должен содержать название и цель. Ход работы с комментариями и выполненное задания из Приложения А. Выводы.

ПРИЛОЖЕНИЕ А

1. Дан массив размера N и целые числа K и L ($1 < K \leq L \leq N$). Найти сумму всех элементов массива, кроме элементов с номерами от K до L включительно. Организуйте в Windows приложении ввод массива и вывод результата.

2. Организуйте в Windows приложении ввод массива «Сотрудники», содержащего фамилии сотрудников. Введите массив «Заявка», элементы которого содержат фамилии сотрудников и, следовательно, должны содержаться в массиве сотрудников. Обеспечьте контроль корректности ввода данных и выведите массив «Заявка».

3. Организуйте в Windows приложении ввод массива «Сотрудники», содержащего фамилии сотрудников. Создайте массив «Заявка», элементы которого должны содержаться в массиве сотрудников. Для создания массива «Заявка» постройте форму «Два списка», содержащую два элемента ListBox**,** источником данных для первого из них служит массив «Сотрудники». Пользователь переносит данные из первого списка во второй, формируя данные для массива «Заявка». После формирования данные переносятся в массив.

4. Организуйте в Windows приложении ввод массива «Студенты» из двух столбцов, содержащего фамилии и имена студентов. Введите массив «Общежитие» той же структуры, элементы которого должны содержаться в массиве сотрудников. Обеспечьте контроль корректности ввода данных. Организуйте вывод обоих массивов.

5. Организуйте в Windows приложении ввод и вывод массива «Машины», содержащего 4 столбца: «Владелец», «Марка», «Номер», «Год Выпуска». При вводе данных обеспечьте их корректность. Поле «Владелец» должно быть строкой в формате «фамилия имя», где фамилия и имя должны начинаться с большой буквы и состоять из букв алфавита кириллицы, включая дефис. Номер машины должен соответствовать формату, принятому для номеров машин. При выводе сохраняйте структуру массива.

6. Организуйте в Windows приложении ввод массива «Студенты», содержащего фамилии студентов, и массива «Стипендия». Обеспечьте контроль корректности ввода данных о стипендии, проверяя диапазон возможных значений, диапазон задается в полях на форме.

7. Организуйте в Windows приложении ввод и вывод матрицы – двумерного массива арифметического типа. Реализовать систему автоматического заполнения массива случайными числами.

8. Организуйте в Windows приложении ввод массива декартовых координат n точек на плоскости. Вычислите массив полярных координат этих точек и организуйте вывод этого массива. Обеспечьте контроль вводимых значений.

9. Дана матрица размера M × N. Поменять местами столбец с номером N и последний из столбцов, содержащих только положительные элементы. Если требуемых столбцов нет, то вывести матрицу без изменений.

10. Организуйте в Windows приложении ввод массива полярных координат **n** точек на плоскости. Вычислите массив декартовых координат этих точек и организуйте вывод этого массива. Обеспечьте контроль вводимых значений.

11. Организуйте в Windows приложении ввод массива декартовых координат n точек в трехмерном пространстве. Вычислите массив полярных координат этих точек и организуйте вывод этого массива. Обеспечьте контроль вводимых значений.

12. Даны целые положительные числа M и N. Сформировать целочисленную матрицу размера M × N, у которой все элементы J-го столбца имеют значение5·J(J= 1, …, N). Обеспечьте контроль вводимых значений. Заполнить массив случайными числами и вывести его в соответствии с заданием.

13. Заполнить матрицу случайными числами размера M × N и целое число K($1 \le K \le M$). Вывести элементы K-й строки данной матрицы. Обеспечьте контроль вводимых значений. Обеспечьте контроль вводимых значений.

14. Дана матрица размера M × N. Для каждой строки матрицы найти сумму ее элементов. Обеспечьте контроль вводимых значений.

15. Дана матрица размера M × N. В каждом столбце матрицы найти максимальный элемент. Обеспечьте контроль вводимых значений.

16. Дана матрица размера M × N. В каждой ее строке найти количество элементов, меньших среднего арифметического всех элементов этой строки. Обеспечьте контроль вводимых значений.

17. Дана матрица размера M × N и целые числа K1 и K2($1 \leq K1 < K2 \leq M$). Поменять местами строки матрицы с номерами K1 и K2. Обеспечьте контроль вводимых значений.

18. Дана матрица размера M × N. Преобразовать матрицу, поменяв местами минимальный и максимальный элемент в каждой строке.

19. Дана матрица размера M × N и целое число K($1 \leq K \leq N$). Удалить столбец матрицы с номером K.

20. Дана матрица размера M × N и целое число K($1 \leq K \leq M$). Перед строкой матрицы с номером K вставить строку из значений S. Обеспечьте контроль вводимых значений.

Контрольные вопросы

1. Что такое массивы?
2. Особенности работы с массивами в C#.
3. Компоненты работы с массивами в Windows Form.
4. Методы заполнения двумерных массивов в Windows Form.
5. Опишите принципы работы ListBox в Windows Form.
6. Опишите принципы работы CheckedListBox в Windows Form.
7. Опишите принципы работы ComboBox в Windows Form.
8. Опишите принципы работы DataGridView в Windows Form.

10. Лабораторная работа №7. Реализация алгоритмов поиска. Отладка приложения

Целью работы является изучение порядка работы при организации поиска данных. Результатом практической работы является отчет, в котором должны быть приведено описание разработанных алгоритмов и результаты отладки ПО.

Для выполнения лабораторной работы № 7 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Краткая теория

Поиск – это действие наиболее часто встречающееся в программировании. Он же представляет собой идеальную задачу, на которой можно испытывать различные структуры данных по мере их появления. Существует несколько основных «вариаций этой темы», и для них создано много различных алгоритмов. При дальнейшем рассмотрении мы исходим из такого принципиального допущения: группа данных, в которой необходимо отыскать заданный элемент, фиксирована. Будем считать, что множество из N элементов задано, скажем, в виде такого массива

a: ARRAY[*0..N-1*] OF item

Обычно тип item описывает запись с некоторым полем, выполняющим роль ключа. Задача заключается в поиске элемента, ключ которого равен заданному «аргументу поиска» x. Полученный в результате индекс i, удовлетворяющий условию $a[i].key=x$, обеспечивает доступ к другим полям обнаруженного элемента. Так как нас интересует в первую очередь сам процесс поиска, а не обнаруженные данные, то мы будем считать, что тип *item* включает только ключ, т.е. он есть ключ (*key*).

Линейный поиск

Если нет никакой дополнительной информации о разыскиваемых данных, то очевидный подход – простой последовательный просмотр массива с увеличением шаг за шагом той его части, где желаемого элемента не обнаружено. Такой метод называется линейным поиском. Условия окончания поиска таковы:

1. Элемент найден, т.е. $a[i] = x$.
2. Весь массив просмотрен и совпадения не обнаружено.

Это дает нам линейный алгоритм:

```
i := 0;
```

WHILE (i < N) AND (a[i] <> x) DO
i := i+1 ;
END;

Обратите внимание, что порядок элементов в логическом выражении имеет существенное значение. Инвариант цикла, т.е. условие, выполняющееся перед каждым увеличением индекса i, выглядит так:

(0 £ i < N) AND (A_k : 0 £ k < i : a_k ¹ x)

Он говорит, что для всех значений *k*, меньших чем *i*, совпадения не было. Отсюда и из того факта, что поиск заканчивается только в случае ложности условия в заголовке цикла, можно вывести окончательное условие его окончания:

((i = N) OR (a_i = x)) AND (A_k : 0 £ k < i : a_k ¹ x)

Это условие не только указывает на желаемый результат, но из него же следует, что если элемент найден, то он найден вместе с минимально возможным индексом, т. е. это первый из таких элементов. Равенство *i = N* свидетельствует, что совпадения не существует.

Совершенно очевидно, что окончание цикла гарантировано, поскольку на каждом шаге значение *i* увеличивается, и, следовательно, оно, конечно же, достигнет за конечное число шагов предела *N*; фактически же, если совпадения не было, это произойдет после *N* шагов.

Ясно, что на каждом шаге требуется увеличивать индекс и вычислять логическое выражение. А можно ли эту работу упростить и таким образом убыстрить поиск ?

Единственная возможность – попытаться упростить само логическое выражение, ведь оно состоит из двух членов. Следовательно, единственный шанс на пути к более простому решению – сформулировать простое условие, эквивалентное нашему сложному. Это можно сделать, если мы гарантируем, что совпадение всегда произойдет. Для этого достаточно в конец массива поместить дополнительный элемент со значением *x*. Назовем такой вспомогательный элемент «барьером», ведь он охраняет нас от перехода за пределы массива. Теперь массив будет описан так:

a: ARRAY[0..N] OF INTEGER

и алгоритм линейного поиска с барьером выглядит следующим образом:

a[N] := x;

i := 0;
WHILE a[i] <> x DO
i := i+1;
END;

Результирующее условие, выведенное из того же инварианта, что и прежде:

$(a_i=x)$ *AND* $(A_k : 0 £ k < i : a_k ¹ x)$

Ясно, что равенство $i = N$ свидетельствует о том, что совпадения (если не считать совпадения с барьером) не было.

Поиск делением пополам (двоичный поиск).

Совершенно очевидно, что других способов убыстрения поиска не существует, если, конечно, нет еще какой-либо информации о данных, среди которых идет поиск. Хорошо известно, что поиск можно сделать значительно более эффективным, если данные будут упорядочены. Вообразите себе телефонный справочник, в котором фамилии не будут расположены по порядку. Это нечто совершенно бесполезное! Поэтому мы приводим алгоритм, основанный на знании того, что массив a упорядочен, т. е. удовлетворяет условию

$A_k : 1£ k < N : a_{k-1} ¹ a_k$

Основная идея – выбрать случайно некоторый элемент, предположим a[m], и сравнить его с аргументом поиска x. Если он равен x, то поиск заканчивается, если он меньше x, то мы заключаем, что все элементы с индексами, меньшими или равными m, можно исключить из дальнейшего поиска; если же он больше x, то исключаются индексы больше и равные m. Это соображение приводит нас к следующему алгоритму (он называется «поиском делением пополам»). Здесь две индексные переменные L и R отмечают соответственно левый и правый конец секции массива a, где еще может быть обнаружен требуемый элемент.

L := 0;
R := N-1;
found := FALSE;
WHILE (L < R) AND NOT found DO
m := любое значение между L и R;
IF a[m] = x THEN found := TRUE;
IF a[m] < x THEN L := m+1
ELSE R := m-1;
ENDIF;

ENDWHILE;

Инвариант цикла, т.е. условие, выполняющееся перед каждым шагом, таков:

(L £ R) AND (A_k : 0 £ k < L : a_k < x) AND (A_k : R < k < N : a_k > x)

из чего выводится результат

found OR ((L > R) AND (A_k : 0 £ k < L : a_k < x) AND (A_k : R < k < N : a_k > x))

откуда следует

(a_m = x) OR (A_k : 0 £ k < N : ak ¹ x)

Выбор *m* совершенно произволен в том смысле, что корректность алгоритма от него не зависит. Однако на его эффективность выбор влияет. Ясно, что наша задача – исключить на каждом шагу из дальнейшего поиска, каким бы ни был результат сравнения, как можно больше элементов. Оптимальным решением будет выбор среднего элемента, так как при этом в любом случае будет исключаться половина массива. В результате максимальное число сравнений равно *log N*, округленному до ближайшего целого. Таким образом, приведенный алгоритм существенно выигрывает по сравнению с линейным поиском, ведь там ожидаемое число сравнений – *N/2*.

Эффективность можно несколько улучшить, поменяв местами заголовки условных операторов. Проверку на равенство можно выполнять во вторую очередь, так как она встречается лишь единожды и приводит к окончанию работы. Но более существенно следующее соображение: нельзя ли, как и при линейном поиске, отыскать такое решение, которое опять бы упростило условие окончания. И мы действительно находим такой быстрый алгоритм, как только отказываемся от наивного желания закончить поиск при фиксации совпадения. На первый взгляд это кажется странным, однако при внимательном рассмотрении обнаруживается, что выигрыш в эффективности на каждом шаге превосходит потери от сравнения с несколькими дополнительными элементами. Напомним, что число шагов в худшем случае – *log N*. Быстрый алгоритм основан на следующем инварианте:

(A_k : 0 £ k < L : a_k < x) AND (A_k : R £ k < N : a_k ³ x)

причем поиск продолжается до тех пор, пока обе секции не «накроют» массив целиком.

L := 0;

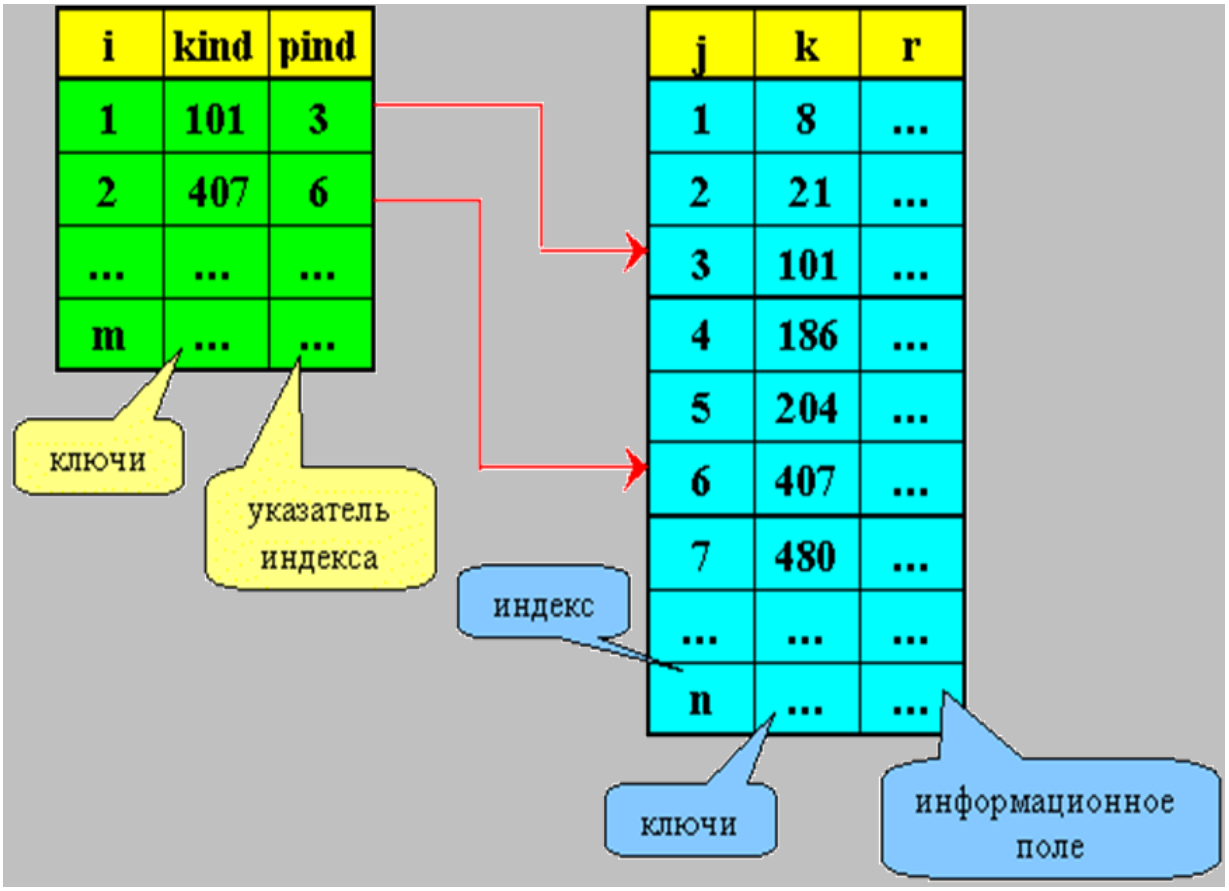
R := N;

```
  WHILE L < R DO
   m := (L+R) DIV 2;
   IF a[m] < x THEN L := m+1
   ELSE R := m ;
  END
END
```

Условие окончания – $L < R$, но достижимо ли оно? Для доказательства этого нам необходимо показать, что при всех обстоятельствах разность R-L на каждом шаге убывает. В начале каждого шага $L < R$. Для среднего арифметического m справедливо условие $L < m < R$. Следовательно, разность действительно убывает, ведь либо L увеличивается при присваивании ему значения $m+1$, либо R уменьшается при присваивании значения m. При L = R повторение цикла заканчивается. Однако наш инвариант и условие $L = R$ еще не свидетельствуют о совпадении. Конечно, при $R = N$никаких совпадений нет. В других же случаях мы должны учитывать, что элемент $a[R]$ в сравнениях никогда не участвует. Следовательно, необходима дополнительная проверка на равенство $a[R] = x$. В отличие от первого нашего решения приведенный алгоритм, как и в случае линейного поиска, находит совпадающий элемент с наименьшим индексом.

Еще одним вариантом убыстрения поиска в случае упорядоченности данных является индексно-последовательный поиск. При таком поиске организуется две таблицы: таблица данных со своими ключами – упорядоченных по возрастанию, и таблица ***индексов***, которая тоже состоит из ключей данных, но эти ключи взяты из основной таблицы через определенный интервал. Другими словами, при прохождении упорядоченной таблицы мы сравниваем с ключом элементы не последовательно, а через определенный интервал, то есть задаем некоторый шаг поиска. Когда на очередном шаге поиска предыдущий элемент меньше значения ключа, а следующий элемент больше значения ключа, то устанавливаются соответственно нижняя low (kind<key) и верхняя hi (kind>key) границы в основной таблице. Между этими границами по основной таблице будет соответственно произвен уже обычный последовательный поиск.

Таблицы индексно-последовательного поиска



В псевдокоде алгоритм индексно-последовательного поиска следующий:

```
i = 1
while (i <= m) and (kind(i) <= key) do
        i=i+1
endwhile
if i = 1 then low = 1
          else low = pind(i-1)
endif
if i = m+1 then hi = n
                  else hi = pind(i)-1
endif
for j = low to hi
        if key = k(j) then
                        search = j
                         return
        endif
next j
```

search = 0
return
Эффективность данного вида поиска величина
$O(\sqrt{n})$

Данный вид поиска, также как и бинарный, может давать значительный эффект при больших размерах таблиц поиска.

В принципе, для достижения наибольшей эффективности поиска при решении конкретных задач пользователь может задавать какой угодно шаг.

Контрольные вопросы

1. Сформулируйте задачу поиска данных.
2. Перечислите основные алгоритмы поиска данных.
3. Приведите эффективность различных алгоритмов поиска данных.

11. Лабораторная работа №8. Реализация обработки табличных данных. Отладка приложения

Целью работы является изучение порядка работы при организации обработки табличных данных. Результатом практической работы является отчет, в котором должны быть приведено описание разработанных алгоритмов и результаты отладки ПО.

