МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ КАЗАХСТАН
КАСПИЙСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕХНОЛОГИЙ И ИНЖИНИРИНГА ИМ. Ш.ЕСЕНОВА

ЖУМАДИЛОВА М.Б.

ИНТЕРФЕЙСЫ КОМПЬЮТЕРНЫХ СИСТЕМ

Актау 2011 год

УДК 004
ББК 32.973.202
Ж 88

Рецензенты: Садыков Н.Р., доктор физико-математических наук, профессор,
Проректор по УМР Каспийского государственного университета
Технологий и Инжиниринга имени Ш.Есенова
Кулиев Ю.М., доктор технических наук, профессор,
ТОО «Проектный институт «Оптимум»
Дуйсембаев Б.М., кандидат физико-математических наук, доцент,
Заместитель директора Актауского колледжа Иностранных языков

Жумадилова М.Б.
Интерфейсы компьютерных систем: учебное пособие/М.Б.Жумадилова - Каспийский государственный университет Технологий и Инжиниринга имени Ш.Есенова, Актау, 2011,-151 с.

ISBN 978-601-7276-75-1

Учебное пособие «Интерфейсы компьютерных систем» предназначено для студентов специальности 050704 «Вычислительная техника и программное обеспечение». Целью учебного пособия «Интерфейсы компьютерных систем» является рассмотрение методов проектирования пользовательских интерфейсов, принципы организации и функционирования программно-аппаратных интерфейсов в компьютерных системах. Пособие охватывает основные главы дисциплины, приводится анализ пользовательских интерфейсов, материал дается в понятной, доступной для пользователя форме.

Основные положения учебного материала излагаются с необходимыми обоснованиями, сопровождаются большим количеством рисунков, указывающих этапы выполнения задания. В учебном пособии содержатся вопросы для самоконтроля студентов, что существенно облегчает изучение материала и помогает в усвоении материала студентам.

Рекомендовано к печати решением Учебно-методического Совета Каспийского государственного университета Технологий и Инжиниринга имени Ш.Есенова

ISBN 978-601-7276-75-1

©КГУТиИ имени Ш.Есенова, 2011

СОДЕРЖАНИЕ

Лекция 1.

Тема: Проблемы разработки сложных программных систем

Цель занятия: Рассмотреть понятие сложной программы и отличия сложных программ от простых, а также основные проблемы разработки сложных программ.

Программы "большие" и "маленькие"

Каждый человек, хоть раз написавший какую-либо программу, достаточно хорошо может представить себе, как разработать "небольшую" программу, решающую обычно одну конкретную несложную задачу и предназначенную, чаще всего, для использования одним человеком или узкой группой людей.

Ущерб от неправильной работы программы практически нулевой (за исключением возможности обрушения ею системы, в которой выполняются и другие, более важные задачи).

Не требуется дополнять программу новыми возможностями, практически никому не нужно разрабатывать ее новые версии или исправлять найденные ошибки.

В связи со сказанным выше, не очень нужно прилагать к программе подробную и понятную документацию — для человека, который ею заинтересуется, не составит большого труда понять, как ею пользоваться, просто по исходному коду.

Сложные или "большие" программы, называемые также **программными системами, программными комплексами, программными продуктами,** отличаются от "небольших" не столько по размерам (хотя обычно они значительно больше), сколько по наличию дополнительных факторов, связанных с их востребованностью и готовностью пользователей платить деньги как за приобретение самой программы, так и за ее сопровождение и даже за специальное обучение работе с ней.

Обычно сложная программа обладает следующими свойствами:

- Она решает одну или несколько связанных задач, зачастую сначала не имеющих четкой постановки, настолько важных для каких-либо лиц или организаций, что те приобретают значимые выгоды от ее использования.
- Существенно, чтобы она была удобной в использовании. В частности, она должна включать достаточно полную и понятную пользователям документацию, возможно, также специальную документацию для администраторов, а также набор документов для обучения работе с программой.
- Ее низкая производительность на реальных данных приводит к значимым потерям для пользователей.
- Ее неправильная работа наносит ощутимый ущерб пользователям и другим организациям и лицам, даже если сбои происходят не слишком часто.
- Для выполнения своих задач она должна взаимодействовать с другими программами и программно-аппаратными системами, работать на разных платформах.

• Пользователи, работающие с ней, приобретают дополнительные выгоды от того, что программа развивается, в нее вносятся новые функции и устраняются ошибки. Необходимо наличие проектной документации, позволяющей развивать ее, возможно, вовсе не тем разработчикам, которые ее создавали, без больших затрат на обратную разработку (реинжиниринг).
• В ее разработку вовлечено значительное количество людей (более 5-ти человек). "Большую" программу практически невозможно написать с первой попытки, с небольшими усилиями и в одиночку.
• Намного больше количество ее возможных пользователей, и еще больше тех лиц, деятельность которых будет так или иначе затронута ее работой и результатами.
Примером "большой" программы может служить стандартная библиотека классов Java, входящая в Java Development Kit [1].

На основании некоторых из перечисленных свойств можно сделать вывод, что "большая" программа или программная система чаще всего представляет собой не просто код или исполняемый файл, а включает еще и набор проектной и пользовательской документации.

Для разработки программных систем требуются особые методы — как уже говорилось, их нельзя написать "нахрапом". Изучением организационных, инженерных и технических аспектов создания ПО, включая методы разработки, занимается дисциплина, называемая программной инженерией. Большая часть трудностей при разработке программных систем связана с организацией экономически эффективной совместной работы многих людей, приводящей к практически полезному результату. Это требует рассмотрения следующих аспектов.

Над программой обычно работает много людей, иногда географически удаленных друг от друга и из различных организаций. Их работа должна быть организована так, чтобы затраты на разработку были бы покрыты доходами от продаж и предоставления услуг, связанных с полученной программой. В затраты входят зарплаты разработчиков, затраты на закупленное оборудование и программные инструменты разработки, на приобретение лицензий и патентование собственных решений, часто еще и затраты на исследование потребностей клиентов, проведение рекламы и другой маркетинговой деятельности.
Значимые доходы могут быть получены, только если программа будет предоставлять пользователям в реальных условиях их работы такие возможности, что они готовы будут заплатить за это деньги (которым, заметим, без труда можно найти другие полезные применения). Для этого нужно учесть множество аспектов. Доходы от продаж значительно снизятся, если многие из пользователей не смогут воспользоваться программой только потому, что в их компьютерах процессоры слишком медленные или мало оперативной памяти, или потому, что данные к системе часто поступают в искаженном виде и она не может их обработать, или потому что они привыкли работать с графическим интерфейсом, а система требует ввода из командной строки, и т.п.

Важно отметить, что **практически полезная** сложная программная система не обязательно является **"правильной".**

Большинство опытных разработчиков и исследователей считают, что практически значимые программные системы всегда содержат ошибки. При переходе от "небольших" программ к "большим" понятие "правильной" программы становится практически бессмысленным. Говоря о программной системе, (в отличие от приведенной выше программы вычисления числа?), нельзя утверждать, что она "правильная", т.е. всегда правильно решает все поставленные перед ней задачи. Этот факт связан как с практической невозможностью полного доказательства или проверки этого, так и с тем, что смысл существования программной системы — удовлетворение потребностей и запросов большого количества различных заинтересованных лиц. А эти потребности не только нечетко определены, различны для разных групп пользователей и иногда противоречивы, но и значительно изменяются с течением времени.

В связи с этим, вместо рассмотрения **"правильных"** и **"неправильных"** программных систем, в силу практического отсутствия первых, рассматривают **"достаточно качественные"** и **"недостаточно качественные".**

Часто программное обеспечение (ПО) нельзя рассматривать отдельно от программно-аппаратной системы, куда оно входит в качестве составной части. Изучением вопросов, связанных с разработкой и эксплуатацией программно-аппаратных систем, занимается **системная инженерия**. В ее рамки попадает огромное количество проблем, связанных с аппаратной частью систем и обеспечением нужного уровня интеграции программной и аппаратной составляющих. Мы только изредка будем затрагивать вопросы, касающиеся системной инженерии в целом, в основном ограничиваясь аспектами, относящимися непосредственно к ПО.

Адекватность, полнота, минимальность и простота интерфейсов.

Этот принцип объединяет ряд свойств, которыми должны обладать хорошо спроектированные интерфейсы.

Адекватность интерфейса означает, что интерфейс модуля дает возможность решать именно те задачи, которые нужны пользователям этого модуля.

Например, добавление в интерфейс очереди метода, позволяющего получить любой ее элемент по его номеру в очереди, сделало бы этот интерфейс не вполне адекватным — он превратился бы почти в интерфейс списка, который используется для решения других задач. Очереди же используются там, где полная функциональность списка не нужна, а реализация очереди может быть сделана более эффективной.

Полнота интерфейса означает, что интерфейс позволяет решать все значимые задачи в рамках функциональности модуля.

Например, отсутствие в интерфейсе очереди метода offer() сделало бы его бесполезным — никому не нужна очередь, из которой можно брать элементы, а класть в нее ничего нельзя.

Более тонкий пример — методы element() и peek(). Нужда в них возникает, если программа не должна изменять очередь, и в то же время ей нужно узнать, какой элемент лежит в ее начале. Отсутствие такой возможности потребовало бы создавать собственное дополнительное хранилище элементов в каждой такой программе.

Минимальность интерфейса означает, что предоставляемые интерфейсом операции решают различные по смыслу задачи и ни одну из них нельзя реализовать с помощью всех остальных (или же такая реализация довольно сложна и неэффективна).

Представленный в примере интерфейс очереди не минимален — методы element() и peek(), а также poll() и remove() можно выразить друг через друга. Минимальный интерфейс очереди получился бы, например, если выбросить пару методов element() и remove().

Простота интерфейса означает, что интерфейсные операции достаточно элементарны и непредставимы в виде композиций некоторых более простых операций на том же уровне абстракции, при том же понимании функциональности модуля.

Скажем, весь интерфейс очереди можно было бы свести к одной операции Object queue(Object o, boolean remove), которая добавляет в очередь объект, указанный в качестве первого параметра, если это не null, а также возвращает объект в голову очереди (или null, если очередь пуста) и удаляет его, если в качестве второго параметра указать true. Однако такой интерфейс явно сложнее для понимания, чем представленный выше.

Разделение ответственности.

Основной принцип выделения модулей — создание отдельных модулей под каждую задачу, решаемую системой или необходимую в качестве составляющей для решения ее основных задач.

Разделение политик и алгоритмов.

Этот принцип используется для отделения постоянных, неизменяемых алгоритмов обработки данных от изменяющихся их частей и для выделения этих частей, называемых **политиками**, в параметры общего алгоритма.

Так, политика, определяющая формат строкового представления даты и времени, задается в виде форматной строки при создании объекта класса java.text.SimpleDateFormat. Сам же алгоритм построения этого представления основывается на этой форматной строке и на самих времени и дате.

Другой пример. Стоимость товара для клиента может зависеть от привилегированности клиента, размера партии, которую он покупает, и сезонных скидок. Все перечисленные элементы можно выделить в виде политик, являющихся, вместе с базовой ценой товара, входными данными для алгоритма вычисления итоговой стоимости.

Разделение интерфейса и реализации.

Этот принцип используется при отделении внешне видимой структуры модуля, описания задач, которые он решает, от способов решения этих задач.

Пример такого разделения — отделение интерфейса абстрактного списка java.util.List<E> от многих возможных реализаций этого интерфейса, например,

java.util.ArrayList<E>, java.util.LinkedList<E>. Первый из этих классов реализует список на основе массива, а второй — на основе ссылочной структуры данных.

Контрольные вопросы по теме №1:

1. Что означает Адекватность интерфейса?
2. Что означает Полнота интерфейса?
3. Что означает Минимальность интерфейса?
4. Что означает Простота интерфейса?
5. Какими свойствами обладают сложные программы?
6. Что изучает дисциплина: «Программная инженерия»?

Лекция 2.

Тема: Жизненный цикл и процессы разработки ПО

Цель занятия: Вводятся понятия жизненного цикла ПО и технологических процессов его разработки. Рассматриваются различные способы организации жизненного цикла ПО, каскадные и итеративные модели жизненного цикла, а также набор стандартов, регулирующих процессы разработки ПО в целом.

Понятие жизненного цикла ПО

Разработка ПО — разновидность человеческой деятельности. Выделить ее компоненты можно, определив набор задач, которые нужно решить для достижения конечной цели — построения достаточно качественной системы в рамках заданных сроков и ресурсов. Для решения каждой такой задачи организуется вспомогательная деятельность, к которой можно также применить декомпозицию на отдельные, более мелкие деятельности, и т.д. В итоге должно стать понятно, как решать каждую отдельную подзадачу и всю задачу целиком на основе имеющихся решений для подзадач.

В качестве примеров деятельностей, которые нужно проводить для построения программной системы, можно привести **проектирование** — выделение отдельных модулей и определение связей между ними с целью минимизации зависимостей между частями проекта и достижения лучшей его обозримости в целом, **кодирование** — разработку кода отдельных модулей, разработку пользовательской документации, которая необходима для достаточно сложной системы.

Однако для корректного с точки зрения инженерии и экономики рассмотрения вопросов создания сложных систем необходимо, чтобы были затронуты и вопросы эксплуатации системы, внесения в нее изменений, а также самые первые действия в ходе ее создания — анализ потребностей пользователей и выработка решений, "изобретение" функций, удовлетворяющих эти потребности. Без этого невозможно, с одной стороны, учесть реальную эффективность системы в виде отношения полученных результатов ко всем сделанным затратам и, с другой стороны, правильно оценивать в ходе разработки степень соответствия системы реальным нуждам пользователей и заказчиков.

Все эти факторы приводят к необходимости рассмотрения всей совокупности деятельностей, связанных с созданием и использованием ПО, начиная с возникновения идеи о новом продукте и заканчивая удалением его последней копии. Весь период существования ПО, связанный с подготовкой к его разработке, разработкой, использованием и модификациями, начиная с того момента, когда принимается решение разработать/приобрести/собрать из имеющихся компонентов новую систему или приходит сама идея о необходимости программы определенного рода, до того момента, когда полностью прекращается всякое ее использование, называют жизненным циклом ПО.

В ходе жизненного цикла ПО проходит через анализ предметной области, сбор требований, проектирование, кодирование, тестирование, сопровождение и другие виды деятельности. Каждый вид представляет собой достаточно однородный набор действий, выполняемых для решения одной задачи или группы тесно связанных задач в рамках разработки и поддержки эксплуатации ПО.

При этом создаются и перерабатываются различного рода **артефакты** — создаваемые человеком информационные сущности, документы в достаточно общем смысле, участвующие в качестве входных данных и получающиеся в качестве результатов различных деятельностей. Примерами артефактов являются: модель предметной области, описание требований, техническое задание, архитектура системы, проектная документация на систему в целом и на ее компоненты, прототипы системы и компонентов, собственно, исходный код, пользовательская документация, документация администратора системы, руководство по развертыванию, база пользовательских запросов, план проекта и пр.

На различных этапах в создание и эксплуатацию ПО вовлекаются люди, выполняющие различные роли. Каждая роль может быть охарактеризована как абстрактная группа заинтересованных лиц, участвующих в деятельности по созданию и эксплуатации системы и решающих одни и те же задачи или имеющих одни и те же интересы по отношению к ней. Примерами ролей являются: бизнес-аналитик, инженер по требованиям, архитектор, проектировщик пользовательского интерфейса, программист-кодировщик, технический писатель, тестировщик, руководитель проекта по разработке, работник отдела продаж, конечный пользователь, администратор системы, инженер по поддержке и т.п.

Общую структуру жизненного цикла любого ПО задать невозможно, поскольку она существенно зависит от целей, для которых это ПО разрабатывается или приобретается, и от решаемых им задач. Структура жизненного цикла будет существенно разной у программы для форматирования кода, которая сначала делалась программистом для себя, а впоследствии была признана перспективной в качестве продукта и переработана, и у комплексной системы автоматизации предприятия, которая с самого начала задумывалась как таковая. Тем не менее, часто определяют основные элементы структуры жизненного цикла в виде модели жизненного цикла ПО. Модель жизненного

цикла ПО выделяет конкретные наборы видов деятельности (обычно разбиваемых на еще более мелкие активности), артефактов, ролей и их взаимосвязи, а также дает рекомендации по организации процесса в целом. Эти рекомендации включают ответы на вопросы о том, какие артефакты являются входными данными у каких видов деятельности, а какие появляются в качестве результатов, какие роли вовлечены в различные деятельности, как различные деятельности связаны друг с другом, каковы критерии качества полученных результатов, как оценить степень соответствия различных артефактов общим задачам проекта и когда можно переходить от одной деятельности к другой.

Жизненный цикл ПО является составной частью жизненного цикла программно-аппаратной системы, в которую это ПО входит. Поэтому часто различные его аспекты рассматриваются в связи с элементами жизненного цикла системы в целом.

Существует набор стандартов, определяющих различные элементы в структуре жизненных циклов ПО и программно-аппаратных систем. В качестве основных таких элементов выделяют **технологические процессы** — структурированные наборы деятельностей, решающих некоторую общую задачу или связанную совокупность задач, такие как процесс сопровождения ПО, процесс обеспечения качества, процесс разработки документации и пр. Процессы могут определять разные этапы жизненного цикла и увязывать их с различными видами деятельностей, артефактами и ролями заинтересованных лиц.

Стоит отметить, что процессом (или технологическим процессом) называют и набор процессов, увязанных для совместного решения более крупной задачи, например, всей совокупности деятельностей, входящих в жизненный цикл ПО. Таким образом, процессы могут разбиваться на подпроцессы, решающие частные подзадачи той задачи, с которой работает общий процесс.

Стандарты жизненного цикла

Чтобы получить представление о возможной структуре жизненного цикла ПО, обратимся сначала к соответствующим стандартам, описывающим технологические процессы. Международными организациями, такими как:

IEEE — читается "ай-трипл-и", Institute of Electrical and Electronic Engineers, Институт инженеров по электротехнике и электронике;

ISO — International Standards Organization, Международная организация по стандартизации;

EIA — Electronic Industry Association, Ассоциация электронной промышленности;

IEC — International Electrotechnical Commission, Международная комиссия по электротехнике;

а также некоторыми национальными и региональными институтами и организациями (в основном, американскими и европейскими, поскольку именно они оказывают наибольшее влияние на развитие технологий разработки ПО во всем мире):

ANSI — American National Standards Institute, Американский национальный институт стандартов;
SEI — Software Engineering Institute, Институт программной инженерии;
ECMA — European Computer Manufactures Association, Европейская ассоциация производителей компьютерного оборудования;
разработан набор стандартов, регламентирующих различные аспекты жизненного цикла и вовлеченных в него процессов. Для примера приведем группу стандартов ISO
ISO/IEC 12207 Standard for Information Technology — Software Life Cycle Processes [1] (процессы жизненного цикла ПО, есть его российский аналог **ГОСТ Р-1999** [3]).

Определяет общую структуру жизненного цикла ПО в виде 3 ступенчатой модели, состоящей из процессов, видов деятельности и задач. Стандарт описывает вводимые элементы в терминах их целей и результатов, тем самым задавая неявно возможные взаимосвязи между ними, но не определяя четко структуру этих связей, возможную организацию элементов в рамках проекта и метрики, по которым можно было бы отслеживать ход работ и их результативность.

Самыми крупными элементами являются процессы жизненного цикла ПО (lifecycle processes). Всего выделено 18 процессов, которые объединены в 4 группы.

Таблица 2.1. Процессы жизненного цикла ПО по ISO 12207

Основные процессы	**Поддерживающие процессы**	**Организационные процессы**	**Адаптация**
Приобретение ПО; Передача ПО (в использование); Разработка ПО; Эксплуатация ПО;	Поддержка ПО Документирование; Управление конфигурациями; Обеспечение качества; Верификация; Валидация; Совместные экспертизы; Аудит; Разрешение проблем	Управление проектом; Управление инфраструктурой; Усовершенствование процессов; Управление персоналом	Адаптация описываемых стандартом процессов под нужды конкретного проекта

Процессы строятся из отдельных видов деятельности (activities).

Стандартом определены 74 вида деятельности, связанной с разработкой и поддержкой ПО. Ниже мы упомянем только некоторые из них.

- Приобретение ПО включает такие деятельности, как инициация приобретения, подготовка запроса предложений, подготовка контракта, анализ поставщиков, получение ПО и завершение приобретения.
- Разработка ПО включает развертывание процесса разработки, анализ системных требований, проектирование программно-аппаратной системы в

целом, анализ требований к ПО, проектирование архитектуры ПО, детальное проектирование, кодирование и отладочное тестирование, интеграцию ПО, квалификационное тестирование ПО, системную интеграцию, квалификационное тестирование системы, развертывание (установку или инсталляцию) ПО, поддержку процесса получения ПО.

•Поддержка ПО включает развертывание процесса поддержки, анализ возникающих проблем и необходимых изменений, внесение изменений, экспертизу и передачу измененного ПО, перенос ПО с одной платформы на другую, изъятие ПО из эксплуатации.

•Управление проектом включает запуск проекта и определение его рамок, планирование, выполнение проекта и надзор за его выполнением, экспертизу и оценку проекта, свертывание проекта.

Каждый вид деятельности нацелен на решение одной или нескольких **задач (tasks)**. Всего определено 224 различные задачи. Например:

•Развертывание процесса разработки состоит из определения модели жизненного цикла, документирования и контроля результатов отдельных работ, выбора используемых стандартов, языков, инструментов и пр.

•Перенос ПО между платформами состоит из разработки плана переноса, оповещения пользователей, выполнения анализа произведенных действий и пр.

Контрольные вопросы по теме №2:

1. Какой период существования ПО называют жизненным циклом ПО?
2. Перечислите стандарты жизненным циклом ПО, описывающим технологические процессы?
3. Перечислите основные процессы жизненного цикла ПО по ISO 12207?
4. Перечислите основные процессы жизненного цикла систем по ISO 15288?
5. Перечислите основные процессы жизненного цикла ПО и систем по ISO 15504?
6. Перечислите основные уровни зрелости организации жизненного цикла?
7. Перечислите основные модели жизненного цикла?

Лекция 3.

Тема: Унифицированный процесс разработки и экстремальное программирование

Цель занятия: Рассматриваются в деталях модели разработки ПО, предлагаемые в рамках унифицированного процесса разработки Rational (RUP) и экстремального программирования (XP).

Два процесса разработки — унифицированный процесс Rational (Rational Unified Process, RUP) и экстремальное программирование (Extreme Programming, XP).

Оба они являются примерами итеративных процессов, но построены на основе различных предположений о природе разработки программного обеспечения и, соответственно, достаточно сильно отличаются.

RUP является примером так называемого "тяжелого" процесса, детально описанного и предполагающего поддержку собственно разработки исходного

кода ПО большим количеством вспомогательных действий. Примерами подобных действий являются разработка планов, технических заданий, многочисленных проектных моделей, проектной документации и пр. Основная цель такого процесса — отделить успешные практики разработки и сопровождения ПО от конкретных людей, умеющих их применять. Многочисленные вспомогательные действия дают надежду сделать возможным успешное решение задач по конструированию и поддержке сложных систем с помощью имеющихся работников, не обязательно являющихся суперпрофессионалами. Для достижения этого выполняется иерархическое пошаговое детальное описание предпринимаемых в той или иной ситуации действий, чтобы можно было научить обычного работника действовать аналогичным образом. В ходе проекта создается много промежуточных документов, позволяющих разработчикам последовательно разбивать стоящие перед ними задачи на более простые. Эти же документы служат для проверки правильности решений, принимаемых на каждом шаге, а также отслеживания общего хода работ и уточнения оценок ресурсов, необходимых для получения желаемых результатов.

Экстремальное программирование, наоборот, представляет так называемые "живые" (agile) методы разработки, называемые также **"легкими"** процессами. Они делают упор на использовании хороших разработчиков, а не хорошо отлаженных процессов разработки. Живые методы избегают фиксации четких схем действий, чтобы обеспечить большую гибкость в каждом конкретном проекте, а также выступают против разработки дополнительных документов, не вносящих непосредственного вклада в получение готовой работающей программы.

Унифицированный процесс Rational RUP является довольно сложной, детально проработанной итеративной моделью жизненного цикла ПО.

Исторически RUP является развитием модели процесса разработки, принятой в компании Ericsson в 70–80-х годах XX века. Эта модель была создана Джекобсоном (Ivar Jacobson), впоследствии, в 1987 году, основавшим собственную компанию Objectory AB именно для развития технологического процесса разработки ПО как отдельного продукта, который можно было бы переносить в другие организации. После вливания Objectory в Rational в 1995 году разработки Джекобсона были интегрированы с работами Ройса (Walker Royce, сын автора "классической" каскадной модели), Крухтена (Philippe Kruchten) и Буча (Grady Booch), а также с развивавшимся параллельно **универсальным языком моделирования (Unified Modeling Language, UML)**.

RUP основан на трех ключевых идеях:

- Весь ход работ направляется итоговыми целями проекта, выраженными в виде **вариантов использования (use cases)** — сценариев взаимодействия результирующей программной системы с пользователями или другими системами, при выполнении которых пользователи получают значимые для них результаты и услуги. Разработка начинается с выделения вариантов использования и на каждом шаге контролируется степенью приближения к их реализации.

•Основным решением, принимаемым в ходе проекта, является **архитектура** результирующей программной системы. Архитектура устанавливает набор компонентов, из которых будет построено ПО, ответственность каждого из компонентов (т.е. решаемые им подзадачи в рамках общих задач системы), четко определяет интерфейсы, через которые они могут взаимодействовать, а также способы взаимодействия компонентов друг с другом.

•Архитектура является одновременно основой для получения качественного ПО и базой для планирования работ и оценок проекта в терминах времени и ресурсов, необходимых для достижения определенных результатов. Она оформляется в виде набора графических моделей на языке UML.

Основой процесса разработки являются **планируемые** и **управляемые итерации**, объем которых (реализуемая в рамках итерации функциональность и набор компонентов) определяется на основе архитектуры.

RUP выделяет в жизненном цикле 4 основные фазы, в рамках каждой из которых возможно проведение нескольких итераций. Кроме того, разработка системы может пройти через несколько циклов, включающих все 4 фазы.

1. Фаза начала проекта (Inception)

Основная цель этой фазы — достичь компромисса между всеми заинтересованными лицами относительно задач проекта и выделяемых на него ресурсов.

На этой стадии определяются основные цели проекта, руководитель и бюджет, основные средства выполнения — технологии, инструменты, ключевые исполнители. Также, возможно, происходит апробация выбранных технологий, чтобы убедиться в возможности достичь целей с их помощью, и составляются предварительные планы проекта.

На эту фазу может уходить около 10% времени и 5% трудоемкости одного цикла.

2. Фаза проектирования (Elaboration)

Основная цель этой фазы — на базе основных, наиболее существенных требований разработать стабильную базовую архитектуру продукта, которая позволяет решать поставленные перед системой задачи и в дальнейшем используется как основа разработки системы.

На эту фазу может уходить около 30% времени и 20% трудоемкости одного цикла.

3. Фаза построения (Construction)

Основная цель этой фазы — детальное прояснение требований и разработка системы, удовлетворяющей им, на основе спроектированной ранее архитектуры. В результате должна получиться система, реализующая все выделенные варианты использования.

На эту фазу уходит около 50% времени и 65% трудоемкости одного цикла.

4. Фаза внедрения (Transition)

Цель этой фазы — сделать систему полностью доступной конечным пользователям. На этой стадии происходит развертывание системы в ее рабочей среде, бета-тестирование, подгонка мелких деталей под нужды пользователей.

На эту фазу может уходить около 10% времени и 10% трудоемкости одного цикла.

Наиболее важные с точки зрения RUP артефакты проекта — это модели, описывающие различные аспекты будущей системы. Большинство моделей представляют собой наборы диаграмм UML. Основные используемые виды моделей следующие:

Модель вариантов использования (Use-Case Model).
Модель анализа (Analysis Model).
Модель проектирования (Design Model)
Модель реализации (Implementation Model).
Модель развертывания (Deployment Model)
Модель тестирования (Test Model или Test Suite)
Моделирование предметной области (бизнес-моделирование, Business Modeling)

Экстремальное программирование

Экстремальное программирование (Extreme Programming, XP) [4] возникло как эволюционный метод разработки ПО "снизу вверх". Этот подход является примером так называемого метода "живой" разработки (Agile Development Method). В группу "живых" методов входят, помимо экстремального программирования, методы SCRUM, DSDM (Dynamic Systems Development Method, метод разработки динамичных систем), Feature-Driven Development (разработка, управляемая функциями системы) и др.

"Живые" методы появились как протест против чрезмерной бюрократизации разработки ПО, обилия побочных, не являющихся необходимыми для получения конечного результата документов, которые приходится оформлять при проведении проекта в соответствии с большинством "тяжелых" процессов, дополнительной работы по поддержке фиксированного процесса организации, как это требуется в рамках, например, CMM. Большая часть таких работ и документов не имеет прямого отношения к разработке ПО и обеспечению его качества, а предназначена для соблюдения формальных пунктов контрактов на разработку, получения и подтверждения сертификатов на соответствие различным стандартам.

"Живые" методы позволяют большую часть усилий разработчиков сосредоточить собственно на задачах разработки и удовлетворения реальных потребностей пользователей. Отсутствие кипы документов и необходимости поддерживать их в связном состоянии позволяет более быстро и качественно реагировать на изменения в требованиях и в окружении, в котором придется работать будущей программе.

По утверждению авторов XP, эта методика представляет собой не столько следование каким-то общим схемам действий, сколько применение комбинации

следующих техник. При этом каждая техника важна, и без ее использования разработка считается идущей не по XP, согласно утверждению Кента Бека (Kent Beck) [4], одного из авторов этого подхода наряду с Уордом Каннингемом (Ward Cunningham) и Роном Джефрисом (Ron Jeffries).

Живое планирование (planning game)

Его задача — как можно быстрее определить объем работ, которые нужно сделать до следующей версии ПО. Решение принимается, в первую очередь, на основе приоритетов заказчика (т.е. его потребностей, того, что нужно ему от системы для более успешного ведения своего бизнеса) и, во вторую, на основе технических оценок (т.е. оценок трудоемкости разработки, совместимости с остальными элементами системы и пр.). Планы изменяются, как только они начинают расходиться с действительностью или пожеланиями заказчика.

Частая смена версий (small releases)

Самая первая работающая версия должна появиться как можно быстрее и тут же должна начать использоваться. Следующие версии подготавливаются через достаточно короткие промежутки времени (от нескольких часов при небольших изменениях в небольшой программе, до месяца-двух при серьезной переработке крупной системы).

Метафора (metaphor) системы

Метафора в достаточно простом и понятном команде виде должна описывать основной механизм работы системы. Это понятие напоминает архитектуру, но должно гораздо проще, всего в виде одной-двух фраз описывать основную суть принятых технических решений.

Простые проектные решения (simple design)

В каждый момент времени система должна быть сконструирована настолько просто, насколько это возможно. Не надо добавлять функции заранее — только после явной просьбы об этом. Вся лишняя сложность удаляется, как только обнаруживается.

Разработка на основе тестирования (test-driven development)

Разработчики сначала пишут тесты, потом пытаются реализовать свои модули так, чтобы тесты срабатывали. Заказчики заранее пишут тесты, демонстрирующие основные возможности системы, чтобы можно было увидеть, что система действительно заработала.

Постоянная переработка (refactoring)

Программисты постоянно перерабатывают систему для устранения излишней сложности, увеличения понятности кода, повышения его гибкости, но без изменений в его поведении, что проверяется прогоном после каждой переделки тестов. При этом предпочтение отдается более элегантным и гибким решениям, по сравнению с просто дающими нужный результат. Неудачно переработанные компоненты должны выявляться при выполнении тестов и откатываться к последнему целостному состоянию (вместе с зависимыми от них компонентами).

Программирование парами (pair programming)

Кодирование выполняется двумя программистами на одном компьютере. Объединение в пары произвольно и меняется от задачи к задаче. Тот, в чьих

руках клавиатура, пытается наилучшим способом решить текущую задачу. Второй программист анализирует работу первого и дает советы, обдумывает последствия тех или иных решений, новые тесты, менее прямые, но более гибкие решения.

Контрольные вопросы по теме №3:

1. Что такое унифицированный процесс Rational (Rational Unified Process, RUP)?
2. Что такое экстремальное программирование (Extreme Programming, XP)?
3. Перечислите основные фазы жизненного цикла в RUP?
4. Перечислите модели, описывающие различные аспекты будущей системы, с точки зрения RUP артефакты проекта?
5. Перечислите дисциплины, включающие различные наборы деятельностей, которые в разных комбинациях и с разной интенсивностью выполняются на разных фазах, по определению RUP?
6. Перечислите основные техники XP, без их использования разработка считается идущей не по XP?
7. Что является достоинствами XP?

Лекция 4.

Тема: Анализ предметной области и требования к ПО

Цель занятия: Рассматриваются вопросы, связанные с анализом предметной области и выделением требований к разрабатываемой программной системе, а также основные графические модели, используемые в этих деятельностях — диаграммы потоков данных и вариантов использования.

Анализ предметной области

Для того чтобы разработать программную систему, приносящую реальные выгоды определенным пользователям, необходимо сначала выяснить, какие же задачи она должна решать для этих людей и какими свойствами обладать.

Требования к ПО определяют, какие свойства и характеристики оно должно иметь для удовлетворения потребностей пользователей и других заинтересованных лиц. Однако сформулировать требования к сложной системе не так легко. В большинстве случаев будущие пользователи могут перечислить набор свойств, который они хотели бы видеть, но никто не даст гарантий, что это — исчерпывающий список. Кроме того, часто сама формулировка этих свойств будет непонятна большинству программистов: могут прозвучать фразы типа "должно использоваться и частотное, и временное уплотнение каналов", "передача клиента должна быть мягкой", "для обычных швов отмечайте бригаду, а для доверительных — конкретных сварщиков", и это еще не самые тяжелые для понимания примеры.

После этого можно определять область ответственности будущей программной системы — какие именно из выявленных задач будут ею решаться, при решении каких задач она может оказать существенную помощь и чем именно. Определив эти задачи в рамках общей системы задач и

деятельностей пользователей, можно уже более точно сформулировать требования к ПО.

Анализом предметной области занимаются системные аналитики или бизнес-аналитики, которые передают полученные ими знания другим членам проектной команды, сформулировав их на более понятном разработчикам языке. Для передачи этих знаний обычно служит некоторый набор моделей, в виде графических схем и текстовых документов.

Наиболее удобной формой представления информации при анализе предметной области являются графические диаграммы различного рода. Они позволяют достаточно быстро зафиксировать полученные знания, быстро восстанавливать их в памяти и успешно объясняться с заказчиками и другими заинтересованными лицами. Набросать рисунок из прямоугольников и связывающих их стрелок обычно можно гораздо быстрее, чем записать соответствующий объем информации, и на рисунке за один взгляд видно гораздо больше, чем в тексте. Изредка встречаются люди, лучше ориентирующиеся в текстах и более адекватно их понимающие, но чаще рисунки все же более удобны для иллюстрации мыслей и объяснения сложных вещей.

Хотя методы структурного анализа могут значительно помочь при анализе систем и организаций, дальнейшая разработка системы, поддерживающей их деятельность, с использованием объектно-ориентированного подхода часто требует дополнительной работы по переводу полученной информации в объектно-ориентированные модели.

Методы объектно-ориентированного анализа предназначены для обеспечения более удобной передачи информации между моделями анализируемых систем и моделями разрабатываемого ПО. В качестве графических моделей в этих методах вместо диаграмм потоков данных используются рассматривавшиеся при обсуждении RUP диаграммы вариантов использования, а вместо диаграмм сущностей и связей — диаграммы классов.

Выделение и анализ требований

После получения общего представления о деятельности и целях организаций, в которых будет работать будущая программная система, и о ее предметной области, можно определить более четко, какие именно задачи система будет решать. Кроме того, важно понимать, какие из задач стоят наиболее остро и обязательно должны быть поддержаны уже в первой версии, а какие могут быть отложены до следующих версий или вообще вынесены за рамки области ответственности системы. Эта информация выявляется при анализе потребностей возможных пользователей и заказчиков.

Потребности определяются на основе наиболее актуальных проблем и задач, которые пользователи и заказчики видят перед собой. При этом требуется аккуратное выявление значимых проблем, определение того, насколько хорошо они решаются при текущем положении дел, и расстановка приоритетов при рассмотрении недостаточно хорошо решаемых, поскольку чаще всего решить сразу все проблемы невозможно.

После выделения основных потребностей нужно решить вопрос о разграничении области ответственности будущей системы, т.е. определить, какие из потребностей надо пытаться удовлетворить в ее рамках, а какие — нет.

На основе выделенных потребностей пользователей, отнесенных к области ответственности системы, формулируются возможные функции будущей системы, которые представляют собой услуги, предоставляемые системой и удовлетворяющие потребности одной или нескольких групп пользователей (или других заинтересованных лиц). Идеи для определения таких функций можно брать из имеющегося опыта разработчиков (наиболее часто используемый источник) или из результатов мозговых штурмов и других форм выработки идей.

Формулировка функций должна быть достаточно короткой, ясной для пользователей, без лишних деталей.

Имея набор функций, достаточно хорошо поддерживающий решение наиболее существенных задач, с которыми придется работать разрабатываемой системе, можно составлять требования к ней, представляющие собой детализацию работы этих функций. Соотношение между проблемами, потребностями, функциями и требованиями показано на рис.

При этом часто нужно учитывать, что ПО является частью программно-аппаратной системы, требования к которой надо преобразовать в требования к программной и аппаратной ее составляющим. В последнее время, в связи со значительным падением цен на мощное аппаратное обеспечение общего назначения, фокус внимания переместился, в основном, на программное обеспечение. Во многих проектах аппаратная платформа определяется из общих соображений, а поддержку большинства нужных функций осуществляет ПО.

Каждое требование раскрывает детали поведения системы при выполнении ею некоторой функции в некоторых обстоятельствах. При этом часть требований исходит из потребностей и пожеланий заинтересованных лиц и решений, удовлетворяющих эти потребности и пожелания, а часть — из внешних ограничений, накладываемых на систему, например, основными законами той предметной области, в рамках которой системе придется работать, государственным законодательством, корпоративной политикой и пр.

Еще до перехода от функций к требованиям полезно расставить приоритеты и оценить трудоемкость их реализации и рискованность. Это позволит отказаться от реализации наименее важных и наиболее трудоемких,

не соответствующих бюджету проекта функций еще до их детальной проработки, а также выявить возможные проблемные места проекта — наиболее трудоемкие и неясные из

Правила работы с требования к ПО и более общими системными требованиями (к программно-аппаратной системе), определяются следующими двумя стандартами IEEE:

IEEE 830-1998 Recommended Practice for Software Requirements Specifications [7] (рекомендуемые методы спецификации требований к ПО).

IEEE 1233-1998, 2002 Guide for Developing System Requirements Specifications [8] (руководство по разработке спецификаций требований к системам).

Варианты использования

Наиболее широко распространенными техниками фиксации требований в настоящий момент являются структурированные текстовые документы и диаграммы вариантов использования, о которых уже заходила речь при обсуждении RUP.

Вариантом использования (use case) называют некоторый сценарий действий системы, который обеспечивает ощутимый и значимый для ее пользователей результат. На практике в виде одного варианта использования оформляется сценарий действий системы, который будет, скорее всего, неоднократно возникать во время ее работы и имеет достаточно четко определенные условия начала выполнения и завершения.

В языке UML вариант использования изображается в виде овала, помеченного именем представляемого варианта. Варианты использования могут быть связаны с участвующими в них действующими лицами (actors), изображаемыми в виде человечков и представляющими различные роли пользователей системы или внешние системы, взаимодействующие с ней.