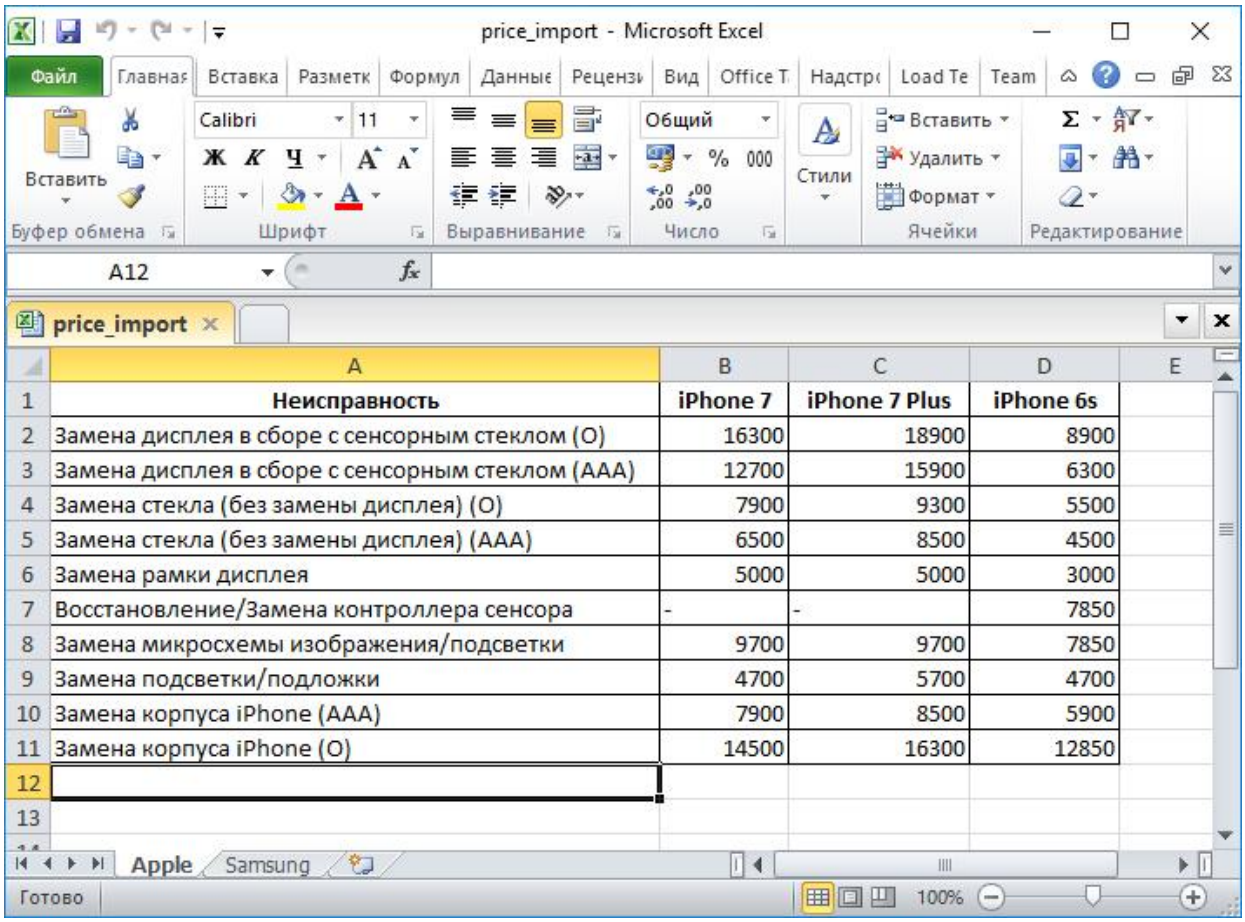
Для выполнения лабораторной работы № 8 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Работа с типом файлов Microsoft Excel на сайте – это довольно частая задача. Импорт/экспорт прайса или номенклатуры товаров в интернет-магазине, выгрузка отчета из базы данных в виде таблицы, да все что угодно. В этом уроке посмотрим, как можно обрабатывать файлы такого типа.

На сегодняшний день существует достаточно много готовых решений (библиотек) для взаимодействия с файлами Excel, чтобы не работать напрямую с системными GridView и DataTable. Не будем изобретать велосипед и рассмотрим одну из популярных и удобных библиотек, которой я пользуюсь чаще других.

Данная библиотека называется ClosedXML. По этой ссылке находится официальный адрес проекта на GitHub. Библиотека позволяет легко манипулировать файлами MS Excel в удобной объектно-ориентированной манере. Она может быть использована на любом .NET языке программирования (C#, VisualBasic.NET), а также в проектах не только типа WebApplication.

Представим ситуацию, что у нас есть под рукой вот такой Excel-файл с ценами на ремонт телефонов, в котором на листах расположены отдельные бренды, колонки на листе представляют конкретные модели для этого бренда, а строки представляют конкретные неисправности и сколько будет стоить ремонт для той или иной модели.



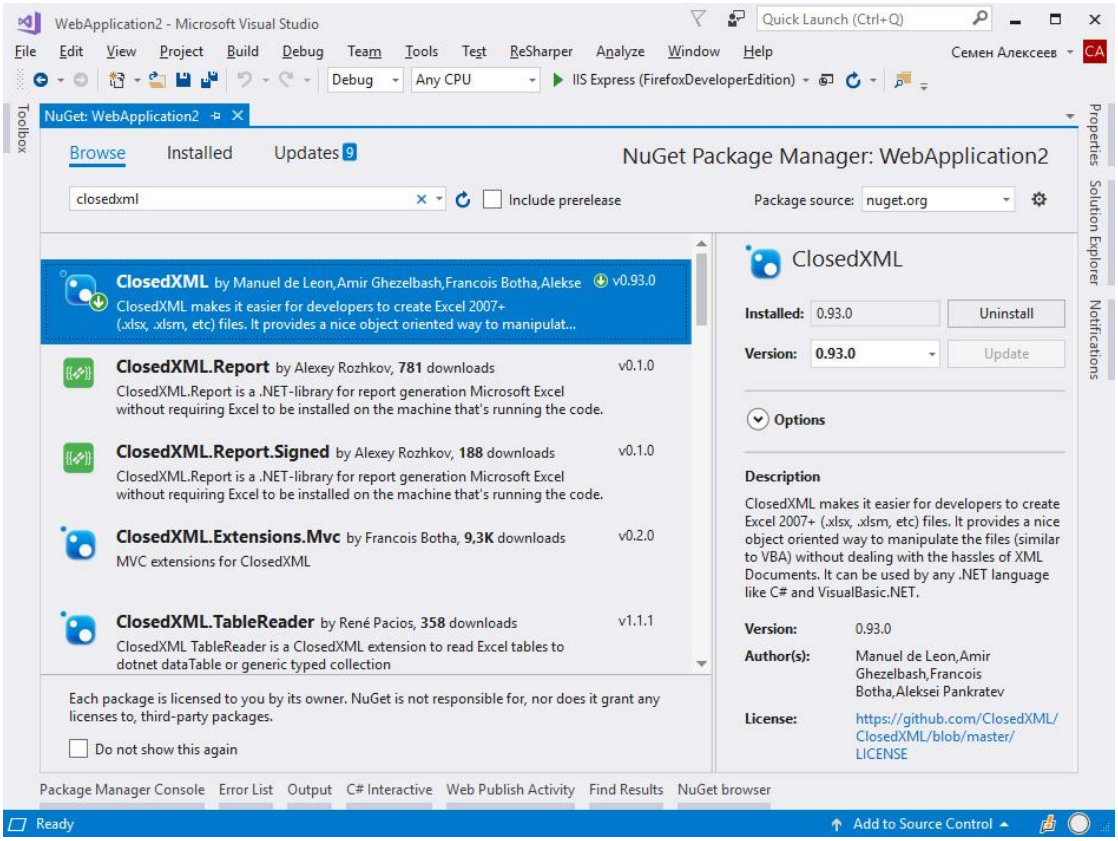
price_import - Microsoft Excel

	A	B	C	D	E
1	**Неисправность**	**iPhone 7**	**iPhone 7 Plus**	**iPhone 6s**	
2	Замена дисплея в сборе с сенсорным стеклом (O)	16300	18900	8900	
3	Замена дисплея в сборе с сенсорным стеклом (AAA)	12700	15900	6300	
4	Замена стекла (без замены дисплея) (O)	7900	9300	5500	
5	Замена стекла (без замены дисплея) (AAA)	6500	8500	4500	
6	Замена рамки дисплея	5000	5000	3000	
7	Восстановление/Замена контроллера сенсора	-	-	7850	
8	Замена микросхемы изображения/подсветки	9700	9700	7850	
9	Замена подсветки/подложки	4700	5700	4700	
10	Замена корпуса iPhone (AAA)	7900	8500	5900	
11	Замена корпуса iPhone (O)	14500	16300	12850	
12					
13					

Прайс-лист на услуги в виде Excel-файла

Давайте создадим веб-приложение, которое сможет распарсить этот файл так, чтобы мы могли работать с записями в объектно-ориентированной манере, например, сохранять/обновлять бренды, модели и позиции в прайсе в базу данных. Или, например, представлять этот прайс в виде обычной html-таблицы в браузере пользователя.

Создаем в Visual Studio проект типа MVC, открываем Nuget Manager и скачиваем нужный нам пакет.



Добавляем пакет ClosedXML в проект

Создадим все доменные классы-модели, которые описывают предметную область.

Кстати, внизу страницы есть ссылка на архив с исходным кодом проекта.

```
namespace WebApplication2.Models
{
    public class PricePosition
    {
        //Неисправность
        public string Problem { get; set; }

        //Стоимость ремонта
        public string Price { get; set; }
    }

    public class PhoneModel
    {
        public PhoneModel()
        {
            PricePositions = new List<PricePosition>();
```

```
        }

        //Название модели телефона
        public string Title { get; set; }

        public List<PricePosition> PricePositions { get; set; }
    }

    public class PhoneBrand
    {
        public PhoneBrand()
        {
            PhoneModels = new List<PhoneModel>();
        }

        //Название бренда
        public string Title { get; set; }

        public List<PhoneModel> PhoneModels { get; set; }
    }
}
```

Описанные модели максимально простые, чтобы не усложнять пример. В них нет и не отслеживается уникальных идентификаторов, все свойства типа String и т. д.

Также определим т. н. ViewModel, то есть модель для представления. В нее заключим все объекты доменной модели, которые мы хотим показать пользователю в браузере:

```
public class PriceViewModel
{
    public PriceViewModel()
    {
        PhoneBrands = new List<PhoneBrand>();
    }

    public List<PhoneBrand> PhoneBrands { get; set; }

    //кол-во ошибок при импорте
    public int ErrorsTotal { get; set; }
```

```
}
```

Далее в вашем проекте определите какое-нибудь действие в нужном контроллере, которое будет отвечать за загрузку Excel-файла из браузера на сервер. Пример кода с html-формой в соответствующем представлении:

```
<div>
    @using (Html.BeginForm(«Import», «Home», FormMethod.Post, new { enctype = «multipart/form-data», id = «frm-excel» }))
    {
        <div>
            Загрузите Excel-файл:
            <input type=«file» name=«fileExcel» id=«fileExcel» />
            <div>
                <input type=«submit» value=«Загрузить» />
            </div>
        </div>
    }
</div>
```

Из примера видно, что в форме мы обращаемся к методу Import в контроллере Home. Создадим подобный метод:

```
[HttpPost]
public ActionResult Import(HttpPostedFileBase fileExcel)
{
    if (ModelState.IsValid)
    {
        PriceViewModel viewModel = new PriceViewModel();
        using (XLWorkbook workBook = new XLWorkbook(fileExcel.InputStream, XLEventTracking.Disabled))
        {
            foreach (IXLWorksheet worksheet in workBook.Worksheets)
            {
                PhoneBrand phoneBrand = new PhoneBrand();
                phoneBrand.Title = worksheet.Name;

                foreach (IXLColumn column in worksheet.ColumnsUsed().Skip(1))
                {
```

```
                    PhoneModel phoneModel = new PhoneModel();
                    phoneModel.Title = column.Cell(1).Value.ToString();

                    foreach (IXLRow row in work-
sheet.RowsUsed().Skip(1))
                    {
                        try
                        {
                            PricePosition pricePosition = new PricePosition();
                            pricePosition.Problem                                =
row.Cell(1).Value.ToString();
                            pricePosition.Price                                  =
row.Cell(column.ColumnNumber()).Value.ToString();
                            phoneModel.PricePositions.Add(pricePosition);

                        }
                        catch (Exception e)
                        {
                            //logging
                            viewModel.ErrorsTotal++;
                        }
                    }

                    phoneBrand.PhoneModels.Add(phoneModel);
                }
                viewModel.PhoneBrands.Add(phoneBrand);
            }
        }
        //например, здесь сохраняем все позиции из прайса в БД

        return View(viewModel);
    }
    return RedirectToAction(«Index»);
}
```

В этом методе мы парсим Excel-файл и манипулируем записями в объектно-ориентированной манере. Код в методе довольно простой. Более подробно он объясняется в видео-версии этой статьи. Здесь отмечу основные моменты:

- в нескольких циклах foreach{ } мы пробегаемся по всем записям в файле, параллельно создавая объекты классов наших доменных моделей;
- получается сформированная коллекция брендов телефонов, в каждом из которых содержится коллекция конкретных моделей, в каждой из которых содержится коллекция позиций прайса с ценами на ремонт;
- также создается ViewModel, где мы считаем количество ошибок при импорте и в которую вкладываем заполненную коллекцию брендов;
- в итоге мы можем либо сохранить полученные объекты в базу данных, либо отправить ViewModel в представление.

В качестве интерактива Вы можете в соответствующем представлении создать HTML-таблицу с прайсом и отправить ее в браузер пользователя.

Также возможна другая ситуация, когда у нас нет исходного Excel-файла, вместо этого веб-приложение должно сформировать его динамически, и пользователь сайта сможет его скачать. Например, тот же список брендов и моделей телефонов.

Создадим соответствующее действие в контроллере:

```
public ActionResult Export()
{
   List<PhoneBrand> phoneBrands = new List<PhoneBrand>();
   phoneBrands.Add(new PhoneBrand()
   {
      Title = «Apple»,
      PhoneModels = new List<PhoneModel>()
   {
      new PhoneModel() { Title = «iPhone 7»},
      new PhoneModel() { Title = «iPhone 7 Plus»}
   }});
   phoneBrands.Add(new PhoneBrand()
   {
      Title = «Samsung»,
      PhoneModels = new List<PhoneModel>()
   {
      new PhoneModel() { Title = «A3»},
      new PhoneModel() { Title = «A3 2016»},
```

```
            new PhoneModel() { Title = «A3 2017»}
        }});

        using (XLWorkbook workbook = new XLWorkbook(XLEventTracking.Disabled))
        {
            var worksheet = workbook.Worksheets.Add(«Brands»);

            worksheet.Cell(«A1»).Value = «Бренд»;
            worksheet.Cell(«B1»).Value = «Модели»;
            worksheet.Row(1).Style.Font.Bold = true;

            //нумерация строк/столбцов начинается с индекса 1 (не 0)
            for (int i = 0; i < phoneBrands.Count; i++)
            {
                worksheet.Cell(i + 2, 1).Value = phoneBrands[i].Title;
                worksheet.Cell(i + 2, 2).Value = string.Join(«, «, phoneBrands[i].PhoneModels.Select(x => x.Title));
            }

            using (var stream = new MemoryStream())
            {
                workbook.SaveAs(stream);
                stream.Flush();

                return new FileContentResult(stream.ToArray(),
                    «application/vnd.openxmlformats-officedocument.spreadsheetml.sheet»)
                {
                    FileDownloadName = $»brands_{DateTime.UtcNow.ToShortDateString()}.xlsx»
                };
            }
        }
    }
```

Отмечу ключевые моменты из листинга выше:

- в нашем простом примере список брендов и моделей мы жестко закодировали, но его также можно получить и из БД;
- к ячейкам на рабочем листе можно обращаться как по литеральному значению, так и по индексу. Нумерация строк/столбцов начинается с индекса 1 (не 0);
- в качестве MIME-type для файлов Excel следует указывать application/vnd.openxmlformats-officedocument.spreadsheetml.sheet;

Контрольные вопросы

1. Укажите порядок экспорта данных в Excel.
2. Какие базовые пакеты необходимо установить в VS для работы с данными в Excel?

12. Лабораторная работа №9. Разработка и отладка генератора случайных символов

Целью работы является изучение порядка работы генератора случайных чисел. Результатом практической работы является отчет, в котором должны быть приведено описание разработанных алгоритмов и результаты отладки ПО.

Для выполнения лабораторной работы № 9 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

1. Способы получения случайных чисел

В программировании достаточно часто находят применение последовательности чисел, выбранных случайным образом из некоторого множества. В качестве примеров задач, в которых используются случайные числа, можно привести следующие:

- тестирование алгоритмов;
- имитационное моделирование;
- некоторые задачи численного анализа;
- имитация пользовательского ввода.

Для получения случайных чисел можно использовать различные способы. В общем случае все методы генерирования случайных чисел можно разделить на аппаратные и программные. Устройства или алгоритмы получения случайных чисел называют генераторами случайных чисел (ГСЧ) или датчиками случайных чисел.

Аппаратные ГСЧ представляют собой устройства, преобразующие в цифровую форму какой-либо параметр окружающей среды или физического процесса. Параметр и процесс выбираются таким образом, чтобы обеспечить хорошую «случайность» значений при считывании. Очень часто используются паразитные процессы в электронике (токи утечки, туннельный пробой диодов, цифровой шум видеокамеры, шумы на микрофонном входе звуковой карты и т. п.). Формируемая таким образом последовательность чисел, как правило, носит абсолютно случайный характер и не может быть воспроизведена заново по желанию пользователя.

К программным ГСЧ относятся различные алгоритмы генерирования последовательности чисел, которая по своим характеристикам напоминает случайную. Для формирования очередного числа

последовательности используются различные алгебраические преобразования. Одним из первых программных ГСЧ является метод средин квадратов, предложенный в 1946 г. Дж. фон Нейманом. Этот ГСЧ формирует следующий элемент последовательности на основе предыдущего путем возведения его в квадрат и выделения средних цифр полученного числа. Например, мы хотим получить 10-значное число и предыдущее число равнялось 5772156649. Возводим его в квадрат и получаем 33317792380594909201; значит, следующим числом будет 7923805949. Очевидным недостатком этого метода является зацикливание в случае, если очередное число будет равно нулю. Кроме того, существуют и другие сравнительно короткие циклы.

Любые программные ГСЧ, не использующие внешних «источников энтропии» и формирующие очередное число только алгебраическими преобразованиями, не дают чисто случайных чисел. Последовательность на выходе такого ГСЧ выглядит как случайная, но на самом деле подчиняется некоторому закону и, как правило, рано или поздно зацикливается. Такие числа называются псевдослучайными.

В дальнейшем мы будем рассматривать лишь программные генераторы псевдослучайных чисел.

2. Характеристики ГСЧ

Последовательности случайных чисел, формируемых тем или иным ГСЧ, должны удовлетворять ряду требований. Во-первых, числа должны выбираться из определенного множества (чаще всего это действительные числа в интервале от 0 до 1 либо целые от 0 до N). Во-вторых, последовательность должна подчиняться определенному распределению на заданном множестве (чаще всего распределение равномерное). Необязательным является требование воспроизводимости последовательности. Если ГСЧ позволяет воспроизвести заново однажды сформированную последовательность, отладка программ с использованием такого ГСЧ значительно упрощается. Кроме того, требование воспроизводимости часто выдвигается при использовании ГСЧ в криптографии.

Поскольку псевдослучайные числа не являются действительно случайными, качество ГСЧ очень часто оценивается по «случайно-

сти» получаемых чисел. В эту оценку могут входить различные показатели, например, длина цикла (количество итераций, после которого ГСЧ зацикливается), взаимозависимости между соседними числами (могут выявляться с помощью различных методов теории вероятностей и математической статистики) и т. п. Подробнее оценка качества ГСЧ рассмотрена ниже.