Варианты использования могут быть связаны друг с другом тремя видами связей: **обобщением (generalization)**, **расширением (extend relationship)** и **включением (include relationship)**. Действующие лица также могут быть связаны друг с другом с помощью связей **обобщения (generalization)**.

Контрольные вопросы по теме №4:

1. Что определяет Требования к ПО?
2. Какую работу называют анализом предметной области или бизнес-моделированием?
3. Для чего применяется схема Захмана или архитектурная схема предприятия (enterprise architecture framework)?
4. Где используются диаграммы потоков данных (data flow diagrams)?
5. Где используются диаграммы сущностей и связей (entity-relationship diagrams, ER diagrams)?
6. Что описывает стандарт IEEE 830-1998 Recommended Practice for Software Requirements Specifications (рекомендуемые методы спецификации требований к ПО)?

7. Что описывает стандарт IEEE 1233-1998, 2002 Guide for Developing System Requirements Specifications (руководство по разработке спецификаций требований к системам)?

Лекция 5.

Тема: Качество ПО и методы его контроля

Цель занятия: Рассматривается понятие качества ПО, характеристики и атрибуты качества, связь атрибутов качества с требованиями. Дается краткий обзор различных методов контроля качества ПО, с более детальным рассмотрением тестирования и проверки свойств на моделях.

Качество программного обеспечения

Как проверить, что требования определены достаточно полно и описывают все, что ожидается от будущей программной системы? Это можно сделать, проследив, все ли необходимые аспекты качества ПО отражены в них. Именно понятие качественного ПО соответствует представлению о том, что программа достаточно успешно справляется со всеми возложенными на нее задачами и не приносит проблем ни конечным пользователям, ни их начальству, ни службе поддержки, ни специалистам по продажам. Да и самим разработчикам создание качественной программы приносит гораздо больше удовольствия.

Качество ПО может быть описано большим набором разнородных характеристик. Такой подход к описанию сложных понятий называется **холистическим** (от греческого слова ????, целое). Он не дает единой концептуальной основы для рассмотрения затрагиваемых вопросов, какую дает целостная система представлений (например, Ньтоновская механика в физике или классическая теория вычислимости на основе машин Тьюринга), но позволяет, по крайней мере, не упустить ничего существенного.

Качество программного обеспечения определяется в стандарте ISO 9126 [1] как вся совокупность его характеристик, относящихся к возможности удовлетворять высказанные или подразумеваемые потребности всех заинтересованных лиц.

Тот же стандарт ISO 9126 [1,2,3,4] дает следующее представление качества.

Различаются понятия **внутреннего качества**, связанного с характеристиками ПО самого по себе, без учета его поведения; **внешнего качества,** характеризующего ПО с точки зрения его поведения; и качества ПО **при использовании** в различных контекстах — того качества, которое ощущается пользователями при конкретных сценариях работы ПО. Для всех этих аспектов качества введены метрики, позволяющие оценить их.

Общие принципы обеспечения качества процессов производства во всех отраслях экономики регулируются набором стандартов ISO 9000. Наиболее важные для разработки ПО стандарты в его составе следующие:

ISO 9000:2000 Quality management systems — Fundamentals and vocabulary [5].

Системы управления качеством — Основы и словарь. (Аналог — ГОСТ Р-2001).

ISO 9001:2000 Quality management systems — Requirements. Models for quality assurance in design, development, production, installation, and servicing [6].

Системы управления качеством — Требования. Модели для обеспечения качества при проектировании, разработке, коммерциализации, установке и обслуживании.

Характеристики и атрибуты качества ПО позволяют систематически описывать требования к нему, определяя, какие свойства ПО по данной характеристике хотят видеть заинтересованные стороны.

Методы контроля качества

Как контролировать качество системы? Как точно узнать, что программа делает именно то, что нужно, и ничего другого? Как определить, что она достаточно надежна, переносима, удобна в использовании? Ответы на эти вопросы можно получить с помощью процессов верификации и валидации.

Верификация обозначает проверку того, что ПО разработано в соответствии со всеми требованиями к нему, или что результаты очередного этапа разработки соответствуют ограничениям, сформулированным на предшествующих этапах.

Валидация — это проверка того, что сам продукт правилен, т.е. подтверждение того, что он действительно удовлетворяет потребностям и ожиданиям пользователей, заказчиков и других заинтересованных сторон.

Эффективность верификации и валидации, как и эффективность разработки ПО в целом, зависит от полноты и корректности формулировки требований к программному продукту.

Основой любой системы обеспечения качества являются методы его обеспечения и контроля. **Методы обеспечения качества** [9] представляют собой техники, гарантирующие достижение определенных показателей качества при их применении. Мы будем рассматривать подобные методы на протяжении всего курса.

Методы контроля качества позволяют убедиться, что определенные характеристики качества ПО достигнуты. Сами по себе они не могут помочь их достижению, они лишь помогают определить, удалось ли получить в результате то, что хотелось, или нет, а также найти ошибки, дефекты и отклонения от требований. Методы контроля качества ПО можно классифицировать следующим образом:

Методы и техники, связанные с выяснением свойств ПО во время его работы.

Это, прежде всего, все виды тестирования, а также **профилирование** и измерение количественных показателей качества, которые можно определить по результатам работы ПО — эффективности по времени и другим ресурсам, надежности, доступности и пр.

Методы и техники определения показателей качества на основе симуляции работы ПО с помощью моделей разного рода.

К этому виду относятся **проверка на моделях (model checking)**, а также **прототипирование (макетирование)**, используемое для оценки качества принимаемых решений.

Методы и техники, нацеленные на выявление нарушений формализованных правил построения исходного кода ПО, проектных моделей и документации.

К методам такого рода относится **инспектирование кода**, заключающееся в целенаправленном поиске определенных дефектов и нарушений требований в коде на основе набора шаблонов, автоматизированные методы поиска ошибок в коде, не основанные на его выполнении, методы проверки документации на согласованность и соответствие стандартам.

Методы и техники обычного или формализованного анализа проектной документации и исходного кода для выявления их свойств.

К этой группе относятся многочисленные **методы анализа архитектуры ПО**, о которых пойдет речь в следующей лекции, методы формального доказательства свойств ПО и формального анализа эффективности применяемых алгоритмов.

Тестирование — это проверка соответствия ПО требованиям, осуществляемая с помощью наблюдения за его работой в специальных, искусственно построенных ситуациях. Такого рода ситуации называют тестовыми или просто **тестами.**

Тестирование — конечная процедура. Набор ситуаций, в которых будет проверяться тестируемое ПО, всегда конечен. Более того, он должен быть настолько мал, чтобы тестирование можно было провести в рамках проекта разработки ПО, не слишком увеличивая его бюджет и сроки. Это означает, что при тестировании всегда проверяется очень небольшая доля всех возможных ситуаций. По этому поводу Дейкстра (Dijkstra) заметил, что тестирование позволяет точно определить, что в программе есть ошибка, но не позволяет утверждать, что там нет ошибок.

Тем не менее, тестирование может использоваться для достаточно уверенного вынесения оценок о качестве ПО. Для этого необходимо иметь **критерии полноты** тестирования, описывающие важность различных ситуаций для оценки качества, а также эквивалентности и зависимости между ними. Этот критерий может утверждать, что все равно в какой из ситуаций, A или B, проверять правильность работы ПО, или, если программа правильно работает в ситуации A, то, скорее всего, в ситуации B все тоже будет правильно. Часто критерий полноты тестирования задается при помощи определения эквивалентности ситуаций, дающей конечный набор классов ситуаций. В этом случае считают, что все равно, какую из ситуаций одного класса использовать в качестве теста. Такой критерий называют **критерием тестового покрытия**, а процент классов эквивалентности ситуаций, случившихся во вр емя тестирования, — достигнутым **тестовым покрытием.**

Таким образом, основные задачи тестирования: построить такой набор ситуаций, который был бы достаточно представителен и позволял бы завершить тестирование с достаточной степенью уверенности в правильности

ПО вообще, и убедиться, что в конкретной ситуации ПО работает правильно, в соответствии с требованиями.

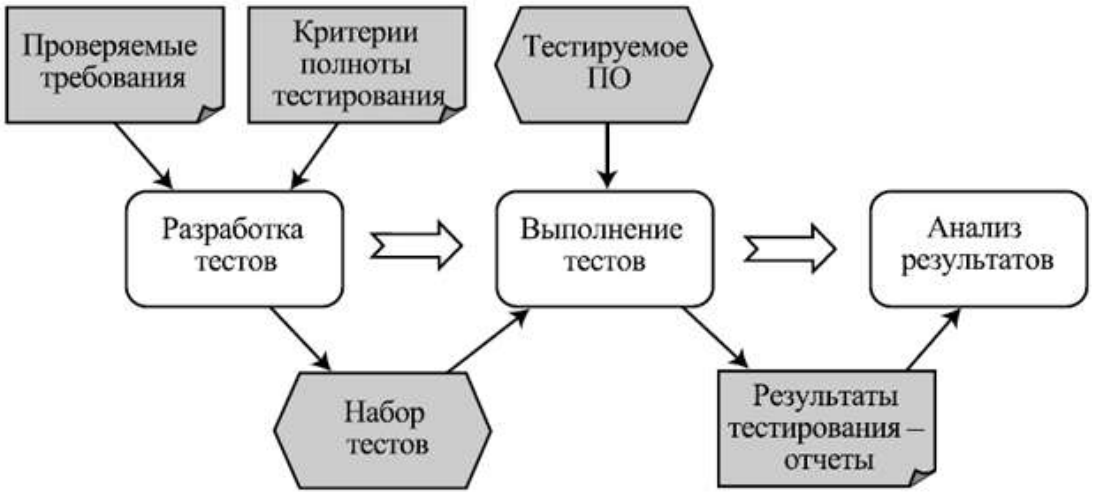


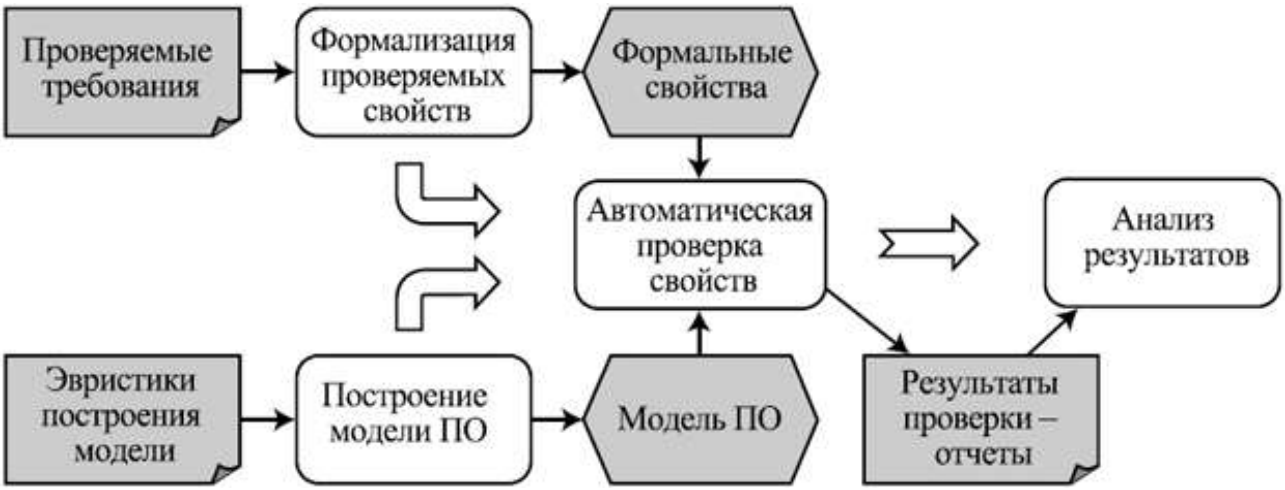
Рис.3. Схема процесса тестирования

Тестирование — наиболее широко применяемый метод контроля качества. Для оценки многих атрибутов качества не существует других эффективных способов, кроме тестирования.
Описывает требования к процедурам тестирования программных систем.

Тестировать можно соблюдение любых требований, соответствие которым выявляется во время работы ПО. Из характеристик качества по ISO 9126 этим свойством не обладают только атрибуты удобства сопровождения. Поэтому выделяют виды тестирования, связанные с проверкой определенных характеристик и атрибутов качества — тестирование функциональности, надежности, удобства использования, переносимости и производительности, а также тестирование защищенности, функциональной пригодности и пр. Кроме того, особо выделяют **нагрузочное** или **стрессовое** тестирование, проверяющее работоспособность ПО и показатели его производительности в условиях повышенных нагрузок — при большом количестве пользователей, интенсивном обмене данными с другими системами, большом объеме передаваемых или используемых данных и пр.

Проверка на моделях

Проверка свойств на моделях (model checking) [10] — проверка соответствия ПО требованиям при помощи формализации проверяемых свойств, построения формальных моделей проверяемого ПО (чаще всего в виде автоматов различных видов) и автоматической проверки выполнения этих свойств на построенных моделях. Проверка свойств на моделях позволяет проверять достаточно сложные свойства автоматически, при минимальном участии человека. Однако она оставляет открытым вопрос о том, насколько выявленные свойства модели можно переносить на само ПО.



http://www.intuit.ru/department/se/compprog/5/5_4.gif

Рис. 4. Схема процесса проверки свойств ПО на моделях

Обычно при помощи проверки свойств на моделях анализируют два вида свойств алгоритмов, использованных при построении ПО. **Свойства безопасности (safety properties)** утверждают, что нечто нежелательное никогда не случится в ходе работы ПО. **Свойства живучести (liveness properties)** утверждают, наоборот, что нечто желательное при любом развитии событий произойдет в ходе его работы.

Примером свойства первого типа служит отсутствие **взаимных блокировок (deadlocks)**. Взаимная блокировка возникает, если каждый из группы параллельно работающих в проверяемом ПО процессов или потоков ожидает прибытия данных или снятия блокировки ресурса от одного из других, а тот не может продолжить выполнение, ожидая того же от первого или от третьего процесса, и т.д.

Примером свойства живости служит гарантированная доставка сообщения, обеспечиваемая некоторыми протоколами — как бы ни развивались события, если сетевое соединение между машинами будет работать, посланное с одной стороны (процессом на первой машине) сообщение будет доставлено другой стороне (процессу на второй машине).

В классическом подходе к проверке на моделях проверяемые свойства формализуются в виде формул так называемых **временных логик**. Их общей чертой является наличие операторов "всегда в будущем" и "когда-то в будущем". Заметим, что второй оператор может быть выражен с помощью первого и отрицания — то, что некоторое свойство когда-то будет выполнено, эквивалентно тому, что отрицание этого свойства не будет выполнено всегда. Свойства безопасности легко записываются в виде "всегда будет выполнено отрицание нежелательного свойства", а свойства живости — в виде "когда-то обязательно будет выполнено желаемое".

Контрольные вопросы по теме №5:

1. Что такое качественное ПО?
2. Перечислите основные аспекты качества ПО по ISO 9126?

3. Перечислите характеристики и атрибуты качества ПО по ISO 9126?
4. Что такое Верификация?
5. Что такое Валидация?
6. Как проводится тестирования, т.е. проверка соответствия ПО требованиям?
7. Что такое Проверка свойств на моделях (model checking)?

Лекция 6.

Тема: Архитектура программного обеспечения

Цель занятия:Рассматривается понятие архитектуры ПО, влияние архитектуры на свойства ПО, а также методы оценки архитектуры. Рассказывается об основных элементах унифицированного языка моделирования UML.

Анализ области решений

Имеющийся опыт разработчиков сразу подсказывает, как можно решать поставленные задачи. Однако иногда, все-таки, возникает потребность сначала понять, как это можно сделать и, вообще, возможно ли это сделать и при каких ограничениях.

Таким образом, явно или неявно, проводится **анализ области решений.** Целью этой деятельности является понимание, можно ли вообще решить стоящие перед разрабатываемой системой задачи, при каких условиях и ограничениях это можно сделать, как они решаются, если решение есть, а если нет — нельзя ли придумать способ его найти или получить хотя бы приблизительное решение, и т.п. Обычно задача хорошо исследована в рамках какой-либо области человеческих знаний, но иногда приходится потратить некоторые усилия на выработку собственных решений. Кроме того, решений обычно несколько и они различаются по некоторым характеристикам, способным впоследствии сыграть важную роль в процессе развития и эксплуатации созданной на их основе программы. Поэтому важно взвесить их плюсы и минусы и определить, какие из них наиболее подходят в рамках данного проекта, или решить, что все они должны использоваться для обеспечения большей гибкости ПО.

Когда определены принципиальные способы решения всех поставленных задач (быть может, в каких-то ограничениях), основной проблемой становится способ организации программной системы, который позволил бы реализовать все эти решения и при этом удовлетворить требованиям, касающимся нефункциональных аспектов разрабатываемой программы. Искомый способ организации ПО в виде системы взаимодействующих компонентов называют **архитектурой**, а процесс ее создания — **проектированием** архитектуры ПО.

Архитектура программного обеспечения

Под архитектурой ПО понимают набор внутренних структур ПО, которые видны с различных точек зрения и состоят из компонентов, их связей и возможных взаимодействий между компонентами, а также доступных извне свойств этих компонентов [1].

Под компонентом в этом определении имеется в виду достаточно произвольный структурный элемент ПО, который можно выделить, определив интерфейс взаимодействия между этим компонентом и всем, что его окружает. Обычно при разработке ПО термин "компонент" (см. далее при обсуждении компонентных технологий) имеет несколько другой, более узкий смысл — это единица развертывания, самая маленькая часть системы, которую можно включить или не включить в ее состав. Такой компонент также имеет определенный интерфейс и удовлетворяет некоторому набору правил, называемому компонентной моделью. Там, где возможны недоразумения, будет указано, в каком смысле употребляется этот термин. В этой лекции до обсуждения UML мы будем использовать преимущественно широкое понимание этого термина, а в дальше — наоборот, узкое.

В определении архитектуры упоминается набор структур, а не одна структура. Это означает, что в качестве различных аспектов архитектуры, различных взглядов на нее выделяются различные структуры, соответствующие разным аспектам взаимодействия компонентов. Примеры таких аспектов — описание типов компонентов и типов статических связей между ними при помощи диаграмм классов, описание композиции компонентов при помощи структур ссылающихся друг на друга объектов, описание поведения компонентов при помощи моделирования их как набора взаимодействующих, передающих друг другу некоторые события, конечных автоматов.

Архитектура программной системы похожа на набор карт некоторой территории. Карты имеют разные масштабы, на них показаны разные элементы (административно-политическое деление, рельеф и тип местности — лес, степь, пустыня, болота и пр., экономическая деятельность и связи), но они объединяются тем, что все представленные на них сведения соотносятся с географическим положением. Точно так же архитектура ПО представляет собой набор структур или представлений, имеющих различные уровни абстракции и показывающих разные аспекты (структуру классов ПО, структуру развертывания, т.е. привязки компонентов ПО к физическим машинам, возможные сценарии взаимодействий компонентов и пр.), объединяемых сопоставлением всех представленных данных со структурными элементами ПО. При этом уровень абстракции данного представления является аналогом масштаба географической карты.

Стандарт IEEE 1471 отмечает необходимость использования архитектуры системы для решения таких задач, как следующие:

- Анализ альтернативных проектов системы.
- Планирование перепроектирования системы, внесения изменений в ее организацию.
- Общение по поводу системы между различными организациями, вовлеченными в ее разработку, эксплуатацию, сопровождение, приобретающими систему или продающими ее.
- Выработка критериев приемки системы при ее сдаче в эксплуатацию.

- Разработка документации по ее использованию и сопровождению, включая обучающие и маркетинговые материалы.
- Проектирование и разработка отдельных элементов системы.
- Сопровождение, эксплуатация, управление конфигурациями и внесение изменений и поправок.
- Планирование бюджета и использования других ресурсов в проектах, связанных с разработкой, сопровождением или эксплуатацией системы.
- Проведение обзоров, анализ и оценка качества системы.
- Разработка и оценка архитектуры на основе сценариев

При проектировании архитектуры системы на основе требований, зафиксированных в виде вариантов использования, первые возможные шаги состоят в следующем.

- Выделение компонентов
- Выбирается набор "основных" сценариев использования — наиболее существенных и выполняемых чаще других.
- Исходя из опыта проектировщиков, выбранного архитектурного стиля (см. следующую лекцию) и требований к переносимости и удобству сопровождения системы определяются компоненты, отвечающие за определенные действия в рамках этих сценариев, т.е. за решение определенных подзадач.
- Каждый сценарий использования системы представляется в виде последовательности обмена сообщениями между полученными компонентами.

При возникновении дополнительных хорошо выделенных подзадач добавляются новые компоненты, и сценарии уточняются.

Определение интерфейсов компонентов

Для каждого компонента в результате выделяется его интерфейс — набор сообщений, которые он принимает от других компонентов и посылает им.

Рассматриваются "неосновные" сценарии, которые так же разбиваются на последовательности обмена сообщениями с использованием, по возможности, уже определенных интерфейсов.

Если интерфейсы недостаточны, они расширяются.

Если интерфейс компонента слишком велик, или компонент отвечает за слишком многое, он разбивается на более мелкие.

Уточнение набора компонентов

Там, где это необходимо в силу требований эффективности или удобства сопровождения, несколько компонентов могут быть объединены в один.

Там, где это необходимо для удобства сопровождения или надежности, один компонент может быть разделен на несколько.

Достижение нужных свойств.

Все это делается до тех пор, пока не выполнятся следующие условия:

Все сценарии использования реализуются в виде последовательностей обмена сообщениями между компонентами в рамках их интерфейсов.

Набор компонентов достаточен для обеспечения всей нужной функциональности, удобен для сопровождения или портирования на другие платформы и не вызывает заметных проблем производительности.

Каждый компонент имеет небольшой и четко очерченный круг решаемых задач и строго определенный, сбалансированный по размеру интерфейс.

На основе возможных сценариев использования или модификации системы возможен также анализ характеристик архитектуры и оценка ее пригодности для поставленных задач или сравнительный анализ нескольких архитектур. Это так называемый метод анализа архитектуры ПО (Software Architecture Analysis Method, SAAM) [1,4].

Для представления архитектуры, а, точнее, различных входящих в нее структур, удобно использовать графические языки. На настоящий момент наиболее проработанным и наиболее широко используемым из них является **унифицированный язык моделирования (Unified Modeling Language, UML)** [5,6,7], хотя достаточно часто архитектуру системы описывают просто набором именованных прямоугольников, соединенных линиями и стрелками, которые представляют возможные связи.

UML предлагает использовать для описания архитектуры 8 видов диаграмм. 9-й вид UML диаграмм, диаграммы вариантов использования (см. лекцию 4), не относится к архитектурным представлениям. Кроме того, и другие виды диаграмм можно использовать для описания внутренней структуры компонентов или сценариев действий пользователей и прочих элементов, к архитектуре часто не относящихся. В этом курсе мы не будем разбирать диаграммы UML в деталях, а ограничимся обзором их основных элементов, необходимым для общего понимания смысла того, что изображено на таких диаграммах.

Диаграммы UML делятся на две группы — **статические** и **динамические диаграммы.**

Статические диаграммы

Статические диаграммы представляют либо постоянно присутствующие в системе сущности и связи между ними, либо суммарную информацию о сущностях и связях, либо сущности и связи, существующие в какой-то определенный момент времени. Они не показывают способов поведения этих сущностей. К этому типу относятся диаграммы классов, **объектов, компонентов** и диаграммы развертывания.

Диаграммы классов (class diagrams) показывают **классы** или **типы** сущностей системы, характеристики классов (**поля** и **операции**) и возможные связи между ними.

Классы представляются прямоугольниками, поделенными на три части. В верхней части показывают имя класса, в средней — набор его полей, с именами, типами, модификаторами доступа (public ‘+’, protected ‘#’, private ‘-’) и начальными значениями, в нижней — набор операций класса. Для каждой операции показывается ее модификатор доступа и сигнатура.

Наиболее часто используется три вида связей между классами — связи по композиции, ссылки, связи по наследованию и реализации.

Композиция описывает ситуацию, в которой объекты класса A включают в себя объекты класса B, причем последние не могут разделяться (объект класса B, являющийся частью объекта класса A, не может являться частью другого объекта класса A) и существуют только в рамках объемлющих объектов (уничтожаются при уничтожении объемлющего объекта).

Ссылочная связь (или **слабая агрегация**) обозначает, что объект некоторого класса A имеет в качестве поля ссылку на объект другого (или того же самого) класса B, причем ссылки на один и тот же объект класса B могут иметься в нескольких объектах класса A.

И композиция, и ссылочная связь изображаются стрелками, ведущими от класса A к классу B. Композиция дополнительно имеет закрашенный ромбик у начала этой стрелки. Двусторонние ссылочные связи, обозначающие, что объекты могут иметь ссылки друг на друга, показываются линиями без стрелок. Такая связь показана на рис. 6.5 между классами Account и Client.

Эти связи могут иметь описание **множественности**, показывающее, сколько объектов класса B может быть связано с одним объектом класса A. Оно изображается в виде текстовой метки около конца стрелки, содержащей точное число или нижние и верхние границы, причем бесконечность изображается звездочкой или буквой n. Для двусторонних связей множественности могут показываться с обеих сторон. На рис. 6.5 множественности, изображенные для связи между классами Account и Client, обозначают, что один клиент может иметь много счетов, а может и не иметь ни одного, и счет всегда привязан ровно к одному клиенту.

Диаграммы состояний используются часто, хотя требуется довольно много усилий, чтобы разработать их с достаточной степенью детальности.

Контрольные вопросы по теме №6:

1. Что такое анализ области решений?
2. Что понимают под архитектурой ПО?
3. Что понимают под компонентом?
4. Что представляют собой Статические диаграммы?
5. Что представляют собой Динамические диаграммы?
6. Перечислите Статические диаграммы?
7. Перечислите Динамические диаграммы?

Лекция 7.

Тема: Образцы проектирования

Цель занятия: Рассматривается понятие образца проектирования, классификация образцов проектирования и некоторые широко используемые примеры образцов анализа и архитектурных стилей.

Образцы человеческой деятельности

Чем отличается работа опытного проектировщика программного обеспечения от работы новичка? Имеющийся у эксперта опыт позволяет ему аккуратнее определять задачи, которые необходимо решить, точнее выделять среди них наиболее важные и менее значимые, четче представлять

ограничения, в рамках которых должна работать будущая система. Но важнее всего то, что эксперт отличается накопленными знаниями о приемлемых или неприемлемых в определенных ситуациях решениях, о свойствах программных систем, обеспечиваемых ими, и способностью быстро подготовить качественное решение сложной проблемы, опираясь на эти знания.

Давней мечтой преподавателей всех дисциплин является выделение таких знаний "в чистом виде" и эффективная передача их следующим поколениям специалистов. В области проектирования сложных систем на роль такого представления накопленного опыта во второй половине XX века стали претендовать образцы проектирования (**design patterns** или просто **patterns**), называемые также типовыми решениями или шаблонами. Наиболее широко образцы применяются при построении сложных систем, на которые накладывается множество разнообразных требований. Одной из первых работ, которая систематически излагает довольно большой набор образцов, относящихся именно к разработке программ, стала книга [1].

На основе имеющегося опыта исследователями и практиками разработки ПО выделено множество образцов — типовых архитектур, проектных решений для отдельных подсистем и модулей или просто программистских приемов, — позволяющих получить достаточно качественные решения типовых задач, а не изобретать каждый раз велосипед.

Более того, люди, наиболее активно вовлеченные в поиск образцов проектирования в середине 90-х годов прошлого века, пытались создать основанные на образцах языки, которые, хотя и были бы специфичными для определенных предметных областей, имели бы более высокий уровень абстракции, чем обычные языки программирования. Предполагалось, что человек, знакомый с таким языком, практически без усилий сможет создавать приложения в данной предметной области, компонуя подходящие образцы нужным способом. Эту программу реализовать так и не удалось, однако выявленные образцы, несомненно, являются одним из самых значимых средств передачи опыта проектирования сложных программных систем.

Образец (pattern) представляет собой шаблон решения типовой, достаточно часто встречающейся задачи в некотором контексте, т.е. при некоторых ограничениях на ожидаемые решения и определенном наборе требований к ним.

Образец проектирования нельзя выдумать или изобрести. Некоторый шаблон решения можно считать кандидатом в образцы проектирования, если он неоднократно применялся для решения одной и той же задачи на практике, если решения на его основе использовались в нескольких (как минимум, трех) случаях, в различных системах.

Образцы проектирования часто сильно связаны друг с другом в силу того, что они решают смежные задачи. Поэтому часто наборы связанных, поддерживающих друг друга образцов представляются вместе в виде **систем образцов (pattern system)** или **языка образцов (pattern language)**, в которых указаны возникающие между ними связи и описываются ситуации, в которых полезно совместное использование нескольких образцов:

По типу решаемых задач выделяют следующие разновидности образцов.

Образцы анализа **(analysis patterns).**

Они представляют собой типовые решения при моделировании сложных взаимоотношений между понятиями некоторой предметной области. Обычно они являются представлением этих понятий и отношений между ними с помощью набора классов и их связей, подходящего для любого объектно-ориентированного языка. Такие представления обладают важными атрибутами качественных модельных решений — способностью отображать понятным образом большое многообразие ситуаций, возникающих в реальной жизни, отсутствием необходимости вносить изменения в модель при небольших изменениях в требованиях к основанному на ней программному обеспечению и удобством внесения изменений, вызванных естественными изменениями в понимании моделируемых понятий. В частности, небольшое расширение данных, связанных с некоторым понятием, приводит к небольшим изменениям в структуре, чаще всего, лишь одного класса модели.

Образцы анализа могут относиться к определенной предметной области, как следует из их определения, но могут также и с успехом быть использованы для моделирования понятий в разных предметных областях.

Такие образцы представляют собой типовые способы организации системы в целом или крупных подсистем, задающие некоторые правила выделения компонентов и реализации взаимодействий между ними.

Образцы проектирования **(design patterns) в узком смысле.**

Они определяют типовые проектные решения для часто встречающихся задач среднего уровня, касающиеся структуры одной подсистемы или организации взаимодействия двух-трех компонентов.

Идиомы **(idioms, programming patterns).**

Идиомы являются специфическими для некоторого языка программирования способами организации элементов программного кода, позволяющими, опять же, решить некоторую часто встречающуюся задачу.

Образцы организации **(organizational patterns) и** образцы процессов **(process patterns).**

Образцы этого типа описывают успешные практики организации разработки ПО или другой сложной деятельности, позволяющие решать определенные задачи в рамках некоторого контекста, который включает ограничения на возможные решения.

Для описания образцов были выработаны определенные шаблоны. Далее мы будем использовать один из таких шаблонов для описания архитектурных стилей, образцов проектирования и идиом. Этот шаблон включает в себя следующие элементы:

- Название образца, а также другие имена, под которыми этот образец используется.
- Назначение — задачи, которые решаются с помощью данного образца. В этот же пункт включается описание контекста, в котором данный образец может быть использован.

- Действующие силы — ограничения, требования и идеи, под воздействием которых вырабатывается решение.
- Решение — основные идеи используемого решения. Включает следующие подпункты:
- Структура — структура компонентов, принимающих участие в данном образце, и связей между ними. В рамках образца компоненты принято именовать, исходя из ролей, которые они в нем играют.
- Динамика — основные сценарии совместной работы компонентов образца.
- Реализация — возможные проблемы при реализации и способы их преодоления, примеры кода на различных языках (в данном курсе мы будем использовать для примеров только язык Java). Варианты и способы уточнения данного образца.
- Следствия применения образца — какими дополнительными свойствами, достоинствами и недостатками обладают полученные на его основе решения.
- Известные примеры использования данного образца.
- Другие образцы, связанные с данным.

Архитектурные стили

Архитектурный стиль определяет основные правила выделения компонентов и организации взаимодействия между ними в рамках системы или подсистемы в целом. Различные архитектурные стили подходят для решения различных задач в плане обеспечения нефункциональных требований — различных уровней производительности, удобства использования, переносимости и удобства сопровождения. Одну и ту же функциональность можно реализовать, используя разные стили.

Работа по выделению и классификации архитектурных стилей была проведена в середине 1990-х годов. Ее результаты представлены в работах [5,6]. Ниже приведена таблица некоторых архитектурных стилей, выделенных в этих работах.

Таблица 7.1. Некоторые архитектурные стили

Виды стилей и конкретные стили	**Контекст использования и основные решения**	**Примеры**
Конвейер обработки данных (data flow)	Система выдает четко определенные выходные данные в результате обработки четко определенных входных данных, при этом процесс обработки не зависит от времени, применяется многократно, одинаково к любым данным на входе. Обработка организуется в виде набора (необязательно последовательности) отдельных компонентов-обработчиков, передающих свои результаты на вход другим обработчикам или на выход всей системы. Важными свойствами являются четко	

		определенная структура данных и возможность интеграции с другими системами	
	Пакетная обработка (batch sequential)	Один-единственный вывод производится на основе чтения некоторого одного набора данных на входе, промежуточные преобразования организуются в виде последовательности	Сборка программной системы: компиляция, сборка системы, сборка документации, выполнение тестов
	Каналы и фильтры (pipe-and-filter)	Нужно обеспечить преобразование непрерывных потоков данных. При этом преобразования инкрементальны и следующее может быть начато до окончания предыдущего. Имеется, возможно, несколько входов и несколько выходов. В дальнейшем возможно добавление дополнительных преобразований	Утилиты UNIX
	Замкнутый цикл управления (closed-loop control)	Нужно обеспечить обработку постоянно поступающих событий в плохо предсказуемом окружении. Используется общий диспетчер событий, который классифицирует событие и отдает его на асинхронную обработку обработчику событий такого типа, после чего диспетчер снова готов воспринимать события	Встроенные системы управления в автомобилях, авиации, спутниках. Обработка запросов на сильно загруженных Web-серверах. Обработка действий пользователя в GUI
Вызов-возврат (call-return)		Порядок выполнения действий четко определен, отдельные компоненты не могут выполнять полезную работу, не получая обращения от других	
	Процедурная декомпозиция	Данные неизменны, процедуры работы с ними могут немного меняться, могут возникать новые. Выделяется набор процедур, схема передачи управления между которыми представляет собой дерево с основной процедурой в его корне	Основная схема построения программ для языков C, Pascal, Ada
	Абстрактные типы данных (abstract data types)	В системе много данных, структура которых может меняться. Важны возможности внесения изменений и интеграции с другими системами. Выделяется набор абстрактных типов данных, каждый из которых предоставляет набор операций для работы с данными такого типа. Внутреннее представление данных скрывается	Библиотеки классов и компонентов
	Многоуровне-вая	Имеется естественное расслоение	Телекоммуникацион-ные

	система (layers)	задач системы на наборы задач, которые можно было бы решать последовательно — сначала задачи первого уровня, затем, используя полученные решения, — второго, и т.д. Важны переносимость и возможность многократного использования отдельных компонентов. Компоненты разделяются на несколько уровней таким образом, что компоненты данного уровня могут использовать для своей работы только соседей или компоненты предыдущего уровня. Могут быть более слабые ограничения, например, компонентам верхних уровней разрешено использовать компоненты всех нижележащих уровней	протоколы в модели OSI (7 уровней), реальные протоколы сетей передачи данных (обычно 5 уровней или меньше). Системы автоматизации предприятий (уровни интерфейса пользователя–обработки запросов–хранения данных)
	Клиент-сервер	Решаемые задачи естественно распределяются между инициаторами и обработчиками запросов, возможно изменение внешнего представления данных и способов их обработки	Основная модель бизнес-приложений: клиентские приложения, воспринимающие запросы пользователей и сервера, выполняющие эти запросы
Интерактивные системы		Необходимость достаточно быстро реагировать на действия пользователя, изменчивость пользовательского интерфейса	
	Данные–представление–обработка (model–view-controller, MVC)	Изменения во внешнем представлении достаточно вероятны, одна и та же информация представляется по-разному в нескольких местах, система должна быстро реагировать на изменения данных. Выделяется набор компонентов, ответственных за хранение данных, компоненты, ответственные за их представления для пользователей, и компоненты, воспринимающие команды, преобразующие данные и обновляющие их представления	Наиболее часто используется при построении приложений с GUI. Document-View в MFC (Microsoft Foundation Classes) — документ в этой схеме объединяет роли данных и обработчика
	Представление–абстракция–управление (presentation–abstraction–control)	Интерактивная система на основе агентов, имеющих собственные состояния и пользовательский интерфейс, возможно добавление новых агентов. Отличие от предыдущей схемы в том, что для каждого отдельного набора данных его модель, представление и управляющий компонент объединяются в агента, ответственного	

		за всю работу именно с этим набором данных. Агенты взаимодействуют друг с другом только через четко определенную часть интерфейса управляющих компонентов	
Системы на основе хранилища данных		Основные функции системы связаны с хранением, обработкой и представлением больших количеств данных	
	Репозиторий (repository)	Порядок работы определяется только потоком внешних событий. Выделяется общее хранилище данных — репозиторий. Каждый обработчик запускается в ответ на соответствующее ему событие и как-то преобразует часть данных в репозитории	Среды разработки и CASE-системы
	Классная доска (blackboard)	Способ решения задачи в целом неизвестен или слишком трудоемок, но известны методы, частично решающие задачу, композиция которых способна выдавать приемлемые результаты, возможно добавление новых потребителей данных или обработчиков. Отдельные обработчики запускаются, только если данные репозитории для их работы подготовлены. Подготовленность данных определяется с помощью некоторой системы шаблонов. Если можно запустить несколько обработчиков, используется система их приоритетов	Системы распознавания текста

Многие из представленных стилей носят достаточно общий характер и часто встречаются в разных системах. Кроме того, часто можно обнаружить, что в одной системе используются несколько архитектурных стилей — в одной части преобладает один, в другой — другой, или же один стиль используется для выделения крупных подсистем, а другой — для организации более мелких компонентов в подсистемах.

Контрольные вопросы по теме №7:

1. Что такое Образец анализа?
2. Что понимают под Архитектурным стилем?
3. Что понимают под Многоуровневой системой?
4. Что представляют собой Образец (pattern)?
5. Что представляют собой Адаптер (adapter)?
6. Перечислите разновидности образцов проектирования?

7. Перечислите виды архитектурных стилей?

Лекция 8.

Тема: Принципы создания удобного пользовательского интерфейса

Цель занятия: Рассматриваются основные факторы удобства использования ПО, а также психо-физиологические особенности человека, делающие предметы удобными и неудобными для него. Рассказывается о методике проектирования, ориентированного на удобство использования.

Удобство использования программного обеспечения

Одним из важных показателей качества программного обеспечения является удобство его использования. Оно описывается с помощью таких характеристик, как понятность пользовательского интерфейса, легкость обучения работе с ним, трудоемкость решения определенных задач с его помощью, производительность работы пользователя с ПО, частота появления ошибок и жалоб на неудобства. Для построения действительно удобных программ нужен учет контекста их использования, психологии пользователей, необходимости помогать начинающим пользователям и предоставлять все нужное для работы опытных. Однако самым значимым фактором является то, помогает ли данная программа решать действительно значимые для пользователей задачи.

Многие программисты имеют технический или математический склад ума. Для таких людей "понятность", "легкость обучения" представляются весьма субъективными факторами. Сами они достаточно легко воспринимают сложные вещи, если те представлены в рамках непротиворечивой системы понятий, как бы дико эта система и входящие в нее понятия ни выглядели для постороннего наблюдателя. Такие люди чаще всего изучают новое ПО при помощи документации и искренне убеждены в том, что пользователи будут разбираться с написанной ими программой тем же способом. Типичный подход программиста при разработке пользовательского интерфейса — предоставить пользователю те же рычаги и кнопки, с помощью которых программист сам хотел бы управлять своей программой.

К пользователям, у которых возникли проблемы с программой, многие программисты достаточно суровы. Любимый их ответ в такой ситуации — "RTFM!" (read this fucking manual, прочти эту чертову инструкцию). Они любят рассказывать друг другу анекдоты о "ламерах", которые не в силах понять, что файлы нужно сохранять.

Посмотрим теперь, что будет с "обычным человеком", впервые попытавшимся воспользоваться компьютером вообще и текстовым редактором в частности (как ни тяжело представить такое сейчас).

Пользователь открывает редактор, скажем, Microsoft Word, как-то набирает текст, затем печатает его на принтере и выключает компьютер. Когда он включает компьютер в следующий раз и не находит важный документ, который он набрал (вы же сами помните!), он страшно раздосадован. Что вы говорите? Надо было сохранить документ? Что значит "сохранить"? Куда? Он же набрал документ и своими глазами видел, что "тот в компьютере есть".

Зачем его еще как-то "сохранять"? Ну ладно, ну хорошо, раз вы так уверяете, что нужно нажать эту кнопочку, он будет ее нажимать. Да-да, каждые 10 минут, специально для вас, он будет нажимать эту кнопочку (зачем она нужна?...). Конечно же, спустя некоторое время он забудет это сделать.

Человек "понимает" смысл и назначение вещей и действий с ними, только если они в его сознании находятся в рамках некоторой **системы связанных друг с другом понятий**. Набор текста на компьютере больше всего напоминает набор текста на печатной машинке (и специально сделан выглядящим так в редакторах WYSIWYG), чуть менее он близок к письму. В обоих этих случаях, написав или напечатав некоторый текст на листе бумаги, мы получим достаточно долго хранящийся документ. Чтобы избавиться от него, нужно предпринимать специальные действия — смять, порвать, пролить на него кофе, выкинуть в мусорную корзину. Если такой документ пропадает без наших действий, значит кто-то (сотрудник, начальник, жена, ребенок или уборщица) взял его. Человек, только что столкнувшийся с электронными документами, воспринимает их как аналоги бумажных и ожидает от них тех же свойств.

Документы "в компьютере" не такие. У компьютера есть два вида памяти — оперативная, или временная, и постоянная. В большинстве случаев набранный в редакторе текст находится в оперативной памяти, содержимое которой пропадает при отключении питания. Чтобы текст смог "пережить" это отключение, он должен быть перемещен в постоянную память. Именно для этого служит операция "сохранить документ".

В предыдущем абзаце описана некоторая система понятий, непривычная для новичка и не доступная с помощью непосредственного созерцания компьютера и размышлений. Ее необходимо как-то передать новому пользователю, иначе он не сможет понять, зачем же сохранять уже написанные документы, ведь они и так есть. Иначе, максимум, что он сможет сделать — выучить **ритуал**, согласно которому нужно иногда нажимать на кнопку "Сохранить". Очень многие люди работают с компьютерами и другой сложной техникой с помощью ритуалов, поскольку не всегда в силах разобраться в новой для них системе понятий, в рамках которой действует эта техника. Но гораздо чаще — потому, что ее производитель и разработчики не тратят столько усилий, сколько нужно, чтобы научить этой системе каждого пользователя.

Если же подпускать пользователей к компьютеру только после прочтения необходимой документации и усвоения ее информации, редко кто из них вообще заинтересуется использованием компьютера. Они используют компьютер и программное обеспечение только как инструменты для решения своих задач (единственный вид программ, где можно хоть как-то рассчитывать на чтение документации, — игры и развлечения), и им не хочется тратить время на чтение инструкций и осмысление правил, не относящихся напрямую к их области деятельности. Почему бы этим программам не быть столь же наглядными, как молоток и отвертка — никто не станет же всерьез читать инструкцию к отвертке?

Можно поспорить с этим на том основании, что компьютер и ПО намного сложнее отвертки — с их помощью можно выполнять гораздо больше действий и сами действия гораздо сложнее. Но, с другой стороны, умеют же сейчас делать автомобили, для вождения которых нужно знать только правила движения и основные элементы управления, а если что-то идет не так — пусть разбираются автослесари. Автомобиль сравним по сложности с самыми сложными программами, а то и превосходит их. И многие водители (по крайней мере, на Западе), используют автомобили, ничего не понимая в их устройстве. Пользователи изначально не хотят читать инструкции и не будут этого делать, пока эти инструкции занимают сотни страниц, написаны непонятным и сухим языком, требуют внимательности и обдумывания, не отвечают сразу на вопросы, которые интересуют пользователя в данный момент, а также пока начальство не скажет, что инструкцию все-таки прочитать надо. Но ведь есть еще и естественная человеческая забывчивость...

Удобство обычной, "некомпьютерной" модели работы с документами подтверждается тем, что Palm Pilot, первый компьютер без разделения памяти на временную и постоянную, разошелся небывалым для такого устройства тиражом — за первые два года было продано около 2 миллионов экземпляров.

Все сказанное выше служит иллюстрацией того факта, что "простота" и "легкость обучения" все-таки не совсем субъективны, а имеют объективные составляющие, которые необходимо учитывать при разработке части программного обеспечения, предназначенной для непосредственного взаимодействия с человеком — пользовательского интерфейса. Если посмотреть внимательнее, непонимание программистами пользователей в большой степени вызвано их, программистов, собственной ленью и нежеланием задумываться над непривычными вещами.

На применяемые в этой области решения огромное влияние оказывают общие законы психологии и физиологии человека — ведь вещи удобны или неудобны большей частью не из-за субъективных предпочтений, а из-за того, что строение человеческого тела и законы работы сознания помогают или мешают использовать их эффективно.

Фундаментальной основой для определения удобств и неудобств понимания человеком функционирования и способов использования различных предметов является **когнитивная психология**, которая изучает любые познавательные процессы человеческого сознания. Психология использования машин, инструментов, оборудования и предметов обихода в ходе практической деятельности человека обычно называется **инженерной психологией** [1,2]. За рубежом выделена особая наука, изучающая психологические, физиологические и анатомические аспекты взаимодействия человека и компьютера, которая так и называется — **взаимодействие человека и компьютера (Human-Computer Interaction, HCI)**.

При рассмотрении задач построения удобного ПО используют много информации из перечисленных дисциплин. Наиболее важные для разработки пользовательского интерфейса результаты этих дисциплин можно сформулировать следующим образом.

Человеку свойственно ошибаться

Обычный человек в нормальном состоянии постоянно делает ошибки разного рода. Можно сказать, что человек, в отличие от компьютера, является адаптивной аналоговой системой и успешность его "функционирования" в гораздо большей степени определяется не точностью выполнения действий и формулировки мыслей, а способностью быстро выдать хорошее приближение к нужному результату и достаточно быстро поправиться, если это необходимо.

Один из принципов построения удобных систем — терпимость к человеческим ошибкам, умение не замечать их, "понимая" пользователя правильно, несмотря на его не вполне корректные действия, а также наличие возможностей полного устранения последствий совсем уж неверных действий. Многими специалистами по удобству использования достаточно серьезно воспринимается радикальный тезис, состоящий в том, что пользователи не ошибаются, а лишь выполняют "действия, не направленные на достижение своих собственных целей".

К тому же сообщения об ошибках, которыми программы пугают неопытных пользователей, являются чаще всего отвлекающим от работы фактором, приводят к раздражению, а иногда — к отказу от работы с программой, которая "слишком умничает". Именно так пользователь воспринимает указания программы, которая, явно не понимая, что же хочет человек, пытается заявлять о неверности его действий. Сообщений об ошибках в удобной программе практически не должно быть, а те, которые все-таки не удается убрать, ни в коем случае не должны формулироваться недостаточно информативно и категоричным тоном: "Неправильно! Некорректное значение!", как будто пользователь сдает экзамен. Человеку будет гораздо комфортнее, если система признается, что не может понять, что он хочет, объяснит, какие именно из введенных им данных вызывают проблемы и почему, а также предложит возможные варианты выхода из создавшейся ситуации. Но еще лучше — если ПО все понимает "правильно", даже если пользователь ошибся, и не обращает внимания на подобные мелочи.

С другой стороны, действия, связанные с большим риском и невозможностью отмены, не должны быть легко доступны — иначе простота доступа в сочетании с человеческими ошибками будет часто приводить к крайне нежелательным последствиям. В самих программах таких действий немного — лишь потеря большого количества ценных данных может претендовать на такой статус, но ПО может управлять и кораблем, и самолетом, и атомной электростанцией. В подобных случаях необходимо особо выделять действия, выполнение которых может привести к тяжким последствиям. Не стоит делать запуск ракеты сравнимым по простоте с удалением файла.

Внимание человека

Человеку очень тяжело долго сохранять сосредоточенность и не отвлекаться от какой-то деятельности. Чаще всего, помимо воли человека, через некоторое время ему в голову приходят разные мысли, и взгляд сам собой уползает куда-то в сторону.

Это означает, что нельзя требовать от человека длительного внимания к работе. Само понимание того, что необходимо сосредоточиться, создает у него стресс и вызывает негативные эмоции. Только в самых рискованных ситуациях, когда от его действий зависит очень многое — пике самолета, превышающий все нормы разогрев химического реактора — пользователь может сосредоточиться на значительное время, но все равно мы все предпочитаем избегать подобных обстоятельств. Такие ситуации должны явно отмечаться какими-то характерными и хорошо ощущаемыми сигналами (сирена, красный мигающий свет и пр.), и ни в коем случае эти же или похожие сигналы нельзя использовать для гораздо менее серьезных случаев, иначе пользователи будут сбиты с толку и могут их перепутать, что приводит к печальным последствиям.

В остальных же ситуациях ПО должно быть готово к постоянным переключениям внимания пользователя и ненавязчиво помогать ему восстановить фокус внимания на последних действиях, которые он выполнял, а также вспомнить их контекст — что он вообще делал, на каком шаге он сейчас находится, что еще нужно сделать и пр.

Поэтому, например, экранные формы, заполнение которых входит в одну процедуру, хорошо бы помечать так, чтобы пользователь сразу видел, сколько шагов он уже выполнил и сколько еще осталось. А текущее поле ввода стоит как-то выделять среди всех полей ввода на форме.

Для привлечения внимания пользователей к каким-то сообщениям или элементам управления нужно помнить правило: **сначала движение, затем яркий цвет, затем все остальное**. Внимание человека прежде всего акцентируется на движущихся объектах, потом — на выделенных цветом, и только потом — на остальных формах выделения. Лучшим способом привлечения внимания является появление анимации, чуть менее действенным — яркие цветные окна и сообщения. Для мягкого, не режущего глаз привлечения внимания можно использовать выделение при помощи цветов мягких оттенков или изменения шрифта сообщений.

Наоборот, появление на экране или страничке ненужных цветных анимированных картинок является лучшим способом отвлечь человека от выполняемой им работы: глаза поневоле переводятся на такую картинку, и требуется психологическое усилие и некоторое время, чтобы от нее оторваться. Некоторые люди даже закрывают анимацию рукой, чтобы она не мешала сосредоточиться на полезной информации на экране!

Ярко выделенных элементов на одном экране не должно быть много, не больше, чем один-два, иначе человек перестает обращать внимание на такое выделение.

Другой фактор, связанный с вниманием пользователей, — их оценка времени выполнения системой заданных действий. Если человек ожидает, что система будет печатать документ достаточно долго, то, послав его на печать, он пойдет налить себе чаю и вернется к тому времени, когда, по его оценке, система должна закончить работу. Поэтому, прежде чем выполнять долгие операции, нужно убедиться, что все необходимые данные от пользователя получены. Иначе и пользователь, и система будут терять время, первый —

ожидая на кухне, когда же система выполнит работу (которую он вроде бы запустил), а вторая — ожидая ввода дополнительных данных.

Контрольные вопросы по теме №8:

1. Что такое Инженерная психология?
2. Что понимают под Ментальной моделью?
3. Перечислите основные факторы, с помощью которых можно оценить или даже измерить удобство использования программы?
4. Перечислите правила, позволяющие оценивать удобство интерфейса?
5. Перечислите принципы позволяющие находить решения, повышающие удобство пользовательского интерфейса?
6. Основная идея метода проектирование, ориентированное на использование (usage-centered design), предложенное Л. Константайном и Л. Локвуд (L.Constantine, L.Lockwood)?
7. Как организуется Эвристическое инспектирование?

Лекция 9 - 10.

Тема: Основные конструкции языков Java и C#.

Цель занятия: Рассматриваются базовые элементы технологий Java и .NET и основные конструкции языков Java и C#. Рассказывается о лексике, базовых типах, выражениях и инструкциях обоих языков, а также о правилах описания пользовательских типов.

Платформы Java и .NET

На данный момент наиболее активно развиваются две конкурирующие линии технологий создания ПО на основе компонентов — технологии Java и .NET. В этой и следующих лекциях мы рассмотрим несколько элементов этих технологий, являющихся ключевыми в создании широко востребованного в настоящее время и достаточно сложного вида приложений. Это Web-приложения, т.е. распределенное программное обеспечение, использующее базовую инфраструктуру Интернета для связи между различными своими компонентами, а стандартные инструменты для навигации по Web — браузеры — как основу для своего пользовательского интерфейса.

Технологии Java представляют собой набор стандартов, инструментов и библиотек, предназначенных для разработки приложений разных типов и связанных друг с другом использованием языка программирования Java. Торговая марка Java принадлежит компании Sun Microsystems, и эта компания во многом определяет развитие технологий Java, но в нем активно участвуют и другие игроки — IBM, Intel, Oracle, Hewlett-Packard, SAP, Bea и пр.

В этот набор входят следующие основные элементы:

Платформа Java Platform Standard Edition (J2SE) - Предназначена для разработки обычных, в основном, однопользовательских приложений.

Платформа Java Platform Enterprise Edition (J2EE) - Предназначена для разработки распределенных Web-приложений уровня предприятия.

Платформа Java **Platform Micro Edition (J2ME) -** Предназначена для разработки встроенных приложений, работающих на ограниченных ресурсах, в

основном, в мобильных телефонах и компьютеризированных бытовых устройствах.

Платформа Java **Card -** Предназначена для разработки ПО, управляющего функционированием цифровых карт. Ресурсы, имеющиеся в распоряжении такого ПО, ограничены в наибольшей степени.

С некоторыми оговорками можно считать, что J2ME является подмножеством J2SE, а та, в свою очередь, подмножеством J2EE. Java Card представляет собой, по существу, особый набор средств разработки, связанный с остальными платформами только поддержкой (в сильно урезанном виде) языка Java.

Язык Java — это объектно-ориентированный язык программирования, который транслируется не непосредственно в машинно-зависимый код, а в так называемый байт-код, исполняемый специальным интерпретатором, **виртуальной** Java- **машиной (**Java **Virtual Machine, JVM)**. Такая организация работы Java-программ позволяет им быть переносимыми без изменений и одинаково работать на разных платформах, если на этих платформах есть реализация JVM, соответствующая опубликованным спецификациям виртуальной машины.

Кроме того, интерпретация кода позволяет реализовывать различные политики безопасности для одних и тех же приложений, выполняемых в разных средах, — к каким ресурсам (файлам, устройствам и пр.) приложение может иметь доступ, а к каким нет, можно определять при запуске виртуальной машины. Таким способом можно обеспечить запускаемое пользователем вручную приложение (за вред, причиненный которым, будет отвечать этот пользователь) большими правами, чем аплет, загруженный автоматически с какого-то сайта в Интернете.

Режим интерпретации приводит обычно к более низкой производительности программ по сравнению с программами, оттранслированными в машинно-специфический код. Для преодоления этой проблемы JVM может работать в режиме **динамической компиляции (just-in-time-compilation, JIT)**, в котором байт-код на лету компилируется в машинно-зависимый, а часто исполняемые участки кода подвергаются дополнительной оптимизации.

.NET представляет собой похожий набор стандартов, инструментов и библиотек, но разработка приложений в рамках .NET возможна с использованием различных языков программирования. Основой .NET являются виртуальная машина для **промежуточного языка** (**Intermediate Language, IL,** иногда встречается сокращение MSIL, Microsoft IL), в который транслируются все .NET программы, также называемая **общей средой выполнения (Common Language Runtime, CLR)**, и общая библиотека классов (.NET Framework class library), доступная из всех .NET приложений.

Промежуточный язык является полноценным языком программирования, но он не предназначен для использования людьми. Разработка в рамках .NET ведется на одном из языков, для которых имеется транслятор в промежуточный язык —Visual Basic.NET, C++, C#, Java (транслятор Java в .NET называется J#,

и он не обеспечивает одинаковой работы программ на Java, оттранслированных в .NET и выполняемых на JVM) и пр. Однако разные языки достаточно сильно отличаются друг от друга, и чтобы гарантировать возможность из одного языка работать с компонентами, написанными на другом языке, необходимо при разработке этих компонентов придерживаться **общих правил (Common Language Specifications, CLS)**, определяющих, какими конструкциями можно пользоваться во всех .NET языках без потери возможности взаимодействия между результатами. Наиболее близок к промежуточному языку C# — этот язык был специально разработан вместе с платформой .NET.

Некоторым отличием от Java является то, что код на промежуточном языке в .NET не интерпретируется, а всегда выполняется в режиме динамической компиляции (JIT).

Компания Microsoft инициировала разработку платформы .NET и принятие стандартов, описывающих ее отдельные элементы (к сожалению, пока не все), и она же является основным поставщиком реализаций этой платформы и инструментов разработки. Благодаря наличию стандартов возможна независимая реализация .NET (например, такая реализация разработана в рамках проекта Mono [6]), но, в силу молодости платформы и опасений по поводу монопольного влияния Microsoft на ее дальнейшее развитие, реализации .NET не от Microsoft используются достаточно редко.

Прежде чем перейти к более детальному рассмотрению компонентных технологий Java и .NET, ознакомимся с языками, на которых создаются компоненты в их рамках. Для Java-технологий базовым языком является Java, а при изучении правил построения компонентов для .NET мы будем использовать язык C#. Он наиболее удобен при работе в этой среде и наиболее похож на Java.

Данная лекция дает лишь базовую информацию о языках Java и C#, которая достаточна для понимания приводимого далее кода компонентов и общих правил, регламентирующих их разработку, и может служить основой для дальнейшего их изучения. Оба языка достаточно сложны, и всем их деталям просто невозможно уделить внимание в рамках двух лекций. Оба языка имеют мощные выразительные возможности объектно-ориентированных языков последнего поколения, поддерживающих автоматическое управление памятью и работу в многопоточном режиме. Они весьма похожи, но имеют большое число мелких отличий в деталях. Наиболее существенны для построения программ различия, касающиеся наличия в C# невиртуальных методов, возможности объявления и использования пользовательских типов значений и делегатных типов в C# и возможности передачи значений параметров в C# по ссылке. Обсуждение и одновременно сравнение характеристик языков мы будем проводить по следующему плану:

1. Лексика.
2. Общая структура программ.

Общие черты Java и C# описываются далее обычным текстом, а особенности — в колонках.

В левой колонке будут описываться особенности Java. В правой колонке будут описываться особенности C#.

Лексика

Программы на обоих рассматриваемых языках, C# и Java, могут быть написаны с использованием набора символов Unicode, каждый символ в котором представляется при помощи 16-ти бит. Поскольку последние версии стандарта Unicode [10] определяют более широкое множество символов, включая символы от U+10000 до U+10FFFF (т.е. имеющие коды от 216 до 220+216–1), такие символы представляются в кодировке UTF-16, т.е. двумя 16-битными символами, первый в интервале U+D800–U+DBFF, второй — в интервале U+DC00–U+DFFF.

Лексически программы состоят из разделителей строк (символы возврата каретки, перевода строки или их комбинация), комментариев, пустых символов (пробелы и табуляции), идентификаторов, ключевых слов, литералов, операторов и разделительных символов.

В обоих языках можно использовать как однострочный комментарий, начинающийся с символов // и продолжающийся до конца строки, так и выделительный, открывающийся символами /* и заканчивающийся при помощи */.

Идентификаторы должны начинаться с буквы (символа, который считается буквой в Unicode, или символа _) и продолжаться буквами или цифрами. В качестве символа идентификатора может использоваться последовательность \uxxxx, где x — символы 0-9, a-f или A-F, обозначающая символ Unicode с шестнадцатеричным кодом xxxx. Корректными идентификаторами являются, например, myIdentifier123, идентификатор765 (если последние два представлены в Unicode). Ключевые слова устроены также (без возможности использовать Unicode-последовательности в C#), но используются для построения деклараций, инструкций и выражений языка или для обозначения специальных констант.

В Java ключевые слова не могут использоваться в качестве идентификаторов.	Добавив в начало ключевого слова символ @, в C# можно получить идентификатор, посимвольно совпадающий с этим ключевым словом. Этот механизм используется для обращения к элементам библиотек .NET, написанным на других языках, в которых могут использоваться такие идентификаторы — @class, @int, @public и пр. Можно получать идентификаторы, добавляя @ и в начало идентификатора, но делать это не рекомендуется стандартом языка. Другой способ получить идентификатор, совпадающий по символам с ключевым словом, — использовать в нем Unicode-последовательность вместо соответствующего символа ASCII. Кроме того, в C# есть специальные идентификаторы, которые только в некотором контексте используются в качестве ключевых слов. Таковы add, alias, get, global, partial, remove, set, value, where, yield.

В обоих языках имеется литерал null для обозначения пустой ссылки на объект, булевские литералы true и false, символьные и строковые литералы, целочисленные литералы и литералы, представляющие числа с плавающей точкой.

Символьный литерал, обозначающий отдельный символ, представляется как этот символ, заключенный в одинарные кавычки (или апострофы). Так, например, можно представить символы 'a', '#', 'Ы'. Чтобы представить символы одинарной кавычки, обратного слэша и некоторые другие, используются так называемые ESC-последовательности, начинающиеся с обратного слэша — '\'' (одинарная кавычка), '\\' (обратный слэш), '\"' (обычная кавычка), '\n' (перевод строки), '\r' (возврат каретки), '\t' (табуляция). Внутри одинарных кавычек можно использовать и Unicode-последовательности, но осторожно — если попытаться представить так, например, символ перевода строки \u000a, то, поскольку такие последовательности заменяются соответствующими символами в самом начале лексич еского анализа, кавычки будут раздел ены переводом строки, что вызовет ошибку.

В Java можно строить символьные литералы в виде восьмеричных ESC-последовательностей из не более чем трех цифр — '\010', '\142', '\377'. Такая последовательность может представлять только символы из интервала U+0000–U+00FF.	В C# можно использовать шестнадцатеричные ESC-последовательности из не более чем четырех цифр для построения символьных литералов. Такая последовательность обозначает Unicode-символ с соответствующим кодом.

Строковые литералы представляются последовательностями символов (за исключением переводов строк) в кавычках. В качестве символов могут использоваться и ESC-последовательности, разрешенные в данном языке. Строковый литерал может быть разбит на несколько частей, между которыми стоят знаки +. Значения литералов "Hello, world" и "Hello," + " world" совпадают.

	В C# можно строить **буквальные строковые литералы (verbatim string literals)**, в которых ESC-последовательности и Unicode-последовательности не преобразуются в их значения. Для этого нужно перед открывающей кавычкой поставить знак @. В такой строке могут встречаться любые символы, кроме ". Чтобы поместить туда и кавычку, надо повторить ее два раза. Например, "Hello \t world" отличается от @"Hello \t world", а "\"" совпадает с @"""".

Целочисленные литералы представляют собой последовательности цифр, быть может, со знаком — 1234, -7654. Имеются обычные десятичные литералы и шестнадцатеричные, начинающиеся с 0x или 0X. По умолчанию целочисленные литералы относятся к типу int. Целочисленные литералы, имеющие тип длинного целого числа long, оканчиваются на букву l или L.

В Java имеются также восьмеричные целочисленные литералы, которые начинаются с цифры 0.	В C#, в отличие от Java, имеются беззнаковые целочисленные типы uint и ulong. Литералы этих типов оканчиваются на буквы u или U, и на любую комбинацию букв u/U и l/L, соответственно.

Литералы, представляющие числа с плавающей точкой, могут быть представлены в обычной записи (3.1415926) или экспоненциальной (314.15926e-2 и 0.31415926e1). По умолчанию такие литералы относятся к типу double, и могут иметь в конце символ d или D. Литералы типа float оканчиваются буквами f или F.

В Java литералы с плавающей точкой могут иметь шестнадцатеричное представление с двоичной экспонентой. При этом литерал начинается с 0x или 0X, экспонента должна быть обязательно и должна начинаться с буквы p или P.	В C# есть тип с плавающей точкой decimal для более точного представления чисел при финансовых расчетах. Литералы этого типа оканчиваются на букву m или M.

Операторы и разделители обоих языков:

(	)	{	}	[	]	;	,	.	:	?	~
=	<	>	!	+	-	*	/	%	&	|	^
==	<=	>=	!=	+=	-=	*=	/=	%=	&=	|=	^=
&&	||	++	--	<<	>>	<<=	>>=				

Дополнительные операторы Java: >>>>>>=	Дополнительные операторы C#: -> :: ?? В C#, помимо ранее перечисленных лексических конструкций, имеются директивы препроцессора, служащие для управления компиляцией. Директивы препроцессора не могут находиться внутри кавычек, начинаются со знака # и пишутся в отдельной строке, эта же строка может заканчиваться комментарием. Директивы #define и #undef служат для того, чтобы определять и удалять опции для условной компиляции (такая опция может быть произвольным идентификатором, отличным от true и false). Директивы #if, #elif, #else и #endif служат для того, чтобы вставлять в код и выбрасывать из него некоторые части в зависимости от декларированных с помощью предыдущих директив опций. В качестве условий, проверяемых директивами #if и #elif, могут использоваться выражения, составленные из опций и констант true и false при помощи скобок и операций &&, ||, ==, !=. Например using System; #define Debug public class Assert {

```
  public void Assert (bool x)
  {
    #if Debug
    if(!x) throw
      new Exception("Assert failed");
    #endif
  }
}
```

Директивы #error и #warning служат для генерации сообщений об ошибках и предупреждениях, аналогичных таким же сообщениям об ошибках компиляции. В качестве сообщения выдается весь текст, следующий в строке за такой директивой.

Директива #line служит для управления механизмом сообщений об ошибках с учетом строк. Вслед за такой директивой в той же строке может следовать число, число и имя файла в кавычках или слово default. В первом случае компилятор считает, что строка, следующая после строки с этой директивой, имеет указанный номер, во втором — помимо номера строки в сообщениях изменяется имя файла, в третьем компилятор переключается в режим по умолчанию, забывая об измененных номерах строк.

Директива #pragma warning, добавленная в C# 2.0, служит для включения или отключения предупреждений определенного вида при компиляции. Она используется в виде

#pragma warning disable n_1, …, n_k
#pragma warning restore n_1, …, n_k

где n_1, … , n_k — номера отключаемых/ включаемых предупреждений.

Общая структура программы

Программа на любом из двух рассматриваемых языков представляет собой набор пользовательских типов данных — в основном, классов и интерфейсов, с их методами. При запуске программы выполняется определенный метод некоторого типа. В ходе работы программы создаются объекты различных типов и выполняются их методы (операции над ними). Объектами особого типа представляются различные потоки выполнения, которые могут быть запущены параллельно.

Во избежание конфликтов по именам и для лучшей структуризации программ пользовательские типы размещаются в специальных пространствах имен, которые в Java называются пакетами (packages), а в C# — пространствами имен (namespaces). Имена пакетов и пространств имен могут состоять из нескольких идентификаторов, разделенных точками. Из любого места можно сослаться на некоторый тип, используя его длинное имя,

состоящее из имени содержащего его пространства имен или пакета, точки и имени самого типа.

В обоих случаях программный код компилируется в бинарный код, исполняемый виртуальной машиной. Правила размещения исходного кода по файлам несколько отличаются.

Код пользовательских типов Java размещается в файлах с расширением .Java.
При этом каждый файл относится к тому пакету, чье имя указывается в самом начале файла с помощью декларации package mypackage;
При отсутствии этой декларации код такого файла попадает в пакет с пустым именем.
	Код пользовательских типов C# размещается в файлах с расширением .cs.
	Декларация пространства имен начинается с конструкции namespace mynamespace { и заканчивается закрывающей фигурной скобкой. Все типы, описанные в этих фигурных скобках, попадают в это пространство имен. Типы, описанные вне декларации пространства имен, попадают в пространство имен с пустым именем.
	Пространства имен могут быть вложены в другие пространства имен. При этом следующие декларации дают эквивалентные результаты.
	namespace A.B { … }
	namespace A
	{
	namespace B { … }
	}

В одном файле может быть описан только один общедоступный (public) пользовательский тип верхнего уровня (т.е. не вложенный в описание другого типа), причем имя этого типа должно совпадать с именем файла без расширения. В том же файле может быть декларировано сколько угодно необщедоступных типов.	В одном файле можно декларировать много типов, относящихся к разным пространствам имен, элементы одних и тех же пространств имен могут описываться в разных файлах.
Пользовательский тип описывается полностью в одном файле.	Пользовательский тип описывается целиком в одном файле, за исключением **частичных типов** (введены в C# 2.0), помеченных модификатором partial — их элементы можно описывать в разных файлах, и эти описания объединяются, если не противоречат друг другу.
Чтобы ссылаться на типы, декларированные в других	Чтобы ссылаться на типы, декларированные в других пространствах имен, по их

пакетах, по их коротким именам, можно воспользоваться директивами импорта. Если в начале файла после декларации пакета присутствует директива import Java.util.ArrayList; то всюду в рамках этого файла можно ссылаться на тип ArrayList по его короткому имени. Если же присутствует директива import Java.util.*; то в данном файле можно ссылаться на любой тип пакета Java.util по его короткому имени. Директива import static Java.lang.Math.cos; (введена в Java 5) позволяет в рамках файла вызывать статический метод cos() класса java.lang.Math просто по его имени, без указания имени объемлющего типа. Во всех файлах по умолчанию присутствует директива import java.lang.*; Таким образом, на типы из пакета java.lang можно ссылаться по их коротким именам (если, конечно, в файле не декларированы типы с такими же именами — локально декларированные типы всегда имеют преимущество перед внешними).	коротким именам, можно воспользоваться директивами использования. Директива using System.Collections; делает возможным ссылки с помощью короткого имени на любой тип (или вложенное пространство имен) пространства имен System.Collections в рамках кода пространства имен или типа, содержащего эту директиву или в рамках всего файла, если директива не вложена ни в какое пространство имен. Можно определять новые имена (синонимы или алиасы) для декларированных извне типов и пространств имен. Например, директива using Z=System.Collections.ArrayList; позволяет затем ссылаться на тип System.Collections.ArrayList по имени Z.
Файлы должны располагаться в файловой системе определенным образом. Выделяется одна или несколько корневых директорий, которые при компиляции указываются в опции -sourcepath компилятора. Файлы из пакета без имени должны лежать в одной из	Нет никаких ограничений на именование файлов и содержащихся в них типов, а также на расположение файлов в файловой системе и имена декларированных в них пространств имен.

корневых директорий. Все остальные должны находиться в поддиректориях этих корневых директорий так, чтобы имя содержащего пакета, к которому файл относится, совпадало бы с именем содержащей сам файл директории относительно включающей ее корневой (с заменой точки на разделитель имен директорий).	
Результаты компиляции располагаются в файлах с расширением .class, по одному типу на файл. Хранящие их директории организуются по тому же принципу, что и исходный код, — в соответствии с именами пакетов, начиная от некоторого (возможно другого) набора корневых директорий. Указать компилятору корневую директорию, в которую нужно складывать результаты компиляции, можно с помощью опции -d. Чтобы эти типы были доступны при компиляции других, корневые директории, содержащие соответвующие им .class-файлы, должны быть указаны в опции компилятора -classpath. В этой же опции могут быть указаны архивные файлы с расширением .jar, в которых много .class файлов хранится в соответствии со структурой пакетов.	Результат компиляции C# программы — динамически загружаемая библиотека (с расширением .dll в системе Windows) или исполняемый файл (.exe), имеющие особую структуру. Такие библиотеки называются **сборками (assembly)**. Для того чтобы использовать типы, находящиеся в некоторой сборке с расширением .dll, достаточно указать ее файл компилятору в качестве внешней библиотеки.
Входной точкой программы является метод public static void main (String[]) одного из классов. Его параметр представляет собой массив строк,	Входной точкой программы является метод public static void Main () одного из классов. Такой метод может также иметь параметр типа string[] (представляющий параметры командной

передаваемых как параметры командной строки при запуске. При этом полное имя класса, чей метод main() выбирается в качестве входной точки, указывается в качестве параметра виртуальной машине при запуске (параметры командной строки следуют за ним).	строки, как и в Java) и/или возвращать значение типа int. Класс, чей метод выбирается в качестве входной точки, указывается в качестве стартового класса при сборке исполняемого файла. Собранный таким образом файл всегда будет запускать метод Main() указанного класса.

Ниже приведены программы на обоих языках, вычисляющие и печатающие на экране значение факториала неотрицательного целого числа (0! = 1, n! = 1*2*...*n), передаваемого им в качестве первого аргумента командной строки. Также приводятся командные строки для их компиляции и запуска.

В отличие от Java, параметры компилятора и способ запуска программ в C# не стандартизованы. Приведена командная строка для компилятора, входящего в состав Microsoft Visual Studio 2005 Beta 2. Предполагается, что директории, в которых находятся компиляторы, указаны в переменной окружения $path или %PATH%, и все команды выполняются в той же директории, где располагаются файлы с исходным кодом.

Компилятор Java и Java-машина располагаются в поддиректории bin той директории, в которую устанавливается набор для разработки Java Development Kit. Компилятор C# располагается в поддиректории Microsoft.NET\Framework\v<номер версии установленной среды .NET> системной директории Windows (обычно Windows или WINNT).

```
public class Counter
{
  public int factorial(int n)
  {
    if(n == 0) return 1;
    else if(n > 0)
      return n * factorial(n - 1);
    else
      throw new IllegalArgumentException(
        "Argument should be >= 0, " +
        "current value n = " + n);
  }

  public static void main(String[] args)
  {
    int n = 2;
    if(args.length > 0)
    {
      try
```

```
using System;

public class Counter
{
  public int Factorial(int n)
  {
    if (n == 0) return 1;
    else if (n > 0)
      return n * Factorial(n - 1);
    else
      throw new ArgumentException(
        "Argument should be >= 0, " +
        "current value n = " + n);
  }

  public static void Main(string[] args)
  {
    int n = 2;
    if (args.Length > 0)
```

```
    {
      n = Integer.parseInt(args[0]);
    }
    catch(NumberFormatException e)
    {
      n = 2;
    }
  }

  if(n < 0) n = 2;
  Counter f = new Counter();
  System.out.println(f.factorial(n));
 }
}
```

```
  {
    try
    {
      n = Int32.Parse(args[0]);
    }
    catch (Exception)
    {
      n = 2;
    }
  }

  if (n < 0) n = 2;
  Counter f = new Counter();
  Console.WriteLine(f.Factorial(n));
 }
}
```

Компиляция javac Counter.java	Компиляция csc.exe Counter.cs
Выполнение java Counter 5	Выполнение Counter.exe 5
Результат 120	Результат 120

Контрольные вопросы по теме №9:

1. Перечислите основные элементы из набора стандартов, инструментов и библиотек, предназначенных для разработки приложений разных типов и связанных друг с другом использованием языка программирования Java?
2. По какому плану будет обсуждение и одновременно сравнение характеристик языков?
3. Программы на обоих рассматриваемых языках, C# и Java, могут быть написаны с использованием какого набора символов?
4. Что в Java называются пакетами (packages), а в C# — пространствами имен (namespaces)?
5. Назовите примитивные типы, являющиеся типами значений, для представления простых данных?
6. Что такое инструкция и выражения в обоих языках?
7. Что определяет пользовательские ссылочные типы?
8. Что такое наследование?
9. Что такое элементы или члены (members) пользовательских типов?
10. Какие типы и операции называют шаблонными?
11. Что позволяет описывать операции с неопределенным числом параметров (как в функции printf стандартной библиотеки C)?
12. Что нужно для описания так называемых метаданных — данных, описывающих элементы кода?
13. Что называют монитором при синхронизации?

14.Дайте краткий обзор основных библиотек для обеих платформ, как Java, так и .NET?

Лекция 11. Тема: Компонентные технологии и разработка распределенного ПО

Цель занятия: Рассматриваются основные понятия компонентных технологий разработки ПО и понятие компонента. Рассказывается об общих принципах разработки распределенного ПО и об организации взаимодействия его компонентов в рамках удаленного вызова процедур и транзакций.

Основные понятия компонентных технологий

Понятие программного компонента (software component) является одним из ключевых в современной инженерии ПО. Этим термином обозначают несколько различных вещей, часто не уточняя подразумеваемого в каждом конкретном случае смысла.

Если речь идет об архитектуре ПО (или ведет ее архитектор ПО), под компонентом имеется в виду то же, что часто называется **программным модулем**. Это достаточно произвольный и абстрактный элемент структуры системы, определенным образом выделенный среди окружения, решающий некоторые подзадачи в рамках общих задач системы и взаимодействующий с окружением через определенный интерфейс. В этом курсе для этого понятия употребляется термин **архитектурный компонент** или **компонент архитектуры.**

На диаграммах компонентов в языке UML часто изображаются компоненты, являющиеся единицами сборки и конфигурационного управления, — файлы с кодом на каком-то языке, бинарные файлы, какие-либо документы, входящие в состав системы. Иногда там же появляются компоненты, представляющие собой единицы развертывания системы, — это компоненты уже в третьем, следующем смысле.

Компоненты развертывания являются блоками, из которых строится компонентное программное обеспечение. Эти же компоненты имеются в виду, когда говорят о компонентных технологиях, компонентной или компонентно-ориентированной (component based) разработке ПО, компонентах JavaBeans, EJB, CORBA, ActiveX, VBA, COM, DCOM, .Net, Web-службах (web services), а также о компонентном подходе, упоминаемом в названии данного курса. Согласно [1], такой **компонент** представляет собой структурную единицу программной системы, обладающую четко определенным интерфейсом, который полностью описывает ее зависимости от окружения. Такой компонент может быть независимо поставлен или не поставлен, добавлен в состав некоторой системы или удален из нее, в том числе, может включаться в состав систем других поставщиков.

Различие между этими понятиями, хотя и достаточно тонкое, все же может привести к серьезному непониманию текста или собеседника в некоторых ситуациях.

В определении архитектурного компонента или модуля делается акцент на выделение структурных элементов системы в целом и декомпозицию

решаемых ею задач на более мелкие подзадачи. Представление такого компонента в виде единиц хранения информации (файлов, баз данных и пр.) не имеет никакого значения. Его интерфейс хотя и определен, но может быть уточнен и расширен в процессе разработки.

Понятие компонента сборки и конфигурационного управления связано со структурными элементами системы, с которыми приходится иметь дело инструментам, осуществляющим сборку и конфигурационное управление, а также людям, работающим с этими инструментами. Для этих видов компонентов интерфейсы взаимодействия с другими такими же элементами системы могут быть не определены.

В данной лекции и в большинстве следующих мы будем иметь дело с компонентами в третьем смысле. Это понятие имеет несколько аспектов:

Компонент в этом смысле — **выделенная структурная единица с четко определенным** интерфейсом. Он имеет более строгие требования к четкости определения интерфейса, чем архитектурный компонент. Абсолютно все его зависимости от окружения должны быть описаны в рамках этого интерфейса. Один компонент может также иметь несколько интерфейсов, играя несколько разных ролей в системе.