3. Применение ГСЧ

Одна из задач, в которых применяются ГСЧ, – это грубая оценка объемов сложных областей в евклидовом пространстве более чем четырех или пяти измерений. Разумеется, сюда входит и приближенное вычисление интегралов. Обозначим область через R; обычно она определяется рядом неравенств. Предположим, что R – подмножество n-мерного единичного куба K. Вычисление объема множества R методом Монте-Карло сводится к тому, чтобы случайным образом выбрать в K большое число N точек, которые с одинаковой вероятностью могут оказаться в любой части K. Затем подсчитывают число M точек, попавших в R, т. е. удовлетворяющих неравенствам, определяющим R. Тогда M/N есть оценка объема R. Можно показать, что точность такой оценки будет довольно низкой. Тем не менее, выборка из 10 000 точек обеспечит точность около 1%, если только объем не слишком близок к 0 или 1. Такой точности часто бывает достаточно, и добиться лучшего другими методами может оказаться очень трудно.

В качестве примера можно рассмотреть вычисление площади фигуры, заданной некоторой системой неравенств. Пусть фигура будет определена следующим образом:

$$\begin{cases} y \le x^2 \\ 0 \le x \le 1 \\ y \ge -1 \end{cases}.$$

Сначала необходимо определить прямоугольную область, из которой будут выбираться случайные точки. Это может быть любая область, полностью содержащая фигуру, площадь которой требуется найти. Возьмем в качестве исходной области прямоугольник с координатами углов (0; –1) – (1; 1). Будем последовательно генери-

ровать точки, равномерно распределенные внутри этого прямоугольника, и для каждой точки проверять неравенства, описывающие фигуру. Если точка удовлетворяет всем неравенствам, значит, она принадлежит фигуре. При достаточно большом числе таких экспериментов отношение числа точек NF, удовлетворяющих неравенствам, к общему числу сгенерированных точек NR показывает долю площади прямоугольника, которую занимает фигура. Площадь прямоугольника SR известна (в нашем случае она равна 2), площадь фигуры SF вычисляется тривиально:

$$S_F = \frac{N_F}{N_R} S_R .$$

Очевидно, что для такой простой области можно легко посчитать область через определенный интеграл. Тем не менее, описанный метод применим и в случае гораздо более сложных фигур, когда рассчитать площадь другим способом становится слишком сложно.

Другим примером приближенного взятия определенного интеграла с помощью ГСЧ является вычисление объема шара в n-мерном пространстве. Объем n-мерного шара выражается формулой:

$$V_n = \frac{\pi^{\frac{n}{2}} r^n}{\Gamma\left(\frac{n}{2}+1\right)},$$

где Γ(z) – некоторая гамма-функция, определяемая следующим соотношением:

$$\Gamma(z+1) = z \cdot \Gamma(z),$$
$$\Gamma(1) = 1.$$

Таким образом, для натуральных *z* гамма-функция равна факториалу *z*. Для вычисления знаменателя можно воспользоваться известным значением $\Gamma\left(\frac{1}{2}\right)$:

$$\Gamma\left(\frac{1}{2}\right)=\sqrt{\pi}\,.$$

Можно показать, что для шара единичного радиуса при увеличении размерности n объем стремится к нулю. Наиболее просто это можно объяснить тем, что числитель растет со скоростью степенной функции, а знаменатель – с факториальной. Таким образом, для больших n метод вычисления через случайные числа будет давать значительные погрешности.

4. Генерирование равномерно распределенных случайных чисел

Почти повсеместно используемый метод генерирования псевдослучайных целых чисел состоит в выборе некоторой функции f, отображающей множество целых чисел в себя. Выбирается какое-нибудь начальное число x_0, а каждое следующее число порождается с помощью рекуррентного соотношения:

$$x_{k+1} = f(x_k)$$

Число x_k часто называется зерном (англ. seed) ГСЧ и полностью определяет текущее состояние ГСЧ и следующее генерируемое значение.

Поначалу функции f выбирались как можно более сложные и трудно понимаемые. Например, f(x) определялась как целое число, двоичное представление которого составляет средний 31 разряд 62-разрядного квадрата числа x (модификация метода средин квадратов). Но отсутствие теории относительно f приводило к катастрофическим последствиям. Для метода средин квадратов это уже упоминавшееся зацикливание при обращении очередного числа в нуль. Поэтому уже довольно давно перешли к использованию функций, свойства которых вполне известны. Всякая последовательность целых чисел из интервала (0, 231–1) должна содержать повторения самое большое после 231≈109 элементов. Используя теорию чисел, можно выбрать такую функцию f, для которой наперед будет известно, что ее период максимально возможный или близкий к максимальному. Этим избегается преждевременное окончание или за-

цикливание последовательности. Дальнейшее использование теории чисел может более или менее предсказать характер последовательности, давая пользователю некоторую степень уверенности в том, что она будет достаточно хорошо моделировать случайную последовательность чисел.

Представим генерирование чисел в диапазоне [0; 1] рекуррентым методом графически (см. рис. 1). Очевидно, функция f(x) должна быть определена на всем отрезке [0; 1] и иметь на этом отрезке непрерывную область значений [0; 1], в противном случае генерируемые числа будут составлять лишь несобственное подмножество указанного отрезка.

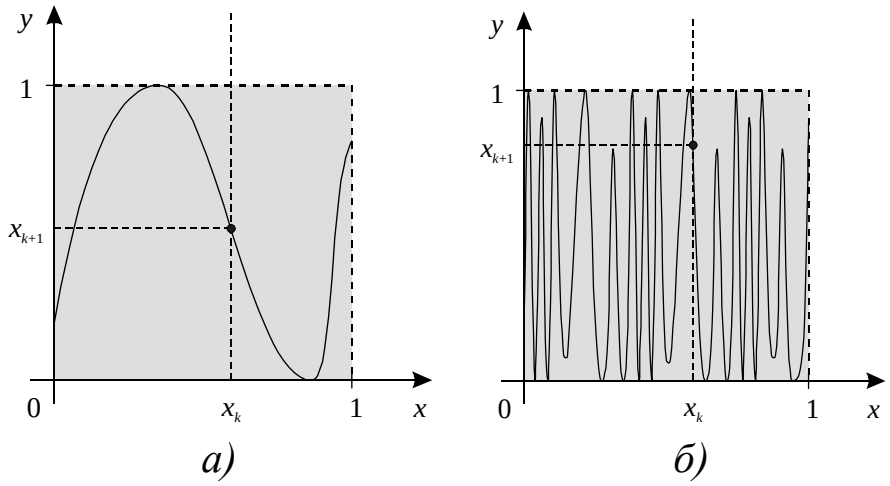


Рис. 1. Графическое представление рекуррентного ГСЧ:
а) с «плохой» функцией *f*(*x*); б) с «хорошей» функцией *f*(*x*)

Считается, что функция f(x) тем лучше подходит для генерирования случайных чисел, чем более плотно и равномерно ее график заполняет область $x \in [0; 1]$, $y \in [0; 1]$. Например, функция, приведенная на рис. 1, а, будет давать последовательность чисел с сильной корреляционной зависимостью соседних элементов. В случае функции, приведенная на рис. 1, б, эта зависимость будет значительно слабее.

В настоящее время широкое распространение получили линейные конгруэнтные ГСЧ. В таком ГСЧ каждое следующее число получается на основе единственного предыдущего, при этом используется функция f вида:

$$f(x) = (ax+c) \bmod m,$$

где для n-разрядных двоичных целых чисел m обычно равно 2n.

Конгруэнтный ГСЧ выдает псевдослучайные целые числа в интервале (0, m). Параметры x0, a и c – целые числа из той же области, выбираемые исходя из следующих соображений:

1. x_0 может быть произвольно. Для проверки программы возможно x_0=1. В дальнейшем в качестве x_0 можно брать текущее время, преобразованное в число из интервала (0, m). Такой подход обеспечивает различные последовательности для различных запусков программы.

2. Выбор a должен удовлетворять трем требованиям (для двоичных машин):

a) $a \bmod 8 = 5$;

b) $\frac{m}{100} < a < m - \sqrt{m}$;

c) двоичные знаки a не должны иметь очевидного шаблона.

3. В качестве c следует выбирать нечетное число, такое, что

$$\frac{c}{m} \approx \frac{1}{2} - \frac{1}{6}\sqrt{3} \approx 0{,}21132.$$

Более подробные рекомендации по выбору параметров можно найти у Д. Кнута.

При использовании конгруэнтного ГСЧ следует помнить, что наименее значимые двоичные цифры xk будут «не очень случайными». Поэтому, если, например, вы хотите использовать число xk для случайного выбора одной из 16 возможных ветвей, берите наиболее значимые разряды xk, а не наименее значимые. Наконец, для большей надежности полезно предварительно испытать случайные числа на какой-либо задаче с известным ответом, схожей с реальным приложением.

5. Генерирование чисел с произвольным распределением

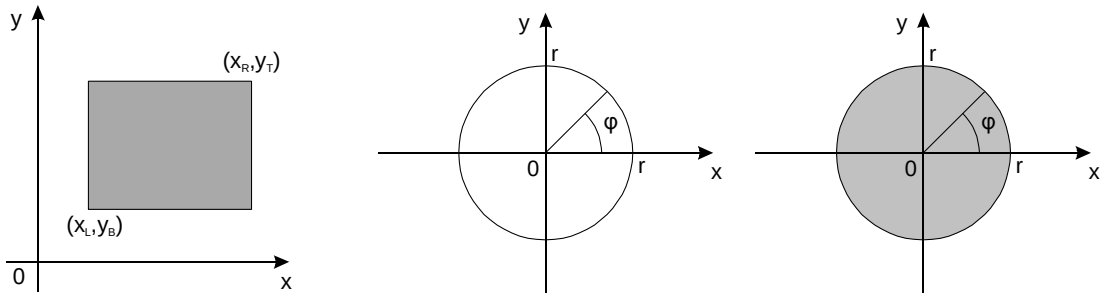
Достаточно часто возникает необходимость сгенерировать последовательность случайных чисел y_i, равномерно распределенных на данном конечном интервале [a, b], с помощью ГСЧ, выдающего числа x_i на интервале [0, m]. Приведение диапазона ГСЧ к нужному

интервалу в этом случае осуществляется простым линейным преобразованием:

$$y_i = \frac{b-a}{m} x_i + a.$$

Распределение чисел после такого преобразования остается равномерным.

Более сложным случаем является генерирование случайных точек из некоторого множества в n-мерном пространстве Rn, например, точек из некоторой области на плоскости. Рассмотрим формирование случайных точек для нескольких простых областей: прямоугольника, окружности и круга.



а) *б)* *в)*

Рис. 2. Области, из которых выбираются точки

Для получения равномерно распределенных случайных чисел из прямоугольника, стороны которого параллельны осям координат (см. рис. 2, а), достаточно извлекать из ГСЧ последовательно пары чисел, приводить их к нужным интервалам и использовать как координаты точки:

$$x_i = \frac{x_R - x_L}{m} u_{2i} + x_L$$
$$y_i = \frac{y_T - y_B}{m} u_{2i+1} + y_B,$$

где u_j – равномерно распределенное случайное число из отрезка [0, m].

Окружность можно представить одномерным множеством точек с угловой координатой φ, принимающей значения на интервале (0, 2π). Таким образом, декартовы координаты очередной точки можно вычислить следующим образом:

$$\phi_i = \frac{2\pi}{m} u_i$$
$$x_i = r \cos \phi_i$$
$$y_i = r \sin \phi_i .$$

где uj – равномерно распределенное случайное число из интервала (0, m); r – радиус окружности.

В случае круга первое, что приходит в голову – воспользоваться полярной системой координат (ρ, φ), в которой данное множество фактически представляет собой прямоугольник (а для него способ генерации чисел известен). Однако при переходе от полярных координат к декартовым нарушается распределение случайных чисел: оно становится неравномерным; плотность распределения в центре круга выше, чем по краям.

Существует несколько способов получения равномерного распределения по кругу. Рассмотрим один из них. Будем генерировать случайные пары (x, y) и для каждой из них ставить внутри круга соответствующую точку, заполняя таким образом эту область. Исходя из представлений о равномерном распределении можно предположить, что при достаточно большой длине сгенерированной последовательности на единицу площади круга будет приходиться примерно одно и то же количество точек вне зависимости от их расположения (другими словами, при равномерном распределении плотность точек по кругу будет одинакова).

Воспользуемся полярной системой координат для генерирования точек. При этом будем выбирать угол φ равномерно распределенным на интервале (0; 2π), а распределение ρ построим следующим образом:

$$\rho = r\sqrt{x},$$

где x – равномерно распределенная на отрезке [0; 1] случайная величина. Можно показать, что при таком способе формирования координат случайные точки будут равномерно распределены по всей площади круга.

Помимо выбора из произвольного множества, часто требуется формировать числа с распределением, отличным от равномерного. Распределение обычно задается функцией плотности распределения f(x) либо функцией распределения F(x). Функция распределения в произвольной точке x показывает вероятность того, что случайная величина X окажется меньше данного значения x:

$$F(x)=P\ (X<x).$$

Функция плотности распределения представляет собой производную $F(x)$:

$$f(x)=\frac{dF(x)}{dx}\ .$$

Функция $F(x)$ для любой случайной величины является неубывающей на всем интервале $(-\infty; +\infty)$, стремится к 0 при $x\to -\infty$ и к 1 при $x\to +\infty$. Для получения случайных чисел с заданным распределением $F(x)$ необходимо найти функцию, обратную к $F(x)$, т.е. такую функцию G, что для всех $y=F(x)$ выполняется $G(y)=x$. Это можно пояснить следующим образом. Предположим, что мы многократно выбираем число y, равномерно распределенное на интервале [0; 1]; каждому y мы ставим в соответствие некоторое x=G(y). Выбору 50000 игреков соответствует выбор 50000 иксов. Таким образом, доля выбранных y, лежащих между двумя фиксированными значениями, скажем y_1 и y_2, в точности равна доле x, лежащих в интервале $[x_1; x_2]$. Но вероятность первого из названных событий равна $|\ y_2 - y_1\ |$, если y распределено равномерно; следовательно, верна цепочка равенств:

доля чисел в интервале $[x_1; x_2]$ = доля чисел в интервале $[y_1; y_2]$

$$= y_2 - y_1 = F(x_2) - F(x_1) = \int_{-\infty}^{x_2} f(\tau)d\tau - \int_{-\infty}^{x_1} f(\tau)d\tau = \int_{x_1}^{x_2} f(\tau)d\tau\ ,$$

которая и показывает, что в случае равномерного распределения игреков x имеет распределение с плотностью $f(\tau)$. Сложной проблемой

в этом подходе является достаточно быстрое и точное формирование обратной функции распределения $G(y)$.

Рассмотрим в качестве примера получение случайного числа с экспоненциальным распределением. Это распределение характеризуется одним параметром $\lambda>0$ и имеет следующие функции распределения и плотности распределения:

$$f(x)=\lambda e^{-\lambda x},\ x\geq 0;$$
$$F(x)=\int_0^x f(t)dt=\int_0^x \lambda e^{-\lambda t}dt=1-e^{-\lambda x}.$$

Для этого распределения легко получить $F^{-}1$ (y), т. е. разрешить уравнение $F(x)=y$. Решение имеет вид

$$x=-\frac{\ln(1-y)}{\lambda}.$$

Для получения x с искомым распределением нужно сгенерировать y, равномерно распределенное на (0,1), и применить эту формулу. Если говорить о практической стороне дела, то существуют более эффективные способы, в которых не используется медленная операция вычисления логарифма для каждого случайного числа. Данный способ продемонстрирован лишь как пример более общего подхода с использованием обратной функции распределения.

6. Тестирование ГСЧ

Качество ГСЧ в значительной мере влияет на результаты работы программ, использующих случайные числа. Поэтому все применяемые генераторы случайных чисел должны пройти перед моделированием системы предварительное тестирование, которое представляет собой комплекс проверок по различным стохастическим критериям, включая в качестве основных тесты на равномерность, стохастичность и независимость (рассматриваются только ГСЧ с равномерным распределением).

Проверка равномерности последовательностей псевдослучайных равномерно распределенных чисел $\{x_i\}$ может быть выполнена по гистограмме с присваиванием косвенных признаков. Суть про-

верки по гистограмме сводится к следующему. Выдвигается гипотеза о равномерности распределения чисел (0, 1). Затем интервал (0, 1) разбивается на m равных частей, тогда при генерации последовательности {xi} каждое из чисел xi с вероятностью $p_j = \frac{1}{m}$, $j = \overline{1, m}$, попадет в один из подынтервалов. Всего в каждый j-й подынтервал попадает Ni чисел последовательности {xi}, $i = \overline{1, N}$, причём $N = \sum_{j=1}^{m} N_j$. Относительная частота попадания случайных чисел из последовательности {xi} в каждый из подынтервалов будет равна Nj/N. Очевидно, что если числа xi принадлежат псевдослучайной квазиравномерно распределенной последовательности, то при достаточно больших N экспериментальная гистограмма (ломаная линия на рис.3, а) приближается к теоретической прямой 1/m. Оценка степени приближения, т. е. равномерности последовательности {xi}, может быть проведена с использованием критериев согласия.

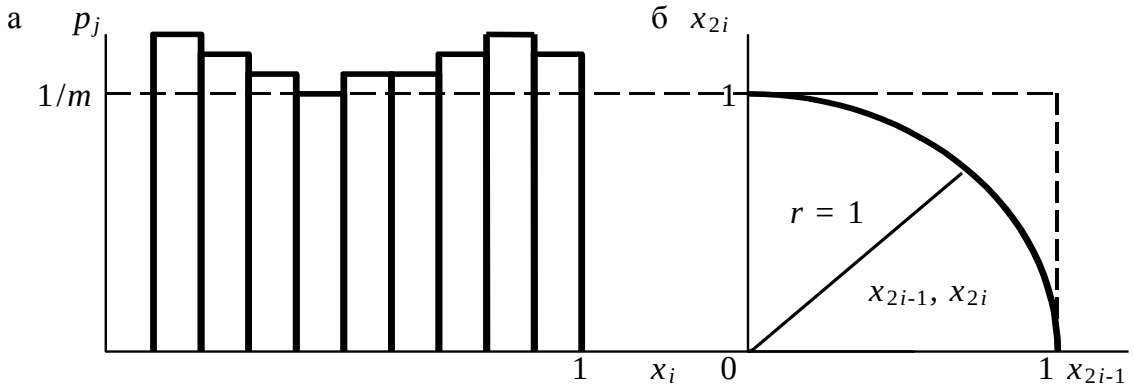


Рис. 3. Проверка равномерности последовательности

Существуют и другие способы проверки равномерности распределения.

Проверка стохастичности последовательности псевдослучайных чисел {xi} наиболее часто проводится методами комбинаций и серий. Сущность метода сводится к определению закона распределения длин участков между единицами (нулями) или закона распределения (появления) числа единиц (нулей) в n-разрядном двоичном числе Xi.