При описании интерфейса компонента важна не только сигнатура операций, которые можно выполнять с его помощью. Становится важным и то, какие другие компоненты он может задействовать при работе, а также каким ограничениям должны удовлетворять входные данные операций и какие свойства выполняются для результатов их работы.

Эти ограничения являются так называемым **интерфейсным контрактом** или программным контрактом компонента. Интерфейс компонента включает набор операций, которые можно вызвать у любого компонента, реализующего данный интерфейс, и набор операций, которые этот компонент может вызвать в ответ у других компонентов. Интерфейсный контракт для каждой операции самого компонента (или используемой им) определяет **предусловие** и **постусловие** ее вызова. Предусловие операции должно быть выполнено при ее вызове, иначе корректность результатов не гарантируется. Если эта операция вызывается у компонента, то обязанность позаботиться о выполнении предусловия лежит на клиенте, вызывающем операцию. Если же эта операция вызывается компонентом у другого компонента, он сам обязуется выполнить это предусловие. С постусловием все наоборот — постусловие вызванной у компонента операции должно быть выполнено после ее вызова, и это — обязанность компонента. Постусловие операции определяет, какие ее результаты счита ются корректными. В отношении вызываемых компонентом операций выполнение их постусловий должно гарантироваться теми компонентами, у которых они вызываются, а вызывающий компонент может на них опираться в своей работе.

Набор правил определения интерфейсов компонентов и их реализаций, а также правил, по которым компоненты работают в системе и взаимодействуют друг с другом, принято объединять под именем компонентной модели (component model) [2]. В компонентную модель входят правила,

регламентирующие жизненный цикл компонента, т.е. то, через какие состояния он проходит при своем существовании в рамках некоторой системы (незагружен, загружен и пассивен, активен, находится в кэше и пр.) и как выполняются переходы между этими состояниями.

Существуют несколько компонентных моделей. Правильно взаимодействовать друг с другом могут только компоненты, построенные в рамках одной модели, поскольку компонентная модель определяет "язык", на котором компоненты могут общаться друг с другом.

Помимо компонентной модели, для работы компонентов необходим некоторый набор базовых служб (basic services). Например, компоненты должны уметь находить друг друга в среде, которая, возможно, распределена на несколько машин. Компоненты должны уметь передавать друг другу данные, опять же, может быть, при помощи сетевых взаимодействий, но реализации отдельных компонентов сами по себе не должны зависеть от вида используемой связи и от расположения их партнеров по взаимодействию. Набор таких базовых, необходимых для функционирования большинства компонентов служб, вместе с поддерживаемой с их помощью компонентной моделью называется компонентной средой (или компонентным каркасом, component framework). Примеры известных компонентных сред — различные реализации J2EE, .NET, CORBA. Сами по себе J2EE, .NET и CORBA являются спецификациями компонентных моделей и набора базовых служб, которые должны поддерживаться их реализациями.

Компоненты, которые работают в компонентных средах, по-разному реализующих одну и ту же компонентную модель и одни и те же спецификации базовых служб, должны быть в состоянии свободно взаимодействовать. На практике этого, к сожалению, не всегда удается достичь, но любое препятствие к такому взаимодействию рассматривается как серьезная, подлежащая скорейшему разрешению проблема.

Общие принципы построения распределенных систем

В дальнейшем мы будем рассматривать компонентные технологии в связи с разработкой распределенных программных систем. Прежде чем углубляться в их изучение, полезно разобраться в общих принципах построения таких систем, без привязки к компонентному подходу. Тогда многие из решений, применяемых в рамках таких технологий, становятся гораздо более понятными.

Построение распределенных систем высокого качества является одной из наиболее сложных задач разработки ПО. Технологии типа J2EE и .NET создаются как раз для того, чтобы сделать разработку широко встречающихся видов распределенных систем — так называемых бизнес-приложений, поддерживающих решение бизнес-задач некоторой организации, — достаточно простой и доступной практически любому программисту. Основная задача, которую пытаются решить с помощью распределенных систем — обеспечение максимально простого доступа к возможно большему количеству ресурсов как можно большему числу пользователей. Наиболее важными свойствами такой

системы являются прозрачность, открытость, масштабируемость и безопасность.

При реализации очень больших систем (поддерживающих работу тысяч и более пользователей, включающих сотни и более машин) хорошая масштабируемость может быть достигнута только с помощью децентрализации основных служб системы и управляющих ею алгоритмов. Вариантами такого подхода являются следующие.

- Децентрализация обработки запросов за счет использования для этого нескольких машин.
- Децентрализация данных за счет использования нескольких хранилищ данных или нескольких копий одного хранилища.

Децентрализация алгоритмов работы за счет использования для алгоритмов:

- не требующих полной информации о состоянии системы;
- способных продолжать работу при сбое одного или нескольких ресурсов системы;
- не предполагающих единого хода времени на всех машинах, входящих в систему.

Использование, где это возможно, **асинхронной связи** — передачи сообщений без приостановки работы до прихода ответа.

Использование комбинированных систем организации взаимодействия, основанных на следующих схемах.

Иерархическая организация систем, хорошо масштабирующая задачи поиска информации и ресурсов.

Репликация — построение копий данных и их распределение по системе для балансировки нагрузки на разные ее элементы. Частным случаем репликации является кэширование, при котором результаты наиболее часто используемых запросов запоминаются и хранятся как можно ближе к клиенту, чтобы переиспользовать их при повторении запросов.

Взаимодействие точка-точка (peer-to-peer, P2P) обеспечивает независимость взаимодействующих машин от других машин в системе.

Безопасность (safety).

Так как распределенные системы вовлекают в свою работу множество пользователей, машин и географически разделенных элементов, вопросы их безопасности получают гораздо большее значение, чем при работе обычных приложений, сосредоточенных на одной физической машине. Это связано как с невозможностью надежно контролировать доступ к различным элементам такой системы, так и с ее доступностью для гораздо более широкого и разнообразного по своему поведению сообщества пользователей.

Понятие безопасности включает следующие характеристики:

Сохранность и целостность данных.

При обеспечении групповой работы многих пользователей с одними и теми же данными нужно обеспечивать их сохранность (т.е. предотвращать исчезновение данных, введенных одним из пользователей) и в тоже время целостность, т.е. непротиворечивость, выполнение всех присущих данным ограничений.

Это непростая задача, которая не имеет решения, удовлетворяющего все стороны во всех ситуациях, — при одновременном изменении одного и того же элемента данных разными пользователями итоговый результат должен быть непротиворечив и поэтому часто может совпадать только с вводом одного из них. Как будет обработана такая ситуация и возможно ли ее возникновение вообще, зависит от дополнительных требований к системе, от принятых протоколов работы, от того, какие риски — потерять данные одного из пользователей или значительно усложнить работу пользователей с системой — будут сочтены более важными.

Защищенность данных и коммуникаций.

При работе с коммерческими системами, содержащими большие объемы персональной и бизнес-информации, а также с системами обслуживания пользователей государственных ведомств очень важна защищенность как информации, постоянно хранящейся в системе, так и информации одного сеанса работы. Для распределенных систем обеспечить защищенность гораздо сложнее, поскольку нельзя физически изолировать все элементы системы и разрешить доступ к ней только проверенным и обладающим необходимыми знаниями и умениями людям.

Отказоустойчивость и способность к восстановлению после ошибок.

Одним из достоинств распределенных систем является возможность построения более надежно работающей системы из не вполне надежных компонентов. Однако для того, чтобы это достоинство стало реальным, необходимо тщательное проектирование систем с тем, чтобы избежать зависимости работоспособности системы в целом от ее отдельных элементов. Иначе достоинство превращается в недостаток, поскольку в распределенной системе элементов больше и выше вероятность того, что хотя бы один элемент выйдет из строя и хотя бы один ресурс окажется недоступным.

ще важнее для распределенных систем уметь восстанавливаться после сбоев. Уровни этого восстановления могут быть различными. Обычно данные одного короткого сеанса работы считается возможным не восстанавливать, поскольку такие данные часто малозначимы или легко восстанавливаются (иначе стоит серьезно рассмотреть необходимость восстановления сеансов). Но так называемые постоянно хранимые (persistent) данные чаще всего требуется восстанавливать в их последнем непротиворечивом состоянии.

Перед разработчиками систем, удовлетворяющих перечисленным свойствам, встает огромное количество проблем. Решать их все сразу просто невозможно в силу ограниченности человеческих способностей. Чтобы хоть как-то структурировать эти проблемы, их разделяют по следующим аспектам [3].

- Организация связи и передачи данных между элементами системы.
- Поддержка идентификации и поиска отдельных ресурсов внутри системы.
- Организация работ в рамках процессов и потоков.
- Синхронизация параллельно выполняемых потоков работ.
- Поддержка целостности данных и непротиворечивости вносимых изменений.

- Организация отказоустойчивой работы.
- Организация защищенности данных и коммуникаций.

Контрольные вопросы по теме №11:

1. Что называют программным модулем?
2. Что такое выделенная структурная единица с четко определенным интерфейсом?
3. Что объединяться под именем компонентной модели (component model)?
4. Перечислить наиболее важные свойства распределенных систем?
5. Какой вид взаимодействия называют синхронным (synchronous) или блокирующим (blocking)?
6. Какой вид взаимодействия называют асинхронным (asynchronous) или не блокирующим (non blocking)?
7. Перечислить наиболее важные свойства понятии транзакции?

Лекция 12. Тема: Компонентные технологии разработки web-приложений

Цель занятия: Рассматриваются основные элементы компонентных сред Java 2 Enterprise Edition и .NET. Показывается, как в рамках этих технологий решаются основные задачи построения распределенных Web-приложений.

Web-приложения

После обзора общих концепций, связанных с компонентными технологиями и распределенным программным обеспечением, отметим дополнительные общие черты таких технологий в их сегодняшнем виде.

Программное обеспечение в современном мире становится все сложнее и приобретает все больше функций. Коммерческие компании и государственные организации стремятся автоматизировать все больше своих процессов, как внутренних, так и тех, что связаны с общением с внешним миром. При этом, однако, разработка таких приложений, их внедрение и поддержка становятся все дороже.

Есть, тем не менее, фактор, который помогает значительно снизить расходы — широчайшее распространение Интернет. Если ваше программное обеспечение использует для связи между своими элементами базовые протоколы Интернет (TCP/IP и HTTP) и предоставляет пользовательский интерфейс с помощью HTML, который можно просматривать в любом браузере, то практически каждый его потенциальный пользователь не имеет технических препятствий для обращения к этому ПО. Не нужно распространять специальные клиентские компоненты, ставить клиентам специальное оборудование, не нужно тратить много времени и средств на обучение пользователей работе со специфическим интерфейсом, настройке связи с серверами и т.д. Интернет предоставляет готовую инфраструктуру для создания крупномасштабных программных систем, в рамках которых десятки тысяч компонентов могли бы работать совместно и миллионы пользователей могли бы пользоваться их услугами.

Поэтому вполне логично, что Web-приложения, т.е. программные системы, использующие для связи протоколы Интернета, а в качестве пользовательского интерфейса — HTML-страницы, стали одним из самых востребованных видов ПО. Однако, чтобы сделать потенциальные выгоды от использования Интернета реальными, необходимы технологии разработки Web-приложений, которые позволяли бы строить их на компонентной основе, минимизируя затраты на интеграцию отдельных компонентов, их развертывание и поддержку в рабочем состоянии.

Другим важным фактором является распространение расширяемого языка разметки (Extensible Markup Language, XML) как практически универсального формата данных. XML предоставляет стандартную лексическую форму для представления текстовой информации различной структуры и стандартные же способы описания этой структуры. Многие аспекты создания и работы Web-приложений связаны с обменом разнообразно структурированными данными между отдельными компонентами или представлением информации об организации, свойствах и конфигурации системы, имеющей гибкую структуризацию. Использование XML позволяет решить часть возникающих здесь проблем.

Поскольку все современные технологии разработки Web-приложений так или иначе используют XML, следующий раздел посвящен основным элементам этого языка.

Расширяемый язык разметки XML

Данный раздел содержит краткий обзор основных конструкций XML.

XML [3,4,5] является **языком разметки**: различные элементы данных в рамках XML-документов выделяются **тегами** — каждый элемент начинается с открывающего тега <tag> и заканчивается закрывающим </tag>. Здесь tag — идентификатор тега, который обычно является английским словом или набором слов, разделяемых знаками ‘-’, которое(-ые) описывают назначение этого элемента данных. Элементы данных могут быть вложены друг в друга, образуя дерево документа. Кроме того, каждый элемент может иметь набор значений атрибутов, которые представляют собой строки, числа или логические значения. Значения атрибутов для данного элемента помещаются внутри его открывающего тега. Элемент данных, не имеющий вложенных подэлементов, может быть оформлен в виде конструкции <tag … />, т.е. не иметь отдельного закрывающего тега.

Ниже приведен пример описания информации о книге в виде XML:

```
<book
   title = "Pattern-Oriented Software Architecture, Volume 1: A System of Patterns"
   ISBN = "047195869"
   year = 1996
   hardcover = true
   pages = 476
   language = "English">
  <author>Frank Buschmann</author>
  <author>Regine Meunier</author>
```

```
  <author>Hans Rohnert</author>
  <author>Peter Sommerlad</author>
  <author>Michael Stal</author>
  <publisher
     title = "John Wiley & Sons"
     address = "605 Third Avenue, New York, NY 10158-0012, USA" />
</book>
```

В этом примере тег <book>, представляющий описание книги, имеет вложенные теги <author> и <publisher>, представляющие ее авторов (таких тегов может быть несколько) и издателя. Он также имеет атрибуты title, ISBN, year, hardcover, pages и language (название книги, ее международный стандартный номер, т.е. International Standard Book Number или ISBN, плюс год издания, наличие твердой обложки, число страниц и язык). Тег <publisher>, в свою очередь, имеет атрибуты title и address (название и юридический адрес издательской организации).

Элементы XML-документа, называемые также **сущностями**, являются в некотором смысле аналогами значений структурных типов в .NET, а значения их атрибутов — аналогами соответствующих значений полей. При этом теги играют роль самих типов, а атрибуты и вложенные теги — роль их полей, имеющих, соответственно, примитивные и структурные типы. Расширяемым XML назван потому, что можно задать специальную структуру тегов и их атрибутов для некоторого вида документов. Эта структура описывается в отдельном документе, называемом схемой, который сам написан на специальном подмножестве XML, DTD (**Document Type Declaration, декларация типа документа**) [3,4,5] или XMLSchema [6].

XML-документ всегда начинается заголовком, описывающим версию XML, которой соответствует документ, и используемую кодировку. По умолчанию используется кодировка UTF-8.

Затем чаще всего идет описание типа документа, указывающее схему, которой он соответствует, и тег корневого элемента, содержащего все остальные элементы данного документа. Схема может задаваться в формате DTD или XMLSchema. Второй, хотя и является более новым, пока еще используется реже, потому что достаточно много документов определяется с помощью DTD и очень многие инструменты для обработки XML могут пользоваться этим форматом. Используемая схема определяется сразу двумя способами — при помощи строки, которая может служить ключом для поиска схемы на данной машине, и при помощи унифицированного идентификатора документа (Unified Resource Identifier, URI), содержащего ее описание и используемого в том случае, если ее не удалось найти локально.

Ниже приводится пример заголовка и описания типа документа для дескриптора развертывания EJB компонентов (см. подробности далее).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 8.1 EJB 2.1//EN"
    "http://www.sun.com/software/appserver/dtds/sun-ejb-jar_2_1-1.dtd">
```

```
<sun-ejb-jar>
…
</sun-ejb-jar>
```

Другой пример показывает заголовок документа DocBook — основанного на XML формата для технической документации, которая может быть автоматически преобразована в HTML, PDF и другие документы с определенными для них правилами верстки.