Теоретически закон появления j единиц в l разрядах двоичного числа Xi описывается, исходя из независимости отдельных разрядов, биномиальным законом распределения:

$$P(j,l) = C_l^j p^j(1)[1 - p(1)]^{l-j} = C_l^j p^l(1),$$

где P (j, l) – вероятность появления j единиц в l разрядах числа Xi; p(1) = p(0) = 0,5 – вероятность появления единицы и нуля в любом разряде числа Xi;

$$C_l^j = \frac{l!}{j!(l-j)!}.$$

Тогда при фиксированной точке выборки N теоретически ожидаемое число появления случайных чисел Xi с j единицами в проверяемых l разрядах будет равно $n_j = NC_l^j p^l(1)$.

После нахождения теоретических и экспериментальных вероятностей P (j, l) или чисел nj при различных значениях l ≤ n гипотеза о стохастичности проверяется с использованием критериев согласия, которые подробно рассматриваются в курсе математической статистики.

При анализе стохастичности последовательности чисел {xi} методом серий последовательность разбивается на элементы первого и второго рода (a и b), т. е.

$$x_i = \begin{cases} a, & \text{если } x_i < p, \\ b & \text{в противном случае,} \end{cases}$$

где 0 < p < 1.

Серией называется отрезок последовательности {xi}, состоящий из идущих друг за другом элементов одного и того же рода. Число элементов в отрезке (a или b) называется длиной серии.

После разбиения последовательности {xi} на серии первого и второго рода будем иметь, например, серию вида

…..aabbbbaaabbbaabbab….

Так как случайные числа a и b в данной последовательности независимы и принадлежат последовательности {xi}, равномерно распределённой на интервале (0, 1), то теоретическая вероятность появления серии длиной j в N опытах (под опытом здесь понимается генерация числа xi и проверка условия xi < p) определится формулой Бернулли:

$$P(j,l) = C_l^j p(1-p)^{l-j}, \quad j = \overline{0,l}, \quad l = \overline{1,n}.$$

В случае экспериментальной проверки оцениваются частоты появления серий длиной j. В результате получаются экспериментальная и теоретическая зависимости P (j, l), сходимость которых проверяется по известным критериям, причем проверку целесообразно проводить при разных значениях l и p, 0 < p < 1.

8. Практические задания

8.1. Случайные числа в заданном диапазоне

Выдайте на экран 10 случайных равномерно распределенных чисел в диапазоне:

1. От 3 до 12, целые.
2. Из множества {–3, 0, 6, 9, 12, 15}.
3. От 3 до 12, вещественные.
4. От –2,3 до 10,7 с шагом 0,1.
5. Из множества {–30; 10; 63; 59; 120; 175}.
6. Из множества {1; 0,1; 0,01; …; 10^{-15}}.

8.2. Двумерные случайные величины

Написать функцию генерации случайной точки в двумерном круге с параметрами r, x_0, y_0.

8.3. Генерация одномерной случайной величины

Постройте случайную последовательность плотностью распределения которой принимает значение 1/4 на отрезке [0; 2] и 1/2 на отрезке [4; 5].

8.4. Оценить вероятность

В урне 5 белых, 10 черных и 15 красных шаров. Вынимают три шара. Оцените программным способом вероятность того, что все шары разного цвета.

8.5. Медианы треугольника

Известно, что две медианы в треугольнике пересекаются в точке, которая делит их в отношении 2:1. Используя ГСЧ и векторную алгебру, докажите этот факт.

9. Лабораторные задания

9.1. ГСЧ фон Неймана

Реализуйте программно метод средин квадратов для двоичных 8-разрядных чисел. Покажите, что ГСЧ зацикливается после прихода в ноль.

Замечания:

1. Квадрат числа будет занимать 16 бит, что может вызвать переполнение знакового типа *int*. Рекомендуется использовать типы *unsigned int* или *long* для промежуточных вычислений.

2. Для выделения средней части следует использовать операции сдвига и преобразования типа (либо побитового «И»).

9.2. Случайная матрица

Заполните динамическую матрицу 40×50 целыми случайными числами от –3 до 2. Найдите среднее арифметическое всех элементов этой матрицы. Зная точное значение данной величины $\frac{2-(-3)}{2}$ ( ), вычислите ее относительную погрешность (в процентах) по формуле:

100% * (ТочноеЗначение – ПриблЗначение) / ДлинаДиапазона

Замечания:

1. Количество целых чисел в диапазоне от –3 до 2 равно 2 – (–3) + 1 = 6.

2. Чтобы напечатать символ %, используйте в функции *printf* спецификатор «%%».

9.3. Площадь фигуры

С помощью встроенного ГСЧ вычислите площадь фигуры, ограниченной линиями:

$2 \le x \le 5$,
$4 \le y \le 25$,
$y \le x^2$.

Вычислите относительную погрешность (в процентах) в двух случаях, когда количество случайных точек равно 1000 и 10000.

Замечания: точное значение площади в данном примере равно 125/3 – 8/3 – 12.

9.4. Случайная величина с заданными свойствами

Напишите функцию, генерирующую случайные числа с заданным распределением методом обратной функции распределения.

Распределения, для которых требуется генерировать случайные числа:

1. Равномерное на отрезках $[a, b] \cup [c, d]$.
2. Треугольное с параметрами $[a, b]$.

10. Дополнительные задания

10.1. Многомерные случайные величины

Напишите функцию генерации случайной точки в n-мерном шаре с центром в начале координат и радиусом r.

Контрольные вопросы

1. Что такое ГСЧ?
2. Какие алгоритмы генерации случайных чисел существуют?
3. Как проверить случайность созданной последовательности?

13. Лабораторная работа №10. Разработка приложений для моделирования процессов и явлений. Отладка приложения

Целью работы является изучение порядка работы по моделированию процессов и явлений. Результатом практической работы является отчет, в котором должны быть приведено описание разработанных алгоритмов и результаты отладки ПО.

Для выполнения лабораторной работы № 10 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

МОДЕЛИРОВАНИЕ ДВИЖЕНИЯ КОСМИЧЕСКИХ ТЕЛ

Задача о движении тела в центральном поле тяготения является хорошим примером, демонстрирующим возможности использования ПК для изучения поведения объекта, подчиняющегося некоторым общим физическим законам. Процесс выведения спутника на орбиту обычно разбивается на два этапа. На первом этапе спутник поднимается над атмосферой практически вертикально на некоторую высоту. Затем обычно последняя ступень ракетоносителя придает спутнику необходимую горизонтальную скорость, и далее он движется по инерции. Рассмотрим модель инерционного движения космического тела (спутника) под действием силы всемирного тяготения в гравитационном поле, создаваемом телом с многократно большей массой (Землей). Будем интересоваться тем, какие траектории спутника возможны, какой должна быть его минимальная скорость вблизи поверхности Земли, чтобы он, двигаясь по круговой траектории, не упал на Землю (первая космическая скорость), какой должна быть минимальная начальная скорость спутника, чтобы получилась незамкнутая траектория и спутник «ушел» от Земли (вторая космическая скорость). В численном эксперименте можно также проверить законы Кеплера. В основу модели положено уравнение Ньютона, имеющее вид:

$$m\vec{a} = -G\frac{Mm}{r^3}\vec{r} \qquad (1)$$

Здесь m и M – масса спутника и масса притягивающего центра, G – гравитационная постоянная, r – радиус-вектор, задающий по-

ложение спутника относительно притягивающего центра, *a* – ускорение спутника.

В проекциях на оси x, y (1) примет вид:

$$\frac{d^2x}{dt^2} = -GM\frac{x}{\sqrt{\left(x^2+y^2\right)^3}}, \quad \frac{d^2y}{dt^2} = -GM\frac{y}{\sqrt{\left(x^2+y^2\right)^3}}, \tag{2}$$

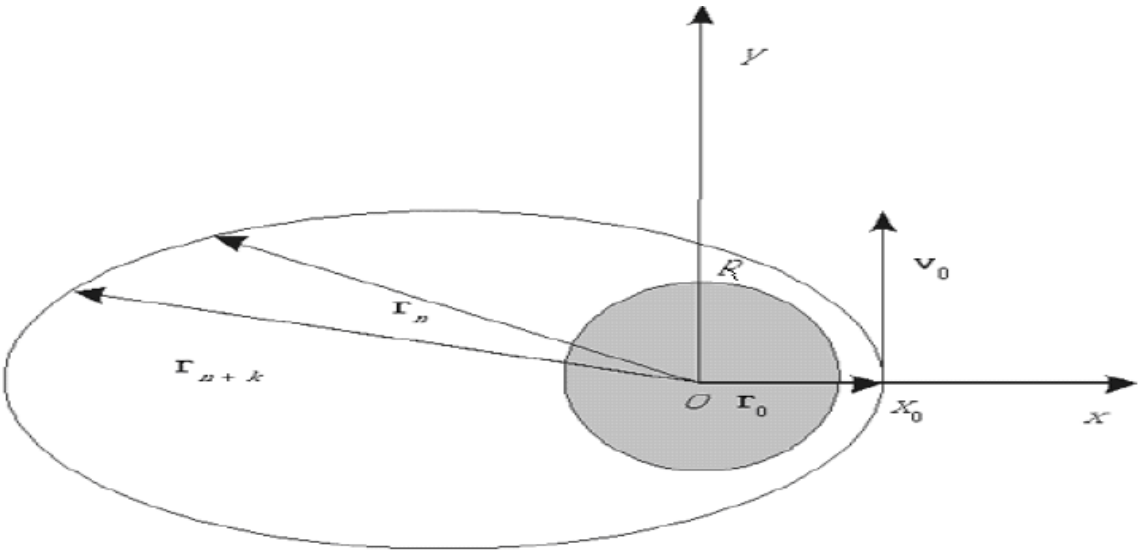


Рис. 1. Система координат и возможная траектория спутника

Сведем (2) к системе четырех дифференциальных уравнений первого порядка:

$$\begin{cases} \dfrac{dx}{dt} = v_x \\ \dfrac{dv_x}{dt} = -GM\dfrac{x}{\sqrt{\left(x^2+y^2\right)^3}} \\ \dfrac{dy}{dt} = v_y \\ \dfrac{dv_y}{dt} = -GM\dfrac{y}{\sqrt{\left(x^2+y^2\right)^3}} \end{cases} \tag{3}$$

Движение тела под влиянием центральной силы происходит в одной плоскости, положение которой определяется векторами и ,

задающими начальное положение тела и его начальную скорость. Декартову систему координат с началом в центре тяготения и начало отсчета времени выберем так, чтобы движение происходило в плоскости OXY и в начальный момент скорость тела была перпендикулярна оси *X*.

Тогда начальные условия можно записать в виде:

t = 0: x = x0, y = 0, vx = 0, vy = v0 (4)

Система уравнений (3) вместе с условиями (4) полностью определяют траекторию спутника и все ее свойства.

Осуществим обезразмеривание задачи. Численный анализ задачи удобно проводить, используя в качестве единиц измерения характерные масштабы задачи. В качестве единицы длины удобно взять *x0*. Если разговор идет о спутнике Земли, то эта величина имеет порядок радиуса Земли *R* и равняется *R* + *h*, где *h* – высота спутника над поверхностью Земли. Всякое расстояние теперь будет задаваться числом, которое показывает, сколько раз в нем укладывается *x0*. Безразмерное *X* будет равняться *x*, измеренному в метрах, деленному на *x0*, также измеренному в метрах. Единицу времени удобно построить, используя гравитационную постоянную и характеристики притягивающего центра. Из уравнений (3) легко видеть, что множитель *GM/r2* имеет размерность ускорения (*м/с2*). Вместо расстояния *r* возьмем *x0* и сформируем выражение с размерностью времени (*с*): *(GM/x03)-1/2*. Его и выберем в качестве единицы времени. В качестве единицы скорости тогда естественно взять *x0/(GM/x03)-1/2*, т.е. *(GM/x0)1/2*

$$\begin{cases} \dfrac{dX}{d\tau} = V_x \\ \dfrac{dV_x}{d\tau} = -\dfrac{X}{\sqrt{\left(X^2+Y^2\right)^3}} \\ \dfrac{dY}{d\tau} = V_y \\ \dfrac{dV_y}{d\tau} = -\dfrac{Y}{\sqrt{\left(X^2+Y^2\right)^3}} \end{cases} \quad (5)$$

Для нахождения в различные моменты времени проекций скорости спутника и его координат на временной оси выберем дискретные точки *tn,* отстоящие друг от друга на малые интервалы Δ*t.* Тогда проекции скорости в момент времени *t n+1* будут приближенно (считаем, что ускорение на этом интервале времени не изменилось) представляться выражениями:

$$V_x^{(n+1)} = V_x^{(n)} + \Delta t \cdot a_x^{(n)}$$

$$V_y^{(n+1)} = V_y^{(n)} + \Delta t \cdot a_y^{(n)} \qquad (6)$$

Координаты в этот момент будем вычислять, как при равномерном движении (опять считая, что интервал времени *Δt* мал, и скорость в течение него такая, как в конце интервала):

$$x^{(n+1)} = x^{(n)} + \Delta t \cdot V_x^{(n+1)}$$

$$y^{(n+1)} = y^{(n)} + \Delta t \cdot V_y^{(n+1)} \qquad (7)$$

В начальный момент времени проекции скорости и координаты спутника известны:

$$x^{(0)} = 1, \quad y^{(0)} = 0, \quad V_x^{(0)} = 0, \quad V_y^{(0)} = V_0 \qquad (8)$$

Система (3), (5)–(8) позволяет шаг за шагом, при малом Δ*t,* достаточно точно вычислить траекторию спутника и все ее характеристики.

Контрольные вопросы

1. Как моделируется движение спутника?
2. Как решается дифференциальное уравнение движения спутника?

14. Лабораторная работа №11. Интеграция модуля в информационную систему

Целью работы является изучение порядка работы по работе с модулями в приложении. Результатом практической работы является отчет, в котором должны быть приведено описание разработанных алгоритмов и результаты отладки ПО.

Для выполнения лабораторной работы № 11 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Этапы проектирования модульных приложений

Для разработки модульного приложения, прежде всего, необходимо выделить тот функционал, который должен расширяться с помощью модулей.

Далее разрабатываются интерфейсы, с помощью которых система будет обращаться к сторонним реализациям за этим функционалом.

Самым тонким моментом становится вопрос о том, как динамически добавлять реализации интерфейсов.

Reflection

В .Net Framework присутствует мощная технология reflection. Reflection позволяет программе отслеживать и модифицировать собственную структуру и поведение во время выполнения.

Применительно к нашей задаче, reflection позволяет загружать библиотеки в память и проверять все реализованные в ней классы на предмет реализации необходимых интерфейсов.

То есть, можно добавлять библиотеки (с реализацией выделенных интерфейсов), представляющие собой модули, в специальную директорию и с помощью технологии reflection находить и инстанцировать необходимые классы.Это наиболее популярное решение, которое часто встречается на просторах интернета. Но у подобного подхода есть существенные недостатки, связанные с высокой ресурсоемкостью.

Загрузка сборки в память и перебор всех доступных в ней классов, в поисках реализаций интерфейсов, требует большого количества оперативной памяти, и ощутимого времени на выполнение перебора. Все усложняется в том случае, если реализация метода выходит за рамки одной библиотеки и имеет зависимости от сторонних библиотек. Тогда под перебор попадают сборки, вообще не

содержащие реализаций необходимых интерфейсов и процессорное время, затрачиваемое на их исследование, тратится впустую.

Структура модуля

Очевидно, чтобы исключить лишние библиотеки из перебора, необходима дополнительная информация о модуле. Подобным источником информации может выступить текстовый файл, сопровождающий библиотеки модуля и предоставляющий информацию о них. Таким образом, перед нами встает задача разработки требований к структуре модуля, одним из пунктов которого можно предложить требование наличия файла, указывающего на главную библиотеку с реализацией интерфейсов и содержащего перечень всех необходимых зависимостей.

Простейшим решением устройства модуля может быть следующее:

1. Модуль представляет собой архив всех необходимых библиотек. В качестве алгоритма сжатия может выступать zip. Причем, как такового сжатия не требуется (бинарные библиотеки плохо поддаются архивированию), необходимо просто объединить все составляющие модуля в один файл.

2. Кроме библиотек, модуль должен содержать текстовый файл (называемый дескриптором), содержащий информацию о главной библиотеке, зависимостях, и, для повышения быстродействия путем исключения перебора, название реализуемого интерфейса вместе с полным именем класса его реализующего.

Для реализации дескриптора плагина представляется удобным воспользоваться XML.

Простейшая структура такого документа может иметь следующий вид:

```
<?xml version=«1.0»?>
<plugin>
    <type>Имя интерфейса</type>
    <name>Имя модуля</name>
    <description>Описание модуля.</description>
    <version>Номер версии модуля</version>
    <class>
     Полное имя класса, реализующего указанный интерфейс
    </class>
    <assembly>Главная библиотека</assembly>
```

```
    <dependences>
        <dependence>Дополнительная библиотека</dependence>
    </dependences>
</plugin>
```

Добавление и удаление модулей

Добавление нового модуля в систему может происходить в следующей последовательности:

1. Системе передается полный путь файла с добавляемым модулем.

2. Добавляемый модуль проверяется на соответствие своему дескриптору: проверяется наличие всех указанных библиотек, наличие главного класса и реализация им указанного интерфейса.

3. В директории системы, отведенной под хранение модулей, создается новая поддиректория для добавляемого модуля. Все библиотеки модуля копируются в эту директорию.

4. Вычисляется уникальный идентификатор модуля (как вариант, можно взять хеш от имени и версии модуля) на случай, если возникнет необходимость сохранения информации об использовании модуля в прошлой сессии работы в системе.

5. Вся информация из дескриптора модуля и вычисленный идентификатор записываются в системный реестр модулей (xml-файл, хранящий информацию об установленных в системе модулях).

В простейшем случае реестр модулей может представлять собой xml-файл со схожей с приведенной выше структурой, с той лишь разницей, что записей о модулях в ней будет много:

```
<?xml version=«1.0»?>
<plugins>
<plugin>
<id>534523</id>
    <type>Имя интерфейса</type>
    <name>Имя модуля 1</name>
    <description>Описание модуля 1</description>
    <version>Номер версии модуля 1</version>
    <class>
      Полное имя класса, реализующего указанный интерфейс
    </class>
    <assembly>Главная библиотека</assembly>
    <dependences>
```

```
            <dependence>Дополнительная библиотека</dependence>
        </dependences>
    </plugin>
    <plugin>>
    <id>79568</id>
        <type>Имя интерфейса</type>
        <name>Имя модуля 2</name>
        <description>Описание модуля 2</description>
        <version>Номер версии модуля 2</version>
        <class>
          Полное имя класса, реализующего указанный интерфейс
        </class>
        <assembly>Главная библиотека</assembly>
        <dependences>
            <dependence>Дополнительная библиотека</dependence>
        </dependences>
    </plugin>
    . . . .
    </plugins>
```

Последовательность действий для удаления модуля:

1. Удаление директории модуля.
2. Удаление информации о модуле из реестра.

Структура классов

PluginDescriptor – предоставляет информацию о модуле, достаточную для его инстанцирования.

PluginRegister – выполняет операции чтения и записи в реестр модулей. Возвращает дескриптор модуля, соответствующего указанному идентификатору. Может вернуть полный список дескрипторов всех доступных модулей.

Plugin – предоставляет информацию о модуле, основываясь на соответствующей информации в дескрипторе. Содержит ссылку на объект, представляющий собой реализацию модуля. Возможно, существование этого класса может показаться избыточным. Но его ценность проявляется в том случае, если решение о целесообразности использования модуля пользователь принимает на основании его (модуля) описании. В таком случае, мы можем предоставить всю необходимую пользователю информацию не загружая библиотек модуля.