```
<?xml version="1.0" encoding="windows-1251"?>
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.3//EN"
"http://www.oasis-open.org/docbook/xml/4.3/docbookx.dtd">
<article>
…
</article>
```

Помимо элементов данных и заголовка с описанием типа документа, XML-документ может содержать комментарии, помещаемые в теги <!-- … -->, **инструкции обработки** вида <? processor-name … ?> (здесь processor-name — идентификатор обработчика, которому предназначена инструкция) и **секции символьных данных** CDATA, которые начинаются набором символов <![CDATA[, а заканчиваются с помощью]]>. Внутри секций символьных данных могут быть любые символы, за исключением закрывающей комбинации. В остальных местах некоторые специальные символы должны быть представлены комбинациями символов в соответствии с таблицей 13.1.

Таблица 13.1. Представления специальных символов в XML

Символ	Представление в XML
<	&lt;
>	&gt;
&	&amp;
"	&quot;
'	&apos;

XML содержит много других конструкций, помимо уже перечисленных, но их рассмотрение выходит за рамки данного курса. Читатель, желающий узнать больше об этом языке и связанных с ним технологиях, может обратиться к [1,2,3,4,5,6,7].

Платформа Java 2 Enterprise Edition

Платформа J2EE предназначена в первую очередь для разработки распределенных Web-приложений и поддерживает следующие 4 вида компонентов [8]:

Enterprise JavaBeans (EJB).

Компоненты EJB предназначены для реализации на их основе бизнес-логики приложения и операций над данными. Любые компоненты, разработанные на Java, принято называть бинами (bean, боб или фасолина, в разговорном языке имеет также значения головы и монеты). Компоненты Enterprise JavaBean

отличаются от "обычных" тем, что работают в рамках EJB-контейнера, который является для них компонентной средой.

В целом EJB-контейнер представляет собой пример **объектного монитора транзакций (object transaction monitor)** — ПО промежуточного уровня, поддерживающего в рамках объектно-ориентированной парадигмы удаленные вызовы методов и распределенные транзакции.

Web-компоненты (Web-components).

Эти компоненты служат для предоставления интерфейса к корпоративным программным системам поверх широко используемых протоколов Интернета, а именно, HTTP. Предоставляемые интерфейсы могут быть как интерфейсами для людей (WebUI), так и специализированными программными интерфейсами, работающими подобно удаленному вызову методов, но поверх HTTP.

Обычные приложения на Java.

J2EE является расширением J2SE и поэтому все Java приложения могут работать и в этой среде. Однако, в дополнение к обычным возможностям J2SE, эти приложения могут использовать в своей работе Web-компоненты и EJB, как напрямую, так и удаленно, связываясь с ними по HTTP.

Аплеты (applets).

Это небольшие компоненты, имеющие графический интерфейс пользователя и предназначенные для работы внутри стандартного Web-браузера. Они используются в тех случаях, когда не хватает выразительных возможностей пользовательского интерфейса на базе HTML, и могут связываться с удаленными Web-компонентами, работающими на сервере, по HTTP.

Компонент любого из этих видов оформляется как небольшой набор классов и интерфейсов на Java, а также имеет дескриптор развертывания (deployment descriptor) — описание в определенном формате на основе XML конфигурации компонента в рамках контейнера, в который он помещается. Приложение в целом также имеет дескриптор развертывания. Дескрипторы развертывания играют важную роль, позволяя менять некоторые параметры функционирования компонента и привязывать их к параметрам среды, в рамках которой компонент работает, не затрагивая его код.

Платформа J2EE приспособлена для разработки многоуровневых Web-приложений. При работе с такими приложениями пользователь формирует свои запросы, заполняя HTML-формы в браузере, который упаковывает их в HTTP-сообщения и пересылает Web-серверу. Web-сервер передает эти сообщения Web-компонентам, выделяющим из них исходные запросы пользователя и передающим их для обработки компонентам EJB. Результаты работы EJB компонентов превращаются Web-компонентами в динамически генерируемые HTML-страницы, и отправляются обратно пользователю, представая перед ним в окне браузера. Аплеты используются там, где нужен более функциональный интерфейс, чем стандартные формы и страницы HTML.

Выделены компоненты, разрабатываемые вручную

Таким образом, приложения на базе J2EE строятся с использованием трех основных архитектурных стилей:

Многоуровневая система.

Самые крупные подсистемы организованы как уровни, решающие различные задачи.

Интерфейс взаимодействия с внешней средой, включая пользователей, реализуется при помощи Web-компонентов.

Уровень бизнес-логики и модели данных реализуется при помощи EJB-компонентов.

Уровень управления ресурсами строится на основе коммерческих систем управления базами данных (СУБД). Можно также подключать другие виды ресурсов, для которых имеется реализация интерфейса поставщика служб J2EE (J2EE service provider interface, J2EE SPI).

Независимые компоненты.

Первые два уровня построены из отдельных компонентов, каждый из которых имеет собственную область ответственности, но может привлекать для решения частных задач другие компоненты.

Данные–представление–обработка (MVC).

Работа компонентов в рамках обработки группы тесно связанных запросов организуется по образцу MVC. При этом сервлеты и обработчики Web-событий служат обработчиками, компоненты JSP — представлением, а компоненты EJB — моделью данных.

Рассмотрим теперь, как решаются общие задачи построения распределенных систем [9] на базе платформы J2EE.

Платформа .NET

Среда .NET предназначена для более широкого использования, чем платформа J2EE. Однако ее функциональность в части, предназначенной для разработки распределенных Web-приложений, очень похожа на J2EE.

В рамках .NET имеются аналоги основных видов компонентов J2EE. Web-компонентам соответствуют компоненты, построенные по технологии ASP.NET, а компонентам EJB, связывающим приложение с СУБД, — компоненты ADO.NET. Компонентная среда .NET обычно рассматриваются как однородная. Однако существующие небольшие отличия в правилах, управляющих созданием и работой компонентов ASP.NET и остальных, позволяют предположить, что в рамках .NET присутствует аналог Web-контейнера, отдельная компонентная среда для ASP.NET, и отдельная — для остальных видов компонентов. Тем не менее, даже если это так, эти среды тесно связаны и, как и контейнеры J2EE, позволяют взаимодействовать компонентам, размещенным в разных средах. Компонентная среда для ASP.NET, в отличие от Web-контейнера в J2EE, поддерживает автоматические распределенные транзакции.

Тем самым, типовая архитектура Web-приложений на основе .NET может быть изображена так, как это сделано на рис. 13.

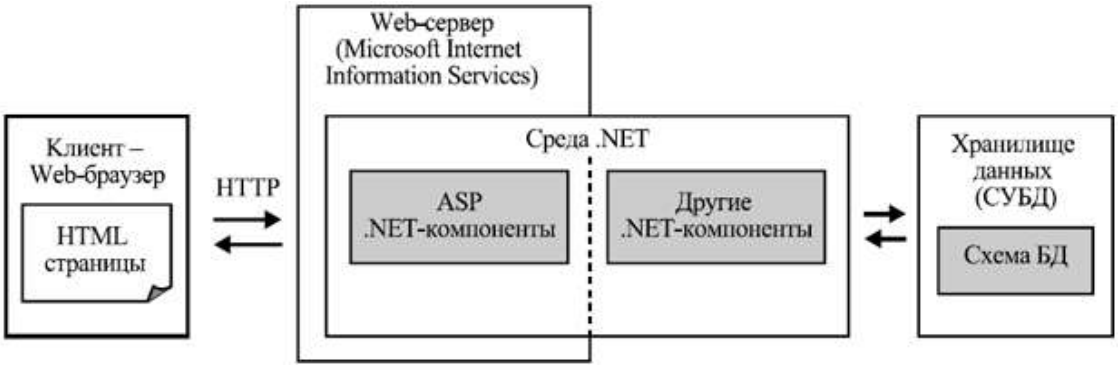


Рис. 13 Типовая архитектура Web-приложения на основе .NET

В целом, Web-приложения на основе .NET используют тот же набор архитектурных стилей, что и аналогичные J2EE-приложения.

Обычно .NET-компоненты представляют собой набор .NET-классов и конфигурационных файлов, играющих роль дескрипторов развертывания и также представленных в некотором формате на основе XML. Для приложений в целом тоже пишутся особые конфигурационные файлы.

Контрольные вопросы по теме №12

1. Что такое Web-приложения?
2. Что является языком разметки?
3. Какой интерфейс называют удаленным интерфейсом (remote interface) и наследующий java.rmi.Remote?
4. Что называют в службе именования контекстами?
5. В каких случаях используются механизмы, реализующие интерфейсы управления транзакциями Java (Java Transaction API, JTA)?
6. Какими способами поддерживается защищенность J2EE приложения?
7. Расшифровать типовую архитектуру Web-приложений на основе .NET?

Лекция 13. Тема : Разработка различных уровней web приложений в J2EE и .NET

Цель занятия: Рассматриваются используемые в рамках Java Enterprise Edition и .NET техники разработки компонентов Web-приложений, связывающих приложение с базой данных и представляющих собой элементы пользовательского интерфейса.

Общая архитектура Web-приложений

Платформы J2EE и .NET предоставляют специальную поддержку для разработки компонентов на двух уровнях: уровне интерфейса пользователя (WebUI) и уровне связи с данными.

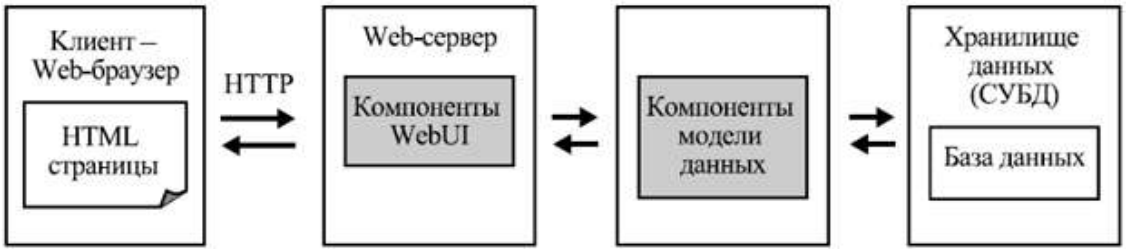


Рис. Общая схема архитектуры Web-приложений J2EE и .NET

Пользовательский интерфейс Web-приложений основан на генерации динамических страниц HTML, содержащих данные, которые запрашивает пользователь. Уровень модели данных предоставляет приложению возможность работать с данными, обычно хранящимися в виде набора таблиц и связей между ними, как с набором связанных объектов.

Основные отличия между техниками разработки компонентов этих двух уровней, используемыми в рамках J2EE и .NET, можно сформулировать следующим образом.

В J2EE компоненты EJB предназначены не только для представления данных приложения в виде объектов, но и для реализации его бизнес-логики, т.е. объектов предметной области и основных способов работы с ними.

В .NET нет специально выделенного вида компонентов, предназначенного для реализации бизнес-логики — она может реализовываться с помощью обычных классов, что часто удобнее. Это положение должно измениться с выходом EJB 3.0.

EJB-компоненты являются согласованным с объектно-ориентированным подходом представлением данных приложения. Работа с ними организуется так же, как с объектами обычных классов (с точностью до некоторых деталей).

В .NET-приложениях все предлагаемые способы представления данных являются объектными обертками вокруг реляционного представления — в любом случае приходится работать с данными как с набором таблиц. В .NET нет автоматической поддержки их преобразования в систему взаимосвязанных объектов и обратно.

Уровень бизнес-логики и модели данных в J2EE

В рамках приложений, построенных по технологии J2EE, связь с базой данных и бизнес-логику, скрытую от пользователя, принято реализовывать с помощью компонентов Enterprise JavaBeans. На момент написания этой лекции последней версией технологии EJB является версия 2.1, в первой половине 2006 года должны появиться инструменты для работы с EJB 3.0 (в рамках J2EE 5.0).

Возможны и другие способы реализации этих функций. Например, бизнес-логика может быть реализована непосредственно в методах объектов пользовательского интерфейса, а обмен данными с базой данных — через интерфейс JDBC. При этом, однако, теряется возможность переиспользования функций бизнес-логики в разных приложениях на основе единой базы данных, а также становится невозможным использование автоматических транзакций при работе с данными. Транзакции в этом случае нужно организовывать с помощью явных обращений к JTA .

Компонент Enterprise JavaBeans (EJB) является компонентом, представляющим в J2EE-приложении элемент данных или внутренней, невидимой для пользователя логики приложения. Для компонентов EJB определен жизненный цикл в рамках рабочего процесса приложения — набор состояний, через которые проходит один экземпляр такого компонента. Компоненты EJB работают внутри EJB-контейнера, являющегося для них компонентной средой. Функции EJB-контейнера следующие:

Управление набором имеющихся EJB-компонентов, например, поддержкой пула компонентов для обеспечения большей производительности, а также жизненным циклом каждого отдельного компонента, в частности, его инициализацией и уничтожением.

Передача вызовов между EJB-компонентами, а также их удаленных вызовов. Несколько EJB-контейнеров, работающих на разных машинах, обеспечивают взаимодействие наборов компонентов, управляемых каждым из них.

- Поддержка параллельной обработки запросов.
- Поддержка связи между EJB-компонентами и базой данных приложения и синхронизация их данных.
- Поддержка целостности данных приложения с помощью механизма транзакций.

Защита приложения с помощью механизма ролей: передача прав ролей при вызовах между компонентами и проверка допустимости обращения в рамках роли к методам компонентов.

Для разработки набора компонентов EJB нужно, во-первых, для каждого компонента создать один или несколько классов и интерфейсов Java, обеспечивающих реализацию самой функциональности компонента и определение интерфейсов для удаленных обращений к нему, и, во-вторых, написать дескриптор развертывания — XML-файл, описывающий следующее:

- Набор EJB-компонентов приложения.
- Совокупность элементов кода на Java, образующих один компонент.
- Связь свойств компонента с полями таблиц БД и связями между таблицами.
- Набор ролей, правила доступа различных ролей к методам компонентов, правила передачи ролей при вызовах одними компонентами других.
- Политику компонентов и их методов по отношению к транзакциям.
- Набор ресурсов, которыми компоненты могут пользоваться в своей работе.

Правила создания EJB-компонента зависят от его вида. Различают три таких вида EJB-компонентов:

- Компоненты данных (сущностные, entity beans).
- Представляют данные приложения и основные методы работы с ними.
- Сеансовые компоненты (session beans).

Представляют независимую от пользовательского интерфейса и конкретных типов данных логику работы приложения, называемую иногда бизнес-логикой.

Компоненты, управляемые сообщениями (message driven beans).

Тоже предназначены для реализации бизнес-логики. Но, если сеансовые компоненты предоставляют интерфейс для синхронных вызовов, компоненты, управляемые сообщениями, предоставляют асинхронный интерфейс. Клиент, вызывающий метод в сеансовом компоненте, ждет, пока вызванный компонент не завершит свою работу. Компоненту же, управляемому сообщениями, можно отослать сообщение и продолжать работу сразу после окончания его передачи, не дожидаясь окончания его обработки.

Далее описываются основные правила построения EJB компонентов разных видов. Более детальное описание этих правил можно найти в [1,2].

Компоненты данных и сеансовые компоненты

Компонент данных или сеансовый компонент могут состоять из следующих элементов: пара интерфейсов для работы с самим компонентом — **удаленный интерфейс** и **локальный интерфейс**; пара интерфейсов для поиска и создания компонентов — **удаленный исходный интерфейс** и **локальный исходный интерфейс**; класс компонента, реализующий методы работы с ним; и, для компонентов данных**,** — **класс первичного ключа**. Обязательно должен быть декларирован класс компонента и один из интерфейсов — удаленный или локальный. Для компонентов данных обязательно должен быть определен класс первичного ключа.

Удаленный интерфейс (remote interface).

Этот интерфейс декларирует методы компонента, к которым можно обращаться удаленно, т.е. из компонентов, работающих в рамках другого процесса или на другой машине.

Удаленный интерфейс должен наследовать интерфейс javax.ejb.EJBObject (в свою очередь, наследующий java.rmi.Remote).

Для компонента данных он определяет набор свойств (в смысле JavaBeans, т.е. пар методов Type getName()/void setName(Type)), служащих для работы с отдельными полями данных или компонентами, связанными с этим компонентом по ссылкам. Это могут быть и часто используемые дополнительные операции, как-то выражающиеся через операции с отдельными полями данных, в том числе и вычислимые свойства. Например, для книги в базе данных приложения хранится набор ссылок на данные об ее авторах, а число авторов может быть часто используемым свойством книги, вычисляемым по этому набору ссылок.

Для сеансового компонента методы удаленного интерфейса служат для реализации некоторых операций бизнес-логики или предметной области, вовлекающих несколько компонентов данных.

Локальный интерфейс (local interface).

По назначению этот интерфейс полностью аналогичен удаленному, но декларирует методы компонента, которые можно вызывать только в рамках того же процесса. Этот интерфейс служит для увеличения производительности приложений, в которых взаимодействия между компонентами происходят в основном в рамках одного процесса. При этом они могут использовать локальные интерфейсы друг друга, не привлекая сложный механизм реализации удаленных вызовов методов. Однако при использовании локального интерфейса компонентао нужно обеспечить развертывание этого компонента в рамках того же EJB-контейнера, что и вызывающий его компонент.

Локальный интерфейс должен наследовать интерфейсу javax.ejb.EJBLocalObject.

Удаленный исходный интерфейс (remote home interface).

Исходные интерфейсы служат для поиска и создания компонентов. Такой интерфейс может декларировать метод поиска компонента данных по значению его первичного ключа findByPrimaryKey(…) и метод создания такого

компонента с указанным значением первичного ключа create(…). Могут быть также определены методы, создающие компонент по набору его данных или возвращающие коллекцию компонентов, данные которых соответствуют аргументам такого метода. Например, метод, создающий компонент, который представляет книгу с данным названием, createByTitle(String title), или метод, находящий все книги с данным набором авторов Collection findByAuthors(Collection authors).

Удаленный исходный интерфейс предназначен для создания и поиска компонентов извне того процесса, в котором они работают. Его методы возвращают ссылку на удаленный интерфейс компонента или коллекцию таких ссылок.

Такой интерфейс должен наследовать интерфейсу javax.ejb.EJBHome (являющемуся наследником java.rmi.Remote).

Локальный исходный интерфейс (local home interface).

Имеет то же общее назначение, что и удаленный исходный интерфейс, но служит для работы с компонентами в рамках одного процесса. Соответственно, его методы поиска или создания возвращают ссылку на локальный интерфейс компонента или коллекцию таких ссылок.

Должен наследовать интерфейсу javax.ejb.EJBLocalHome.

Класс компонента **(bean class).**

Этот класс реализует методы удаленного и локального интерфейсов (но не должен реализовывать сами эти интерфейсы!). Он определяет основную функциональность компонента.

Для компонентов данных такой класс должен быть абстрактным классом, реализующим интерфейс javax.ejb.EntityBean. Свойства, соответствующие полям хранимых данных или ссылкам на другие компоненты данных, должны быть определены в виде абстрактных пар методов getName()/setName(). В этом случае EJB-контейнер может взять на себя управление синхронизацией их значений с базой данных. Вычислимые свойства, значения которых не хранятся в базе данных, реализуются в виде пар неабстрактных методов.

Для сеансовых компонентов класс компонента должен быть неабстрактным классом, реализующим интерфейс javax.ejb.SessionBean и все методы удаленного и локального интерфейсов.

Кроме того, класс компонента может (а иногда и должен) реализовывать некоторые методы, которые вызываются EJB-контейнером при переходе между различными этапами жизненного цикла компонента.

Например, при инициализации экземпляра компонента всегда вызывается метод ejbCreate(). Для компонентов данных он принимает на вход и возвращает значение типа первичного ключа компонента. Если первичный ключ — составной, он должен принимать на вход значения отдельных его элементов. Такой метод для компонента данных должен возвращать null и всегда должен быть реализован в классе компонента. Для сеансовых компонентов он имеет тип результата void, а на вход принимает какие-то параметры, служащие для инициализации экземпляра компонента. Для каждого метода исходных интерфейсов с именем createSomeSuffix(…) в классе компонента должен быть

реализован метод ejbCreateSomeSuffix(…) с теми же типами параметров. Для компонентов данных все такие методы возвращают значение типа первичного ключа, для сеансовых — void.

Другие методы жизненного цикла компонента, которые можно перегружать в классе компонента, декларированы в базовых интерфейсах компонентов соответствующего вида (javax.ejb.EntityBean или javax.ejb.SessionBean). Это, например, ejbActivate() и ejbPassivate(), вызываемые при активизации и деактивизации экземпляра компонента; ejbRemove(), вызываемый перед удалением экземпляра компонента из памяти; для компонентов данных — ejbStore() и ejbLoad(), вызываемые при сохранении данных экземпляра в базу приложения или при их загрузке оттуда.

 Класс первичного ключа (primary key class).

Декларируется только для компонентов данных, если в этом качестве нельзя использовать подходящий библиотечный класс.

Определяет набор данных, которые образуют первичный ключ записи базы данных, соответствующей одному экземпляру компонента.

Чаще всего это библиотечный класс, например, String или Integer. Пользовательский класс необходим, если первичный ключ составной, т.е. состоит из нескольких значений простых типов данных. В таком классе должен быть определен конструктор без параметров и правильно перегружены методы equals() и hashCode(), чтобы EJB-контейнер мог корректно управлять коллекциями экземпляров компонентов с такими первичными ключами. Такой класс также должен реализовывать интерфейс java.io.Serializable.

Уровень пользовательского интерфейса в J2EE

Компоненты пользовательского интерфейса в Web-приложениях, построенных как по технологии J2EE, так и по .NET, реализуют обработку HTTP-запросов, приходящих от браузера, и выдают в качестве результатов HTTP-ответы, содержащие сгенерированные HTML-документы с запрашиваемыми данными. Сами запросы автоматически строятся браузером на основе действий пользователя — в основном, переходов по ссылкам и действий с элементами управления в HTML-формах.

Если стандартных элементов управления HTML не хватает для реализации функций приложения или они становятся неудобными, используются специальные библиотеки элементов управления WebUI, предоставляющие более широкие возможности для пользователя и более удобные с точки зрения интеграции с остальными компонентами приложения.

В рамках J2EE версии 1.4 два основных вида компонентов WebUI — сервлеты (servlets) и серверные страницы Java (Java Server Pages, JSP) — отвечают, соответственно, за обработку действий пользователя и представление данных в ответе на его запросы. В следующей версии J2EE 5.0 будут также использоваться компоненты **серверного интерфейса Java (Java Server Faces, JSF)** — библиотека элементов управления WebUI.

Сервлеты представляют собой классы Java, реализующие обработку запросов HTTP и генерацию ответных сообщений в формате этого протокола. Страницы JSP являются упрощенным представлением сервлетов, основанным на

описании генерируемого в качестве ответа HTML-документа при помощи смеси из его постоянных элементов и кода на Java, генерирующего его изменяемые части. При развертывании Web-приложения содержащиеся в нем страницы JSP транслируются в сервлеты и далее работают в таком виде. Описание генерируемых документов на смеси из HTML и Java делает страницы JSP более удобными для разработки и значительно менее объемными, чем получаемый из них и эквивалентный по функциональности класс-сервлет.

Сервлеты

Интерфейс Java-сервлетов определяется набором классов и интерфейсов, входящих в состав пакетов javax.servlet и javax.servlet.http, являющихся частью J2EE SDK. Первый пакет содержит классы, описывающие независимые от протокола сервлеты, второй — сервлеты, работающие с помощью протокола HTTP.

Основные классы и интерфейсы пакета javax.servlet.http следующие.

HttpServlet

Предназначен для реализации сервлетов, работающих с HTTP-сообщениями. Содержит защищенные методы, обрабатывающие отдельные методы HTTP-запросов, из которых наиболее важны void doGet(HttpServletRequest, HttpServletResponse), определяющий обработку GET-запросов, и void doPost(HttpServletRequest, HttpServletResponse), обрабатывающий POST-запросы. В обоих методах первый параметр содержит всю информацию о запросе, а второй — о генерируемом ответе.

HttpServletRequest и HttpServletResponse — интерфейсы, содержащие методы для получения и установки (второй) заголовков и других атрибутов HTTP-запросов и ответов. Второй интерфейс также содержит метод, возвращающий поток вывода для построения содержимого ответа.

Cookie

Класс, представляющий закладки сервера, которые хранятся на клиентской машине для запоминания информации о данном пользователе.

HTTPSession

Интерфейс, предоставляющий методы для управления сеансом обмена HTTP-сообщениями. Информация о сеансе используется в том случае, если она должна быть доступна нескольким сервлетам.

Серверные страницы Java

Серверные страницы Java [7,8] представляют собой компоненты, разрабатываемые на смеси из HTML и Java и предназначенные для динамического создания HTML-документов, содержащих результаты обработки запросов пользователя. Таким образом, JSP обычно играют роль представления в образце "данные–представление–обработчик", принятом за основу архитектуры приложений J2EE. Результатом работы JSP является HTML-страничка, а вставки Java кода служат для построения некоторых ее элементов на основе результатов работы приложения.

При работе Web-приложения JSP компилируются в сервлеты специального вида. При этом основное содержание страницы JSP превращается в метод doGet(), в котором HTML-элементы записываются в поток

содержимого ответа в неизменном виде, а элементы Java-кода преобразуются в код, записывающий некоторые данные в тот же поток на основании параметров запроса или данных приложения.

Контрольные вопросы по теме №13:

1. Из каких элементов состоят Компонент данных или сеансовый компонент?
2. Что представляет собой протокол HTTP (Hypertext Transfer Protocol, протокол передачи гипертекста)?
3. Что представляет собой Сервлеты?
4. Что представляет собой Серверные страницы Java?
5. Перечислить виды Скриптовых элементов?
6. Перечислить виды компонентов в .NET?
7. Расшифровать общую схему архитектуры Web-приложений J2EE и .NET?

Лекция 14. Тема: Развитие компонентных технологий

Цель занятия: Рассказывается о некоторых компонентных средах и технологиях, обрисовывающих направления дальнейшего развития стандартных платформ разработки Web-приложений. Также рассматриваются Web-службы, представляющие собой компонентную технологию другого уровня.

В данной лекции рассказывается о развитии компонентных технологий разработки Web-приложений, нацеленном на повышение их гибкости, удобства их создания и поддержки, а также на снижение трудоемкости внесения изменений в приложения такого рода.

В ряде аспектов разработка отдельных компонентов в рамках .NET несколько проще, чем разработка компонентов с той же функциональностью в рамках J2EE версии 1.4. В то же время, разработка приложений в целом в рамках J2EE проще для начинающих разработчиков, поскольку имеющаяся по этой платформе документация четче определяет общую структуру приложений и распределение ответственности между разными типами компонентов в нем.

Большим достоинством J2EE является прозрачность и предсказуемость ее развития, поскольку все его шаги открыты в рамках четко определенного процесса компании Sun для внесения изменений в спецификации платформы и на каждом из этих шагов учитываются интересы множества участников. Развитие платформы J2EE определяется большим количеством открытых проектов отдельных разработчиков и организаций, предлагающих свои решения по построению сред функционирования Web-приложений (Web application frameworks).

Развитие же платформы .NET находится целиком в руках компании Microsoft и пока не является прозрачным для тех, кто не работает в ней или в одной из близких к ней компаний-партнеров. На основании выступлений отдельных представителей компании можно делать выводы, касающиеся лишь общих планов развития платформы, без каких-либо технических деталей. Поэтому в данной лекции рассматриваются, в основном, направления развития технологий J2EE.

Развитие технологий J2EE

Ряд разработчиков выделяет следующие проблемы удобства разработки и поддержки приложений J2EE версии 1.4.

• Громоздкость разработки компонентов EJB и неудобство их использования для описания структуры предметной области. Для разработки такого простейшего компонента необходимо определить два интерфейса, класс компонента и написать дескриптор развертывания.Полученные классы и интерфейсы достаточно сильно отличаются от обычных классов Java, с помощью которых разработчики описывали бы предметную область в рамках обычного приложения на платформе J2SE. Поэтому гораздо тяжелее вносить в них изменения, связанные с изменением требований к соответствующим объектам предметной области.

• Отсутствие удобной поддержки для отображения иерархии наследования классов в структуру базы данных приложения. Данные класса-предка и класса-наследника могут храниться в одной таблице, в разных и несвязанных таблицах, или общая часть данных может храниться в одной таблице, а специфические данные класса-наследника — в другой. Однако для обеспечения правильной синхронизации данных в каждом из этих случаев достаточно много кода надо написать вручную. Поскольку во многих приложениях объектные модели данных содержат классы, связанные отношением наследования, отсутствие вспомогательных механизмов, автоматически обеспечивающих отображение таких классов на структуру базы данных, приносит много неудобств.

• Невозможность использовать в рамках приложения компоненты EJB, соответствующие данным в базе данных, и временные объекты того же типа, для которых не нужно иметь соответствующих записей в таблицах баз данных. Часто такая возможность оказывается удобной при программировании различных методов обработки данных внутри приложений.

• Громоздкость разработки сервлетов для обработки простых (тем более, сложных) запросов пользователя. При этом необходимо полностью проанализировать запрос, часто — найти в содержащемся в нем документе HTML поля формы, заполненной пользователем, и указанную им операцию их обработки. Только после этого можно переходить к собственно выполнению этого запроса, что, казалось бы, является основной функцией сервлета. Таким образом, большое количество усилий тратится только на то, чтобы выделить из запроса операцию, которую пользователь хочет произвести, а также ее аргументы.

• Неудобство использования в рамках JSP-страниц специализированных элементов пользовательского интерфейса. Для сравнения: в рамках ASP.NET можно использовать библиотечные и пользовательские элементы управления, которые помещаются на страницу при помощи специального тега, а в параметрах этого тега указываются, в частности, методы для обработки событий, связанных с действиями пользователя.

Для решения этих проблем используются различные библиотеки, инструменты и компонентные среды, созданные в сообществе Java-

разработчиков. Некоторые такие библиотеки и техники станут стандартными средствами в рамках платформы J2EE новой версии 5.0 [1].

Jakarta Struts

Среда Jakarta Struts [2,3] создавалась для того, чтобы упростить разработку компонентов Web-приложения, предназначенных для обработки запросов пользователей, и сделать эту обработку более гибкой.

Основные решаемые такими компонентами задачи можно сформулировать следующим образом:

- выделить сам логический запрос и его параметры из HTML-документа, содержащегося в HTTP-запросе;
- проверить корректность параметров запроса и сообщить пользователю об обнаруженной некорректности наиболее информативным образом;
- преобразовать корректный логический запрос и его параметры в вызовы соответствующих операций над объектами предметной области;
- передать результаты сделанных вызовов компонентам, ответственным за построение их представления для пользователя.

Как и в рамках базовой платформы J2EE, в Struts основным архитектурным стилем для Web-приложений является образец "данные–представление–обработка". При этом роль представления играют JSP-страницы, а роль обработчиков — сервлеты. Основные отличия Struts от стандартной техники J2EE связаны с большей специализацией сервлетов и некоторой стандартизацией обмена данными между сервлетом, обрабатывающим запросы пользователя, и JSP-страницей, представляющей их результаты.

В рамках приложения на основе Struts используется ровно один стандартизированный сервлет (ActionServlet), анализирующий запросы пользователя и выделяющий из каждого запроса **действие (action)**, которое пользователь пытается выполнить. Для Интернет-магазина такими действиями, например, могут быть аутентификация (предоставление своего имени и пароля), получение данных о товаре, поиск товара, добавление товара к уже заказанным, изменение заказа, предоставление прав на скидку, выполнение заказа, получение статуса выполнения заказа, отмена заказа и пр. Для каждого действия создается отдельный **класс действия**. Такой класс должен быть наследником класса org.apache.struts.action.Action из библиотеки Struts и перегружать метод ActionForward execute(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse) — именно он и вызывается для выполнения этого действия.

Java Server Faces

Java Server Faces (JSF) [4,5] включают библиотеку элементов управления WebUI javax.faces и две библиотеки пользовательских тегов, предназначенных для использования этих элементов управления в рамках серверных страниц Java. С помощью тегов библиотеки jsf/html элементы управления размещаются на странице, а с помощью тегов из jsf/core описывается обработка событий, связанных с этими элементами, и проверка корректности действий пользователя.

В аспекте построения WebUI на основе серверных страниц Java-технология Java Server Faces является развитием подхода Struts (Struts включают решения и для других аспектов разработки приложений), предлагая более богатые библиотеки элементов WebUI и более гибкую модель управления ими. Эта модель включает следующие элементы:

Возможность различного изображения абстрактного элемента управления (например, элемент управления "выбор одного из многих" может быть изображен как группа радио-кнопок, комбо-бокс или список).

В дополнение к библиотекам элементов WebUI JSF предлагает определять правила навигации между страницами в конфигурационном файле приложения. Каждое правило относится к некоторому множеству страниц и при выполнении определенного действия или наступлении события предписывает переходить на некоторую страницу. Действия и события связываются с действиями пользователя или логическими результатами их обработки (такими результатами могут быть, например, успешная регистрация заказа в системе, попытка входа пользователя в систему с неправильным паролем и пр.).

Технология Java Server Face версии 1.2 войдет в состав будущей версии 5.0 платформы J2EE .

Управление данными приложения. Hibernate

Технологии обеспечения синхронизации внутренних данных приложения и его базы данных развиваются в настоящий момент достаточно активно. Технология EJB предоставляет соответствующие механизмы, но за счет значительного снижения удобства разработки и модификации компонентов. Обеспечение той же функциональности при более простой внутренней организации кода является основным направлением развития в данной области. Возможным решением этой задачи являются объектно-реляционные преобразователи (object-relation mappers, ORM), которые обеспечивают автоматическую синхронизацию между данными приложения в виде наборов связанных объектов и данными, хранящимися в системе управления базами данных (СУБД) в реляционном виде, т.е. в форме записей в нескольких таблицах, ссылающихся друг на друга с помощью внешних ключей.

Одним из наиболее широко применяемых и развитых в технологическом плане объектно-реляционных преобразователей является Hibernate [7,8,9].

Базовая парадигма, лежащая в основе избранного Hibernate подхода, — это использование объектов обычных классов Java (быть может, оформленных в соответствии с требованиями спецификации JavaBeans — с четко выделенными свойствами) в качестве объектного представления данных приложения. Такой подход даже имеет название-акроним POJO (plain old Java objects, простые старые Java-объекты), призванное показать его отличие от сложных техник построения компонентов, похожих на EJB.

Большое достоинство подобного подхода — возможность использовать один раз созданные наборы классов, представляющих понятия предметной области, в качестве модели данных любых приложений на основе Java, независимо от того, являются ли они распределенными или локальными,

требуется ли в них синхронизация с базой данных и сохранение данных объектов или нет.

С помощью дополнительной службы NHibernate [7] возможности среды Hibernate могут быть использованы и из приложений на базе .NET.

Java Data Objects

Еще более упростить разработку объектно-ориентированных приложений, данные которых могут храниться в базах данных, призвана технология Java Data Objects (JDO) [10,11].

В ее основе тоже лежит использование для работы с хранимыми данными обычных классов на Java, но в качестве хранилища данных может выступать не только реляционная СУБД, но вообще любое хранилище данных, имеющее соответствующий специализированный адаптер (в рамках JDO это должна быть реализация интерфейса javax.jdo.PersistenceManager). Основная функция этого адаптера — прозрачная для разработчиков синхронизация хранилища данных и набора хранимых объектов в памяти приложения. Он должен также обеспечивать достаточно высокую производительность приложения, несмотря на наличие нескольких промежуточных слоев между классами самого приложения и хранилищем данных, представляемых ими.

Использование JDO очень похоже на использование Hibernate. Конфигурационные файлы, хранящие информацию о привязке объектов Java-классов к записям в определенных таблицах, а также о привязке коллекций ссылок на объекты к ссылкам между записями, также похожи на аналогичные файлы Hibernate. Обычно первые даже несколько проще, поскольку большую часть работы по отображению полей объектов в поля записей базы данных берет на себя специализированный адаптер.

В дополнение JDO предоставляет средства для построения запросов с помощью описания свойств объектов, без использования более привычного встроенного SQL.

В целом, подход JDO является обобщением подхода ORM на произвольные хранилища данных, но он требует реализации более сложных специализированных адаптеров для каждого вида таких хранилищ, в то время как один и тот же ORM может использоваться для разных реляционных СУБД, требуя для своей работы только наличия более простого драйвера JDBC.

Среда Spring

Среда Spring представляет собой одну из наиболее технологичных на данный момент сред разработки Web-приложений. В ее рамках получили дальнейшее развитие идеи специализации обработчиков запросов, использованные в Struts. В Spring также используются элементы аспектно-ориентированного подхода к разработке ПО, позволяющие значительно повысить гибкость и удобство модификации построенных на ее основе Web-приложений.

Основная задача, на решение которой нацелена среда Spring, — интеграция разнородных механизмов, используемых при разработке компонентных Web-приложений. В качестве двух основных средств такой интеграции используются идеи **обращения управления (inversion of control)** и

аспектно-ориентированного программирования (aspect-oriented programming, AOP).

Обращением управления называют отсутствие обращений из кода компонентов приложения к какому-либо API, предоставляемому средой и ее библиотеками. Вместо этого компоненты приложения реализуют только функции, необходимые для работы в рамках предметной области и решения тех задач, с которыми приложению придется иметь дело. Построение из этих компонентов готового приложения, конфигурация отдельных его элементов и связей между ними — это дело среды, которая сама в определенные моменты обращается к нужным операциям компонентов. Конфигурация компонентов приложения в среде Spring осуществляется при помощи XML-файла springapp-servlet.xml. Этот файл содержит описание набора компонентов, которые настраиваются при помощи указания параметров инициализации соответствующих классов и значений своих свойств в смысле JavaBeans.

Ajax

Рассказывая о развитии технологий разработки Web-приложений, невозможно обойти вниманием набор техник, известный под названием Ajax и используемый для снижения времени реакции Web-интерфейсов на действия пользователя.

Вообще говоря, Web-технологии не очень хорошо приспособлены для построения пользовательского интерфейса интерактивных приложений, т.е. таких, где пользователь достаточно часто выполняет какие-то действия, на которые приложение должно реагировать. Они изначально разрабатывались для предоставления доступа к статической информации, которая меняется редко и представлена в виде набора HTML-страниц. Обычно при обмене данными между Web-клиентом и Web-сервером клиент изредка посылает серверу простые и небольшие по объему запросы, а тот в ответ может присылать достаточно объемные документы.

В интерактивных приложениях обмен данными между интерфейсными элементами приложения и обработчиками запросов несколько иной. Обработчик достаточно часто получает запросы и небольшие наборы их параметров, а изменения, которые происходят в интерфейсе после получения ответа на запрос, обычно тоже невелики. Часто нужно изменить содержание лишь части показываемой браузером страницы, в то время как остальные ее элементы представляют более стабильную информацию, являющуюся элементом дизайна сайта или набором пунктов его меню. Для отражения этих изменений не обязательно пересылать с сервера весь HTML-документ, что предполагается в рамках традиционного обмена информацией с помощью Web. Точно так же, если бы корректность вводимых пользователем данных можно было бы проверить на стороне клиента, обработка некорректного ввода происходила бы гораздо быстрее и не требовала бы вообще никакого обмена данными с сервером.

Ajax пытается решить эти задачи при помощи комбинации кода на JavaScript, выполняющегося в браузере, и передаваемых время от времени между клиентом и сервером специальных XML-сообщений, содержащих

только существенную информацию о запросе или изменениях HTML-страницы, которые должны быть показаны. В рамках браузера в отдельном потоке работает ядро Ajax, которое получает сообщения JavaScript-кода о выполнении пользователем определенных действий, выполняет проверку их корректности, преобразует их в посылку соответствующего запроса на сервер, преобразует ответ сервера в новую страницу или же выдает уже имеющийся в специальном кэше ответ. Запросы на сервер и их обработка осуществляются часто асинхронно с действиями пользователя, позволяя заметно снизить ощущаемое время реакции системы на них.

Web-службы

В настоящее время совместно с компонентными технологиями на базе J2EE и .NET широкое распространение получают Web-службы или **Web-сервисы (Web services)**, представляющие собой компонентную технологию другого рода, реализуемую поверх этих платформ. Основными отличительными признаками **служб (services)** любого вида служат следующие.

Служба чаще всего предоставляет некоторую функцию, которая полезна достаточно широкому кругу пользователей.

Как и другие компоненты, службы имеют четко описанный интерфейс, выполняющий определенный контракт. Именно контракт фиксирует выполняемую службой функцию.

Контракт службы не должен зависеть от платформ, операционных систем и языков программирования, с помощью которых эта служба может быть реализована. Интерфейс и реализация службы строго отделяются друг от друга.

Службы вызываются асинхронно. Ждать или нет результатов вызова службы, решает сам клиент, обратившийся к ней.

Службы совершенно независимы друг от друга, могут быть вызваны в произвольных комбинациях и всякий раз работают в соответствии со своим контрактом.

Контракт службы не должен зависеть от истории обращений к ней или к другим службам. Но он может зависеть, например, от состояния каких-либо хранилищ данных.

Обращение к службе возможно с любой платформы, из любого языка программирования, с любой машины, имеющей какую-либо связь с той, на которой размещается реализация службы. Реализации служб должны предоставлять возможность обратиться к ним и динамически обнаружить их в сети, а также предоставлять необходимую дополнительную информацию, с помощью которой можно понять, как с ними работать.

Архитектура приложений, построенных из компонентов, являющихся такими службами, называется **архитектурой, основанной на службах (service oriented architecture,** SOA).

Практически единственным широко используемым видом служб являются Web-службы. Web-службами называют службы, построенные с помощью ряда определенных стандартных протоколов, т.е. использующие протокол SOAP и инфраструктуру Интернет для передачи данных о вызовах операций и их результатах, язык WSDL для описания интерфейсов служб и реестры UDDI для

регистрации служб, имеющихся в сети. Существуют многочисленные дополнительные технологии, протоколы и стандарты, относящиеся к другим аспектам работы Web-служб, однако они пока применяются гораздо реже. Web-службы могут быть реализованы на основе J2EE, .NET или других технологий.

Контрольные вопросы по теме №14:

1. Для чего создавалась среда Jakarta Struts?
2. Какие библиотеки включают в себя Java Server Faces (JSF)?
3. Что представляет собой объектно-реляционные преобразователи (object-relation mappers, ORM)?
4. Что представляет собой технология Java Data Objects (JDO)?
5. Что представляет собой среда Spring?
6. Перечислить основные отличительные признаки Web-службы или Web-сервисы (Web services)?
7. Расшифровать схему архитектуры приложений на основе Web-служб?

Лекция 15. Тема: Управление разработкой ПО

Цель занятия: Рассматриваются основные деятельности, входящие в компетенцию руководителей проектов. В общем рассказе о некоторых аспектах управления ресурсами, персоналом, рисками и коммуникациями проекта выделены особенности управления проектами по созданию ПО.

Задачи управления проектами

Эта лекция посвящена управлению проектами по разработке, поддержке или модификации программного обеспечения. С достаточно общих позиций можно считать задачей управления проектами эффективное использование ресурсов для достижения нужных результатов. Всегда нужно получить как можно более хорошие результаты, используя при этом как можно меньше ресурсов. При этом в первую очередь возникают два вопроса: что именно считается "хорошим" результатом проекта и чем, какими ресурсами, можно пользоваться для его достижения.

Чтобы ответить на вопрос о том, какими критериями руководствуются при оценке проектов, и чего нужно добиваться, надо рассмотреть его с разных аспектов.

Одним из самых важных критериев является **экономическая эффективность проекта**, т.е. отношение суммы доходов разного рода, полученных в его результате, ко всем затраченным ресурсам. К сожалению, эти доходы чаще всего невозможно определить заранее. Поэтому при оценках проекта вместо дохода от его результатов рассматривают качество создаваемого ПО во всех его аспектах, т.е. набор имеющихся функций, надежность работы, производительность, удобство для всех категорий пользователей, а также удобство расширения и внесения изменений. Характеристики качества должно соотноситься с требованиями рынка или заказчика и с уже имеющимися продуктами.

С более общих позиций, и экономические, и неэкономические показатели результативности проекта объединяют в понятие **ценностей**, создаваемых в его ходе. Эти ценности могут иметь различную природу в зависимости от проекта,

от организации, в рамках которой он проводится, от национально-культурных особенностей рынка и персонала и пр. Кроме того, ценности выстраиваются в иерархию в зависимости от уровней рассмотрения проектов.

В одном конкретном проекте основной ценностью может быть достижение запланированного качества результатов в указанный срок и в рамках определенного бюджета.

В то же время, могут быть получены и другие ценности: достигнута высокая сплоченность команды; новые члены коллектива приобретут серьезные знания и полезные навыки; команда овладеет новыми технологиями; ее члены получат повышение и/или поощрения, которые повысят их лояльность компании и т.п.

На уровне нескольких зависящих друг от друга проектов (такую группу проектов называют **программой**), в ходе которых создаются и дорабатываются несколько продуктов на единой платформе, а также могут оказываться различные услуги, связанные с этими продуктами, ценности связаны, прежде всего, с качеством общей архитектуры создаваемых продуктов.

В одном проекте работа иногда ведется по принципу "сдали и забыли", т.е. основные усилия направлены на то, чтобы заказчик подписал акт приемки работ или его аналог, после чего поставщик перестает отвечать за результаты, поэтому часто такой аспект качества ПО, как удобство внесения изменений, игнорируется. Однако для бизнеса организации в целом проведение таких проектов небезопасно. Среди исследователей и экспертов-практиков преобладает взгляд на любую программную систему как на систему **развивающуюся**, полезность которой значительно снижается, если нет возможности расширять ее, тем более — исправлять серьезные ошибки, которые всегда есть в сложных программах. Заказчик всегда сталкивается с проблемами поддержки ПО и рано или поздно столкнется и с необходимостью его развития. На уровне группы проектов игнорирование удобства модификации ПО, а также вопросов, связанных с организационными и экономическими последствиями из менений в общей архитектуре, просто губительно.

На уровне организации в целом или подразделения, в рамках которого может одновременно проводиться много проектов, связанных по предметной области, используемым технологиям и просто по вовлеченным в них людям, возникают другие ценности. Это может быть отлаженность производственных процессов, высокая технологическая экспертиза и технологическое лидерство в своей области, низкая текучка кадров, повышение оборота, прибыли, капитализации, доли продаж в рамках отрасли, занимаемого среди поставщиков такой же продукции места по экономическим и технологическим показателям.

Поскольку каждый проект проводится в рамках какой-то организации, то принятая в ней система ценностей влияет и на оценку каждого конкретного проекта (см. далее).

Основные виды ресурсов, используемых в любом проекте, следующие.

- Время.
- Бюджет.
- Персонал.

– Используемое оборудование, инструменты, материалы, и т.п.

Программные системы практически всегда уникальны. Каждая из них обладает своим набором характеристик (включая все реализуемые функции, производительность при их выполнении, все элементы пользовательского интерфейса и т.п), так или иначе отличающихся от характеристик других программ, даже делающих "то же самое". Если обладающая нужными свойствами (в том числе и подходящей ценой) программа уже имеется, незачем создавать ее заново — достаточно приобрести ее или взять ее код и скомпилировать. Поэтому практически каждая разрабатываемая программа уникальна — она должна иметь такие характеристики, которыми не обладает ни одна уже созданная.

Тем самым, почти каждый проект по разработке ПО включает элементы творчества, создания того, чего еще никто не делал. Крупные же проекты требуют решения сразу нескольких ранее не решенных задач. Управление проектами с элементами творческой деятельности очень сильно отличается от управления проектами, в которых заранее ясно, что именно надо делать и как.

Другое следствие этой уникальности ПО — отсутствие стандартных процессов разработки. Нет целостных подходов к созданию ПО, которые годились бы для всех случаев, а не только для определенного вида проектов. Кроме того, для хорошо определенных процессов, таких как RUP, XP, Microsoft Solution Framework или DSDM (Dynamic Systems Development Method, Метод разработки динамичных систем), недостаточно четко очерчены области их применимости. Каждый раз менеджеру проекта приходится только на основании своего опыта и советов экспертов принимать решение о том, какой процесс разработки использовать и как его модифицировать для достижения большей эффективности в конкретном проекте.

Есть много аргументов в пользу того, что программный код является **проектом**, а не **конечным продуктом.**

При разработке ПО переход от проекта к продукту почти полностью автоматизирован — требуется лишь скомпилировать код и развернуть систему в том окружении, где она будет работать. А само программирование гораздо больше напоминает разработку проекта здания, чем его строительство по уже готовому проекту. То же, что в разработке ПО обычно называется проектом или дизайном, представляет собой лишь набросок окончательного проекта, определяющий основные его черты и требующий дальнейшей детализации. Таким образом, разработка программ отличается от других инженерных видов деятельности тем, что в основном состоит из проектирования, а не изготовления продукта.

Это еще одна причина того, что программирование всегда включает элемент творчества. Кроме того, проблемы, с которыми сталкивается руководитель проекта разработки ПО, гораздо более похожи на проблемы периода проектирования здания, самолета или корабля, чем на проблемы периода их постройки.

Среди аспектов окружения проекта, оказывающих на его ход существенное влияние, отметим структуру проводящей проект организации,

организационную культуру различных вовлеченных в проект организаций, которую руководителю проекта надо учитывать при выработке стратегии поведения, а также заинтересованных в проекте лиц.

Полномочия руководителя и ход проекта в значительной мере зависят от структуры организации, в рамках которой проводится проект, т.е. от тех правил, согласно которым в этой организации группируются ресурсы и происходит выделение ресурсов под проекты. Различают следующие структуры организаций [1].

Функциональная. В такой организации подразделения выделяются по их области деятельности или этапам производственных процессов — в ней есть финансовый, плановый, маркетинговый, опытно-конструкторский и производственный отделы. Проекты ведутся сотрудниками нескольких разных подразделений, а руководство проектом осуществляется за счет координации их деятельности, через руководителей соответствующих отделов. Руководитель проекта практически всегда член дирекции. Выделение ресурсов, необходимых проекту, должно осуществляться на уровне дирекции, которая дает поручения руководству отделов выделить соответствующую часть ресурсов.

Такая схема позволяет собрать вместе сотрудников, обладающих знаниями и умениями в одной области, и развивать их экспертизу. Она помогает выполнять очень крупные проекты. С другой стороны, она не слишком гибка и предполагает высокую косвенность управления проектом и ограниченность общего количества проектов, проводимых организацией.

Проектная. В организации такого типа подразделения выделяются для проведения конкретных проектов. Руководитель такого временного подразделения является руководителем соответствующего проекта и полностью распоряжается выделенными для него ресурсами.

Эта схема обладает высокой гибкостью и приспособляемостью под нужды проекта, но может требовать дополнительных усилий для составления проектной команды, поскольку слабо мотивирует развитие персонала.

Продуктовая. Подразделения такой организации отвечают за разработку, развитие и поддержку определенных продуктов или семейств близких продуктов. В каждом таком подразделении может одновременно выполняться несколько проектов, связанных с данным продуктом. Руководителями проектов обычно являются сотрудники этого отдела, которые вполне распоряжаются выделенными для проекта ресурсами.

Продуктовая схема позволяет дополнить гибкость и простоту управления проектами в проектной схеме легкостью подбора подходящего персонала. Недостатком ее может являться выработка слишком узкой специализации у сотрудников и трудности расформирования большого подразделения при отказе от продолжения работ над некоторым продуктом.

Ориентированная на клиента. Подразделения таких организаций формируются для удовлетворения нужд крупных клиентов или групп клиентов. Проекты для такого клиента ведутся внутри соответствующего подразделения.

Эта схема позволяет уменьшить усилия, необходимые для понимания нужд клиентов. В целом она похожа на продуктовую, но при возникновении нужды в

новом продукте может осложнить подбор персонала в соответствующую группу.

Территориальная. Подразделения формируются согласно географическому положению. Проекты бывают локальными, целиком проводящимися в рамках одного подразделения, или распределенными — включающими ресурсы нескольких подразделений.

В ее рамках удобнее проводить локальные проекты, а распределенные всегда требуют дополнительных усилий по координации работ.

Матричная. Это гибрид нескольких схем, обычно проектной или продуктовой и функциональной. В такой организации есть и функциональные подразделения, в которых группируются ресурсы, и проектные группы, формируемые под конкретный проект из служащих функциональных подразделений. Ресурсы проекта передаются в соответствующую группу, и ими распоряжается руководитель проекта. Руководители функциональных подразделений, тем не менее, могут даже во время проекта иметь определенную власть над своими подчиненными.

Эта схема может сочетать достоинства функциональной и проектной, но может и порождать проблемы, связанные с двойной подчиненностью участников проектных групп и разницей между возложенной на них ответственностью и предоставленными полномочиями.

Контрольные вопросы по теме №15:

1. Что такое экономическая эффективность проекта?
2. Перечислить основные виды ресурсов, используемых в любом проекте?
3. Перечислить структуру организации–исполнителя проекта?
4. Перечислить виды организационной культуры организации-исполнителя?
5. Перечислить роли участников или заинтересованных лиц (stakeholders) в проекте?
6. Перечислить Виды деятельности, входящие в управление проектом?
7. Что такое метрика ПО?

Лабораторная работа № 1
Тема: Директивно - диалоговое форма взаимодействия с программной системой.

Интерфейс командной строки (Command Line Interface - CLI).

Цель работы: Изучение и приобретение навыков разработки директивно-диалоговых форм взаимодействия с программной системой на основе командных файлов.

В данной работе на примере командных файлов рассматривается командно-директивная форма взаимодействия. Данная форма диалогового взаимодействия, как правило, предназначена для подготовленного пользователя и требует знания алгоритмов выполнения программы, так и отдельных команд и их параметров. Запуск программ или выполнение отдельных директив проводится с командной строки.

В диалоговом взаимодействии пользователя с программной системой выделяются 2 типа сообщений: входные сообщения, порождаемые

пользователем с помощью средств ввода информации и выходные сообщения, формируемые системой с помощью средств вывода и отображения информации. Первый шаг диалога чаще всего начинается с выдачи системой одного или нескольких выходных сообщений. Выходные сообщения, как правило, отражают результаты выполнения процедурной части, либо состояние системы и диалога Последовательности диалога в свою очередь, делятся на последовательности, где инициатива может принадлежать системе и пользователю. Существует также и третий тип инициативы –смешанная инициатива, предполагающая периодическое перераспределение инициативы с помощью управляющих сигналов. Директивная форма взаимодействия требует определенных знаний системы и управляющих команд операционной системы.

Задание к работе

Выполнение данной работы состоит в создании четырех командных файлов, реализующих директивно-диалоговую форму взаимодействия пользователя с программной системой.

При выполнении работы необходимо создать следующие командные файлы (bat-файлы):

вариант - с параметрами символами;

вариант - с параметрами, использующие имена файлов;

вариант - с использованием внешней команды “CHOICE” (директивно-диалоговая форма взаимодействия).

вариант – с использованием нескольких параметров.

Количество используемых параметров и функции, исполняемые командными файлами, выбираются самим обучающимся.

Рассмотрим примеры таких файлов.

1 вариант.

Здесь при использовании параметра “p” производится просмотр содержимого каталога “stud”.

```
@echo off
if -%1==- goto konez
if %1== p goto work
: work
dir c:\stud
pause
goto konez
: konez
echo Вы хотите закончить работу?
pause
goto exit
: exit
```

2 вариант

В этом случае при использовании в качестве параметра существующего файла a.txt производится просмотр содержимого этого файла.

```
@echo off
if -%1==- goto konez
```

```
if not exist %1 goto mess
echo просмотр файла
type %1
pause
goto konez
: mess
echo файл не найден
pause
goto konez
: konez
echo Вы хотите закончить работу?
pause
goto exit
: exit
```

3 вариант.

В этом примере рассматривается диалогово-директивная форма взаимодействия, в данном случае командный файл выполняет следующие функции: при выборе альтернативы “y” просматривается каталог “stud”, при выборе альтернативы “n”- выход из программы, если же пользователь не производит никакого выбора, ПЭВМ через 10 секунд сама осуществляет выбор, в данном случае производится завершение работы.

```
@echo off
echo Если вы хотите просмотреть каталог “stud” нажмите “Y”
choice /c:YN /t:N,10
if errorlevel 2 goto vyhod
dir C:\stud
pause
goto exit
: vyhod
echo Конец работы
pause
:exit
```

4 вариант (несколько параметров)

```
@ECHO OFF
if /%1==/ goto konez
if not exist %1 goto error
if %2==u goto udal
if %2==c goto copu
if /%2==/ goto konez
:copu
md AAA
echo Katalog cozdan
pause
copy %1 AAA
pause
```

```
goto exit
:udal
del %1
pause
goto konez
:error
echo file ne ukazan
:exit
```

Задание

а) Привести краткие сведения о формах диалогового взаимодействия.
б) Привести основные сведения о командных файлах и их практическом использовании для организации диалога пользователя с программной системой.
в) Привести тексты созданных bat-файлов.

Содержание отчета

1. Цель и задание к лабораторной работе.
2. Результаты работы программы.
3.Аналитические выводы.

Лабораторная работа № 2
Тема: Пользовательские интерфейсы на основе GUI

Цель: Освоить навыки создания и использования в приложениях элементов пользовательского интерфейса на основе GUI(Graphic User Interface).

Общая рекомендация: Перед выполнением работы внимательно прочитайте инструкции, приведенные ниже, а также посмотрите, как работают приложения, проанализируйте коды макросов.

Элементы пользовательского интерфейса GUI(Graphic User Interface)

Грамотно разработанное приложение или комплект процедур не только обладает интеллектуальными возможностями в области принятия решений и обеспечивает отлаженную работу циклов, но также обеспечивает интерактивное общение с пользователем. Приложение или процедура должна при необходимости выдавать экранные сообщения и предлагать пользователю ввести данные. Интерактивно взаимодействуя с программой, пользователь ощущает себя участником процесса и до некоторой степени способен управлять действиями программы, что обеспечит интерес пользователя к работе приложения и уменьшить таким образом вероятность ошибок.

Выведение сообщений на экран - один из лучших (и простых) способов обеспечения интерактивности общения с пользователем. Если операция может занять много времени, сориентируйте пользователя относительно времени и темпов выполнения операции. Если пользователь совершает ошибку (например, вводит неверное значение параметра для пользовательской функции), его следует вежливо уведомить о том, что в будущем лучше таких ошибок не допускать.

Вывод экранных сообщений

Каждое приложение должно иметь механизм вывода информации на экран, чтобы пользователь имел представление о работе программы, или чтобы просто проинформировать его об ошибке. Для общения с пользователем удобнее всего пользоваться функцией MsgBox.

Использование функции MsgBox

Проблема с использованием свойства StatusBar состоит в том, что отображенная на панели состояния информация может быть вообще не замечена пользователем, если он не представляет, для чего нужна панель состояния. Поэтому в случае необходимости можно использовать функцию MsgBox:

MsgBox(prompt, buttons, title, *helpFile, context)*

prompt — это сообщение, выводимое в диалоговое окно.

buttons — это числовое значение или константа, которые среди прочего указывают командные кнопки диалогового окна .Значение по. умолчанию равно нулю.

title — текст, который должен находиться в полосе заголовка. Если нет никакого заголовка, VBA использует заголовок Microsoft Excel.

helpFile — это текст, указывающий файл справки, содержащий тему пользовательской помощи. Если вводится helpFile, придется также включить и context. При вводе helpFile, в диалоговом окне появляется кнопка Help.

context — число, указывающее номер ремарки в helpFile.

Параметры функции MsgBox, как и всех других функций VBA, следует заключать в круглые скобки только когда используется возвращаемое значение функции

В случае длинных сообщений VBA разрывает текст внутри диалогового окна. Для создания собственных переносов строки используют функцию VBA Chr и символ возврата каретки (ASCI 13) после каждой строки, как показано в следующем примере: |

```
msgbox "First line" & Chr (13) & "Second line"
```

Определение стиля MsgBox

Диалоговое окно сообщения по умолчанию содержит только кнопку ОК. Можно включить в диалоговое окно другие кнопки и пиктограммы, используя различные значения для параметра buttons. В табл.указаны все доступные опции.

Таблица Опции параметра buttons функции MsgBox.

Константа Значение Описание

Кнопки

vbOKOnly 0 Выводит только кнопку ОК (по умолчанию).

vbOkCancel 1 Выводит кнопку Ок и Cancel.

vbAbortRetrylgnore 2 Выводит кнопки Abort, Retry, Ignore.

vbYesNoCancel 3 Выводит кнопки Yes, No, Cancel.

vbYesNo 4 Выводит кнопки Yes, No.

vbRetryCancel 5 Выводит кнопки Retry, Cancel.

Пиктограммы

vbCritical 16 Выводит пиктограмму Critical Message
vbQuestion 32 Выводит пиктограмму Warning Query
vbExclamation 48 Выводит пиктограмму Warning Message vbinformation 64 Выводит пиктограмму Information Message
Кнопки по умолчанию
vbDefaultButtonI 0 Первая кнопка является кнопкой по умолчанию.
vbDefaultButton2 256 Вторая кнопка является кнопкой по умолчанию.
vbDefaultButton3 512 Третья кнопка является кнопкой по умолчанию.

Получение возвращаемых значений MsgBox

Диалоговое окно сообщения, содержащее только кнопку ОК., является однонаправленным. Пользователь либо нажимает ОК., либо нажимает Enter, чтобы убрать диалоговое окно с экрана. Многокнопочные стили оформления диалоговых окон имеют некоторые отличия; пользователь в этом случае должен сделать выбор между кнопками, а процедура должна иметь указания по поводу того, что представляет собой выбор пользователя.

Это осуществляется записью возвращаемого значения функции MsgBox в виде переменной. В табл. перечислены семь вариантов записи.

Таблица Возвращаемые значения функции MsgBox.

Константа Значение Кнопка
vbOk 1 OK
vbCancel 2 Cancel
vbAbort 3 Abort
vbRetry 4 Retry
vblgnore 5 Ignore
vbYes 6 Yes
vbNo 7 No

Использование диалоговых окон

Многие методы VBA известны как эквиваленты диалоговых окон, поскольку они позволяют осуществлять выбор тех же опций, которые имеются во встроенных диалоговых окнах Access и Excel. Использование эквивалентов диалоговых окон становится эффективным в том случае, если известно, какие опции можно выбирать; однако иногда случается и так, что определить некоторые из опций диалогового окна нужно самому пользователю.

Если, например, процедура будет производить печать документа (используя метод PrintOut), может понадобиться узнать, сколько копий или сколько страниц документа пользователь желает распечатать. Для получения этой информации можно использовать метод InputBox, но, как правило, проще вывести на экран диалоговое окно Print.

Встроенные диалоговые окна являются Dialog-объектами, которые представляют собой группу из более чем 200 встроенных диалоговых окон. Для отсылки к определенному диалоговому окну используются заранее заданные константы.

Приводимая ниже таблица некоторые из наиболее употребляемых констант для встроенных диалоговых окон.

Таблица. Некоторые из констант для встроенных диалоговых окон Excel.
Константа Диалоговое окно xlDialogChartWizard ChartWizard xlDialogColumnWidth Column Width xlDialogDefineName Define Name
xlDialogFindFile Find File
xlDialogFont Font
xlDialogFormatAuto AutoFormat
xlDialogFormulaFind Find
xlDialogFormulaGoto Go To
xlDialogFormulaReplace Replace
xlDialogFunctionWizard Function Wizard .
xlDialogGoalSeek Goal Seek
xlDialogNew New
xlDialogNote Cell Note
xlDialogOpen Open
xlDialogOptionsCalculation Options (Calculation tab)
xlDialogOptionsEdit Options (Edit tab)
xlDialogOptionsGeneral Options (General tab)
xlDialogOptionsView Options (View tab)
xlDialogPageSetup Page Setup
xlDialogPasteSpecial Paste Special
xlDialogPivotTableWizard PivotTable Wizard
xlDialogPrint Print
xlDialogPrinterSetup Printer Setup
xlDialogPrintPreview Print Preview
xlDialogRowHeight Row Height
xlDialogSaveAs Save As
xlDialogSort Sort
Example
Application.Dialogs(xlDialogPrint).Show

В лабораторной работе необходимо использовать кнопки из панели инструментов "Формы" или "Элементы управления", которым нужно назначить макросы, выполняющие функции вызова функции MsgBox или диалоговых окон

Как создать макрос

Макрос — это относительно небольшая программа, содержащая набор команд (операторов Visual Basic for Application (VBA)), предназначенных для автоматизации решения различных задач в конкретном приложении. Подобно пакетным файлам DOS, макросы сводят несколько операций в единую процедуру, которую можно быстро активизировать. Для запуска макроса могут быть использованы различные графические объекты, кнопки на панели инструментов или сочетание клавиш. Макросы часто используются для следующих целей:

-ускорения часто выполняемых операций;

-объединения сложных команд;

-упрощения доступа к параметрам в окнах диалога;

-автоматизации обработки сложных последовательных действий в задачах.
Есть два способа создания макроса : можно автоматически записать последовательность своих действий или вручную ввести инструкции на особом листе MS EXCEL , называемом модулем.
Для изменения инструкций можно открыть макрос в окне редактирования макросов. После присвоения макросу кнопки панели инструментов, графического объекта, команды меню или сочетания клавиш, для его выполнения будет достаточно выбрать этот объект и активизировать его стандартным способом. При этом не теряется возможность выполнения макроса с помощью команд основного меню MS EXCEL.

Запись макроса

1. В меню *Сервис* выберите подменю *Макрос* и выберите команду *Запись*.
2. Введите имя для макроса в соответствующее поле.

Первым символом имени макроса должна быть буква. Остальные символы могут быть буквами, цифрами или знаками подчеркивания. В имени макроса не допускаются пробелы; в качестве разделителей слов следует использовать знаки подчеркивания.

3. Чтобы выполнить макрос с клавиатуры с помощью сочетания клавиш, введите соответствующую букву в поле *Сочетание клавиш*. Для строчных букв используется сочетание *CTRL+ буква*, а для заглавных — *CTRL+SHIFT+ буква*, где буква — любая клавиша на клавиатуре. Буква, используемая в сочетании клавиш, не может быть цифрой или специальным символом. Заданное сочетание клавиш будет заменять любое установленное по умолчанию в Microsoft Excel, пока книга, содержащая данный макрос, открыта.
4. В поле *Сохранить* в книге выберите книгу, в которой должен быть сохранен макрос.

Чтобы макрос был доступен независимо от того, используется ли в данный момент Microsoft Excel, его следует сохранить в личной книге в папке XLStart.
Чтобы создать краткое описание макроса, введите необходимый текст в поле "О*писание*".

5. Нажмите кнопку *ОК*.

По умолчанию, при записи макроса используются абсолютные ссылки. Макрос, записанный с абсолютными ссылками, при выполнении всегда обрабатывает те же ячейки, которые обрабатывались при его записи. Для того, чтобы с помощью макроса обрабатывать произвольные ячейки, следует записать его с относительными ссылками. Для этого нажмите кнопку *Относительная ссылка* на панели инструментов *Остановка записи*. Относительные ссылки будут использоваться до конца текущего сеанса работы в Microsoft Excel или до повторного нажатия кнопки *Относительная ссылка*.

6. Выполните макрокоманды, которые нужно записать.
7. Нажмите кнопку *Остановить запись*

на соответствующей панели инструментов.

Выполнение макроса в Microsoft Excel.

1. Откройте книгу, которая содержит макрос.
2. В меню *Сервис* установите указатель на пункт *Макрос* и выберите команду Макросы.
3. В поле *Имя* макроса введите имя того макроса, который нужно выполнить.
4.Нажмите кнопку *Выполнить*.

Задание

Разработать 2 приложения в MS EXCEL и в MS WORD с элементами GUI. Обязательными компонентами являются объекты DialogSheet или UserForm для приложения в MS EXCEL и все элементы из опции ФОРМЫ панели инструментов MS WORD.

Содержание отчета

1. Цель и задание к лабораторной работе.
2. Результаты работы программы.
3.Аналитические выводы.

Лабораторная работа № 3
Тема: Компоненты пользовательского интерфейса на основе WUI (Web user interface)

Цель: Отработать навыки создания в HTML-документе компонентов, позволяющих создавать интерактивное взаимодействие пользователя с WWW-сервером (HTTP - сервер).

Программное обеспечение: стандартное (HTML-броузер).

Пояснения: Компоненты пользовательского интерфейса на основе WUI обеспечивают взаимодействие пользователя в сетевых программных приложениях (например в Интернет).К одной из важнейших функций WEB-страниц (при наличии объектов WUI), помимо непосредственного отображения информации для пользователей, относится возможность посылать на WEB - узел определенные данные и производить их обработку на сервере. Для этих целей в код страницы включаются специальные тэги, определяющие в HTML - странице специальные объекты-формы, с помощью которых можно создавать интерактивный интерфейс.

При создании формы ее содержимое заключается между тегами <FORM></FORM>. После открывающегося тега <FORM>, может быть указан сценарий или программа, которая будет обрабатывать запрос. Это определяется с помощью атрибута ACTION.

Например:

<FORM ACTION="/cgi-bin/primer.pl"> -для CGI

<FORM ACTION="/primer.php">

С помощью атрибутов METHOD определяется способ передачи данных на сервер.

Например:

<FORM ACTION="/cgi-bin/primer.pl" METHOD = "POST" >

<FORM ACTION="/primer.php" METHOD = "GET" >

При использовании метода GET информация из формы добавляется в конец URL, который был указан в описании заголовка формы. CGI-программа (CGI-скрипт) получает данные из формы в виде параметра переменной среды QUERY_STRING. При POST вся информация о форме передается после обращения к указанному URL и CGI-программа получает данные из формы в стандартный поток ввода (STDIN). Сервер не пересылает сообщение об окончании пересылки данных. Вместо этого, используется переменная окружения CONTENT_LENGTH для определения количества переданных данных. Данные считываются из стандартного потока ввода.(Программный интерфейс взаимодействия по протоколу HTTP броузер-сервер будет рассмотрен более подробно в последующих лабораторных работах,рекомендуется разработанные по этому заданию программы сохранить для дальнейшего использования)

Когда пользователь заполняет форму и активизирует кнопку SUBMIT - специальный тип кнопки, информация, введенная пользователем в форму, посылается HTTP-серверу для обработки. Рассмотрим элементы формы, обеспечивающие ввод данных и отправку их на сервер. Основные элементы формы определяются атрибутами шести типов:

- Кнопки
- Однострочное текстовое поле
- Текстовые блоки
- Меню
- Флажки
- Переключатели

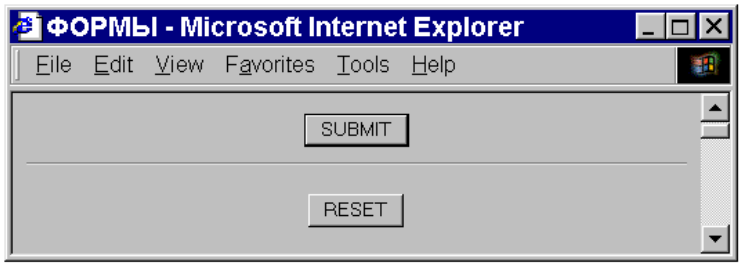
Кнопка Submit (Отправить) используется для передачи всех вводимых данных из полей формы.

<INPUT TYPE = "SUBMIT" VALUE = "SUBMIT" NAME = "SUBMIT">

Кнопка RESET (Сброс) используется для очистки полей формы.

<INPUT TYPE = "RESET" VALUE = "RESET" NAME = "RESET">

Здесь VALUE - надпись на кнопках , NAME - имя, которое передается сценарию.

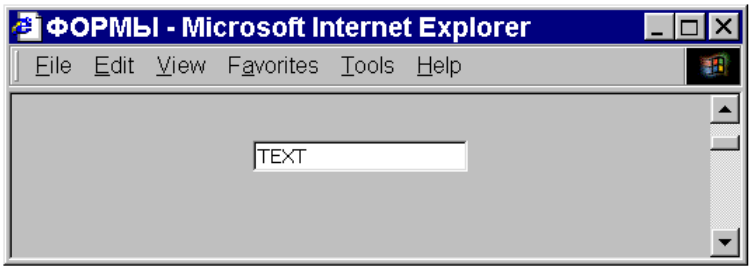


Вид кнопок в броузере

Однострочное текстовое поле.

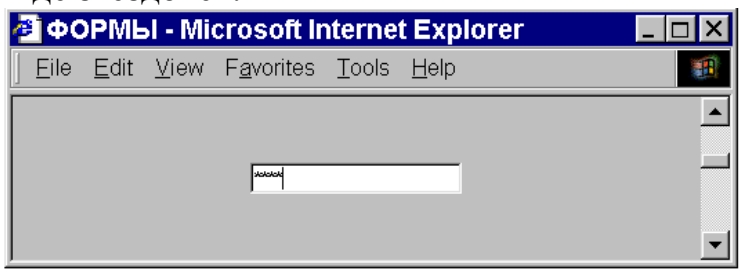
Пример.

<INPUT TYPE = "TEXT" NAME = "NAME" VALUE = "TEXT"
SIZE = "20" MAXLENGTH = "30" >



Вид в броузере

Здесь атрибут TYPE = "TEXT" - указывает броузеру, что это однострочное текстовое поле; VALUE - может содержать некоторый текст в поле ввода ; NAME - имя, которое передается сценарию в качестве уникального идентификатора; MAXLENGTH - указывает максимальное число символов, которое можно вводить в текстовое поле; SIZE - устанавливает значение ширины поля в символах. Если TYPE = "PASSWORD" все вводимые данные отображаются в виде звездочек.

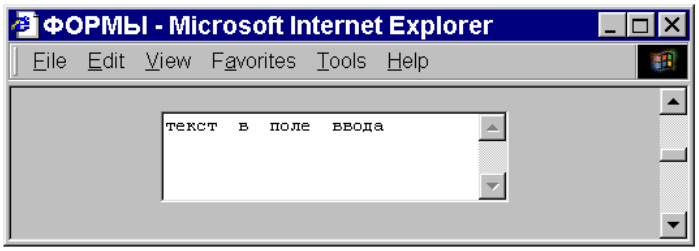


Вид в броузере

Если TYPE = "HIDDEN", поле данного типа не отображается броузером и не дает пользователю возможность изменять присвоенные данному полю значение. Это поле используется для передачи в CGI-программу статической информации, например пароля или другой информации.

Текстовые блоки. Прокручиваемое текстовое поле.

<TEXTAREA NAME = "INFORMATION" ROWS = "4" COLS = "30" WRAP = "virtual">текст в поле ввода</ TEXTAREA >



Вид в броузере

Здесь NAME - имя, которое передается сценарию в качестве уникального идентификатора; ROWS - определяет значение высоты поля в виде количества строк, которые будут отображаться на экране одновременно (до приведения в действие механизма строк). COLS - определяет ширину поля в символах. WRAP = " virtual ", в этом случае текст будет заполнятся построчно. По достижении предельного значения длины строки, указанной атрибутом COLS, текст будет переходить на новую строку автоматически. Переход на новую строку возможен также с помощью клавиши "ENTER". Наличие символов

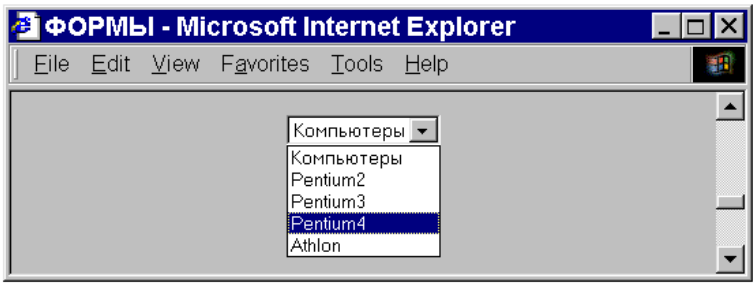
между тегами <TEXTAREA ></ TEXTAREA > указывает, что в прокручиваемом поле имеется заранее введенный текст.

Меню.

Данный вид поля формы отображается в виде однострочного поля с небольшой стрелкой в правой части. Если щелкнуть по стрелке, развернется все меню (пример 1).

Пример 1.

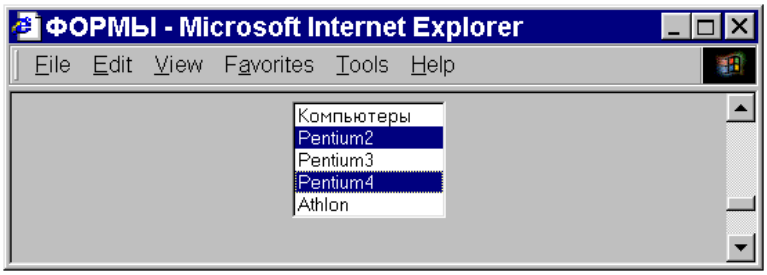
```
<SELECT NAME = "NAME" SIZE = "1">
<OPTION SELECTED VALUE = "Pentium3">Компьютеры</OPTION>
<OPTION VALUE = "Pentium2"> Pentium2</OPTION>
<OPTION VALUE = "Pentium3"> Pentium3</OPTION>
<OPTION VALUE = "Pentium4"> Pentium4</OPTION>
<OPTION VALUE = "Atlon"> Atlon</OPTION>
</ SELECT >
```



Вид в броузере

Пример 2 (применение атрибута MULTIPLE).

```
<SELECT NAME = "NAME" SIZE = "5" MULTIPLE>
<OPTION SELECTED VALUE = "Pentium3">Компьютеры</OPTION>
<OPTION VALUE = "Pentium2"> Pentium2</OPTION>
<OPTION VALUE = "Pentium3"> Pentium3</OPTION>
<OPTION VALUE = "Pentium4"> Pentium4</OPTION>
<OPTION VALUE = "Atlon"> Atlon</OPTION>
</ SELECT >
```



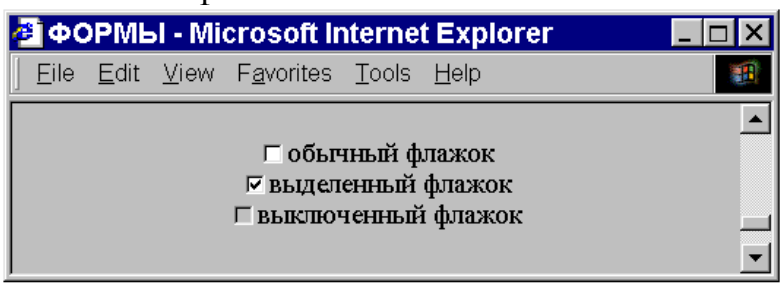
Вид в броузере

Здесь атрибут SELECT указывает броузеру, что следует создать окно меню, атрибут NAME используется в качестве идентификатора данного поля ввода данных, атрибут SIZE указывает сколько будет отображаться строк сначала. Тег OPTION используется для объявления каждой опции, которую необходимо поместить в меню. Атрибуту VALUE присваивается идентификатор для конкретного варианта опции. Атрибут MULTIPLE разрешает выбрать более одной опции меню.

Флажки. Данный тип элементов формы позволяет пользователю выделить несколько опций в наборе флажков.

Пример.

<INPUT TYPE = "CHECKBOX" NAME = "CHECKBOX_1" VALUE = "ON">обычный флажок
<INPUT TYPE = "CHECKBOX" NAME = "CHECKBOX_2" VALUE = "ON" CHECKED>выделенный флажок
<INPUT TYPE = "CHECKBOX" NAME = "CHECKBOX_3" VALUE = "ON" DISABLED>выключенный флажок



Вид в броузере

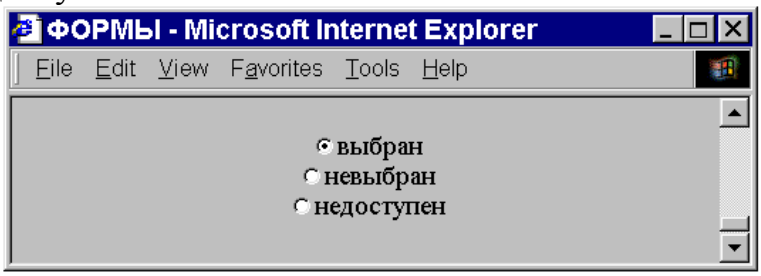
Здесь TYPE = "CHECKBOX" определяет тип элемента формы. Атрибут NAME используется в качестве уникального идентификатора при передаче данной информации сценарию. Если указан атрибут CHECKED для поля INPUT, то он будет иметь статус выбранного. Если указан атрибут DISABLED для поля INPUT, то пользователь не может установить этот флажок. VALUE = "ON" назначается переменной определенной в NAME и затем передается сценарию.

Переключатели.

Этот тип элементов формы позволяет пользователю выбрать только одну опцию из предлагаемого набора.

Пример.

<INPUT TYPE = "RADIO" VALUE = "FALSE" NAME = "CHECK" CHECKED>выбран
<INPUT TYPE = "RADIO" VALUE = "TRUE" NAME = "CHECK">невыбран
<INPUT TYPE = "RADIO" VALUE = "TRUE" NAME = "CHECK" DISABLED>недоступен



Вид в броузере

Здесь TYPE = "RADIO" определяет тип элемента формы. Атрибут NAME используется в качестве уникального идентификатора при передаче данной информации сценарию. Переключатель автоматически устанавливается, если указан атрибут CHECKED. Если указан атрибут DISABLED для поля INPUT,

то пользователь не может установить этот переключатель. Значение в VALUE назначается переменной определенной в NAME и затем передается сценарию.

Задание

1. Составить код HTML-страницы с включением тегов FORM со всеми компонентами, которые формируют интерактивный интерфейс пользователя с программным приложением.

2. Составить код HTML-страницы с определенной смысловой нагрузкой, выбрать компоненты форм, определяющие вводимую информацию.

Содержание отчета

1. Цель и задание к лабораторной работе.
2. Листинг программы.
3. Результаты работы программы.
4.Аналитические выводы.

Лабораторная работа № 4

Тема: Пользовательский интерфейс на основе "Hand User Interface".

Цель: Создание дружественного интерфейса с помощью HTML.

ВведениеДанный вид пользовательского интерфейса используется в основном для карманных компьютеров, для которых характерен дисплей небольшого размера. В таких системах применяются объекты пользовательского интерфейса учитывающие эту особенность.

Данная работа посвящена отработке навыков использования гиперссылок в документах в формате HTML. Гиперссылки являются компонентами пользовательского интерфейса в определенной степени учитывающие требования к проектированию "Hand User Interface".

Инструктивные материалы и краткая теоретическая часть.

Термин “гипертекстовый” означает, что такой документ состоит из нескольких относительно самостоятельных частей. Последовательность переходов от одной части к другой определяется двумя обстоятельствами:

• организацией логической связи между частями документа, которая устанавливается его создателем;

• интересами пользователя, который может пользоваться имеющимися ссылками в произвольном порядке.

Благодаря этому свойству гипертекст позволяет заменить жесткую линейную последовательность просмотра информации, характерную для других форм электронных документов, гибким алгоритмом, напоминающим работу с печатными изданиями, но значительно более эффективным в реализации.

Во многих случаях гипертекстовый документ имеет полносвязную структуру, то есть от одной его части можно перейти за один или несколько шагов к любой другой его части.

Например, на рис. 1. показана структура гипертекстового документа, описывающего работу кухонного комбайна.

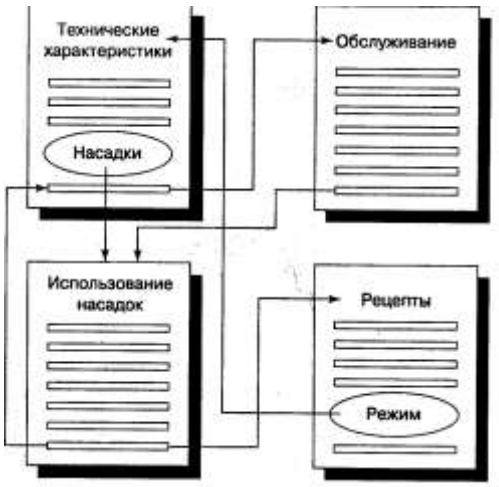


Рис. 1. Пример структуры гипертекстового документа

Используя ссылки, пользователь может ознакомиться с соответствующей информацией как последовательно **(“Технические характеристики” — “Обслуживание” — “Насадки” — “Рецепты”),** так и “перескакивая” с одной страницы на другую, чтобы уточнить тот или иной момент.Связь между частями документа реализуется посредством так называемых гиперссылок.

Гиперссылка **(Hypertext Reference)** — это интерактивная область документа, щелчок мышью на которой приводит к выполнению заданной операции перехода. Переход может выполняться как внутри текущего документа, так и на любой другой информационный ресурс.

В качестве гиперссылки может использоваться слово, фраза или некоторый графический элемент документа. В связи с этим необходимо отметить, что понятие “гипертекстовый” совершенно не ограничивает содержимое документа только текстовой информацией. Его компоненты могут быть реализованы в виде графических изображений, видеоклипов или звукового ряда. Такой широкий диапазон форм представления информации возможен благодаря особенностям языков гипертекстовой разметки.

Язык гипертекстовой разметки, или язык разметки документов **(Markup Language),** — это специальный язык программирования, предназначенный для описания структуры информационного наполнения документа. Другими словами, такой язык позволяет указать, что вот здесь, например, должен быть текст, здесь — картинка, а вот там должна появляться реклама.

Таким образом, любой гипертекстовый документ, описанный с помощью языка разметки, представляет собой программу, результатом выполнения которой является отображение информационной части документа на экране монитора.

С точки зрения пользователя гиперссылка представляет собой интерактивную область документа, обеспечивающую динамический переход между его частями. Однако каждая гиперссылка имеет и оборотную сторону: для создателя HTML-документа это прежде всего адрес ресурса , включенного им в состав документа.

Необходимо отметить, что само понятие ресурса имеет достаточно абстрактный характер. В общем случае его используют для обозначения той информации или данных, которые представляют (или могут представлять)

интерес для пользователя. Соответственно, объем и “способ существования” ресурса могут изменяться в очень широком диапазоне. Например, если посетитель Интернета — любитель футбола, то для него ресурсом будет Web-сайт, посвященный ходу чемпионата Европы по этому виду спорта, если же посетитель сам является Web-дизайнером, то его может интересовать один-единственный файл, содержащий описание какого-нибудь необычного элемента страницы. Общим для всех ресурсов является то, что каждый из них имеет адрес, однозначно идентифицирующий его среди других ресурсов. Адрес ресурса, представленный в символьном виде, называется **Uniform Resource Locator** (универсальный указатель ресурса), сокращенно URL. Поскольку физическим носителем (точнее, хранителем) ресурса является компьютер, то основу URL составляет доменное имя этого компьютера. Однако для обращения к ресурсу-файлу требуется учитывать организацию файловой системы компьютера. Поэтому URL может быть дополнен описанием маршрута доступа к необходимому файлу. Очень часто наряду с собственно адресом ресурса URL содержит также наименование протокола, который должен использоваться при работе с этим ресурсом. Таким образом, в общем виде структуру URL можно представить так :

[тип протокола]://[доменное имя компьютера]/[маршрут доступа].

В лабораторной работе мы будем использовать усеченное понятие URL, т.к. работа выполняется на локальном компьютере, это будет HTML-файл находящийся на другом диске (логическом или физическом) или, даже документ в другом (не текущем) каталоге.

Иллюстративный пример навигационной схемы по группе HTML-файлов приводится на рис.2

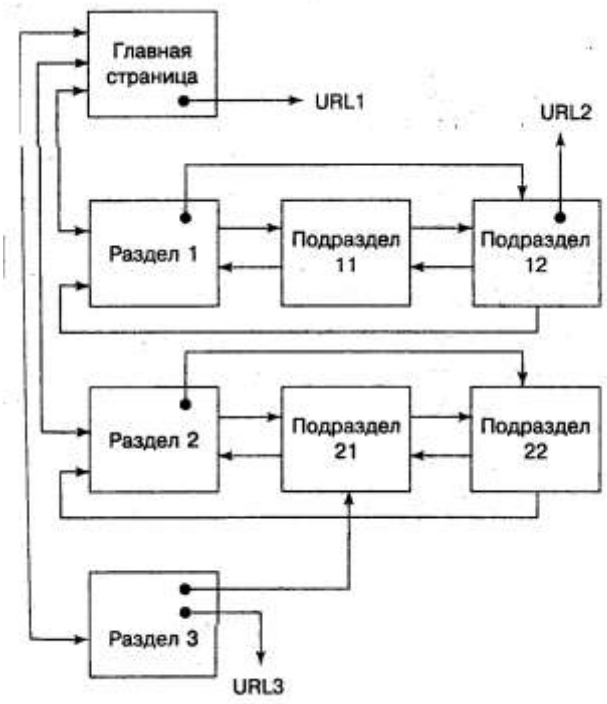


рис.2

Кратко о HTML?

HTML не является языком программирования, это язык разметки текстовых документов. В нем необязательно точное соответствие синтаксису, объявление переменных, описание процедур и классов и прочих атрибутов "обычных" языков программирования. Если вы не укажете такой важный элемент как TITLE или BODY, то броузер просто будет использовать соответствующее значение по умолчанию. Если вы сделаете синтаксическую ошибку, самое страшное, что случится - искажение вида документа в окне броузера, при котором ошибку легко найти и исправить.

Термин HTML (HyperText Markup Language) означает "язык маркировки гипертекстов". Первую версию HTML разработал сотрудник Европейской лаборатории физики элементарных частиц Тим Бернерс-Ли. Со времени создания первой версии HTML претерпел некоторые изменения.

Для освоения HTML вам понадобятся две вещи:

1. Любой браузер, т.е., программа, пригодная для просмотра HTML-файлов - Internet Explorer или Netscape Navigator.
2. Любой редактор текстовых файлов, поддерживающий русский язык в выбранной Вами кодировке. Если на Вашем компьютере установлен Windows, вполне подойдет Notepad или Блокнот.

Мы будем использовать текстовый редактор для подготовки HTML-файлов, а браузер — как инструмент контроля за сделанным. Свои первые HTML-файлы Вы будете разрабатывать у себя на локальном диске. Другими словами, компьютер, на котором Вы будете заниматься, может и не иметь подключения к Интернет

Как устроен HTML-документ?

HTML-документ — это просто текстовый файл с расширением *.htm, *.html. Вот самый простой HTML-документ:

```
<html>
<head>
<title>
Пример 1
</title>
</head>
<body>
<H1>
Привет!
</H1>
<P>
Это простейший пример HTML-документа.
</P>
<P>
Этот *.htm-файл может быть одновременно открыт
и в Notepad (Блокноте), и в Internet Explorer. Сохранив изменения в Notepad(Блокнот),
просто нажмите кнопку Reload ('обновить') в Internet Explorer ,
```

чтобы увидеть эти изменения реализованными в HTML-документе.
</P>
</body>
</html>
Для удобства чтения здесь введены дополнительные отступы, однако в HTML это не обязательно. Более того, браузеры просто игнорируют символы конца строки и множественные пробелы в HTML-файлах. Поэтому наш пример вполне мог бы выглядеть и вот так:

```
<html>
<head>
<title>Пример 1</title>
</head>
<body>
<H1>Привет!</H1>
<P>Это простейший пример HTML-документа.</P>
<P>Этот *.htm-файл может быть одновременно открыт и в Notepad (Блокноте), и в Internet Explorer или Netscape.
Сохранив изменения в Notepad, просто нажмите кнопку Reload ('обновить')
в Netscape, чтобы увидеть эти изменения реализованными в HTML-документе.</P>
</body>
</html>
```

Как видно из примера, вся информация о форматировании документа сосредоточена в его фрагментах, заключенных между знаками "<" и ">". Такой фрагмент (например, <html>) называется меткой (по-английски — tag, читается "тэг"). Большинство HTML-меток — парные, то есть на каждую открывающую метку вида <tag> есть закрывающая метка вида </tag> с тем же именем, но с добавлением "/". Метки можно вводить как большими, так и маленькими буквами. Например, метки <body>, <BODY> и <Body> будут восприняты браузером одинаково. Многие метки, помимо имени, могут содержать атрибуты — элементы, дающие дополнительную информацию о том, как браузер должен обработать текущую метку. В нашем простейшем документе, однако, нет ни одного атрибута.

Обязательные метки

<html> ... </html>
Метка <html> должна открывать HTML-документ. Аналогично, метка </html> должна завершать HTML-документ.
<head> ... </head>
Эта пара меток указывает на начало и конец заголовка документа. Помимо наименования документа (см. описание метки <title> ниже), в этот раздел может включаться множество служебной информации, о которой мы обязательно поговорим чуть позже.
<title> ... </title>

Все, что находится между метками <title> и </title>, толкуется браузером как название документа. Internet Explorer , например, показывает название текущего документа в заголовке окна и печатает его в левом верхнем углу каждой страницы при выводе на принтер. Рекомендуется название не длиннее 64 символов.

<body> ... </body>

Эта пара меток указывает на начало и конец тела HTML-документа, каковое тело, собственно, и определяет содержание документа.

<H1> ... </H1> — <H6> ... </H6>

Метки вида <Hi> (где i — цифра от 1 до 6) описывают заголовки шести различных уровней. Заголовок первого уровня — самый крупный, шестого уровня, естественно — самый мелкий.

<P> ... </P>

Такая пара меток описывает абзац. Все, что заключено между <P> и </P>, воспринимается как один абзац. Метки <Hi> и <P> могут содержать дополнительный атрибут ALIGN (читается "элайн", от английского "выравнивать"), например:

<H1 ALIGN=CENTER>Выравнивание заголовка по центру</H1>

или

<P ALIGN=RIGHT>Образец абзаца с выравниванием по правому краю</P>

Подытожим все, что мы знаем, с помощью примера 2:

```
<html>
<head>
<title>Пример 2</title>
</head>
<body>
<H1 ALIGN=CENTER>Привет!</H1>
<H2>Это чуть более сложный пример HTML-документа</H2>
<P>Теперь мы знаем, что абзац можно выравнивать не только влево, </P>
<P ALIGN=CENTER>но и по центру</P> <P ALIGN=RIGHT>или по правому краю.</P>
</body>
</html>
```

С этого момента Вы знаете достаточно, чтобы создавать простые HTML-документы самостоятельно от начала до конца. В следующем разделе мы поговорим, как можно улучшить наш простой HTML-документ. Начнем с малого — с абзаца.

Непарные метки

В этом разделе мы поговорим о метках, которые не подчиняются двум основным правилам HTML: все они непарные, а некоторые (так называемые &-последовательности) к тому же должны вводиться только маленькими буквами.

**
**

Эта метка используется, если необходимо перейти на новую строку, не прерывая абзаца. Очень удобно при публикации стихов.

<html>

<head>
<title>Пример 3</title>
</head>
<body>
<H1>Стихи</H1>
<H2>Автор</H2>
<P>Строка1
Строка2
Строка3
Строка4</P>
<P>Строка5
Строка6
Строка7
Строка7</P>
</body> </html>

<HR>

Метка <HR> описывает вот такую горизонтальную линию:

Метка может дополнительно включать атрибуты SIZE (определяет толщину линии в пикселах) и/или WIDTH (определяет размах линии в процентах от ширины экрана). В примере 4 приведена небольшая коллекция горизонтальных линий.

```
<html>
<head>
<title>Пример 4</title>
</head>
<body>
<H1>Коллекция горизонтальных линий</H1>
<HR SIZE=2 WIDTH=100%><BR>
<HR SIZE=4 WIDTH=50%><BR>
<HR SIZE=8 WIDTH=25%><BR>
<HR SIZE=16 WIDTH=12%><BR>
</body>
</html>
```

&-последовательности

Поскольку символы "<" и ">" воспринимаются браузерами как начало и конец HTML-меток, возникает вопрос: а как показать эти символы на экране? В HTML это делается с помощью &-последовательностей (их еще называют символьными объектами или эскейп-последовательностями). Браузер показывает на экране символ "<", когда встречает в тексте последовательность < (по первым буквам английских слов less than — меньше, чем). Знак ">" кодируется последовательностью > (по первым буквам английских слов greater than — больше, чем). Символ "&" (амперсанд) кодируется последовательностью & Двойные кавычки (") кодируются последовательностью " Помните: точка с запятой — обязательный

элемент &-последовательности. Кроме того, все буквы, составляющие последовательность, должны быть в нижнем регистре (т.е., маленькие). Использование меток типа " или & не допускается. Вообще говоря, &-последовательности определены для всех символов из второй половины ASCII-таблицы (куда, естественно, входят и русские буквы). Дело в том, что некоторые серверы не поддерживают восьмибитную передачу данных, и поэтому могут передавать символы с ASCII-кодами выше 127 только в виде &-последовательностей.

Форматирование шрифта

HTML допускает два подхода к шрифтовому выделению фрагментов текста. С одной стороны, можно прямо указать, что шрифт на некотором участке текста должен быть жирным или наклонным, то есть изменить физический стиль текста. С другой стороны, можно пометить некоторый фрагмент текста как имеющий некоторый отличный от нормального логический стиль, оставив интерпретацию этого стиля браузеру. Поясним это на примерах.

Физические стили

Под физическом стилем принято понимать прямое указание браузеру на модификацию текущего шрифта. Например, все, что находится между метками <B> и </B>, будет написано жирным шрифтом. Текст между метками <I> и </I> будет написан наклонным шрифтом. Несколько особняком стоит пара меток <TT> и </TT>. Текст, размещенный между этими метками, будет написан шрифтом, имитирующим пишущую машинку, то есть имеющим фиксированную ширину символа.

Логические стили

При использовании логических стилей автор документа не может знать заранее, что увидит на экране читатель. Разные браузеры толкуют одни и те же метки логических стилей по-разному. Некоторые браузеры игнорируют некоторые метки вообще и показывают нормальный текст вместо выделенного логическим стилем. Вот самые распространенные логические стили.

<EM> ... </EM>

От английского emphasis — акцент.

<STRONG> ... </STRONG>

От английского strong emphasis — сильный акцент.

<CODE> ... </CODE>

Рекомендуется использовать для фрагментов исходных текстов.

<SAMP> ... </SAMP>

От английского sample — образец. Рекомендуется использовать для демонстрации образцов сообщений, выводимых на экран программами.

<KBD> ... </KBD>

От английского keyboard — клавиатура. Рекомендуется использовать для указания того, что нужно ввести с клавиатуры.

<VAR> ... </VAR>

От английского variable — переменная. Рекомендуется использовать для написания имен переменных.

Пример

Подытожим наши знания о логических и физических стилях с помощью примера 5. Заодно Вы сможете увидеть, как Ваш браузер показывает те или иные логические стили.