PluginManager – выполняет добавление и удаление модулей в системе. Осуществляет контроль целостности модуля при его добавлении. Реализуется в соответствии с шаблоном «одиночка», для унификации способов получения модулей.

PluginLoader – выполняет инстанцирование модуля. Необходимость введения этого класса вызвана спецификой самого процесса инстанцирования, речь о которой пойдет ниже.

Нюансы динамического подключения модулей, специфичные для .Net Framework

У динамической загрузки сборок в .Net есть особенность – сборки загружаются в так называемые AppDomain – домены приложения, являющиеся изолированной средой, в которой выполняются приложения. Нюанс в том, что выгрузка сборки отдельно невозможна. Ее возможно произвести только выгрузив весь домен целиком. Таким образом, после инициализации модуля все используемые им библиотеки блокируются для удаления из файловой системы, и их выгрузка из памяти становится невозможной.

Для разрешения проблемы с блокированием файлов библиотек существует решение, называемое теневое копирование. Его суть заключается в копировании файла библиотеки и загрузки в память уже ее копии, оригинал при этом остается доступен для удаления.

Решение проблемы, связанной с выгрузкой ранее загруженного модуля из памяти, заключается в создании для модуля отдельного домена. Помимо решения проблемы с выгрузкой модуля из памяти, это решение открывает возможность ограничения прав исполняемого кода. Например, модулю можно запретить чтение и запись в любой директории отличной от той, в которой он располагается. Но у подобного решения есть и обратная сторона, требующая сериализации классов, реализующих интерфейсы модулей.

Для наглядности приведу код класса PluginLoader и часть кода класса Plugin (в приведенном коде не учитывается наличие зависимостей загружаемого модуля):

```
public class BasePlugin
{
    ...

    /// <summary>
    /// Возвращает реализацию инкапсулируемого расширения.
```

```
        /// </summary>
        /// <remarks>
        /// Инстанцирование расширения происходит при первом обращении.
        /// </remarks>
        public Object Instance {
            get {
                if (instance == null) {
                    instance = CreateInstance();
                }
                return instance;
            }
        }

        /// <summary>
        /// Создает домен для загружаемой реализации модуля
        /// и загружает ее в него.
        /// </summary>
        /// <returns>
        /// В случае успеха загруженное расширение, иначе null.
        /// </returns>
        private Object CreateInstance()
        {
            Descriptor d = this.Descriptor;

            /* Настраиваем домен */
            AppDomainSetup setup = new AppDomainSetup();
            setup.ShadowCopyFiles = «true»; // включаем теневое копирование

            //TODO: Задать настройки безопасности для плагина

            /* Создаем домен для плагина */
            AppDomain domain = AppDomain.CreateDomain(d.ToString(), null, setup);

            /* Создаем загрузчик плагина */
            PluginLoader loader =
```

```
                (PluginLoad-
er)domain.CreateInstanceFromAndUnwrap(
                    typeof(PluginLoader).Assembly.Location,
                    typeof(PluginLoader).FullName);

            /* Создаем экземпляр плагина */
            Object obj = null;
            try {
                obj = loader.CreateInstance(d, PluginManager
                                .GetPluginDirectory(this));
            } catch (Exception e) {
                AppDomain.Unload(domain);
                throw new PluginLoadException(d, "", e);
            }
            return obj;
        }
    }

    [Serializable]
    public class PluginLoader
    {
        /// <summary>
        /// Загружает сборку модуля и возвращает реализацию мо-
дуля.
        /// </summary>
        /// <param name="d">Дескриптор загружаемого моду-
ля.</param>
        /// <param name="pluginDirectory">
        /// Полный путь к директории загружаемого модуля.
        /// </param>
        /// <returns>
        /// Реализацию модуля, соответствующего переданному
дескриптору.
        /// </returns>
        public Object CreateInstance(Descriptor d, String pluginDi-
rectory)
        {
```

```
            String assemblyFile = Path.Combine(pluginDirectory,
d.AssemblyName);

            /* Пытаемся загрузить сборку. */
            Assembly assembly = null;
            try {
                AssemblyName asname =
                   AssemblyName.GetAssemblyName(assemblyFile);
                assembly = AppDomain.CurrentDomain.Load(asname);
            } catch (Exception e) {
                throw new AssemblyLoadException(assemblyFile,
                            «Ошибка загрузки сборки.», e);
            }

            /* Пробуем получить объект класса, реализующего мо-
дуль. */
            Object obj = null;
            try {
                obj = assembly.CreateInstance(d.ClassName);
            } catch (Exception e) {
                throw new ClassLoadException(d.ClassName, assem-
bly.FullName,
                            «Ошибка при получении экземпляра клас-
са.», e);
            }
            return obj;
        }
    }
```

Контрольные вопросы

1. Как создать модульное приложение для Net Framework?
2. Как интегрировать готовый модуль в программу?

15. Лабораторная работа №12. Программирование обмена сообщениями между модулями

Целью работы является изучение порядка работы по работе с модулями в приложении. Результатом практической работы является отчет, в котором должны быть приведено описание разработанных алгоритмов и результаты отладки ПО.

Для выполнения лабораторной работы № 12 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Основы WCF

WCF – это среда выполнения и набор интерфейсов API для создания систем, передающие сообщения между службами и клиентами. Те же инфраструктура и интерфейсы API используются для создания приложений, обменивающихся данными с другими приложениями на данном компьютере или на компьютере, который находится в другой компании, и доступ к которому можно получить через Интернет.

Обмен сообщениями и конечные точки

WCF основывается на том обмена сообщениями, и что-то, что моделируется в виде сообщений (например, HTTP-запроса или сообщения Message Queuing (MSMQ)) можно представить единым образом в модели программирования. Это обеспечивает универсальный интерфейс API для разных транспортных механизмов.

В модели различаются *клиентов*, являющиеся приложениями, которые инициируют связь, и *служб*, которые являются приложениями, которые ожидают клиентам взаимодействовать с ними и реагировать на них, обмен данными. Одно приложение может быть как клиентом, так и службой. Примеры, см. в разделе дуплексные службы и сети Peer-to-Peer.

Между конечными точками выполняется обмен сообщениями. *Конечные точки* места, где отправленных или полученных сообщений (или оба), и они определяют все сведения, необходимые для обмена сообщениями. Служба предоставляет одну или несколько конечных точек приложения (а также ноль или более конечных точек инфраструктуры), а клиент создает конечную точку, совместимую с одной из конечных точек службы.

Конечной точки описывает образом основан на стандартах, куда должны отправляться сообщения и как они должны отправ-

ляться сообщения должны выглядеть. Служба может предоставлять эту информацию в виде метаданных, которые клиенты могут обрабатывать для создания соответствующих клиентов WCF и обмен данными *стеки*.

Протоколы связи

Одним из обязательных элементов стека связи является *транспортный протокол*. Сообщения можно отправлять через интрасети или через Интернет с помощью общих транспортов, таких как HTTP и TCP. Предусмотрены другие транспорты, поддерживающие связь с приложениями очереди сообщений и узлами в сетке одноранговой сети. Можно добавить дополнительные транспортные механизмы, с помощью встроенных точек расширения WCF.

Другим обязательным элементом стека связи является кодирование, определяющее способ форматирования любого заданного сообщения. WCF предоставляет следующие кодировки:

- кодировка текста – кодирование с возможностью взаимодействия;
- кодировка подсистемы оптимизации передачи сообщений MTOM – поддерживающий взаимодействие способ эффективной отправки неструктурированных двоичных данных в службу и из нее;
- двоичное кодирование для эффективной передачи.

Дополнительные механизмы кодирования (например, кодирование сжатием) можно добавить с помощью встроенных точек расширения WCF.

Шаблоны сообщений

WCF поддерживает несколько шаблонов обмена сообщениями, включая запрос ответ, одностороннюю и дуплексную связь.Разные транспорты поддерживают разные шаблоны обмена сообщениями и таким образом влияют на типы поддерживаемых взаимодействий. API-интерфейсы WCF и среда выполнения также помогают отправлять сообщения безопасно и надежно.

Термины WCF

Ниже перечислены другие понятия и термины, используемые в документации по WCF.

Сообщение

Автономная единица данных, которая может состоять из нескольких частей, включая текст и заголовки.

Служба

Конструкция, предоставляющая доступ к одной или нескольким конечным точкам, каждая из которых предоставляет доступ к одной или нескольким операциям службы.

Конечная точка

Конструкция, в которой производится отправка или прием сообщений. Он включает в себя расположения (адрес), который определяет, где можно отправлять сообщения, спецификации механизма связи (привязка), который описывает, каким образом должны отправляться сообщения, и определение для набора сообщений, отправленные или полученные (или оба) расположение (контракт службы), описывающий, какие сообщения могут отправляться.

Служба WCF видима внешнему миру как коллекция конечных точек.

Конечная точка приложения

Конечная точка, предоставляемая приложением и соответствующая контракту службы, реализуемому приложением.

Конечная точка инфраструктуры

Конечная точка, предоставляемая инфраструктурой для расширения функциональных возможностей, необходимых для службы или предоставляемых службой и не имеющих отношения к контракту службы. Например, со службой может быть связана конечная точка инфраструктуры, предоставляющая информацию о метаданных.

Адрес

Задает расположение, где принимаются сообщения. Он задается в виде универсального кода ресурса (URI). Часть URI, определяющая схему, задает транспортный механизм для доставки по адресу, например HTTP и TCP. Иерархическая часть URI содержит уникальное расположение, формат которого зависит от транспортного механизма.

Адрес конечной точки позволяет создавать уникальные адреса для каждой конечной точки в службе или при определенных условиях использовать один адрес для нескольких конечных точек. В следующем примере показан адрес, использующий протокол HTTPS с портом, не установленным по умолчанию:

Копировать

HTTPS://cohowinery:8005/ServiceModelSamples/CalculatorService

Привязка

Задает способ связи конечной точки с внешним миром. Она состоит из набора компонентов, называемых элементами привязки, которые компонуются в «стек», один над другим, образуя инфраструктуру связи. Как минимум, привязка определяет используемые транспорт (например, HTTP или TCP) и кодирование (например, текстовое или двоичное). Привязка может содержать элементы привязки, задающие такие сведения, как механизмы безопасности, используемые для защищенных сообщений, или шаблон сообщений, используемый конечной точкой.

Элемент привязки

Представляет определенную часть привязки, такую как транспорт, кодирование, реализация протокола на уровне инфраструктуры (например, WS-ReliableMessaging) или любой другой компонент стека связи.

Поведения

Компонент, управляющий различными аспектами работы службы, конечной точки, определенной операции или клиента во время выполнения. Расширения функциональности группируются в соответствии с областью действия: общие расширения функциональности влияют глобально на все конечные точки, расширения функциональности служб влияют только на аспекты, относящиеся к службам, расширения функциональности конечных точек влияют только на свойства, относящиеся к конечным точкам, а расширения функциональности уровня операции влияют только на конкретные операции. Например, одно расширение функциональности службы регулирующее и определяет реакцию службы при избытке сообщений, превосходящем возможности обработки. С другой стороны, поведение конечной точки управляет только аспектами, относящимися к конечным точкам, например способом поиска учетных данных безопасности и расположением для поиска.

Привязки, предоставляемые системой

WCF содержит ряд привязок, предоставляемых системой. Они являются коллекциями элементов привязки, оптимизированными для конкретных сценариев. Например, класс WSHttpBinding предназначен для взаимодействия со службами, реализующими различные спецификации WS-*. Эти заранее определенные привязки экономят время, предоставляя только те параметры, которые могут быть пра-

вильно применены для конкретного сценария. Если заранее определенная привязка не удовлетворяет необходимым требованиям, можно создать собственную настраиваемую привязку.

Конфигурация и кодирование

Управление приложением возможно с помощью кода, с помощью конфигурации или с помощью и того, и другого. Преимущество конфигурации заключается в том, что после написания кода параметры клиента или службы могут задаваться пользователями (например, системным администратором), а не только разработчиком, при этом отсутствует необходимость в повторной компиляции. Конфигурация не только позволяет задавать значения, такие как адреса конечных точек, но и обеспечивает дополнительные возможности управления, позволяя добавлять конечные точки, привязки и расширения функциональности. Кодирование позволяет разработчику сохранить полный контроль над всеми компонентами службы или клиента; все параметры, заданные при конфигурации, можно проверить и, при необходимости, изменить с помощью кода.

Операция службы

Процедура, определенная в программном коде службы и реализующая функциональные возможности операции. Эта операция видима клиентам в виде методов клиента WCF. Метод может возвращать значение и может иметь ряд необязательных аргументов либо может не иметь аргументов и не возвращать никаких значений. Например, операция, выполняющая функцию простого приветствия «Привет», может использоваться для уведомления о наличии клиента и запуска серии операций.

Контракт службы

Объединяет несколько связанных операций в один функциональный модуль. Контракт может определять параметры уровня службы, такие как пространство имен службы, соответствующий контракт обратного вызова и другие подобные параметры. В большинстве случаев контракт задается путем создания интерфейса на выбранном языке программирования и применения атрибута ServiceContractAttribute к этому интерфейсу. Фактический код службы создается при реализации этого интерфейса.

Контракт операции

Контракт операции определяет параметры операции и тип возвращаемых ею значений. При создании интерфейса, определяющего

контракт службы, контракт операции задается применением атрибута OperationContractAttribute к определению каждого метода, входящего в контракт. Операции могут задаваться как получающие одно сообщение и возвращающие одно сообщение или как получающие набор типов и возвращающие тип. В последнем случае формат сообщений, обмен которыми происходит при выполнении данной операции, определяется системой.

Контракт сообщения

Описывает формат сообщения. Например, в нем описывается, должны элементы сообщения размещаться в заголовках или в теле, уровень безопасности, применяемый к определенным элементам сообщения и т. д.

Контракт сбоя

Может связываться с операцией службы для обозначения ошибок, которые могут возвращаться вызывающему объекту. С операцией могут быть связаны ноль или более сбоев. Эти ошибки представляют собой сбои протокола SOAP, которые моделируются в модели программирования как исключения.

Контракт данных

Хранящееся в метаданных описание типов данных, используемых службой. Это описание позволяет другим объектам работать со службой. Типы данных могут использоваться в любой части сообщения, например в виде типов параметров или возвращаемых значений. Если в службе используются только простые типы, явное использование контрактов данных не требуется.

Размещение

Служба должна быть размещена в некотором процессе. Объект *узла* – это приложение, контролирующее время существования службы. Службы могут быть резидентными (размещенными сами в себе) или управляемыми существующим ведущим процессом.

Резидентная служба

Служба, которая выполняется в приложении-процессе, созданном разработчиком. Разработчик контролирует время существования службы, набор свойств службы, открывает службу (при этом служба переходит в режим ожидания данных) и закрывает службу.

Ведущий процесс

Приложение, предназначенное для размещения служб. В число таких приложений входят службы IIS, службы активации Windows

(WAS) и службы Windows. В этих сценариях основное приложение контролирует время существования службы.Например, с помощью IIS можно создать виртуальный каталог, содержащий сборку службы и файл конфигурации. При получении сообщения IIS запускает службу и контролирует время ее существования.

Создание экземпляров

Со службой связана модель создания экземпляров. Существуют три модели создания экземпляров: «один экземпляр», в которой один объект CLR обслуживает всех клиентов, «по вызовам», в которой для обработки каждого вызова, поступившего от клиента, создается новый объект CLR, и «по сеансам», в которой создается набор объектов CLR, по одному на каждый отдельный сеанс. Выбор модели создания экземпляров зависит от требований к приложению и ожидаемого режима использования службы.

Клиентское приложение

Программа, обменивающаяся сообщениями с одной или несколькими конечными точками. Клиентское приложение начинает работу с создания экземпляра клиента WCF и вызывает методы этого клиента WCF. Обратите внимание, что одно и то же приложение может быть как клиентом, так и службой.

Канал

Конкретная реализация элемента привязки. Привязка представляет собой конфигурацию, а канал является реализацией, связанной с этой конфигурацией. Следовательно, с каждым элементом привязки связан канал. Каналы, собранные в стек друг на другом, образуют конкретную реализацию привязки: стек каналов.

Клиент WCF

Конструкция «клиент-приложение», предоставляющая доступ к операциям службы в виде методов (на выбранном языке программирования .NET Framework, например Visual Basic или Visual C#). Каждое приложение может содержать клиента WCF, включая приложение, содержащее службу. Следовательно, можно создать службу, содержащую клиентов WCF других служб.

Клиент WCF может создаваться автоматически с помощью ServiceModel Metadata Utility Tool (Svcutil.exe) и нацеливания его на работающую службу, публикующую метаданные.

Метаданные

Описывают характеристики службы, которые необходимо знать внешней сущности для обмена данными со службой.Метаданные могут считываться служебным средством ServiceModel Metadata Utility Tool (Svcutil.exe) для создания клиента WCF и сопутствующей конфигурации, которые клиентское приложение можно использовать для взаимодействия со службой.

Метаданные, предоставляемые службой, содержат документы схемы XML, определяющие контракт данных службы, и документы WSDL, описывающие методы службы.

Если метаданные для службы включены, они автоматически создаются WCF путем проверки службы и ее конечных точек. Для публикации метаданных службы необходимо явно задать расширение функциональности метаданных.

Безопасность

В WCF включает в себя конфиденциальность (шифрование сообщений во избежание перехвата), целостность (средства обнаружения подделки сообщения), проверку подлинности (средства проверки серверов и клиентов) и авторизацию (Управление доступом к ресурсы). Эти функции предоставляются либо путем использования существующих механизмов обеспечения безопасности, таких как TLS по HTTP (также называемого HTTPS), либо путем реализации одной или нескольких различных спецификаций безопасности WS-*.

Режим безопасности транспорта

Указывает, что конфиденциальность, целостность и проверка подлинности обеспечиваются механизмами транспортного уровня (такими как HTTPS). При использовании такого транспорта, как HTTPS, преимущества этого режима заключаются в его эффективной производительности и хорошей известности в связи с преобладающим применением в Интернете. Недостаток заключается в том, что безопасность этого типа применяется отдельно на каждом прыжке пути передачи, в результате чего связь становится чувствительной к атакам типа «злоумышленник в середине».

Режим безопасности сообщения

Указывает, что безопасность обеспечивается путем реализации одной или нескольких спецификаций безопасности, таких как спецификация безопасности веб-служб: Безопасность сообщений SOAP. Каждое сообщение содержит необходимые механизмы,

обеспечивающие безопасность во время его передачи и позволяющие получателям обнаруживать подделки и расшифровывать сообщения. В этом отношении безопасность заложена внутри каждого сообщения, обеспечивая сквозную безопасность по всем участкам передачи. Поскольку информация безопасности становится частью сообщения, можно также включить несколько типов учетных данных с сообщением (они называются *утверждений*). Дополнительным преимуществом такого подхода является возможность безопасной передачи сообщения с помощью любого транспортного механизма, включая несколько транспортных механизмов между источником и назначением. К недостатком этого подхода относится сложность используемых криптографических механизмов, требовательных к производительности.

Режим безопасности транспорта с учетными данными сообщения

Задает использование транспортного уровня для обеспечения конфиденциальности, проверки подлинности и целостности сообщений, но каждое сообщение может содержать несколько учетных данных (утверждений), требуемых получателями сообщения.

WS-*

Сокращение для обозначения растущего набора спецификаций веб-служб (WS), таких как WS-Security, WS-ReliableMessaging и т. д., реализованных в WCF.