```
<html>
<head>
<title>Пример 5</title>
</head>
<body>
<H1>Шрифтовое выделение фрагментов текста</H1>
<P>Теперь мы знаем, что фрагменты текста можно выделять
<B>жирным</B> или <I>наклонным</I> шрифтом. Кроме того, можно
включать в текст фрагменты с фиксированной шириной символа
<TT>(имитация пишущей машинки)</TT></P>
<P>Кроме того, существует ряд логических стилей:</P>
<P><EM>EM - от английского emphasis - акцент </EM><BR>
<STRONG>STRONG - от английского strong emphasis - сильный акцент
</STRONG><BR>
<CODE>CODE - для фрагментов исходных текстов</CODE><BR>
<SAMP>SAMP - от английского sample - образец </SAMP><BR>
<KBD>KBD - от английского keyboard - клавиатура</KBD><BR>
<VAR>VAR - от английского variable - переменная </VAR></P>
</body>
</html>
```

Создание внешних и внутренних ссылок (связывание). /данный раздел наиболее тесно связан с заданием лабораторной работы/

Как уже упоминалось в самом начале, сокращение HTML означает "язык маркировки гипертекстов". Прежде всего, что же такое гипертекст? В отличие от обыкновенного текста, который можно читать только от начала к концу, гипертекст позволяет осуществлять мгновенный переход от одного фрагмента текста к другому. Системы помощи многих популярных программных продуктов устроены именно по гипертекстовому принципу. При нажатии левой кнопкой мыши на некоторый выделенный фрагмент текущего документа происходит переход к некоторому заранее назначенному документу или фрагменту документа. В HTML переход от одного фрагмента текста к другому задается с помощью метки вида:

```
<A HREF="[адрес перехода]">выделенный фрагмент текста</A>
```

В качестве параметра [адрес перехода] может использоваться несколько типов аргументов. Самое простое — это задать имя другого HTML-документа, к которому нужно перейти. Например:

```
<A HREF="pr.htm">Перейти к оглавлению</A>
```

Такой фрагмент HTML-текста приведет к появлению в документе выделенного фрагмента Перейти к оглавлению, при нажатии на который в текущее окно будет загружен документ pr.htm. Обратите внимание: если в

адресе перехода не указан каталог, переход будет выполнен внутри текущего каталога. Если в адресе перехода не указан сервер, переход будет выполнен на текущем сервере. Из этого следует одно очень важное практическое соображение. Если Вы подготовили к публикации некоторую группу HTML-документов, которые ссылаются друг на друга только по имени файла и находятся в одном каталоге на Вашем компьютере, вся эта группа документов будет работать точно так же, если ее поместить в любой другой каталог на любом другом компьютере, на локальной сети или... в Интернет! Таким образом, у Вас появляется возможность разрабатывать целые коллекции документов без подключения к Интернет, и только после окончательной готовности, подтвержденной испытаниями, помещать коллекции документов на Интернет целиком. На практике, однако, часто бывает необходимо дать ссылку на документ, находящийся на другом сервере. Например, если Вы хотите дать ссылку на это руководство со своей странички, Вам придется ввести в свой HTML-документ примерно такой фрагмент:

<A HREF="http://www.KSU.com/home/pr.htm">ПРИМЕР</A>

При необходимости можно задать переход не просто к некоторому документу, но и к определенному месту внутри этого документа. Для этого необходимо создать в документе, к которому будет задан переход, некоторую опорную точку, или анкер. Разберем это на примере. Допустим, что необходимо осуществить переход из файла 1.htm к словам "Переход закончен" в файле 2.htm (файлы находятся в одном каталоге). Прежде всего, необходимо создать вот такой анкер в файле 2.htm:

<A NAME="AAA">Переход закончен</A>

Слова "Переход закончен" при этом никак не будут выделены в тексте документа. Затем в файле 1.htm (или в любом другом) можно определить переход на этот анкер:

<A HREF="2.htm#AAA">Переход к анкеру AAA</A>

Кстати говоря, переход к этому анкеру можно определить и внутри самого документа 2.htm — достаточно только включить в него вот такой фрагмент:

<A HREF="#AAA">Переход к анкеру AAA</A>

На практике это очень удобно при создании больших документов. В начале документа можно поместить оглавление, состоящее из ссылок на анкеры, расположенные в заголовках разделов документа. Во избежание недоразумений рекомендуется задавать имена анкеров латинскими буквами. Следите за написанием имен анкеров: большинство браузеров отличают большие буквы от маленьких. Если имя анкера определено как AAA, ссылка на анкер aaa или AaA не выведет Вас на анкер AAA, хотя документ, скорее всего, будет загружен корректно. Пока что мы обсуждали только ссылки на HTML-документы. Однако возможны ссылки и на другие виды ресурсов:

<A HREF="ftp://server/directory/file.ext">Выгрузить файл</A>

Такая ссылка, если ей воспользоваться, запустит протокол передачи файлов и начнет выгрузку файла file.ext, находящегося в каталоге directory на сервере server, на локальный диск пользователя.

<A HREF="mailto:user@mail.box">Послать письмо</A>

Если пользователь совершит переход по такой ссылке, у него на экране откроется окно ввода исходящего сообщения его почтовой программы. В строке To: ("Куда") окна почтовой программы будет указано user@mail.box.
Разберем все, что мы знаем о связывании, с помощью примера :

```
<HTML>
<HEAD>
<TITLE>Пример</TITLE>
</HEAD>
<BODY>
<H1>Связывание </H1>
<P>С помощью ссылок можно переходить к другим файлам
(например, к <A HREF="pr.htm">оглавлению этого
руководства</A>).</P>
<P>Можно выгружать файлы (например,
<A HREF="ftp://KSU.com/home/html-pr.doc">это
файл</A>) по FTP.</P>
<P>Можно дать пользователю возможность послать почту (например,
<A HREF="mailto:a@ksu.com">послать письмо</A>).</P>
</BODY>
</HTML>
```

Изображения в HTML-документе

Встроить изображение в HTML-документ очень просто. Для этого нужно только иметь это самое изображение в формате GIF (файл с расширением *.gif) или JPEG (файл с расширением *.jpg или *.jpeg) и одну строчку в HTML-тексте. Допустим, нам нужно включить в документ изображение, записанное в файл picture.gif, находящийся в одном каталоге с HTML-документом. Тогда строчка будет вот такая:

```
<IMG SRC="picture.gif">
```

Метка <IMG SRC="[имя файла]"> может также включать атрибут ALT="[текст]", например:

```
<IMG SRC="picture.gif" ALT="Картинка">
```

Встретив такую метку, браузер покажет на экране текст Картинка и начнет загружать на его место картинку из файла picture.gif. Атрибут ALT может оказаться необходимым для старых браузеров, которые не поддерживают изображений, а также на случай, если у браузера отключена автоматическая загрузка изображений (при медленном подключении к Интернет это делается для экономии времени). Файл, содержаций изображение, может находиться в другом каталоге или даже на другом сервере. В этом случае стоит указать его полное имя. Разберем все, что мы знаем об изображениях, с помощью примера

```
<HTML>
<HEAD>
<TITLE>Пример</TITLE>
</HEAD>
<BODY>
<H1>Изображения </H1>
```

<P>Изображение можно встроить очень просто: </P>
<P><IMG SRC="picture.gif"></P>
<P>Кроме того, изображение можно сделать "горячим", то есть осуществлять переход при нажатии на изображение:</P>
<P><A HREF="pr.htm"><IMG SRC="picture.gif"></A></P>
</BODY>
</HTML>
Обратите внимание на вторую часть примера. Если ссылка на изображение находится между метками <A HREF="..."> и </A>, изображение фактически становится кнопкой, при нажатии на которую происходит переход по ссылке.

Задание

Спроектировать и реализовать логически организованную группу гипертекстовых документов (HTML-файлов), систему ссылок внутри документов, представляющими пользователю удобную и интуитивно понятную навигацию по гипертексту.

Содержание отчета

1. Цель и задание к лабораторной работе.
2. Листинг программы.
3. Результаты работы программы.
4.Аналитические выводы.

Лабораторная работа № 5
Тема: Ввод и вывод данных. Запись программы JavaScript в HTML-странице.

Цель: ***ознакомить с рабочей области окна программы, научить вводить программу и протестировать ее.***

Теоретические сведения.

Запись программы JavaScript в HTML-странице.

Программы JavaScript можно записывать в любом месте HTML-страницы. В некоторых случаях целесообразно записать одну часть программы в одном месте (например, в заголовке - элементе <HEAD>), а другую - в другом (например, в конце HTML-страницы после тела документа - элемента <BODY> или в значении атрибута формы или еще где-нибудь). В соответствии со стандартом HTML, ярлыки <SCRIPT LANGUAGE = "JavaScript"> и </SCRIPT>, как не принадлежащие к HTML, игнорируются, но текст между этими ярлыками остается виден. Чтобы его скрыть, необходимо заключить этот текст в контейнер - комментарий HTML:

```
<SCRIPT LANGUAGE="JavaScript">
<!--
  Здесь пишется текст программы
-->
</SCRIPT>
```

Комментарии JavaScript бывают однострочные и многострочные. Таким образом, мы только что изучили первый оператор JavaScript - **однострочный**

комментарий. Многострочный комментарий начинается с последовательности символов "/*" и заканчивается теми же символами, написанными в обратном порядке: */".

```
<SCRIPT LANGUAGE="JavaScript">
<!--
/* - начало многострочного комментария
   Это уже многострочный комментарий
   - эта программа состоит из единственного комментария
   и ничего не делает
конец многострочного комментария - */
//-->
</SCRIPT>
```

Вывод текста в модальное окно

```
<SCRIPT LANGUAGE="JavaScript">
<!--
alert("Перва\я программа");
-->
</SCRIPT>
```

Эта программа выводит модальное окно (т.е. такое, которое невозможно закрыть, не выполнив требуемые в нем действия, в данном случае - не нажав кнопку "OK").

Общее задание.

Пример 1.

```
<html>
<head>
 <title>Вызов диалогового окна из onLoad</title>
 <meta content="text/html; charset=windows-1251">
 <script type="text/javascript">
 <!--
 function opendoc(){
 alert("\Д\и\а\л\о\г\о\в\о\е \о\к\н\о \в\ы\з\в\а\н\о")
  }
 // -->
</script>
</head>
<body onload="opendoc()">
<b>
Проверка вызова onLoad
</b>
</body>
</html>
```

Пример 2.

```
<html>
<head>
  <title>Вызов диалогового окна из функции</title>
```

```
 <meta content="text/html; charset=windows-1251">
 <script type="text/javascript">
 <!--
 function opendoc(){
 alert("Диалоговое окно вызвано при помощи кнопки");
  }
 // -->
 </script>
</head>
<body>
<b>
Проверка вызова функции при помощи кнопки
</b>
<form method="POST">
<input type="Button" name="BUTTON1" value="Нажми меня!"
onclick="opendoc()">
</form>
</body>
</html>
```

Контрольные вопросы и задания.

1. В этой лабораторной работе использован тег для размещения на странице кнопки. Как вы думаете, что означают его атрибуты value и onclick ?
2. Для чего предназначен метод alert().
3. Создайте HTML-страничку и разместите на ней описанные выше сценарии, придумав собственные запросы и сообщения, используя методы Confirm(),Promt().

Содержание отчета

1. Цель и задание к лабораторной работе.
2. Листинг программы.
3. Результаты работы программы.
4.Аналитические выводы.

Лабораторная работа №6
Тема: Типы данных. Переменные и операторы присваивания. Имена переменных. Создание переменных.

Цель работы: ознакомить с основными типами данных, операторами присваивания, арифметическими операциями, с понятиями конкатенацией, автоинкремент, автодекремент на языке JavaScript, научить применять полученные знания на практике.

Теоретические сведения

Строки, числа и другие объекты JavaScript можно хранить в переменных. Объекты JavaScript приходится хранить в переменных, если есть необходимость манипулировать ими - "склеивать" строки, складывать числа и т.д.

Переменная-это пара: имя и значение

Переменная задается **оператором определения переменной**, начинающимся с **ключевого слова** var, за которым через пробел следует имя переменной:

```
var a;
var __some_strange_name__;
var MoyaNovayaPeremennaya_1;
```

Имя переменной должно начинаться с латинской буквы или знака подчеркивания и может состоять из латинских букв, знаков подчеркивания и цифр.

Заглавные и строчные буквы в именах переменных JavaScript различаются:

```
var a;
var A;
// Определены две разные переменные.
```

Переменная состоит из имени и значения. Присвоить значение переменной можно с помощью операции присваивания:

```
var a;
a=1;
var myString;
myString="Моя строка";
a=myString;
```

С переменными, содержащими числовые значения, можно выполнять арифметические операции:

Основные арифметические операции JavaScript

Операция	Знак
Сложение	+
Вычитание	-
Умножение	*
Деление	/

Результат выполнения арифметических (и других!) операций можно запомнить в переменной (присвоить ее значению результат выполнения операции):

```
var FondZarplaty=1;
var Nachisl=0.385;
var FZP_i_Nachisl;
FZP_i_Nachisl=FondZarplaty*(1+Nachisl);
```

Часто возникает необходимость добавить к значению какой-то переменной единицу. Для этого в JavaScript существует специальное обозначение:

```
var a=10;
a++;
alert(a)
```

или

```
var a=10;
++a;
alert(a)
```

выведет число 11

Таким образом, чтобы увеличить числовое значение на единицу (выполнить операцию "**автоинкремент**" - автоприращение), следует использовать удвоенный знак "плюс". Аналогичным образом, чтобы уменьшить числовое значение на единицу (выполнить операцию "**автодекремент**" - автоуменьшение), следует использовать удвоенный знак "минус":

```
var a=10;
a--;
alert(a)
```

или

```
var a=10;
--a;
alert(a)
```

выведет число 9

Положение двойного знака	Выполняемые действия
Спереди: a=++b;	Сначала выполнить операцию автоинкремента (добавить к b единицу), а затем полученное значение присвоить переменной a: b=b+1; a=b;
Сзади: a=b++;	Сначала значение b присвоить переменной a, а затем выполнить над b операцию автоинкремента - увеличить значение b на единицу (видимо, для последующего использования): a=b; b=b+1;

Конкатенация(сцепление) строк. Знак + используется в JavaScript и еще для одной цели - конкатенации ("склеивания") строк.

Конкатенация двух строк - это получение новой строки, состоящей из последовательно расположенных конкатенируемых строк

Общее задание

Пример1.

```
<SCRIPT LANGUAGE="JavaScript">
<!--
var a;
a=1;
// Сейчас будет 1
alert(a);
var myString;
myString="Моя строка";
a=myString;
// А сейчас будет "Моя строка"
alert(a);
-->
</SCRIPT>
```

Пример 2

```
<SCRIPT LANGUAGE="JavaScript">
<!--
var a=1;
var myString="Моя строка";
var b;
b=a+myString;
alert(b);
//-->
</SCRIPT>
```

Пример 3

```
<HTML>
<H3>РЕДАКТОР</H3>
код:<BR>
<TEXTAREA id="mycode" ROWS=10 COLS=60></TEXTAREA>
<P>REZULTAT<BR>
<TEXTAREA id="myrezult" ROWS=3 COLS=60></TEXTAREA>
<P>
<BUTTON onclick="document.
all.myrezult.value=eval(mycode.value)">ВЫПОЛНИТЬ</BUTTON>
<BUTTON onclick="document. all.mycode.value=' ';
document. all.mycode.value=' '">ОЧИСТИТЬ </BUTTON>
</HTML>
```

Пример 4

```
<HTML>
<SCRIPT>
alert("ФАМИЛИЯ-ИВАНОВ\n ИМЯ-ИВАН\nОТЧЕСТВО-ИВАНОВИЧ")
</SCRIPT>
</HTML>
```

Задания

1. Составить предложение «Свою первую программу создадим на JavaScript».
2. Вычислить следующие выражения:

А) x=5, y=18. Найти z=x+y.

b) a=22,b=34 . Найти y= (a+b)/b.
c) y=((x+a)/z)*5.
3.Написать программу калькулятор, используя арифметические операции : +,-,/,*.

Содержание отчета

1. Цель и задание к лабораторной работе.
2. Листинг программы.
3. Результаты работы программы.
4.Аналитические выводы.

Лабораторная работа №7

Тема: Операторы: break, continue, for, function, if...else, while.

Цель: ознакомить с основными операторами, научить применять их на практике.

Теоретические сведения.

Операторы служат для управления потоком команд в *JavaScript*. Один объект может быть разбит на несколько строк,или, наоборот в одной строке может бытьнесколько операторов.

Необходимо знать следующее, во-первых,блоки операторов, такие как определения функций, должны быть заключены в фигурные скобки. Во-вторых, точка с запятой служит разделителем отдельных операторов. Если пропустить точку с запятой, поведение программы станет непредсказуемым.

Оператор break

Оператор Прерывает текущий цикл for и while, передавая управление первому оператору после цикла.

Оператор continue

Передает управление оператору проверки истинности условия в цикле while и оператору обнавления значения счетчика в цикле for. Важное отличие от оператора break заключается в том, что оператор *continue* не прерывает цикл, а делает следующее.

- В цикле **while**, это переходит назад к условию.
- В цикле **for**, это переходит к модернизированному выражению.

Оператор for

Создает цикл с тремя необязательными выражениями, заключенными в круглых скобках и разделенными точкой с запятой, сопровождаемыми блоками утверждений, выполненных внутри цикле:

```
for (initialExpression;condition;updateExpression)
  {
   ...строки кода...
  }
```

где

- Выражение *initialExpression* используеться для инициализации переменной счетчика, которую можно создать с помощью оператора *var*. Это выражение необязательно.

- Выражение *condition*, которое вычисляется на каждом проходе через цикл. Если это условие истинно, то выполняются условия внутри цикла. Это условие необязательно. Если оно опущено, то условие всегда считается равным истинному, тогда цикл продолжается до ошибки или до оператора *break*.
- Выражение *updateExpression*, вообще используется для изменения значения переменной счетчика. Оно тоже является необязательным. В этом случае можно обновлять значение переменной счетчика внутри цикла.

Примеры

Оператор **for** создает цикл ,продолжающийся до тех пор пока переменная счетчика *i* меньше чем девять, увеличиваю *i* на однин после каждого прохода через цикл.

```
for (var i = 0; i < 9; i++)
 {
   n += i
   myfunc(n)
 }
```

Следующий пример создает цикл **for**, продолжающийся до наступления ошибки или выполнения оператора *break*. Переменная счетчика увеличивается на 2 , при каждом проходе цикла:

```
for (var i = 0; ; i+=2)
 {
   ...строки кода...
 }
```

Оператор function

Объявляет функции языка *JavaScript*, требует указания имени *name* и списка параметров *param*. Для того чтобы возвращать значение, функция должна иметь в себе оператор return, который указывает возвращаемое функцией значение. Оператор функция вы не можете вложить в другую функцию. Принято следующее написание этого оператора:

```
function name([param] [, param] [..., param])
  {
   ...statements...
  }
```

Примеры

Эта функция возвращает общую долларовую сумму продажи, когда дано число единиц, продаваемых изделий a, b, и c.

```
function calc_sales(units_a, units_b, units_c) {
   return units_a*79 + units_b*129 + units_c*699}
```

Оператор if...else

Оператор **if...else** - это условный оператор, который выполняет первый набор утверждений, если значение *condition* истинно. В противном случае выполняет второй набор утверждений, заключенные в операторе *else*, если *условие* ложно. Если набор утверждений (statments), заключенные в фигурные скобки {} содержит один оператор, то скобки можно опустить. Условные операторы могут быть вложены друг в друга без ограничений.

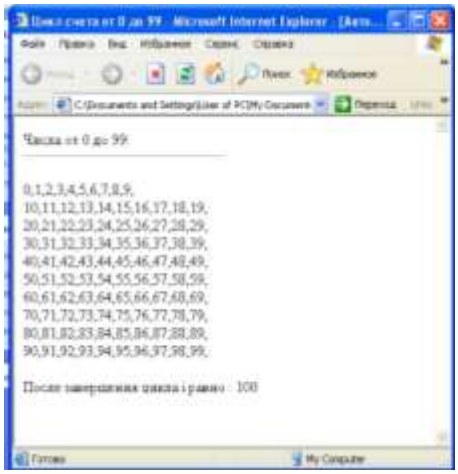
```
if (condition) {
  ...statements...
  }[else {
    else
  ...statements...
  }]
```

Оператор while

Оператор **while** - это оператор цикла, который повторяет цикл, пока значение *condition* есть истинно (true). Как только значение *condition* становится ложным (false), то управление переходит к первому оператору после фигурной скобки, закрывающей тело цикла **while**:

```
 while (condition) {
   ...statment...
 }
```

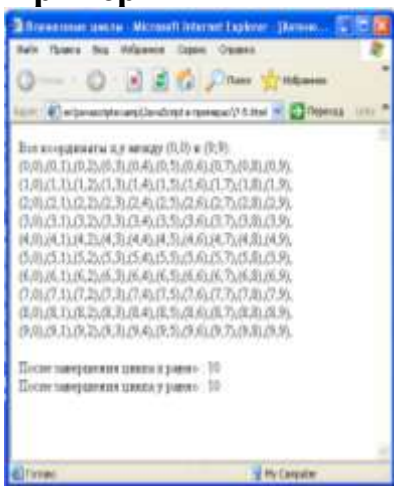
Пример 1



2. Общее задание

```
<html>
 <head>
 <title>Цикл счета от 0 до 99</title>
 <body>
 <script type="text/javascript">
 <!--
 // Вывести заголовочную часть
 document.write("Числа от 0 до 99:");
 document.write('<hr size="0" width="50%"align="left">');
for (var i = 0; i < 100; ++i) {
  // Символ новой строки после каждого 10-го числа
  if(i%10 == 0) {
    document.write('<br>');
  }
  // Вывести собственно число
  document.write(i + ",");
}
// Последнее число, следовательно, завершаемся...
document.write("<br><br>После завершения цикла i
равно : " + i);
// -->
</script></body></html>
```

Пример 2



```
<html>
<head>
  <title>Вложенные циклы</title>
</head>
<body>
  <script type="text/javascript">
  <!--
     document.write("Все координаты x,y между
(0,0) и (9,9):<br>");
     for (var x = 0; x < 10; ++x) {
       for (var y = 0; y < 10; ++y) {
         document.write("(" + x + "," + y + "),");
       }
     document.write('<br>');
     }
     document.write("<br>После завершения цикла
x равно : " + x);
     document.write("<br>После завершения цикла
y равно : " + y);
  // -->
  </script></body></html>
```

Пример 3

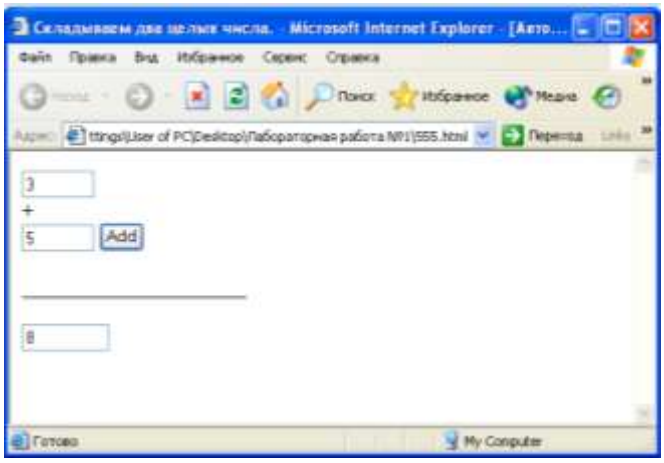

```
<html>
<head>
  <title>Использование цикла while</title>
</head>
<body>
  <script type="text/javascript">
  <!--
    // Объявление переменных
    var i = 0;
    var result = 0;
    var status = true;
    document.write("0");
    while (status) {
    result += ++i;
    document.write(" + " + i);
    if (i == 10) {
   status = false;
     }
    }
    document.writeln(" = " + result);
    // -->  </script></body></html>
```

Пример 4

```
<html><head>  <title>Операторы break и
continue</title></head>
<body>
  <script type="text/javascript">
  <!--
    // Объявление переменных
    var highestNum = 0;
    var n = 175;         // Тестовое значение

    for (var i = 0; i < n; ++i) {
      document.write(i + "<br>");
      if(n < 0){
        document.write("n не может быть
отрицательным.");
        break;
      }
      if (i * i <= n) {
        highestNum = i;
        continue;
      }
      document.write("<br>Сделано!");
      break;
    }
    document.write("<br>Целочисленное
значение, не превышающее квадратный
корень");
    document.write(" из " + n + " = " +
highestNum);
  // -->
```

Пример 5




```
<HTML>
 <HEAD>
  <TITLE>Складываем два целых числа.</TITLE>
<script>
```

```
function addInteger() {
var a=document.add_plus.inputA.value; // a-принимает значение 1-го поля
var b=document.add_plus.inputB.value; // b-принимает значение 2-го поля
// Переводит текстовый формат в целочисленный - parseInt()
document.add_plus.output.value=parseInt(a)+parseFloat(b)
}
</script>
</HEAD>
<BODY>
<form name="add_plus">
<input type="Text" name="inputA" value size=6><BR>
+<br>
<input type="Text" name="inputB" value size=6>
<input type="Button" value="Add" onClick="addInteger()">
<p>_____</p>
<input type="Text" name="output" size=8><BR>
</form>
  </BODY>
</HTML>
```

Задание: Вывести на экран таблицу умножения.

Содержание отчета

1. Цель и задание к лабораторной работе.
2. Листинг программы.
3. Результаты работы программы.
4. Аналитические выводы.

Лабораторная работа № 8

Тема: Простые визуальные эффекты. Раскрывающийся список.

Цель: научить создавать с помощью языка JavaScript простые визуальные эффекты и научить использовать списки для написания веб–страниц.

Теоретические сведения.

Раскрывающийся список.

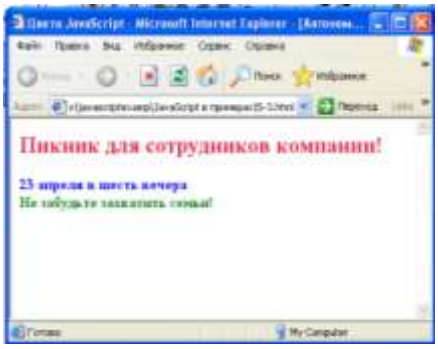
Простейшее меню можно создать с помощью <SELECT> И <OPTION>. Обычно такие конструкции называют раскрывающимися списками. Ниже приводится простейший пример использования раскрывающегося списка. В этом примере раскрывающийся список задается HTML – кодом, а обработка выбора из этого списка – сценарием. Задача сценария заключается просто в обработке номера выбранного элемента из списка. В примере это вывод окна с соответствующим сообщением. Выбор пользователя из раскрывающегося списка производится щелчком левой кнопкой мыши на элементе списка. При этом свойства selectedIndex обьекта элемента документа, соответствующего тегу <SELECT>, приобретает в качестве своего значения номер выбранного

элемента списка (нумерация начинается с 0). Для инициации обработки выбора пользователя здесь служит событие onchange (произошло изменение в выделении элемента списка). Обработка этого события осуществляется функцией myselection (). Начальное выделение и отображение элемента в раскрывающемся списке задается атрибутом SELECTED тега <OPTION>.

Общее задание.

Пример 1. Следующий текст раскрасить в различные цвета.

Пикник для сотрудников компании! - красный
23 апреля в шесть вечера - синий
Не забудьте захватить семьи! -зеленый

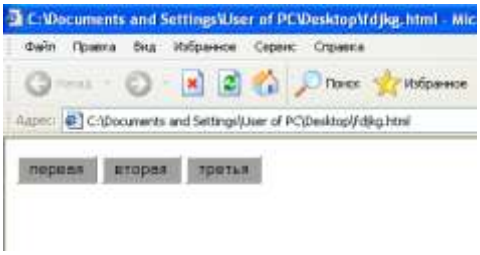



```
<html>
<head>
 <title>Цвета JavaScript</title>
</head>
<body>
 <h2>
 <script type="text/javascript">
<!—
document.writeln("Пикник для сотрудников компании!".fontcolor("crimson"));
    // -->
  </script>
  </h2>
  <h4>
 <script type="text/javascript">
 <!--
  document.writeln("23 апреля в шесть вечера".fontcolor("blue"));
  document.writeln("<br>" + "Не забудьте захватить
  семьи!".fontcolor("#008000"));
   // -->
   </script>
  </h4>
   </body></html>
```

Пример 2.Создать страницу, в которой нажимая на кнопку с названием цвета , страница, закрашивается в соответствующий цвет.

```
<html>
<head>
<title>bgcolor</title>
</head>
<body text=000000 bgcolor=ffffff>
<table border=0 align=center>
<tr><td><form>
<input type=button value="красный" onClick="document.bgColor='ff0000'">
<input type=button value="желтый" onClick="document.bgColor='ffff00'">
<input type=button value="синий" onClick="document.bgColor='0000ff'">
<input type=button value="голубой" onClick="document.bgColor='87ceeb'">
<input type=button value="коралловый" onClick="document.bgColor='ff7f50'">
</form></td>
</table>
</body>
</html>
```

Пример 3.Подсветка кнопок и текста.




```
<html>
<STYLE>
mystyle{font-weight:bold;background-color:a0a0a0}
</STYLE>
<FORM
onmouseover="colorchange('yellow')"onmouseout="colorchange('a0a0a0')">
<INPUT TYPE="BUTTON" VALUE="первая" CLASS="mystyle"
onclick="alert('вы нажали кнопку ')">
```

```
<INPUT TYPE="BUTTON" VALUE="вторая" CLASS="mystyle"
onclick="alert('вы нажали кнопку II')">
<INPUT TYPE="BUTTON" VALUE="третья" CLASS="mystyle"
onclick="alert('вы нажали кнопку III')">
</FORM>
<SCRIPT>
function colorchange(color){
if (event.srcElement.type=="button")
event.srcElement.style.backgroundColor=color;
}
</SCRIPT>
</html>
```

Пример 4.

```
<HTML>
Выберите экзамен, который  хотите сдать
<SELECT NAME ="TEST" onchange = " myselection ()">
<OPTION>Математика
<OPTION SELECTED>Физика
<OPTION>Биология
<OPTION>Химия
</SELECT>
<SCRIPT>
function myselection () {
var testname, testnumber;
testnumber = document.all.TEST. selectedIndex
if (testnumber == 0)
testname="Мат. Анализ"
else{
if (testnumber ==1)
testname="Квантовая физика"
else{
if (testnumber ==2)
testname="Биология"
else
 testname ="Органическая химия"
  }
}
alert ("Вы будете сдавать:"+ testname)
}
</SCRIPT>
</HTML>
```

Прмер2. Сценарий раскрывающегося списка

```
<HTML>
Выберите экзамен, который  хотите сдать:
<SCRIPT>
```

```
var N_sel = 1                          //номер элемента, выбранного по умолчанию
var aoptions =new Array()              //массив элементов списка
aoptions [0] = "Математика"
aoptions [1] = "Физика"
aoptions [2] = "Биология"
aoptions [3] = "Химия"
/*Строка, содержащая теги, формирующие  раскрывающийся список */
xstr='<SELECT ID = "TEST" onhange = "myselection ()">'
for (I = 0; I < aoptions. length; j++ ) {
xprefix = (I == N_sel)? 'SELECTD = ' + N_sel: ' '
xstr+='<OPTION'+ xprefix + '>'+ aoptions [i]
}
xstr+='</SELECT>'
document.write(xstr)                   //запись в документ
function myselection(){
var testname, testnumber;
testnumber = document.all.TEST.selectedIndex
if (testnumber == 0)
testname="Мат. Анализ"
else{
if (testnumber ==1)
testname="Квантовая физика"
else{
if (testnumber ==2)
testname="Биология"
else
testname ="Органическая химия"
}
}
alert ("Вы будете сдавать:"+ testname)
}
</SCRIPT>
</HTML>
```

Задание.

1.Создать страницу, в которой при нажатии на кнопку, меняется цвет текста с красного на зеленый.

2. Создать страницу «Кнопочный светофор».

3. Создать страницу, подсветки текста.

4. Как вы думаете, что означают атрибуты SelectIndex(),Option(),SELECTED?

5. Для чего предназначена функция myselection ()?

6. Что означает эта запись xstr+='</SELECT>'?

7. Создайте HTML-страничку и разместите на ней список товаров, список фирм предоставляющих этот товар, цены на товар.

Содержание отчета

1. Цель и задание к лабораторной работе.
2. Листинг программы.
3. Результаты работы программы.
4.Аналитические выводы.

Лабораторная работа № 9
Тема: Таблицы.

Цель: научить применять таблицы и протестировать имеющиеся программы.

Теоретические сведения

Доступ к элементам таблицы.

Таблицы являются наиболее используемыми элементами в веб- дизайне. Это обусловлено простотой и удобством компоновки документа с помощью тегов таблицы <TABLE>,<TR>,<TD> …. .

Таблица как объект документа имеет две коллекции, посредством которых осуществляется доступ к ее содержимому. Первая из них- коллекция строк ROWS, а вторая коллекция ячеек CELLS. Коллекция ROWS содержит все строки таблицы, включая разделы, соответствующие тегам <THEAD> и<TFOOT>. Коллекция CELLS содержит все элементы таблицы, созданные с помощью тегов <TH>и <TD>. Доступ к элементам коллекции осуществляется либо по индексу, либо по значению атрибута ID в соответствующем теге. Так для доступа к строке таблицы можно использовать значение ID в теге <TR>, а для доступа к ячейке- значение ID в теге <TH>или<TD> .При использовании индекса(номера) следует иметь в виду, что нумерация начинается с 0.При этом ячейки таблицы нумеруются слева направо и сверху вниз.

Добавление и удаление строк таблицы.

Добавление новой строки в таблицу производиться с помощью метода INSERT ROW().Этот метод возвращает ссылку на вновь созданную строку, которая затем используется для вставки ячеек. Ячейки вставляются в строку с помощью метода INSERTCELLS (индекс_ячейки). Данный метод возвращает ссылку на созданную ячейку, которая затем используется для задания содержимого ячейки. Ячейки вставляются в строку по порядку без пропусков начиная с нулевой.

Для удаления строки таблицы служит метод

DELETEROW (индекс_строки). Параметр указывает на номер удаляемой строки.

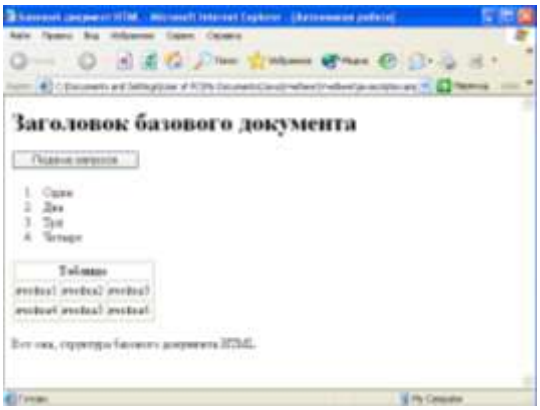
Общее задание

Пример 1. Составить следующую таблицу из 3 столбцов и 4 строк.

```
<HTML>
<SCRIPT>
/*-----------------------------------------*/
ah=new Array("Фамилия","Имя","Должность")
/*-----------------------------------------------*/
```

```
ad=new Array()
ad[0]=new Array("Иванов   ","Иван     ","Директор   ")
ad[1]=new Array(" Петров  ","    Петр ","  Зам директор ")
ad[2]=new Array("  Сидорова ","Элла     ","секретарша   ")
ad[3]=new Array(" Федоров   ","Федор     "," шофер  ")
strtab="<TABLE>"
/*-----------------------------------*/
for (i=0;  i<ah.length; i++) {
strtab+= "<TH>" + ah [i] + </TH>
}
/*-------------------------------------------*/
for (i=0; i<ad.length; i++){
strtab+="<TR>"
for (j=0;j<ad[i].length;j++){
strtab+="<TD>"+ad[i][j]+"</TD>"
}
strtab+="</TR>"
}
strtab+="/TABLE"
document.write(strtab)
</SCRIPT>
</HTML>
```

Пример 2:




```
<html>
  <head>
    <title>Базовый документ HTML</title>
    <script type="text/javascript"></script>
  </head>
<body>
  <h1>Заголовок базового документа</h1>
  <form>
    <input name="EnterBtn" type="SUBMIT">
  </form>
```

```
  <ol type=1>
    <li>Один</li>
    <li>Два</li>
    <li>Три</li>
    <li>Четыре</li>
  </ol>
  <table width="3" height="3" border>
    <tr>
      <th colspan="3">Таблица</th>
    </tr>
    <tr>
      <td>ячейка1</td>
      <td>ячейка2</td>
      <td>ячейка3</td>
    </tr>
    <tr>
      <td>ячейка4</td>
      <td>ячейка5</td>
      <td>ячейка6</td>
    </tr>
  </table>
  <P>
    Вот она, структура базового документа HTML.
  </p>
</body>
</html>
```

Задания.

1. Добавить 2 строки с записями.
2. Вставить 3 ячейки.
3. Удалить верхнюю строку.
4. Создать таблицу из нескольких строк и столбцов, а также вставить изображение с диаграммой к этой таблице. Данные предлагается взять из следующей таблицы:

Таблица 1.1. Доля отдельных развитых стран в ВВП мира (%)

Страна	**1970**	**1980**	**1995**
США	23,7	21,4	20,9
Япония	6,2	7,6	8,6
Германия	5,6	5,0	4,6
Франция	3,8	3,7	3,4
Великобритания	4,3	3,5	3,2
Италия	3,9	3,6	3,3

Содержание отчета
1. Цель и задание к лабораторной работе.
2. Листинг программы.
3. Результаты работы программы.
4. Аналитические выводы.

Лабораторная работа № 10
Тема: Символьные данные.

Цель: научить использовать символьные данные для написания веб-страниц.

Теоретические сведения.

Тексты большого объема, расположенные на веб странице, обычно снабжают поисковой системой. Для решения задачи поиска в тексте используется объект TextRange. Этот объект просто обеспечивает доступ к текстовой информации, находящейся в объектах, которые соответствуют тегам <BODY>,<TEXTAREA>,<BUTTON> <INPUT TYPE=’’text’’>.
Метод createTextRange() создает текстовую область.
Метод findText() производит поиск в текстовой области строки символов.
Метод ScrollIntoView() содержимое окна прокручивается так, чтобы найденная строка символов оказалась видимой.
Метода Select() организует подсвечивание найденного текста.
Не является ли введенный поисковый образ пустым. Ниже приведен вариант кода функции:

```
 function myfind() {
 if (!WORD.value) return                    // если поисковый образ пуст,
                                                                  выходим

 obj= document. Body. createTextRange()
obj=findText(WORD.value)
obj= ScrollIntoView()
obj= Select()
}
```

Для функции поиска использован стандартный метод prompt(), который принимает в качестве параметров пояснительный текст и начальное значение поискового образа (в данном случае пустую строку) и предоставляет поле для ввода значения. В окне есть две кнопки- OK и Отмена. Метод prompt() возвращает введенное пользователем значение либо false, если пользователь щелкнул на кнопке Отмена.

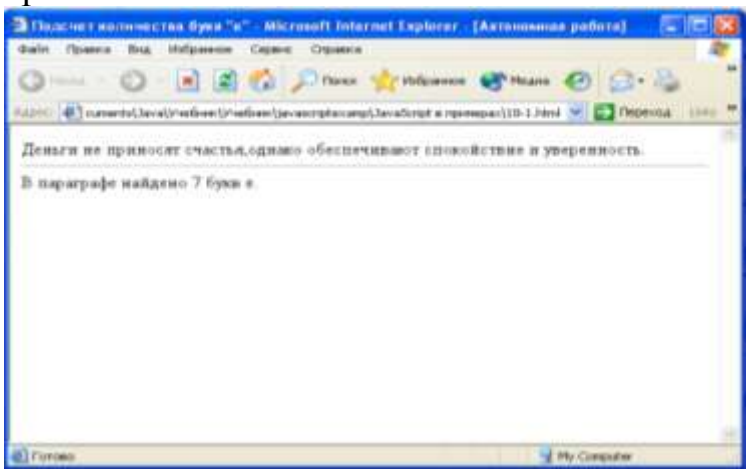
Общее задание

Пример 1.

```
<HTML>
<BODY>
<INPUT TYPE=’’text’’NAME=’’ WORD’’ VALUE=’’’’SIZE=20>
 <BUTTON onclick=’’ myfind()’’> Поиск </BUTTON>
<P>
```

```
<TEXTAREA> ID= “mytext”
<! Здесь расположен текст, в котором производится поиск>
</TEXTAREA>
 <SCRIPT>
  function myfind() {
 if (!WORD.value) return                    /*если поисковый образ пуст,
                                                                  выходим*/
  obj= document. Body. createTextRange()      // создаем текстовую область
obj=findText(WORD.value)                    // производим поиск
  obj= ScrollIntoView()
                                             /* прокручиваем текстовую
                                                    область в окне */
                                              // выделяем найденное
 obj= Select()
  }
</SCRIPT>
 </HTML>
</BODY>
```

Пример 2. Создать страницу, в которой подсчитывается количество букв Е в следующей записи «Деньги не приносят счастья, однако обеспечивает спокойствие и уверенность».




```
<HTML>
<HEAD>
<TITLE>Подсчет количества букв "e"</TITLE>
<script type="text/javascript" language="JavaScript1.1">
<!--
  // Объявление переменных
  var pos = 0;
  var num = -1;
  var i = -1;
     var graf = "Деньги не приносят счастья,"
           + "однако обеспечивают спокойствие и уверенность.";
  // Поиск в строке с подсчетом количества встречаемых "e"
  while (pos != -1) {
```

```
    pos = graf.indexOf("e", i + 1);
    num += 1;
    i = pos;
   }
  // Поместить ответ на страницу
  document.write(graf)
  document.write("<hr size='1'>")
  document.write("В параграфе найдено " + num + " букв е.");  document.close();
//-->
</script>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Пример 3

```
   <HTML>
  <BODY>
  <INPUT TYPE=’’text’’NAME=’’ WORD’’ VALUE=’’’’SIZE=20>
  <BUTTON onclick=’’ myfind()’’> Поиск >,</BUTTON>
   <P>
   <! Здесь расположен текст, в котором производится поиск>
   </BODY>
   <SCRIPT>
   function myfind() {
   obj= document. Body. createTextRange()

   obj=findText(WORD.value)                    // создаем текстовую область
   obj= ScrollIntoView()                      // производим поиск
                                                /* прокручиваем текстовую
                                                   область в окне */

    obj= Select()                                // выделяем найденное
}
  </SCRIPT>
  </HTML>
```

Задания.

1. В слове «Компьютеризация » подсчитать количество букв «и» и «о» результат вывести на экран.
2. В следующей фразе «Рыбак рыбака видит из далека» подсчитать количество букв «р» и «а».
3. Изменить выше рассмотренные функции поиска таким образом, чтобы в случае неудачного поиска в тексте на экране появилось сообщение «Не удалось найти». Для вывода сообщения воспользуйтесь стандартным методом alert().

Содержание отчета

1. Цель и задание к лабораторной работе.
2. Листинг программы.
3. Результаты работы программы.
4. Аналитические выводы.

Лабораторная работа № 11
Тема: Простые базы данных.

Цель: научить использовать простые базы данных для написания веб- страниц.

Теоретические сведения.

Достаточно эффективный способ формирования таблиц- использование специального элемента управления ActiveX, называемого Simple Tabular Data (STD-простые табличные данные). Этот элемент вставляется в документ с помощью тега и позволяет легко управлять данными, хранящимися в обычном текстовом файле. Вы можете удалять, сортировать, искать и фильтровать (выбирать) данные. Элемент ActiveX STD, управляющий данными, встроен в браузер IE4+ , а чтобы вставить его в HTML-

<OBJECT ID="mydbcontrol">
CLASSID="CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83">
<PARAM NAME="FieldDelim"VALUE="I">
< PARAM NAME ="DataURL" value="mydb.txt">
< PARAM NAME ="UseHeader" VALUE=true>
</ OBJECT >

FieldDelim –разделитель полей (ячеек).
DataURL –место расположения текстового файла с данными.
UseHeader-определяет, содержит ли первая строка в текстовом файле имена полей.