Контрольные вопросы

1. Что такое WCF?
2. Как происходит обмен данными через WCF?
3. Какие существуют протоколы связи WCF?

16. Лабораторная работа №13. Организация файлового ввода-вывода данных

Целью работы является изучение порядка работы с файлами. Результатом практической работы является отчет, в котором должны быть приведено описание разработанных алгоритмов и результаты отладки ПО.

Для выполнения лабораторной работы № 13 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Введение

В данной работе рассматривается организация ввода-вывода в языке C#, базовые потоки и производные от них, а также встроенные средства работы с файловой системой в Windows.

Основы ввода-вывода. Потоки

Ввод и вывод данных является одной из наиболее часто встречающихся в программах функций. То, как в языке программирования организован этот процесс, напрямую влияет на удобство работы программиста. На данный момент общепринятой моделью ввода-вывода данных является модель потоков. Потоки – последовательности байт, которые представляют собой отображение некоего источника или наоборот, хранилища данных в память программы. При таком отображении происходит абстрагирование от сущности источника/хранилища, что позволяет вести работу единообразно. С помощью потоков можно работать с такими объектами как последовательные порты, файлы, консоль, оперативная память, сеть и т. д. В C# реализована потоковая модель работы с вводом-выводом.

Для того чтобы использовать ввод-вывод, необходимо подключить к проектируемому коду пространство имен System.IO:

```
using System.IO;
```

после чего станут доступны все необходимые базовые классы.

Система потоков в C# устроена иерархически, все возможные потоки, независимо от их предназначения, являются потомками базового класса Stream. У класса Stream есть ряд производных классов, название которых содержит слово Stream – FileStream, MemoryStream и т. д. Все эти классы реализуют те же методы и свойства что и Stream. Отличия лишь в нюансах использования и возможностях чтения и записи.

Так как потоки предназначены для ввода и вывода данных, программист может, как читать данные из потока, так и записывать их в поток. Соответственно, первая операция необходима для чтения данных, а вторая – для записи данных в источник. Например, для того, чтобы сохранить на диск текстовый файл, программа записывает в поток данного файла необходимый текст. В дальнейшем же она может снова использовать этот же поток, для того чтобы прочитать данные. Кроме того, у потоков есть еще одно важное свойство – это позиция в потоке. Поскольку каждый поток, по факту может быть представлен как массив байтов, то он должен обладать определенной длиной, а также иметь возможность перемещаться по элементам с помощью индекса, аналогичному цифровому индексу элемента в массиве. Такой индекс в потоке называется позицией. После каждой операции чтения или записи позиция в потоке смещается на прочитанное/записанное количество байт. Длина же потока изменяется только в случае записи. Логично, что позиция в потоке не может быть больше, чем его длина. Важно помнить, что далеко не все потоки позволяют получать значения позиции и длины. Например, существуют сетевые потоки, длина которых заранее не определена, соответственно и понятие позиции не имеет смысла. В таких потоках возможно только последовательное чтение или запись. Кроме того, есть потоки, предназначенные только для чтения – записывать в них нельзя.

Рассмотрим основные методы и свойства базового класса Stream:

Read – читает байты из потока в массив и возвращает число прочитанных байтов

Write – записывает данные из массива байтов в поток

Position – текущая позиция в потоке. Некоторые потоки позволяют изменять позицию. Как и в массивах, нумерация начинается с 0.

Length – длина потока в байтах.

Еще одной важной операцией для потока является метод *Close()*. Этот метод закрывает поток и высвобождает все ресурсы, занятые им. Данный метод обязательно вызывается после окончания операций ввода вывода. Оставление потоков открытыми – грубая ошибка программиста, которая может приводить как к утечкам памяти, так и к потере данных, т. к. некоторые операции ввода-вывода завершаются только по закрытии потока.

Потоки, предназначенные для работы с файловой системой

Для работы с файловой системой используется потомок класса Stream – FileStream. Класс FileStream поддерживает все описанные выше методы и свойства, может использоваться для чтения и записи данных, а также позволяет определять длину (которая равна размеру файла в байтах) а также изменять текущую позицию в потоке.

Для того чтобы открыть файл используется конструкция вида

FileStream example=newFileStream(path,FileMode)

где path – путь к файлу в обычном формате Windows, а FileMode – перечисление, принимающее следующие значения

FileMode.Append – открывает файл для записи и новые данные дописывается в конец файла

FileMode.Create – создает новый файл, при этом старый будет удален. Для того чтобы избежать затирания файла, если он существует, используется CreateNew – создает новый файл, однако если такой файл уже существует, будет выброшено исключение.

FileMide.Open – открывает файл только для чтения.

Естественно, что если открыть файл только для чтения, записать в такой поток уже ничего будет нельзя. Пример того, как можно прочитать файл в цикле

```
byte[] buffer= newbyte[10000];
int counter=0;
FileStream file=new FileStream(«C:\\file.txt»);
while((counter=file.Read(buffer,0,10000))!=0)
{
//выполнить действия с полученными данными
}
file.Close();
```

Как видно, для работы с файлом необходимо сначала создать поток, указав имя открываемого файла, после чего содержимое файла будет сразу же доступно для чтения или записи. После выполнения работ, следует сразу же закрыть поток. В случае если поток не будет закрыт, никакие операции с файлом извне будут недоступны (Windows будет сообщать о том, что файл «занят другой программой»). Кроме того, все данные, записанные в файл через поток, будут сохранены лишь в момент закрытия потока. Если поток не за-

крыть и завершить приложение, то все записанные данные будут утеряны.

Кодировки текста. Работа с кодировками в C#

С самого создания средств хранения данных встал вопрос о представлении текстов в памяти ЭВМ. Компьютер может хранить лишь числа, поэтому потребовались схемы соответствия, сопоставляющие определенную последовательность бит с символом или буквой. Такие схемы (правильнее назвать их таблицами или страницами) называются кодовой страницей или проще, кодировкой. Первые кодировки использовали в качестве обозначение символа 1 байт. Например, такой кодировкой является всем известная ASCII. В итоге было создано множество различных кодировок, причем часто получалось так, что одна и та же последовательность бит в разных кодировках обозначала разный символ, что приводило к проблемам отображения текстов, сохраненных в кодировках, не получившими большого распространения. Поэтому, для того чтобы правильно отобразить содержимое любого потока, содержащего текст, необходимо обладать сведениями о кодировке, в которой этот текст был сохранен. Аналогично и при записи.

В C# для хранения текстовых строк в памяти используется кодировка UTF-16, где 16 – число бит на один символ. Таким образом, любая строковая переменная в программе на C# – всего лишь массив байт, интерпретируемый как текст в кодировке UTF-16. Таким образом, по умолчанию, весь ввод-вывод текста также будет происходить в указанной кодировке. Однако до сих пор есть ряд других кодировок, часто встречающихся в использовании – Windows-1251, ASCII, CP866 и др. Многие старые программы просто не понимают кодировку Юникод, используемую в C#. Поэтому часто возникает необходимость корректно прочитать/записать строки из другой кодировки в программе на C#.

Для этого в языке реализован следующий механизм. Внутри программы все строки хранятся в Юникод, однако при вводе или выводе строки программист может указать, в какой кодировке должен быть входной/выходной текст. При этом в момент вызова функции чтения/записи будет произведено перекодирование текста из Юникод в указанную кодировку. Для указания кодировок в C# используется класс Encoding.

Encoding – класс со статическими методами и перечислением. В перечислении указываются некоторые заранее заданные кодировки, а с помощью метода GetEncoding() можно получить любую кодовую страницу по номеру.например, кодировка Windows обозначается как Encoding.GetEncoding(1251).

Чтение и запись текстовых файлов

Для работы с текстовыми файлами удобно использовать классы TextReader и TextWriter, которые скрывают в себе уже указанный выше поток FileStream.

Пример объявления:

TextReader text=new StreamReader(«C:\file.txt», Encoding.UTF8);

TextWriter text=new StreamWriter(«C:\file.txt», Encoding.UTF8);

Обратите внимание на конструктор – здесь используется базовый класс, реализующей методы именно для чтения или записи. В качестве аргументов обязательно передаются имя файла, а также кодировка, в которой будет происходить чтение/запись текста. У данного класса есть также ряд других конструкторов.

После открытия файла, можно записывать в него данные с помощью метода Write, принимающего в качестве аргумента строку, либо читать строки или отдельные символы с помощью методов Read или ReadLine. Последний метод читает строку, пока не встретит символ переноса.

Класс Directory

Работа с файловой системой включает в себя не только чтение и запись файлов, но и работу с папками. Для таких задач в C# предусмотрен класс Directory, имеющий ряд статических методов:

Create – создает директорию согласно указанному пути.

Exists – проверяет, существует ли указанная директория.

Delete – удаляет директорию.

GetFiles – возвращает массив строк с именами файлов, находящихся в данной директории.

Класс File

Для перемещения и удаления файлов используются статические методы класса File:

Exists – проверяет, существует ли файл по указанному пути.

Copy – копирует файл в указанную директорию, при этом можно указать новое имя файла.

Delete – удаляет файл.

Move – перемещает файл в указанную директорию, при это можно указать новое имя файла.

Задания для самостоятельной работы

1. Создать приложение, позволяющее сохранить введенный текст в текстовый файл. Пользователь должен иметь возможность выбрать папку для сохранения файла, а также кодировку (Как минимум, UTF-8, Win1251, DOS 866).
2. Создать приложение, отображающий текст из файла. Пользователь должен иметь возможность выбрать файл, а также кодировку(Как минимум, UTF-8, Win1251, DOS 866).
3. Создать приложение, осуществляющее поиск файлов в папке, в названии которых содержится заданное пользователем ключевое слово. Программа должна выводить список найденных файлов на экран. Пользоваться масками поиска нельзя.
4. Создать приложение, копирующее заданный пользователем файл в указанную им папку. При этом пользователь должен иметь переименовать файл. Копировать в несуществующую папку нельзя.
5. Создать приложение, выводящее список самых часто встречающихся расширений файлов в указанной папке. Укажите 5 самых часто встречающихся расширений. Пользователь должен иметь возможность выбрать папку для анализа.

Контрольные вопросы

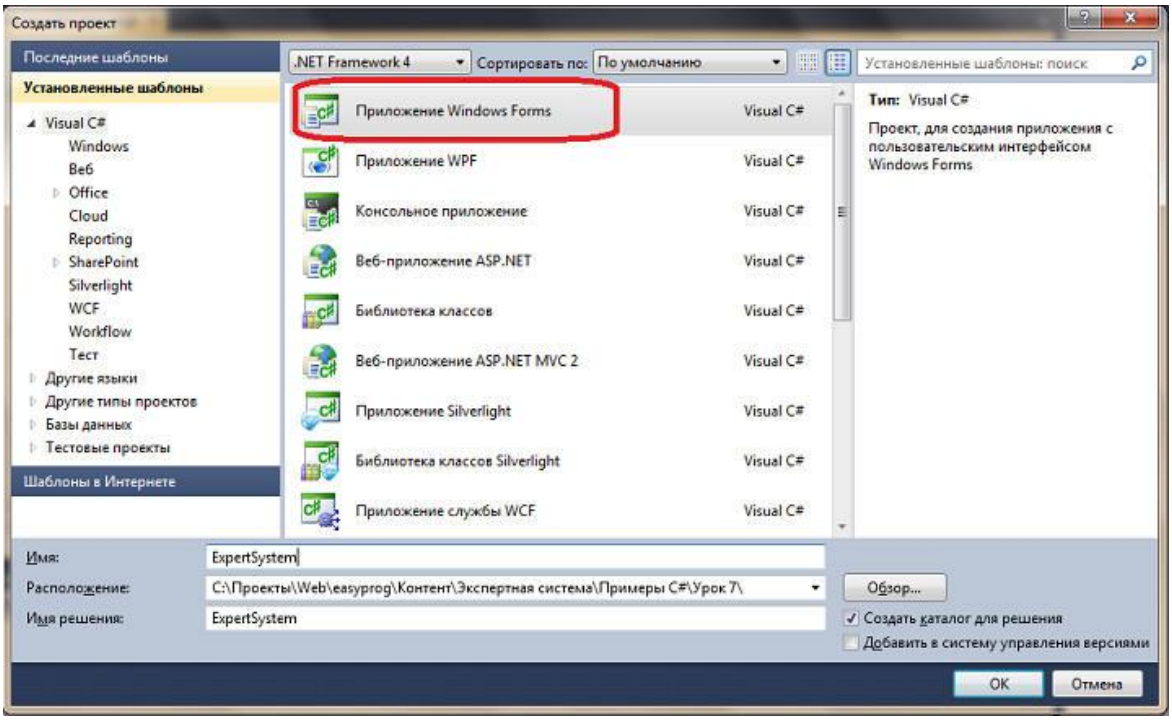
1. Какое пространство имен отвечает за файловый ввод-вывод?
2. Опишите основные методы класса Directory.
3. Опишите порядок работы для создания текстового файла в нужной кодировке.

17. Лабораторная работа №14. Разработка модулей экспертной системы

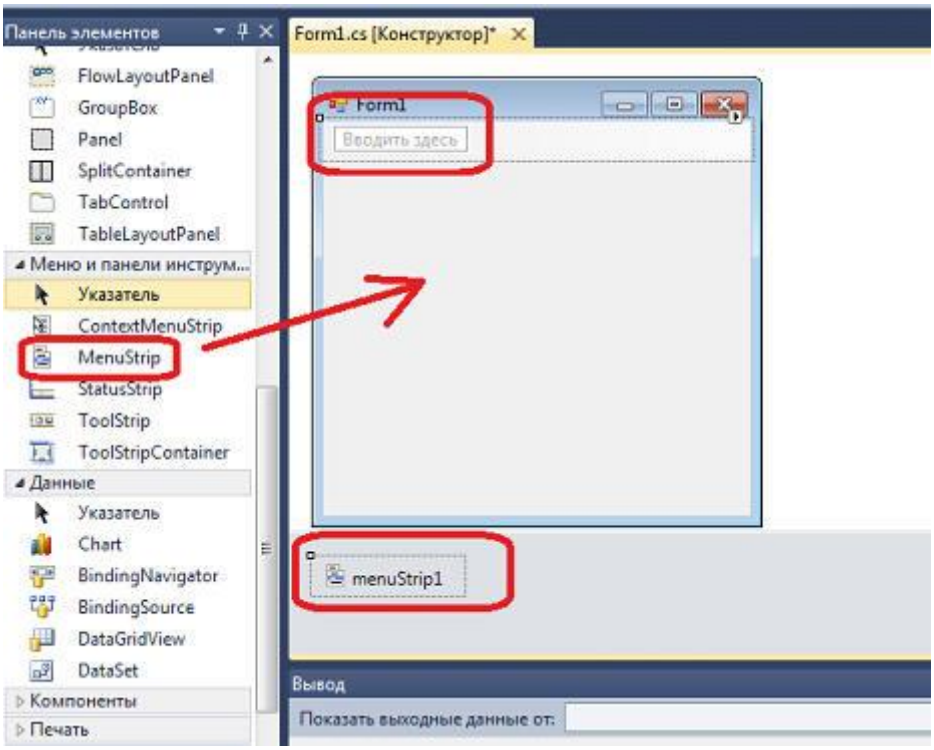
Целью работы является изучение порядка разработки модулей экспертных систем. Результатом практической работы является отчет, в котором должны быть приведено описание разработанных алгоритмов и результаты отладки ПО.

Для выполнения лабораторной работы № 14 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

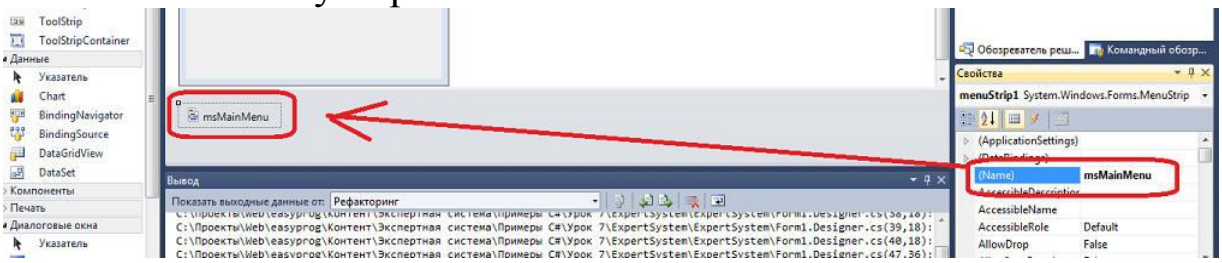
И так, запускаем Visual Studio и создаем новый проект*:



Кинем на форму меню*:

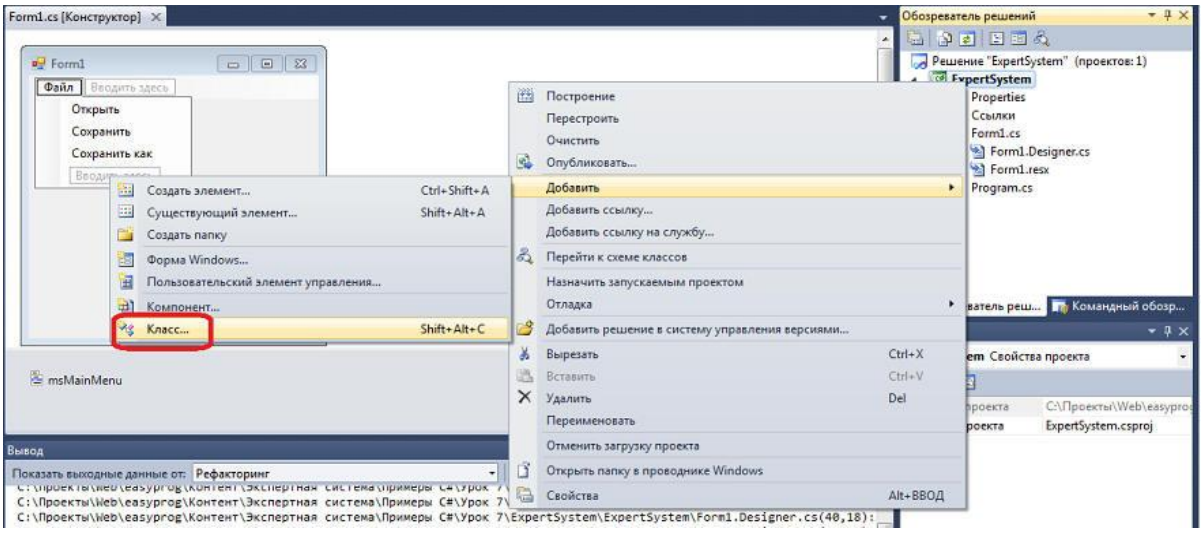


Назначим ему нормальное имя*:

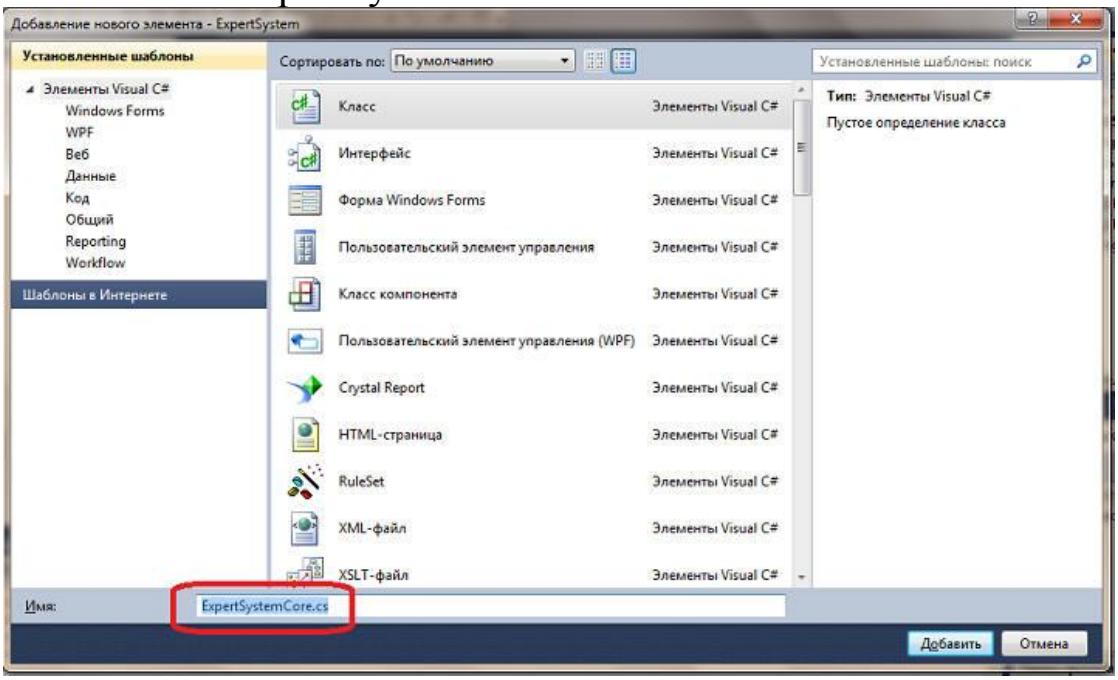


И введем пока очень простое меню*:

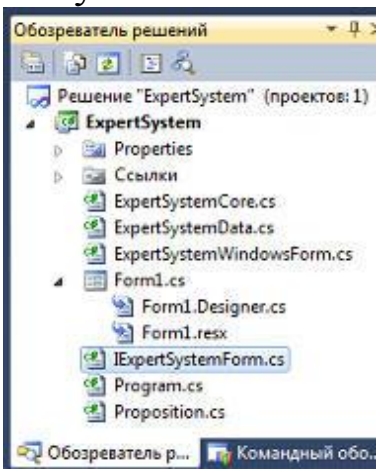
Теперь добавим к нашему проекту новый класс*:



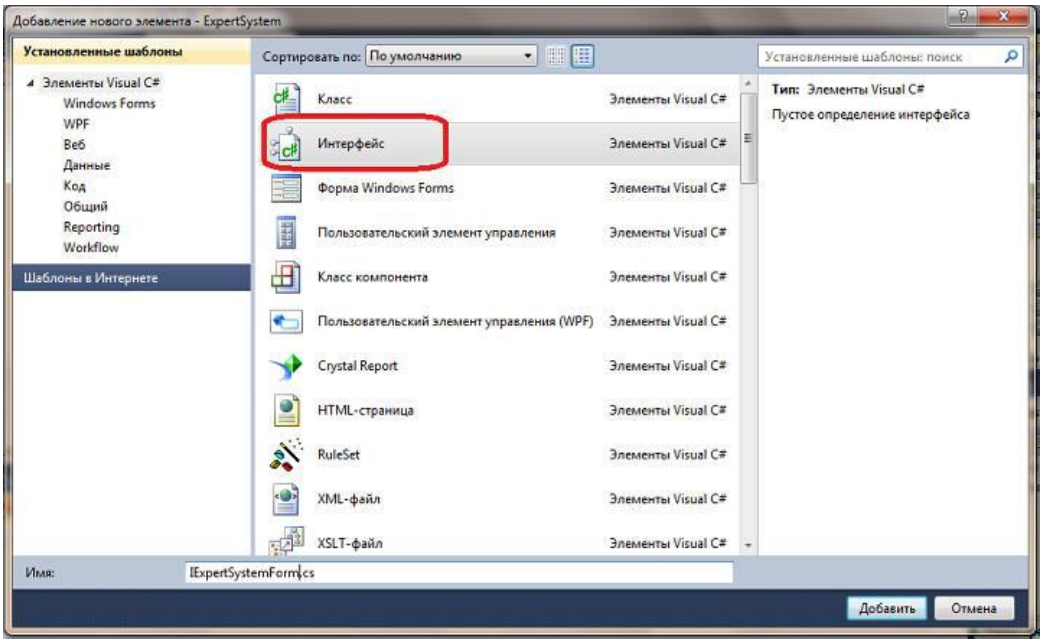
Назвав его ExpertSystemCore*:



Аналогично нам нужно добавить другие классы (ExpertSystemData.cs, ExpertSystemWindowsForm.cs, Proposition.cs), а так же интерфейс IExpertSystemForm.cs*:



Для добавления интерфейса мы выбираем интерфейс*:



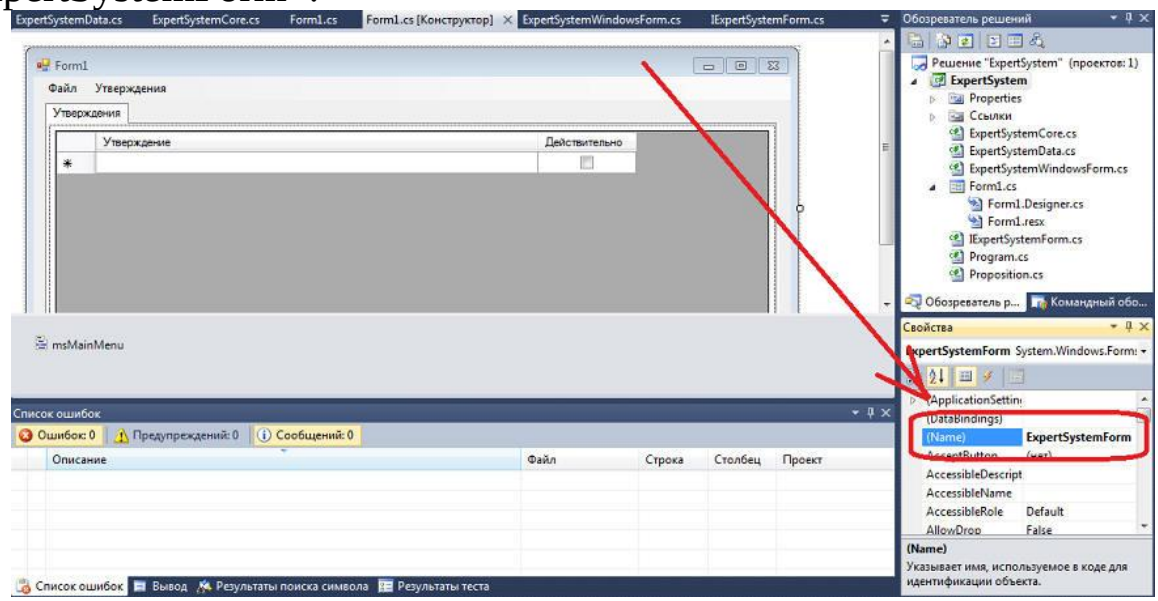
Для чего мы делаем интерфейс? Для связи ядра экспертной системы и диалоговой формы. Вдруг мы в будущем захотим сильно переделать пользовательский интерфейс, или переписать нашу ЭС на технологию **WPF**. В этом случае мы просто меняем реализацию интерфейса IExpertSystemForm и все. А логика ЭС как была так и осталась, мы ее трогать уже не будем.

Для начала мы просто предусмотрим в интерфейсе метод draw_propositions – перерисовать таблицу утверждений и свойство Caption – заголовок формы:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ExpertSystem
{
   interface IExpertSystemForm
   {
      /// <summary>
      /// Заголовок формы
      /// </summary>
      string Caption { get; set; }

      /// <summary>
      /// Перерисовать утверждения
      /// </summary>
```

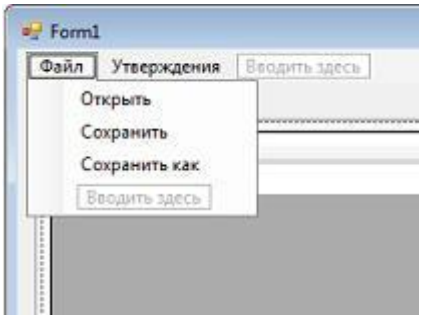
```
        /// <param name=«propositions»>Список утверждений</param>
        void draw_propositions(List<Proposition> propositions);
    }
}
```

Для того чтобы реализовать этот интерфейс, нам потребуется сама форма. В принципе, форма у нас уже есть, она создалась автоматически, дадим ей только нормальное название ExpertSystemForm*:

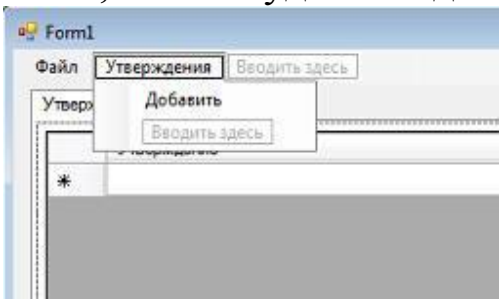


К форме нам надо присобачить меню.

В меню «Файл» у нас пока будет три пункта: «открыть», «сохранить» и «сохранить как»*:

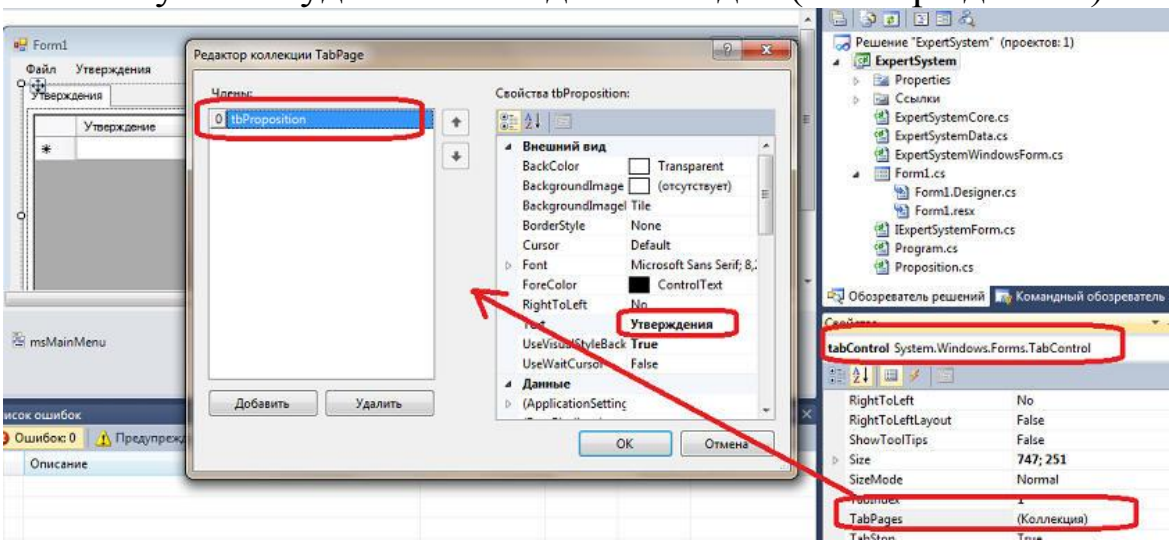


В меню «Утверждения» пока только один, мы сначала создадим заготовку программы, потом будем ее «допиливать»*:

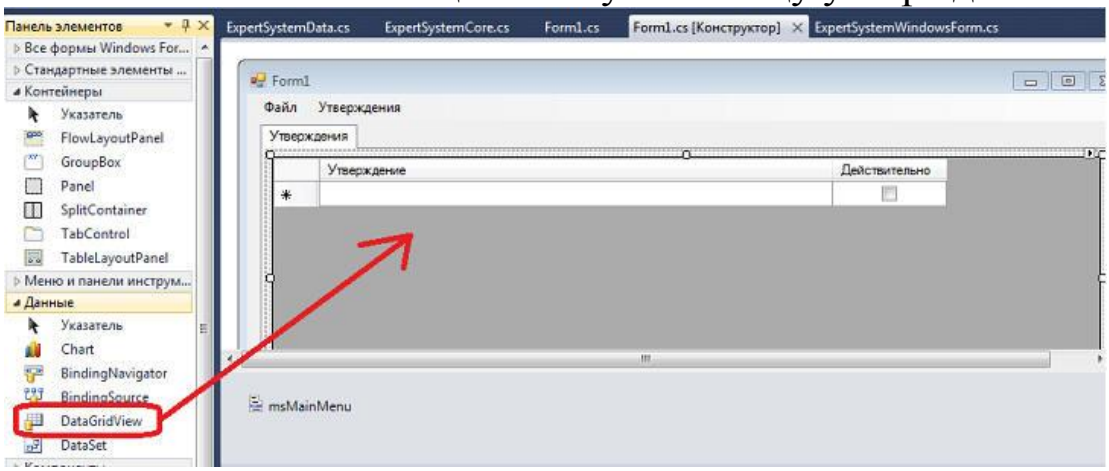


Еще нам надо поместить на форму контейнер для таблицы (TabControl)*:

Пока у него будет только одна закладка («Утверждения»)*:



И на нее то мы и помещаем самую таблицу утверждений*:



Таблицу утверждений мы будем хранить в типизированном списке List, тип элементов Proposition, это класс нам надо создать. Вводим его код в файл Proposition.cs (обратите внимание, что класс сериализуемый, чтобы мы могли сохранить его в файл):

```
using System.Linq;
using System.Text;
```

```
namespace ExpertSystem
{
   /// <summary>
   /// Утверждение. Например «Это летает»,
   /// »Это ползает», «Имеет хвост», «Имеет перья»
   /// </summary>
   [Serializable]
   public class Proposition
   {
      /// <summary>
      /// Текст утверждения
      /// </summary>
      public string caption;

      /// <summary>
      /// Имеет ли место данное утверждение
      /// </summary>
      public bool itis;
   }
}
```

Сами данные экспертной системы у нас будут храниться в объекте класса ExpertSystemData (пока там только утверждения):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ExpertSystem
{
   [Serializable]
   public class ExpertSystemData
   {
      public List<Proposition> propositions;

      public ExpertSystemData()
      {
         propositions = new List<Proposition>();
```

```
        }
    }
}
```

Наконец, управлять экспертной системой будет ядро, реализуем в нем функции сохранения и открытия файлов, а также метод для тестирования добавления нового утверждения:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;


namespace ExpertSystem
{
    public class ExpertSystemCore
    {

        public ExpertSystemCore(ExpertSystemForm app_form)
        {
            data = new ExpertSystemData();
            form = new ExpertSystemWindowsForm();
            (form as ExpertSystemWindowsForm).form = app_form;
            FFileName = »«;
        }

        public void AddProposition()
        {
            Proposition prop = new Proposition();
            prop.caption = »Тест»;
            prop.itis = true;
            data.propositions.Add(prop);
            form.draw_propositions(data.propositions);
        }
```

```
        /// <summary>
        /// Скрытое имя файла
        /// </summary>
        private string FFileName;

        /// <summary>
        /// Имя файла, открытого в даннмый момент
        /// </summary>
        public string FileName
        {
            get
            {
                return FFileName;
            }
        }

        /// <summary>
        /// База знаний экспертной системы,
        /// а так же другие нужные для работы данные
        /// </summary>
        private ExpertSystemData data;

        /// <summary>
        /// Связь с формой приложения через интерфейс IExpertSystemForm
        /// </summary>
        private IExpertSystemForm form;


        /// <summary>
        /// Обновить форму приложения
        /// </summary>
        private void UpdateForm()
        {
            form.Caption = FileName;
        }
```

```
        /// <summary>
        /// Открыть файл экспертной системы
        /// </summary>
        public void Open()
        {
            OpenFileDialog openFileDialog = new OpenFileDialog();
            openFileDialog.Title = »Открыть проект...»;
            openFileDialog.Filter = »Файлы экспертных систем|*.es|Все файлы|*.*»;

            if (openFileDialog.ShowDialog() == System.Windows.Forms.DialogResult.Cancel) return;
            try
            {
                FileStream fs = new FileStream(openFileDialog.FileName, FileMode.Open);
                BinaryFormatter formatter = new BinaryFormatter();
                data = (ExpertSystemData)formatter.Deserialize(fs);
                fs.Close();
                FFileName = openFileDialog.FileName;
                UpdateForm();
                form.draw_propositions(data.propositions);
            }
            catch (IOException ex)
            {
                MessageBox.Show(ex.Message, »Ошибка», MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
        }

        /// <summary>
        /// Сохранить файл экспертной системы
        /// </summary>
        public void Save()
        {
            //Если имя файла не задано то вызываем диалог выбора файла
            if (FFileName == »«)
```

```
            {
               SaveAs();
               return;
            }
            FileStream fs = new FileStream(FFileName, File-
Mode.OpenOrCreate);
            BinaryFormatter formatter = new BinaryFormatter();
            formatter.Serialize(fs, data);
            fs.Close();
            UpdateForm();
         }

         /// <summary>
         /// Сохранить файл экспертной системы с диалогом выбо-
ра файла
         /// </summary>
         public void SaveAs()
         {
            SaveFileDialog saveFileDialog = new SaveFileDialog();
            saveFileDialog.Title = »Сохранение проекта...»;
            saveFileDialog.Filter = »Файлы экспертных сис-
тем|*.es|Все файлы|*.*»;
            if (saveFileDialog.ShowDialog() == Sys-
tem.Windows.Forms.DialogResult.Cancel) return;
            FFileName = saveFileDialog.FileName;
            Save();
         }

      }
   }
```

Теперь можно реализовать и обработчики пунктов меню, вот как будет выглядеть класс главной формы после их реализации:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
```

```
using System.Text;
using System.Windows.Forms;

namespace ExpertSystem
{
   public partial class ExpertSystemForm : Form
   {
      /// <summary>
      /// Ядро экспертной системы
      /// </summary>
      private ExpertSystemCore core;

      public ExpertSystemForm()
      {
         InitializeComponent();
         core = new ExpertSystemCore(this);
      }

      private void tsmiOpen_Click(object sender, EventArgs e)
      {
         core.Open();
      }

      private void tsmiSave_Click(object sender, EventArgs e)
      {
         core.Save();
      }

      private void tsmiSaveAs_Click(object sender, EventArgs e)
      {
         core.SaveAs();
      }

      private void tsmiAddProposition_Click(object sender, Even-
tArgs e)
      {
         core.AddProposition();
      }
```