В текстовом файле с данными может содержаться не только текстовая информация но и HTML–коды. Чтобы эти коды, интерпретировались, необходимо в теги <TD> добавить атрибут DATAFORMATAS="html". Это позволяет вставлять в ячейки таблицы, гиперссылки, кнопки и др. элементы.

В текстовый файл добавить надпись
Иванов Иван <IMG SRC='pict1.jpg'>
В HTML

```
<TD><SPAN DATAFLD="Портрет" DATAFORMATAS="html" ></SPAN></TD>
```

Для установки фильтра предусмотрены следующие три параметра. Если требуется отфильтровать из таблицы все строки, в которых поле Фамилия имеет значение "Иванов", то необходимо использовать следующие теги параметров.

<PARAM NAME="FilterColumn" VALUE= "Фамилия ">
<PARAM NAME="FilterCriterion" VALUE= "= =">
<PARAM NAME="FilterValue" VALUE= "Иванов">

Общее задание

Пример 1. Таблица с помощью элемента STD.

```
<html>
<object id="mydbcontrol"
classid="clsid:333c7bc4-460f-11d0-bc04-0080c7055a83">
<PARAM NAME="FIELDDELIM" VALUE="|">
<PARAM NAME="DATAURL" VALUE="mydb.txt">
<PARAM NAME="UseHeader" VALUE=true>
</object>
<script>
alert(mydbcontrol.innerHTML)</script><html>
<TABLE DATASRC = #mydbcontrol BORDER=5>
<THEAD>
<TH> фамилия сотрудника</th><th>имя</th><th>ФОТО</th>
</THEAD>
<TR>
<TD><SPAN DATAFLD="ФАМИЛИЯ"></SPAN></TD>
<TD><SPAN DATAFLD="ИМЯ"></SPAN></TD>
<TD><SPAN
DATAFLD="ПОРТРЕТ"DATAFORMATAS="HTML"></SPAN></TD>
</TR>
</TABLE>
<HTML>
```

Пример 2. Примет формы с кнопками навигации.

```
<html>
<HEADER><TITLE>ПРИМЕР ФОРМЫ STD</TITLE></HEADER>
<object id="mydbcontrol"
classid="clsid:333c7bc4-460f-11d0-bc04-0080c7055a83">
<PARAM NAME="FIELDDELIM" VALUE="|">
<PARAM NAME="DATAURL" VALUE="mydb.txt">
<PARAM NAME="UseHeader" VALUE=true>
</object>
<!-- ПОЛЯ С ДАННЫМИ -->
<TABLE WIDTH=75%>
<TR><TD>
Фамилия  <INPUT TYPE="text" DATASRC="#mydbcontrol"
DATAFLD="ФАМИЛИЯ">
</TD></TR>
<TR><TD>
Имя  <INPUT TYPE="text" DATASRC="#mydbcontrol"
DATAFLD="ИМЯ">
</TD></TR>
<TR><TH>ФОТО</TH>
<TD>
```

```
<SPAN DATASRC=#mydbcontrol DATAFLD="ПОРТРЕТ"
DATAFORMATAS="html">
 </SPAN></TD></TR></TABLE><P>
<! КНОПКИ ПЕРЕМЕЩЕНИЯ ПО ЗАПИСЯМ >
<INPUT NAME= "cmdFirst" TYPE="BUTTON" VALUE="<<"
ONCLICK="First()">
<INPUT NAME= "cmdPrevious" TYPE="BUTTON" VALUE="<"
ONCLICK="Previous()">
<INPUT NAME= "cmdNext" TYPE="BUTTON" VALUE=">"
ONCLICK="Next()">
<INPUT NAME= "cradLast"  TYPE="BUTTON" VALUE=">>"
ONCLICK="Last()">
<SCRIPT>
function First() {
mydbcontrol.recordset.moveFirst()
}
function Previous(){
if (!mydbcontrol.recordset.bof)
mydbcontrol.recordset.movePrevious()
}
function Next(){
if (!mydbcontrol.recordset.eof)
mydbcontrol.recordset.moveNext()
}
function Last(){
mydbcontrol.recordset.moveLast()
}
</SCRIPT>
</HTML>
```

Пример3. Фильтрация данных таблицы

```
 <HTML>
<H3>Фильтр:</H3>
Поле
<! Раскрывающийся список имён полей>
<SELECT NAME="FLD">
<OPTION value="Фамилия">Фамилия
<OPTION value="Имя">Имя
</SELECT>
<BR>
Значение
<! Поле ввода значения для фильтра и кнопка>
<INPUT NAME="INP" VALUE="" TYPE="text">
<P>
<BUTTON onclick="filter()">Применить</BUTTON>
<HR>
```

```
<! Элемент управления STD>
<OBJECT ID="mydbcontrol" CLASSID="CLSID:333C7BC4-460F-11D0-BC04-
0080C7055A83">
<PARAM NAME="FieldDelim" VALUE="|">
<PARAM NAME="DataURL" VALUE="mydb.txt">
<PARAM NAME="UseHeader" VALUE=true>
</OBJECT>
<! Таблица для вывода данных>
<TABLE DATASRC="#mydbcontrol" border=5>
<THEAD>
<TH>Фамилия</TH>
<TH>Имя</TH>
<TH>Портрет</TH>
</THEAD>
<TR>
<TD><SPAN DATAFLD="Фамилия"></SPAN></TD>
<TD><SPAN DATAFLD="Имя"></SPAN></TD>
<TD><SPAN DATAFLD="Портрет" DATAFORMATAS="html"></SPAN></TD>
</TR>
</TABLE>
<SCRIPT>
/* Сохраняем основные параметры элемента STD в переменной obj */
var obj="<OBJECT ID=\"mydbcontrol\" ";
obj+="CLASSID=\"CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83\">";
obj+="<PARAM NAME=\"FieldDelim\" Value=\"|\">";
obj+="<PARAM NAME=\"DataURL\" Value=\"mydb.txt\">";
obj+="<PARAM NAME=\"UseHeader\" value=true>";

function filter() {// установка фильтра
var cpar=""; // переменная для хранения параметров фильтра
if (INP.value){// если введено значение
cpar="<PARAM NAME=\"FilterColumn\" VALUE=\""+FLD.value+"\">";
cpar+="<PARAM NAME=\"FilterValue\" VALUE=\""+INP.value+"\">";
cpar+="<PARAM NAME=\"FilterCriterion\" VALUE=\"=\">";
cpar+="<PARAM NAME=\"CaseSensitive\" VALUE=false>";
}
document.all.mydbcontrol.outerHTML=obj+cpar+"</OBJECT>";
}
</SCRIPT>
</HTML>
```

Задания.

1.Для чего предназначены следующие записи

а) <INPUT TYPE="TEXT" DATASRC=# mydbcontrol DATAFLD="Имя">

б) {

```
if (!mydbcontrol.recordset.eof)
mydbcontrol.recordset. moveNext ()
}
FUNCTION Last () {
{
 c)  FUNCTION First() {
mydbcontrol.recordset.moveFirst()
}
```

2.Создать базу списка группы, отсортировать по алфавиту.
3.Создать базу "Автомобили" с изображением, и вывести на отдельную страницу определенную марку машины с изображением.

Содержание отчета

1. Цель и задание к лабораторной работе.
2. Листинг программы.
3. Результаты работы программы.
4. Аналитические выводы.

Лабораторная работа № 12
Тема: Слои

Цель: научить использовать слои для написания веб- страниц.

Теоретические сведения

Создание слоев.

Слои - это одна из замечательных новых возможностей.Она позволяет выполнять точное позиционирование таких объектов web-страницы, как изображения. Их можно перемещать, делать объекты невидимыми, управлять слоями можно легко с помощью языка JavaScript. Чтобы создать слой, мы должны использовать либо тэг <layer> либо <ilayer>. Вы можете воспользоваться следующими параметрами:

Параметр	Описание
name="layerName"	Название слоя
left=xPosition	Абсцисса левого верхнего угла
top=yPosition	Ордината левого верхнего угла
z-index=layerIndex	Номер индекса для слоя
width=layerWidth	Ширина слоя в пикселах
clip="x1_offset,y1_offset,x2_offset,y2_offset"	Задает видимую область слоя
above="layerName"	Определяет, какой слой окажется под нашим
below="layerName"	Определяется, какой слой окажется над нашим
Visibility=show|hide|inherit	Видимость этого слоя
bgcolor="rgbColor"	Цвет фона - либо название стандартного цвета, либо rgb-запись
background="imageURL"	Фоновая картинка

Тэг <layer> используется для тех слоев, которые Вы можете точно позиционировать. Если же Вы не указываете положение слоя (с помощью параметров left и top), то по умолчанию он помещается в верхний левый угол

окна.Тэг <ilayer> создает слой, положение которого определяется при формировании документа. Рассмотрим простой пример. Создадим два слоя. В первом из них помещаем изображение, а во втором - текст. Покажем этот текст поверх данного изображения.

Текст поверх изображения

Исходный код:

```
<html>
<layer name=pic z-index=0 left=200 top=100>
<img src="img.gif" width=160 height=120>
</layer>
<layer name=txt z-index=1 left=200 top=100>
<font size=+4> <i> Layers-Demo </i> </font>
</layer>
</html>
```

С помощью тэга <layer> формируем два слоя. Оба слоя позиционируются как 200/100 (через параметры left и top). Все, что находится между тэгами <layer> и </layer> (или тэгами <ilayer> и </ilayer>) принадлежит описываемому слою.

Используем параметр z-index, определяя тем самым порядок появления указанных слоев - то есть, в нашем случае, тем самым сообщаем браузеру, что текст будет написан поверх изображения. В общем случае, именно слой с самым высоким номером z-index будет показан поверх всех остальных. Если в первом тэге <layer> напишем z-index=100, то текст окажется под изображением - его слой номер Z-индекса (z-index=1).

Текст под изображением

Слои и JavaScript

Рассмотрим, как можно получить доступ к слоям через JavaScript. Пример, пользователь получает возможность, нажимая кнопку, прятать или показать некий слой.

Для начала мы должны знать, каким образом слои представлены в JavaScript. Как обычно, для этого имеются несколько способов. Самое лучшее - дать каждому слою свое имя. Так, если мы задаем слой

```
<layer ... name=myLayer>
...
</layer>
```

то в дальнейшем можем получить доступ к нему с помощью конструкции document.layers["myLayer"]. Доступ к этим слоям можно также получить через целочисленный индекс. Так, чтобы получить доступ к самому нижнему слою, мы можем написать document.layers[0]. Обратите внимание, что индекс - это **не** то же самое, что параметр z-index. Если, например, мы имеем два слоя, называемые layer1 и layer2 с номерами z-index 17 и 100, то мы можем получить доступ к этим слоям через document.layers[0] и document.layers[1], а **не** через document.layers[17] и document.layers[100].Слои имеют несколько свойств, которые можно изменять с помощью скрипта на JavaScript. В следующем примере представлена кнопка, которая позволяет Вам скрывать или, наоборот, предоставлять один слой.

Перемещение слоев

Свойства left и top определяют задают положение данного слоя. Мы можем менять его, записывая в эти атрибуты новые значения. Например, в следующей строке задается горизонтальное положение слоя в 200 пикселов:
document.layers["myLayer2"].left= 200;
Перейдем теперь к программе перемещения слоев - она создает нечто вроде линейки прокрутки внутри окна браузера.

Общее задание

Пример 1

```
<html>
<head>
<script language="JavaScript">
<!-- hide
function move() {
  if (pos < 0) direction= true;
  if (pos > 200) direction= false;
  if (direction) pos++
    else pos--;
  document.layers["myLayer2"].left= pos;
}
// -->
</script>
</head>
<body onLoad="setInterval('move()', 20)">
<ilayer name=myLayer2 left=0>
<font size=+1 color="#0000ff"><i>This text is inside a layer</i></font>
</ilayer>
</body>
</html>
```

Пример2.

```
<html>
<head>
<script language="JavaScript">
```

```
<!-- hide
function showHide() {
  if (document.layers["myLayer"].visibility == "show")
    document.layers["myLayer"].visibility= "hide"
  else document.layers["myLayer"].visibility= "show";
}

// -->
</script>
</head>
<body>
<ilayer name=myLayer visibility=show>
<font size=+1 color="#0000ff"><i>This text is inside a layer</i></font>
</ilayer>
<form>
<input type="button" value="Show/Hide layer" onClick="showHide()">
</form>
</body>
</html>
```

Задания.

1. Для чего предназначены следующие записи

а)
```
<!-- hide
function showHide() {
  if (document.layers["myLayer"].visibility == "show")
    document.layers["myLayer"].visibility= "hide"
  else document.layers["myLayer"].visibility= "show";
```

б)
```
<ilayer name=myLayer visibility=show>
```

в)
```
<body onLoad="setInterval('move()', 20)">
```

2. Используя слои написать свою веб- страницу.

Содержание отчета

1. Цель и задание к лабораторной работе.
2. Листинг программы.
3. Результаты работы программы.
4. Аналитические выводы.

Лабораторная работа № 13
Тема: Строка состояния. Таймер.

Цель: научить использовать таймер, строку состояния для написания веб - страниц.

Теоретические сведения

Строка состояния

Программы на JavaScript могут выполнять запись в строку состояния - прямоугольник в нижней части окна Вашего браузера. Все, что необходимо для этого сделать - всего лишь записать нужную строку в *window.status*. В

следующем примере создаются две кнопки, которые можно использовать, чтобы записывать некий текст в строку состояния и, соответственно, затем его стирать.

Таймеры

С помощью функции Timeout (или таймера) можно запрограммировать компьютер на выполнение некоторых команд по истечении некоторого времени. В следующем скрипте демострируется кнопка, которая открывает выпадающее окно не сразу, а по истечении 3 секунд.

Скрипт выглядит следующим образом:

```
<script language="JavaScript">
<!-- hide
function timer() {
  setTimeout("alert('Врем\я истекло!')", 3000);
}
// -->
</script>
...
<form>
  <input type="button" value="Timer" onClick="timer()">
</form>
```

Здесь setTimeout() - это метод объекта window. Он устанавливает интервал времени .Первый аргумент при вызове - это код JavaScript, который следует выполнить по истечении указанного времени. В нашем случае это вызов - alert('Врем\я истекло!'). Обратите пожалуйста внимание, что код на JavaScript должен быть заключен в кавычки. Во втором аргументе компьютеру сообщается, когда этот код следует выполнять. При этом время Вы должны указывать в миллисекундах (3000 миллисекунд = 3 секунда).

Общее задание

Пример 1

```
<html>
<head>
<script language="JavaScript">
<!-- hide
function statbar(txt) {
   window.status = txt;
}
// -->
</script>
</head>
<body>
<form>
 <input type="button" name="look" value="Писать!"
  onClick="statbar('Привет! Это окно состо\яни\я!');">
<input type="button" name="erase" value="Стереть!"
```

```
  onClick="statbar(");">
</form>
</body>
</html>
```

Пример 2

```
<html>
<head>
<script language="JavaScript">
<!-- hide
// выбор текста дл\я прокрутки
var scrtxt = "Это JavaScript! " +
    "Это JavaScript! " +
    "Это JavaScript!";
var len = scrtxt.length;
var width = 100;
var pos = -(width + 2);
function scroll() {
  // напечатать заданный текст справа и установить таймер
  // перейти на исходную позицию для следующего шага
  pos++;
  // вычленить видимую часть текста
  var scroller = "";
  if (pos == len) {
    pos = -(width + 2);
  }
  // если текст еще не дошел до левой границы, то мы должны
  // добавить перед ним несколько пробелов. В противном случае мы должны
  // вырезать начало текста (ту часть, что уже ушла за левую границу
  if (pos < 0) {
    for (var i = 1; i <= Math.abs(pos); i++) {
      scroller = scroller + " ";}
    scroller = scroller + scrtxt.substring(0, width - i + 1);
  }
  else {
    scroller = scroller + scrtxt.substring(pos, width + pos);
  }
  // разместить текст в строке состо\яни\я
  window.status = scroller;
  // вызвать эту функцию вновь через 100 миллисекунд
  setTimeout("scroll()", 100);
}
// -->
</script>
</head>
```

Задания.

1.Для чего предназначены следующие записи

а) window.status = scroller;

б) scroller = scroller + scrtxt.substring(0, width - i + 1);

в) for (var i = 1; i <= Math.abs(pos); i++) {
scroller = scroller + " ";}

г) var pos = -(width + 2);

2. Создать страницу бегущей строки.

Содержание отчета

1. Цель и задание к лабораторной работе.
2. Листинг программы.
3. Результаты работы программы.
4. Аналитические выводы.

Лабораторная работа №14

Тема: Фреймы. Формы.

Цель: научить использовать фреймы и формы для создания веб-страниц, протестировать задания .

Теоретические сведения **Фреймы.**

Один из часто задаваемых вопросов - как фреймы и JavaScript могут работать вместе. В общем случае окно браузера может быть разбито в несколько отдельных фреймов. Это означает, что фрейм определяется как некое выделенное в окне браузера поле в форме прямоугольника. Каждый из фреймов выдает на экран содержимое собственного документа (в большинстве случаев это документы HTML). Для создания фреймов Вам необходимо два тэга: <frameset> и <frame>. При создании web-страниц Вы можете использовать несколько вложенных тэгов <frameset>.

Общее задание

Пример 1.

```
<html>
<frameset rows="50%,50%">
<frame src="page1.htm" name="frame1">
<frame src="page2.htm" name="frame2">
</frameset>
</html>
```

Пример 2

```
<frameset cols="50%,50%">
<frameset rows="50%,50%">
<frame src="cell.htm">
<frame src="cell.htm">
</frameset>
<frameset rows="33%,33%,33%">
<frame src="cell.htm">
<frame src="cell.htm">
```

```
<frame src="cell.htm">
</frameset>
</frameset>
```

Пример 3.

```
<html>
<frameset rows="50%,50%">
<frame src="page1.htm" name="frame1">
<frame src="page2.htm" name="frame2">
Один<BR>
Два
<H1 ID="XXX">Три<H1>
<SCRIPT>
function change(){
parent.left.document.all.XXX.innerText="Ура!"
}
</SCRIPT>
<H1 onclick="change()">Щелкни здесь<H1>
</frameset>
</html>
```

Пример 4.

```
<html>
<head>
<title>Пример документа с фреймами</title>
</head>
<script language="JavaScript">
<!--
var string = location.search;
var current_page=string.substring (1, string.length);
document.write('<frameset cols=\"180,*\">');
document.write('<frame src=\"menu.htm\" name=\"menu\">');
if (location.search == "") {
document.write('<frame src=\"content.htm\" name=\"content\">');
} else {
document.write('<frame src=\'" + current_page + '\" name=\"content\">');
}
document.write('</frameset>');
//-->
</script>
<noscript>
```

Формы.

Проверка информации, введенной в форму

Формы широко используются на Интернет. Информация, введенная в форму, часто посылается обратно на сервер или отправляется по электронной почте на некоторый адрес. Проблема состоит в том, чтобы убедиться, что

введенная пользователем в форму информация корректна. Легко проверить ее перед пересылкой в Интернет можно с помощью языка JavaScript. Сначала можно выполнить проверку формы. А затем рассмотрим, какие есть возможности для пересылки информации по Интернет. В начале необходимо создать простой скрипт.

Допустим, HTML-страница содержит два элемента для ввода текста. В первый из них пользователь должен вписать свое имя, во второй элемент - адрес для электронной почты. Команда if проверяет, чем заканчивается первое или второе сравнения. Если хотя бы одно из них выполняется, то и в целом команда if имеет результатом true, а стало быть будет выполняться следующая команда скрипта. Второй оператор в команде if следит за тем, чтобы введенная строка содержала @.

Если пользователь ввел свое имя (например, 'Stefan') в первый элемент, то скрипт создает выпадающее окно с сообщением '*Hi Stefan!*'.

Проверка на присутствие определенных символов

В некоторых случаях требуется ограничивать информацию, вводимую в форму, лишь некоторым набором символов или чисел. Достаточно вспомнить о телефонных номерах - представленная информация должна содержать лишь цифры (предполагается, что номер телефона, как таковой, не содержит никаких символов). Нам необходимо проверять, являются ли введенные данные числом. Сложность ситуации состоит в том, что большинство людей вставляют в номер телефона еще и разные символы - например: 01234-56789, 01234/56789 or 01234 56789 (с символом пробела внутри). Не следует принуждать пользователя отказываться от таких символов в телефонном номере. А потому мы должны дополнить наш скрипт процедурой проверки цифр и некоторых символов. Решение задачи продемонстрировано в следующем примере2 .Функция test() определяет, какие из введенных символов признаются корректными

Общее задание

Пример 1

```
 <html>
<head>
<script language="JavaScript">
<!-- Скрыть
function test1(form) {
  if (form.text1.value == "")
    alert("Пожалуйста, введите строку!")
  else {
   alert("Hi "+form.text1.value+"! Форма заполнена корректно!");
  }
}
function test2(form) {
  if (form.text2.value == "" ||
      form.text2.value.indexOf('@', 0) == -1)
       alert("Неверно введен адрес e-mail!");
```

```
  else alert("OK!");
}
// -->
</script>
</head>
<body>
<form name="first">
Введите Ваше имя:<br>
<input type="text" name="text1">
<input type="button" name="button1" value="Проверка"
onClick="test1(this.form)">
<P>
Введите Ваш адрес e-mail:<br>
<input type="text" name="text2">
<input type="button" name="button2" value="Проверка"
onClick="test2(this.form)">
</body>
</html>
```

Пример 2

```
<html>
<head>
<script language="JavaScript">
<!-- hide
// ********************************************************
// Script from Stefan Koch - Voodoo's Intro to JavaScript
//     http://rummelplatz.uni-mannheim.de/~skoch/js/
//       JS-book: http://www.dpunkt.de/javascript
//    You can use this code if you leave this message
// ********************************************************
function check(input) {
  var ok = true;
  for (var i = 0; i < input.length; i++) {
    var chr = input.charAt(i);
    var found = false;
    for (var j = 1; j < check.length; j++) {
      if (chr == check[j]) found = true;
    }
    if (!found) ok = false;
  }
   return ok;
}
function test(input) {
  if (!check(input, "1", "2", "3", "4",
        "5", "6", "7", "8", "9", "0", "/", "-", " ")) {
    alert("Input not ok.");
```

```
  }
  else {
    alert("Input ok!");
  }
}
// -->
</script>
</head>
<body>
<form>
Telephone:
<input type="text" name="telephone" value="">
<input type="button" value="Check"
  onClick="test(this.form.telephone.value)">
</form>
</body>
</html>
```

Задания.

1.Для чего предназначены следующие записи

а) <frameset cols="50%,50%">

б) <frame src="cell.htm">

в) <frame src="page2.htm" name="frame2">

2. Создать страницу с плавающими фреймами.

3.Создать страницу с тремя фремами в первой выводит надпись "Готов к работе", во второй надпись "Приступай", в третьей "Выполняй работу."

4.Для чего предназначены следующие записи

а) <input type="button" value="Check"
onClick="test(this.form.telephone.value)">

б) function check(input) {
var ok = true;
for (var i = 0; i < input.length; i++)

в)<input type="button" name="button1" value="Проверка" onClick="test1(this.form)">

5. Создать страницу со своими формами, с рисунками.

Содержание отчета

1. Цель и задание к лабораторной работе.
2. Листинг программы.
3. Результаты работы программы.
4. Аналитические выводы.

Лабораторная работа № 15
Тема: Вставка изображений.

Цель: научить вставлять изображения для написания веб- страниц.

Теоретические сведения.

HTML-документ могут быть включены изображения. Для этого используется ярлык <IMG>:

<IMG SRC="источник" ALT="текст" LOWSRC="источник" NAME="имя">

источник - это URL файла изображения (в частности, это может быть имя файла). Изображение, указанное в атрибуте LOWSRC, загружается до изображения, указанного в атрибуте SRC, а затем заменяется на последнее (предположительно первое имеет меньшее разрешение и, соотвественно, существенно меньший объем файла, т.е. быстро загружается);

текст - это текст, выводимый на месте изображения до его загрузки или в случае отказа пользователя от загрузки изображения;

имя - это имя конкретного элемента <IMG>. Атрибут NAME (так же, как и атрибут LOWSRC) стандартизован только в HTML

Такие элементы, входящие в состав HTML-страницы, находят свое отражение в JavaScript в виде свойств объекта document. заменяющая одно изображение (встроенное в документ img1.htm, первоначально загружаемый в окно) на другое:

```
<SCRIPT LANGUAGE="JavaScript">
<!--
var newWindowFeatures="dependent=1,innerHeight=208,innerWidth=208";
var board=window.open("imge.htm","Board",newWindowFeatures);
if ( board.confirm("Сменим картинку?") )
{
  board.document.images[0].src="img1.gif";
}
//-->
</SCRIPT>
```

Операция в выделенной строчке заменяет прежнее значение свойства src объекта board.document.images[0] на новое (img1.gif), тем самым загружая в документ новое изображение.

Общее задание.

Пример 1. Картинка сменяется другой картинкой при наведении на нее мышкой.

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!--//
browser_name = navigator.appName;
browser_version = parseFloat(navigator.appVersion);
```

```
if (browser_name == "Netscape" && browser_version >= 3.0) { roll = 'true'; }
else if (browser_name == "Microsoft Internet Explorer" && browser_version >= 3.0)
{ roll = 'true'; }
else { roll = 'false'; }
function over(img,ref) { if (roll == 'true') { document.images[img].src = ref; } }
function out(img,ref)  { if (roll == 'true') { document.images[img].src = ref; } }
if (roll == 'true')
{
a1=new Image;a1.src="image1.gif";
a2=new Image;a2.src="image2.gif";
...
aX=new Image;aX.src="imageX.gif";
}
//-->
</SCRIPT>
 <A HREF="page.htm" onMouseOver="over('image_name','image2.gif');"
onMouseOut="out('image_name','image1.gif');"><img name="image_name"
alt="текст" src="image1.gif" width=x height=y border="0"></A>
</HEAD>
</HTML>
```

Пример 2.Используйте html файл приведенный ниже для пересылки со странички на другую страничку.

```
<HTML>
<HEAD>
<BODY>
<META HTTP-EQUIV="REFRESH" CONTENT="1;
URL=http://www.куда.пересылать">
<script language="JavaScript"> <!--
window.location.href = "http://www.куда.пересылать"
// --> </script>
<CENTER><P><B><FONT SIZE=+4><A
HREF="http://www.куда.пересылать">название сайта куда
отсылаем</A></FONT></B></P></CENTER>
</HEAD>
</BODY>
</HTML>
```

Пример 3.

```
<HTML>
<IMG ID = "myimg" SRC = " C:\Documents and Settings\All Users\Documents\My
Pictures\Sample Pictures\Blue hills.jpg" onclick = "imgchange() ">
<SCRIPT>
var  flag = false
function imgchange ( ) {
if  (flag) document.all.myimg.src = "C:\Documents and Settings\All
Users\Documents\My Pictures\Sample Pictures\Blue hills.jpg"
```

```
else
 document.all.myimg.src = "C:\Documents and Settings\All Users\Documents\My
Pictures\Sample Pictures\Sunset.jpg"
flag = ! flag
}
</SCRIPT>
</HTML>
```

Контрольные задания:

1.Для чего предназначены следующие записи

а) <input type="button" value="Check"
onClick="test(this.form.telephone.value)">

б) function check(input) {
var ok = true;
for (var i = 0; i < input.length; i++)

в)<input type="button" name="button1" value="Проверка" onClick="test1(this.form)">

2. Создать веб- страницу для сайта на тему: «Актауский Государственный Униветситет им.Ш.Е.Есенова»

Содержание отчета

1. Цель и задание к лабораторной работе.
2. Листинг программы.
3. Результаты работы программы.
4.Аналитические выводы.

Задания для самостоятельной работы студентов

Темы для самостоятельной работы студентов:

Самостоятельная работа студентов направлена на повышение навыков работы с научной и периодической литературой. Данный вид работы выполняется студентами в виде конспектов лекции и рефератов. Темы рефератов назначается преподавателем по следующему перечню:

1. Специфика управления персоналом проекта по созданию ПО.
2. Web-службы.
3. Протокол http.
4. Расширяемый язык разметки XML
5. Синхронное и асинхронное взаимодействие.
6. Библиотеки готовых компонентов.
7. Средства создания многопоточных программ.
8. Контроль удобства программного обеспечения.
9. Шаблонный метод.
10. Инспекция программ по Фагану.
11. UML (унифицированный язык моделирования). Виды диаграмм UML.
12. Методы контроля качества.
13. Унифицированный процесс Rational.
14. Обзор пользовательских интерфейсов современных программных приложений.
15. Анализ и оценка эффективности диалогового взаимодействия.

Темы для самостоятельной работы студентов с преподователем:
1. Домашние сети. Обзор пользовательского интерфейса, анализ и оценка эффективности диалогового взаимодействия.
2. Муниципальные сети. Обзор пользовательского интерфейса, анализ и оценка эффективности диалогового взаимодействия.
3. Беспроводные сети. Обзор пользовательского интерфейса, анализ и оценка эффективности диалогового взаимодействия.
4. Уровни передачи данных.
5. Локальные вычислительные сети. Обзор пользовательского интерфейса, анализ и оценка эффективности диалогового взаимодействия.
6. Анализ и выбор структуры диалогового взаимодействия.
7. Выбор форм диалогового взаимодействия для различных категорий пользователей.
8. Выбор средств ввода и вывода информации.
9. Выбор методов отображения информации в зависимости от важности и сложности информации для различных категорий пользователей.
10. Комплексное решение вопросов выбора компонентов пользовательских и программно-аппаратных интерфейсов в многоуровневых клиент-серверных системах.
11. Организация навигации по программному приложению (на примере интернет-сайтов, электронных средств обучения).
12. Разработка программного приложения с использованием синтаксически-ограниченных форм пользовательского интерфейса.
13. Разработка пользовательского интерфейса, основанного на реакциях программного приложения на определенные события.
14. Разработка программного приложения с пользовательским интерфейсом, использующим методы эффективной навигации и поиска информации.
15. Выбор комплекса технических и программных средств для операторов различных категорий и квалификации.

2. Задания для самостоятельной работы студентов с преподавателем:

Разработка программного приложения с пользовательским интерфейсом, использующим методы эффективной навигации и поиска информации:
1. Компьютеры и ноутбуки;
2. Принтеры и печатающие устройства;
3. Сервера и сетевые оборудования;
4. Мониторы и проекторные оборудования;
5. Сканеры и аудио-видео оборудования;
6. Модемы и телекоммуникационные оборудования;
7. Компьютерные системы и программы.

Глоссарий

Абстракция – выделение существенных характеристик некоторого объекта, которые отличают его от всех других видов и четко определяют его концептуальные границы относительно дальнейшего рассмотрения и анализа.
Агрергация – отношение «часть - целое»
Ассоциация – отношение между экземплярами классов
Атрибут – любая характеристика сущности, значимая для рассматриваемой предметной области и предназначена для квалификации, идентификации, классификации, количественной характеристики или выражения состояния сущности.
Жизненный цикл – период, который начинается с момента принятия решения о необходимости создания ИС и заканчивается в момент его полного изъятия из эксплуатации

Иерархия – ранжированная или упорядоченная система абстракций, расположение их по уровням.
ИС – информационная система.
Метод проектирования – организованная совокупность процессов создания ряда моделей, которые описывают различные аспекты разрабатываемой системы с использованием четко определенной нотации.
Метод – совокупность трех составляющих:
- концепций и теоретических основ. В качестве таких основ может выступать структурный или объектно-ориентрованный подход;
- нотации, используемых для построения моделей статической структуры и динамики поведения проектируемой системы. В качестве таких нотаций обычно используется графические диаграммы, поскольку они наиболее наглядны и просты в восприятии (диаграммы потоков данных и диаграммы «сущность – связь» для структурного подхода, диаграммы вариантов использования, диаграммы классов и др.- для объектно-ориентированного подхода);
- процедуры, определяющей практическое применение метода (последовательность и правила построения моделей, критерии, используемые для оценки результатов).
Модель (ПО) -полное описание системы ПО с определенной точки зрения.
Модель ЖЦ ПО – структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении ЖЦ.
Модульность – свойство системы, связанное с возможностью ее декомпозиции на ряд внутренне связных, но слабо связанных между собой модулей.
Накопитель данных – абстрактное устройство для хранения информации.
Наследование – построение новых классов на основе существующих с возможностью добавления или переопределения данных и методов.
Нотация (языка моделирования) – совокупность графических объектов, которые используются в моделях.
Объект – осязаемая реальность (tangible entity) – предмет или явление, имеющие четко определяемое поведение.
Объектная декомпозиция – описание структуры системы в терминах объектов и связей между ними, а поведения системы – в терминах обмена сообщениями между объектами.
Операция (метод) – определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию.
Параллелизм – свойство объектов находиться в активном или пассивном состоянии и различать активные и пассивные объекты между собой.
Полиморфизм – способность класса принадлежать более чем одному типу.
Поток данных – информация, передаваемая через некоторое соединение от источника к приемнику.
Программная инженерия – 1. Совокупность инженерных методов и средств создания ПО. 2. Дисциплина, изучающая применение строгого систематического количественного (т.е инженерного) подхода к разработке, эксплуатации и сопровождению ПО.
Программное обеспечение (программный продукт) – совокупность компьютерных программ, процедур и, возможно. связанной с ними документации и данных.
Прототип – действующий программный компонент, реализующий отдельные функции и внешние интерфейсы разрабатываемого ПО.
Процесс (ЖЦ ПО) – совокупность взаимосвязанных действий, преобразующих некоторые входные данные в выходные.
Процесс создания ПО – совокупность упорядоченных во времени, взаимосвязанных и объединенных в стадии работ, выполнение которых необходимо и достаточно для создания ПО, соответствующего заданным требованиям.

Процесс (на диаграмме потоков данных) – преобразование входных потоков данных в выходные в соответствии с определенным алгоритмом.
Разработка ПО – комплекс работ по созданию ПО и его компонентов в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации, подготовку материалов, требуемых для проверки работоспособности и соответствующего качества программных продуктов, материалов, необходимых для организации обучения персонала, и т.д.
Реверсный инжиниринг – перенос существующей системы ПО в новую среду.
Репозиторий – база данных, предназначенная для хранения проектных метаданных (версий проекта и его отдельных компонентов), синхронизации поступления информации от различных разработчиков при групповой разработке, контроля метаданных на полноту и непротиворечивость.
Связь – поименованная ассоциация между двумя сущностями, значимая для рассматриваемой предметной области.
Сопровождение ПО- внесение изменений в ПО в целях исправления ошибок, повышения производительности или адаптации к изменившимся условиям работы или требованиям.
Стадия ЖЦ ПО- часть процесса создания ПО, ограниченная определенными временными рамками и заканчивающаяся выпуском конкретного продукта (моделей ПО, программных компонентов, документации), определяемого заданными для данной стадии требованиями.
Сущность – реальной либо воображаемый объект, имеющий существенное значение для рассматриваемой предметной области.
Тестирование – процесс исполнения программы в целях обнаружения ошибки.
Технология проектирования ПО – совокупность технологических операций проектирования в их последовательности и взаимосвязи, приводящая к разработке проекта ПО.
Типизация – ограничение, накладываемое на класс объектов и препятствующее взаимозаменяемости различных классов (или сильно сужающее ее возможность).
Требование – условие или характеристика, которым должна удовлетворять система.
Уникальный идентификатор - атрибут или совокупность атрибутов и/или связей, предназначенные для уникальной идентификации каждого экземпляра данного типа сущности.
Управление требованиями – 1. Систематический подход к выявлению, организации и документированию требований к системе. 2. Процесс, устанавливающий соглашение между заказчиками и разработчиками относительно изменения требований к системе и обеспечивающий его выполнение.
Устойчивость – свойство объекта существовать во времени (вне зависимости от процесса, породившего данный объект) и/или в пространстве (при перемещении объекта из адресного пространства, в котором он был создан).
Функциональная декомпозиция – описание структуры системы в терминах иерархии ее функций и передачи информации между отдельными функциональными элементами.
Функциональная точка – любой и следующих элементов разрабатываемой системы:
- входной элемент приложения (входной документ или экранная форма);
- выходной элемент приложения (отчет, документ, экранная форма);
- запрос (пара «вопрос/ответ»);
- логический файл (совокупность записей данных, используемых внутри приложения);
- интерфейс приложения (совокупность записей данных, передаваемых другому приложению или получаемых от него).

CASE -средство – программное средство, поддерживающее процессы жизненного цикла ПО (определенные в стандарте ISO / IEC 12207:1995), включая анализ требований к системе, проектирование прикладного ПО и баз данных, генерацию кода, тестирование, документирование, обеспечение качества, конфигурационное управление и управление проектом, а также другие процессы.

Литература:

Основная:

1.Джеф Раскин, Интерфейс: новые направления в проектировании компьютерных систем. - Пер. с англ. - СПб.: Символ-Плюс,2003.
2. Торрес Р.Дж. Практическое руководство по проектированию и разработке пользовательского интерфейса. - Пер. с англ. - М: Вильяме, 2002.
3. Коутс Р., Влеймник И. Интерфейс "человек-машина" - М: Мир, 1990.
4.Алиев Т.М., Вигдоров Д.И., Кривошеее В.П. Системы отображения информации. - М.: Высшая школа, 1988.
5.Гасов В.М., Соломонов Л.А. Инженерно-психологическое проектирование взаимодействия человека с техническими средствами.Практическое пособие. //Под ред.Четверикова В.Н. - М.: Высшая школа, 1990.
6.Соломонов Л.А., Филипивич Ю.Н., Шульгин В.А. Персональные автоматизированные информационные системы. Практическое пособие.//Под ред.Четверикова В.Н. -М.:Высшая школа, 1990.
7.Гасов В.М., Меньков А.В., Соломонов Л.А., Шигин А.В. Системное проектирование взаимодействия человека с техническими системами.Практическое пособие.//Под ред.Четверикова В.Н. - М.: Высшая школа, 1991.
8.Гасов В.М., Коротаев А.И., Сенькин СИ. Отображение информации. Практиеское пособие. //Под ред.Четверикова В.Н. - М: Высшая школа, J 991.
9.Сальников Ю.В., Савченко А.В., Филипов А.Н. Средства общения с ЭВМ. //Под ред. Савельева А.Я. - М.;Высшая школа.,1987.

Дополнительная:

1.Айден К., Колесниченко О,. Крамер М., Фибельман Х., Шишигин И. Аппаратные" средства" PC. - СПб.; BHV,1998.
2. Борзенко А. IBM PC: устройство, ремонт, модернизация. - М.: 1995.
3.Венда В.Ф., Инженерная психология и синтез систем отображения информации. - М.: Машиностроение, 1975.
4.Смоляров А.М. Системы отображения информации и инженерная психология. - М.: Высшая школа, 1982.
5.Дракин В.И., Попов Э.В., Преображенский А.Б. Общение конечных пользователей с системами обработки данных. - М.: Радио и связь, 1988.
6.Основы инженерной психологии. //Под ред. В.Ф.Ломова - М.: Высшая школа. 1986.
7. Жумагалиев Б.И. Средства взаимодействия в автоматизированных системах. Учебное пособие. - Алматы: КазНТУ, 2001.

Формат 60х84 1\12
Объем 151 стр., 12,6 печатных листа
Тираж 20 экз.
Отпечатано
В Редакционно- издательском отделе
КГУТиИ им.Ш.Есенова
г.Актау, 32мкр.