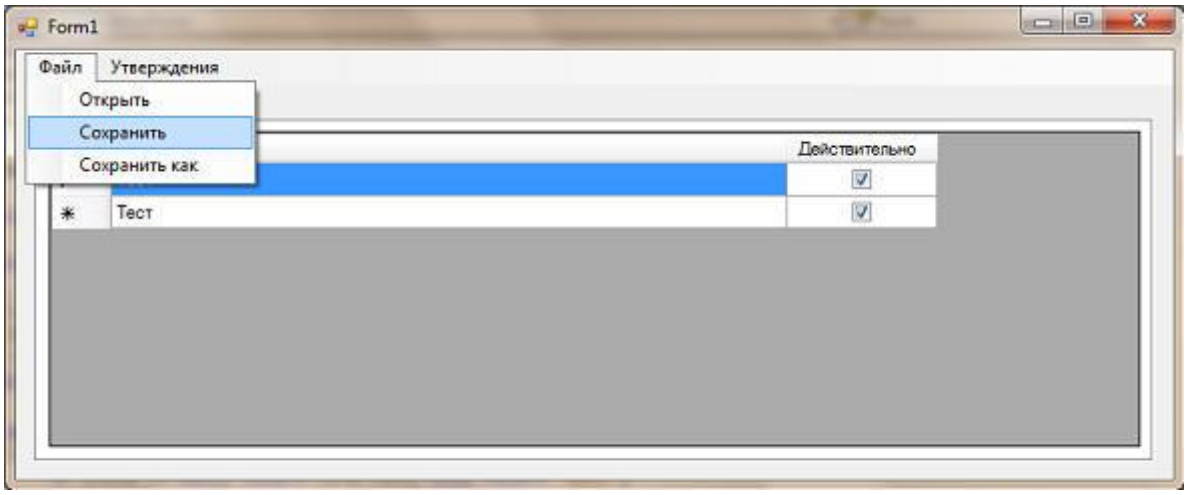
```
        }
    }
```

Ну, и еще у нас осталась реализация интерфейса IExpertSystemForm. Для этого мы создаем класс ExpertSystemWindowsForm:

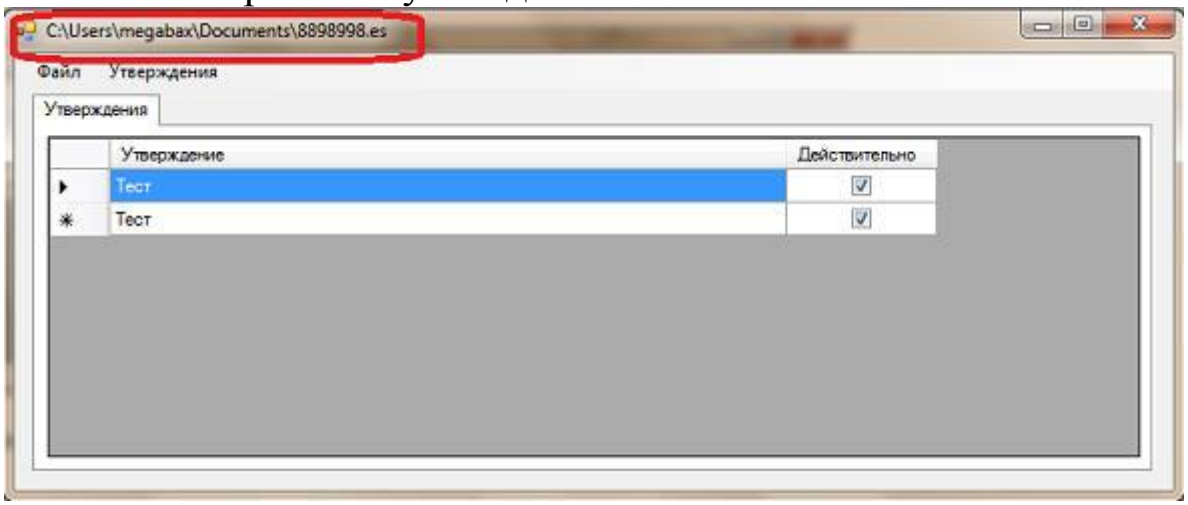
```
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;

    namespace ExpertSystem
    {
        public class ExpertSystemWindowsForm : IExpertSystemForm
        {
            public ExpertSystemForm form { get; set; }

            /// <summary>
            /// Заголовок формы
            /// </summary>
            public string Caption
            {
                get
                {
                    if (form != null) return form.Text; else return »NULL»;
                }
                set
                {
                    if (form != null) form.Text = value;
                }
            }

            /// <summary>
            /// Перерисовать утверждения
            /// </summary>
            /// <param name=«propositions»>Список утверждений</param>
            public void draw_propositions(List<Proposition> proposi-
tions)
```

```
        {
            form.dgvPropositions.Rows.Clear();
            if (propositions.Count == 0) return;
            form.dgvPropositions.RowCount = propositions.Count;
            int j = 0;
            foreach (Proposition item in propositions)
            {
                redraw_row(j, item);
                j++;
            }
        }

        /// <summary>
        /// Перерисовать строку
        /// </summary>
        /// <param name=«j»>Номер строки</param>
        /// <param name=«item»>Утверждение</param>
        private void redraw_row(int j, Proposition item)
        {
            if (j >= form.dgvPropositions.RowCount) return;
            if (j == -1) return;
            form.dgvPropositions.Rows[j].Cells[0].Value = item.caption;
            form.dgvPropositions.Rows[j].Cells[1].Value = item.itis;
        }


    }
}
```

После чего мы можем начать тестирование нашей заготовки программы. Попробуем добавить парочку тестовых утверждений и сохранить файл:



После сохранения у нас должен измениться заголовок:



Проверьте, открывает ли программа сохраненный файл, восстанавливаются ли данные, которые вы сохранили.

Контрольные вопросы

1. Что такое экспертная система?
2. Как можно создать модуль на языке C# для экспертной системы?
3. Как происходит интеграция модулей в одну программу?

18. Лабораторная работа №15. Создание сетевого сервера и сетевого клиента

Целью работы является изучение порядка разработки сетевых приложений. Результатом лабораторной работы является отчет, в котором должны быть приведено описание разработанных алгоритмов и результаты отладки ПО.

Для выполнения лабораторной работы № 15 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Основы работы стека TCP/IP

TCP/IP – это название набора сетевых протоколов. На самом деле передаваемый пакет проходит несколько уровней. (Как на почте: сначала вы пишете письмо, потом помещаете в конверт с адресом, затем на почте на нем ставится штамп и т. д.).

IP протокол – это протокол так называемого сетевого уровня. Задача этого уровня – доставка ip-пакетов от компьютера отправителя к компьютеру получателю. Помимо собственно данных, пакеты этого уровня имеют ip-адрес отправителя и ip-адрес получателя. Номера портов на сетевом уровне не используются. Какому порту, т. е. приложению, адресован этот пакет, был ли этот пакет доставлен или был потерян, на этом уровне неизвестно – это не его задача, это задача транспортного уровня.

TCP и UDP – это протоколы так называемого транспортного уровня. Транспортный уровень находится над сетевым. На этом уровне к пакету добавляется порт отправителя и порт получателя.

TCP – это протокол с установлением соединения и с гарантированной доставкой пакетов. Сначала производится обмен специальными пакетами для установления соединения, происходит что-то вроде рукопожатия (-Привет. -Привет. -Поболтаем? -Давай.). Далее по этому соединению туда и обратно посылаются пакеты (идет беседа), причем с проверкой, дошел ли пакет до получателя. Если пакет не дошел, то он посылается повторно («повтори, не расслышал»).

UDP – это протокол без установления соединения и с негарантированной доставкой пакетов. (Типа: крикнул что-нибудь, а услышат тебя или нет – неважно).

Над транспортным уровнем находится прикладной уровень. На этом уровне работают такие протоколы, как http, ftp и пр. Например

HTTP и FTP – используют надежный протокол TCP, а DNS-сервер работает через ненадежный протокол UDP. Основные определения.

Принципы работы интернет-протоколов TCP/IP по своей сути очень просты и сильно напоминают работу нашей почты.

Вспомните, как работает наша обычная почта. Сначала вы на листке пишете письмо, затем кладете его в конверт, заклеиваете, на обратной стороне конверта пишете адреса отправителя и получателя, а потом относите в ближайшее почтовое отделение. Далее письмо проходит через цепочку почтовых отделений до ближайшего почтового отделения получателя, откуда оно тетей-почтальоном доставляется по указанному адресу получателя и опускается в его почтовый ящик (с номером его квартиры) или вручается лично. Все, письмо дошло до получателя. Когда получатель письма захочет вам ответить, то он в своем ответном письме поменяет местами адреса получателя и отправителя, и письмо отправится к вам по той же цепочке, но в обратном направлении.

На конверте письма будет написано примерно следующее:

Адрес отправителя:
От кого: Иванов Иван Иванович
Откуда: Ивантеевка, ул. Большая , д. 8, кв. 25

Адрес получателя:
Кому: Петров Петр Петрович
Куда: Москва, Усачевский переулок, д. 105, кв. 110

Теперь мы готовы рассмотреть взаимодействие компьютеров и приложений в сети Интернет (да и в локальной сети тоже). Обратите внимание, что аналогия с обычной почтой будет почти полной.

Каждый компьютер (он же: узел, хост) в рамках сети Интернет тоже имеет уникальный адрес, который называется IP-адрес (Internet Protocol Address), например: 195.34.32.116. IP адрес состоит из четырех десятичных чисел (от 0 до 255), разделенных точкой. Но знать только IP адрес компьютера еще недостаточно, т. к. в конечном счете обмениваются информацией не компьютеры сами по себе, а приложения, работающие на них. А на компьютере может одновременно работать сразу несколько приложений (например, почтовый сервер, веб-сервер и пр.). Для доставки обычного бумажного

письма недостаточно знать только адрес дома – необходимо еще знать номер квартиры. Также и каждое программное приложение имеет подобный номер, именуемый номером порта. Большинство серверных приложений имеют стандартные номера, например: почтовый сервис привязан к порту с номером 25 (еще говорят: «слушает» порт, принимает на него сообщения), веб-сервис привязан к порту 80, FTP – к порту 21 и так далее. Когда запускается серверная программа, она устанавливает настройки TCP/IP стека операционной системы таким образом, чтобы пакеты, приходящие на определенный порт, перенаправлялись именно данной программе. Это и называется «слушать порт».

Таким образом, имеем следующую практически полную аналогию с нашим обычным почтовым адресом:

«адрес дома» = «IP компьютера»

«номер квартиры» = «номер порта»

В компьютерных сетях, работающих по протоколам TCP/IP, аналогом бумажного письма в конверте является пакет, который содержит собственно передаваемые данные и адресную информацию – адрес отправителя и адрес получателя, например:

Адрес отправителя (Source address):
IP: 82.146.49.55
Port: 2049

Адрес получателя (Destination address):
IP: 195.34.32.116
Port: 53

Данные пакета:
...

Комбинация: «IP адрес и номер порта» – называется «сокет».

В нашем примере мы с сокета 82.146.49.55:2049 посылаем пакет на сокет 195.34.32.116:53, т. е. пакет пойдет на компьютер, имеющий IP адрес 195.34.32.116, на порт 53. А порту 53 соответствует сервер распознавания имен (DNS-сервер), который примет этот пакет. Зная адрес отправителя, этот сервер сможет после обработки

нашего запроса сформировать ответный пакет, который пойдет в обратном направлении на сокет отправителя 82.146.49.55:2049, который для DNS сервера будет являться сокетом получателя.

Как правило, взаимодействие осуществляется по схеме «клиент-сервер»: «клиент» запрашивает какую-либо информацию (например, страницу сайта), сервер принимает запрос, обрабатывает его и посылает результат. Номера портов серверных приложений общеизвестны, например: почтовый SMTP сервер «слушает» 25-й порт, POP3 сервер, обеспечивающий чтение почты из ваших почтовых ящиков «слушает» 110-порт, веб-сервер – 80-й порт и пр.

Основные классы для работы с сетью

Перед тем, как начать изучать основы передачи данных через сеть, следует изучить работу с IP адресами компьютеров в сети. Каждый адрес представляет собой 4-байтное число, однозначно идентифицирующее компьютер в сети. Для удобства работы с IP адресами каждый байт записывают в виде десятичного числа и между ними ставят точку, например «123.23.24.24». Для хранения IP адресов библиотека .Net Framework предлагает класс *IPAddress* из пространства имен *System.Net*.

Конструкторы этого класса:

IPAddress(Byte[])	Инициализирует новый экземпляр класса IPAddress с указанным адресом, заданным в виде массива *Byte*.
IPAddress(Int64)	Инициализирует новый экземпляр класса IPAddress с указанным адресом, заданным в виде массива *Int64*.

Основные метод, которые предполагается использовать:

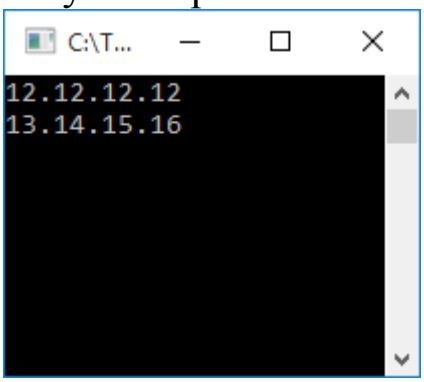
Parse	Преобразует строку IP-адреса в экземпляр класса IPAddress. (Метод статический)
ToString	Преобразует адрес в Интернете в его стандартный формат.

Примеры использования данных методов:

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Net;

namespace TcpClient
{
   class Program
   {
      static void Main(string[] args)
      {
         byte []arr =  {12,12,12,12};

         IPAddress addr = new IPAddress (arr);
         IPAddress addr1 = IPAddress.Parse («13.14.15.16»);


         Console.WriteLine(addr.ToString());
         Console.WriteLine(addr1.ToString());

         Console.ReadKey();
      }
   }
}
```

Результат работы:



Для установления сетевого соединения одного IP адреса недостаточно, нужно знать порт («адрес квартиры»), к которому необходимо подключиться. Комбинация IP адреса и порта в терминах

библиотеки .Net Framework называется Конечной точкой IP и описывается классом *IPEndPoint.*

Конструкторы этого класса:

IPEndPoint(Int64, Int32)	Инициализирует новый экземпляр класса IPEndPoint с заданными адресом и номером порта.
IPEndPoint(IPAddress, Int32)	Инициализирует новый экземпляр класса IPEndPoint с заданными адресом и номером порта.

Для преобразования конечной точки IP в строку используется метод *ToString().*

Пример работы:

```
    static void Main(string[] args)
     {
        byte []arr =  {12,12,12,12};

        IPAddress addr = new IPAddress (arr);
        IPEndPoint endp = new IPEndPoint(addr, 1024);

        Console.WriteLine(endp.ToString());

        Console.ReadKey();
}
```

Следующий класс, который следует рассмотреть – это класс *Socket.*

Сокет – это один конец двустороннего канала связи между двумя программами, работающими в сети. Соединяя вместе два сокета, можно передавать данные между разными процессами (локальными или удаленными). Реализация сокетов обеспечивает инкапсуляцию протоколов сетевого и транспортного уровней.

Основные свойства и методы класса Socket:

LocalEndPoint	Локальная конечная точка.
RemoteEndPoint	Удаленная конечная точка
void Bind(EndPoint localEP)	Связывает объект *Socket* с локальной конечной точкой
void Listen(int backlog)	Устанавливает объект *Socket* в состояние прослушивания, где *backlog* – максимальная длина очереди ожидающих подключений
Socket Accept()	Создает новый объект Socket для заново созданного подключения.
void Connect(EndPoint remoteEP)	Создает подключение к удаленному узлу. У этого метода существует несколько перегрузок.
int Receive(byte[] buffer)	Возвращает данные из связанного объекта *Socket* в приемный буфер *buffer*. У этого метода существует несколько перегрузок.
int Send(byte[] buffer)	Передает данные в подключенный объект *Socket*. У этого метода существует несколько перегрузок.

Работа серверного приложения может быть представлена в виде диаграммы:

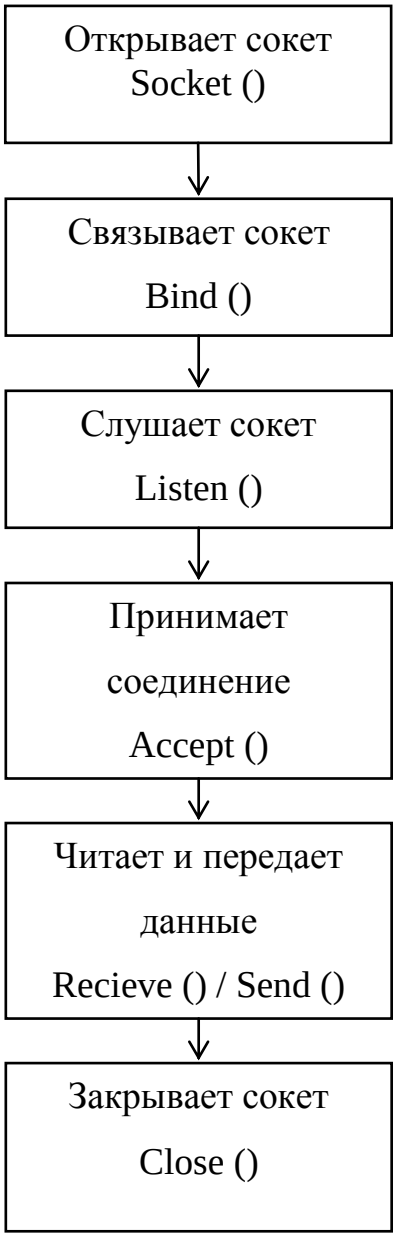


Диаграмма работы сервера

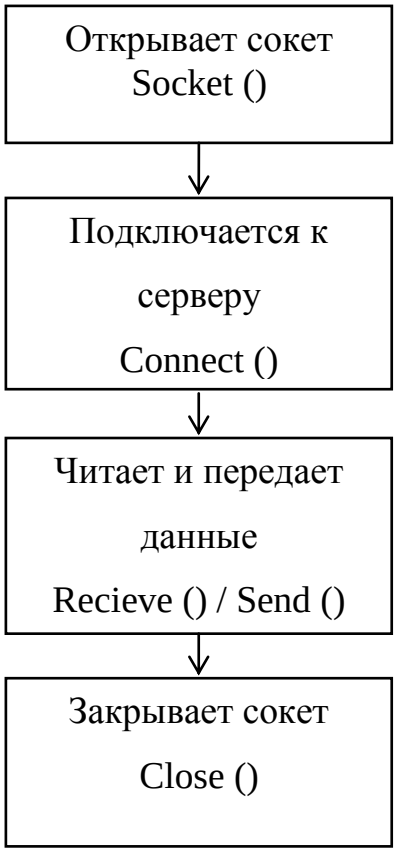


Диаграмма работы клиента

Задания.

1. Разработать чат. Нужен сервер, который принимает сообщения от клиентов. Клиент при подключении к серверу указывает имя пользователя. Далее серверу отправляются сообщения. Сервер, приняв информацию от клиента, должен переправить полученное сообщение всем клиентам с указанием имени пользователя. Клиент просто получает все сообщения от сервера и отображает их в отдельном элементе управления.

2. Разработать простейший HTTP сервер, который реализует лишь метод GET. Файлы для работы брать из указанного в настрой-

ках сервера каталога. Необходимо предусмотреть два типа содержимого – текст и изображение JPG. Проверку работоспособности проводить при помощи браузера.

3. Написать простой клиент отправки электронных писем по протоколу SMTP. Работоспособность демонстрировать с использованием сервера MAIL.KUZSTU.RU логин и пароль использовать тот же, что и для авторизации на прокси-сервере. (Староста когда-то получал их для всей группы в нулевом корпусе)

4. Написать простой клиент для получения электронной почты по протоколу POP3. Примечания см. в задании 3.

Контрольные вопросы

1. Какие классы используются для создания программы, обменивающейся данными через стек протоколов TCP/IP?
2. Какие действия необходимо выполнить, чтобы создать серверную часть приложения?
3. Какие действия необходимо выполнить, чтобы создать клиентское приложение?

СОДЕРЖАНИЕ