Язык UML

Язык UML представляет собой общецелевой язык визуального моделирования, который разработан для спецификации, визуализации, проектирования и документирования компонентов программного обеспечения, бизнес-процессов и других систем. Язык UML является достаточно строгим и мощным средством моделирования, которое может быть эффективно использовано для построения концептуальных, логических и графических моделей сложных систем различного целевого назначения. Этот язык вобрал в себя наилучшие качества и опыт методов программной инженерии, которые с успехом использовались на протяжении последних лет при моделировании больших и сложных систем.

Канонические диаграммы языка UML

В рамках языка UML все представления о модели сложной системы фиксируются в виде специальных графических конструкций, получивших название диаграмм.

Диаграмма (diagram) — графическое представление совокупности элементов модели в форме связного графа, вершинам и ребрам (дугам) которого приписывается определенная семантика.

Раздаточный материал № 29

В языке UML определены следующие виды канонических диаграмм:

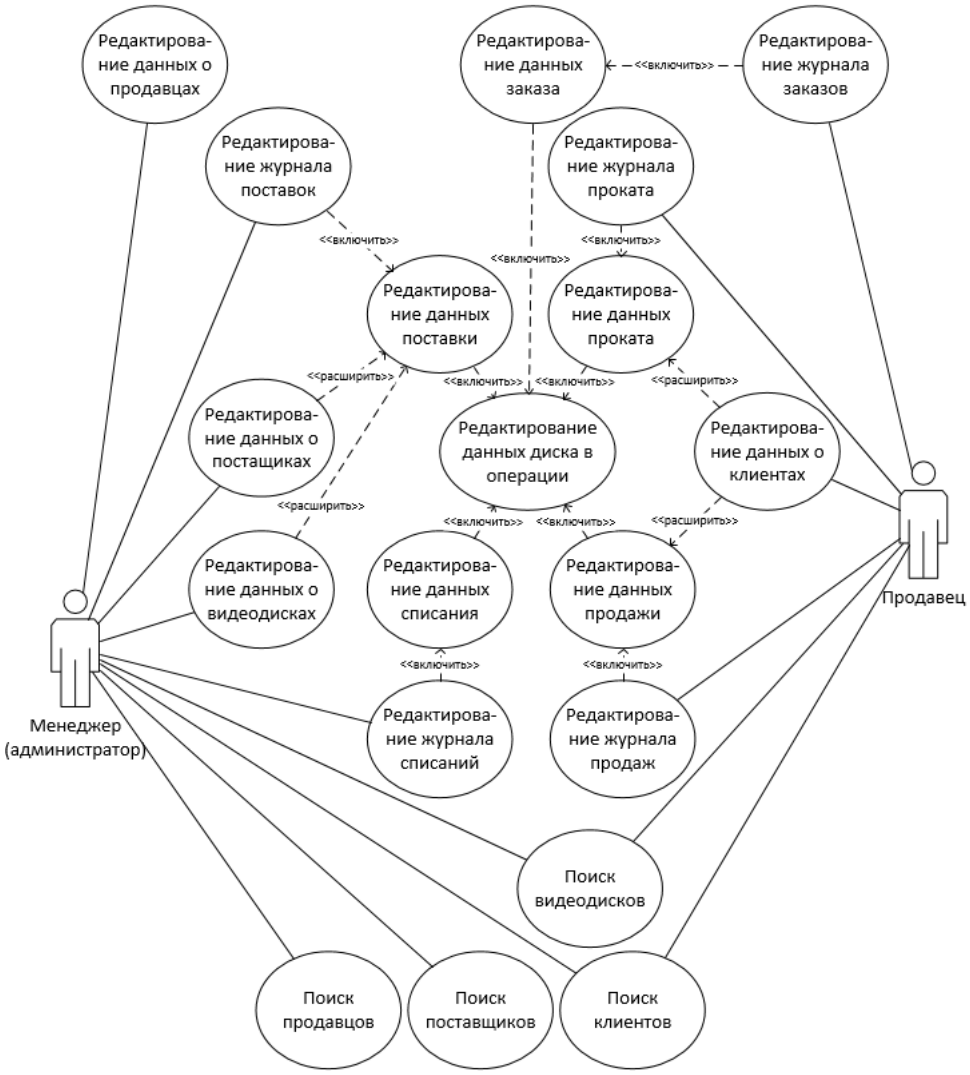
1. вариантов использования (use case diagram)
2. классов (class diagram)
3. кооперации (collaboration diagram)
4. последовательности (sequence diagram)
5. состояний (statechart diagram)
6. деятельности (activity diagram)
7. компонентов (component diagram)
8. развертывания (deployment diagram)

Перечень этих диаграмм и их названия являются каноническими в том смысле, что представляют собой неотъемлемую часть графической нотации языка UML. Более того, процесс ООАП неразрывно связан с процессом построения этих диаграмм. При этом совокупность построенных таким образом диаграмм является самодостаточной в том смысле, что в них содержится вся информация, которая необходима для реализации проекта сложной системы.

Каждая из этих диаграмм детализирует и конкретизирует различные представления о модели сложной системы в терминах языка UML.

Диаграмма вариантов использования представляет собой наиболее общую концептуальную модель сложной системы, которая является исходной для построения всех остальных диаграмм.

Раздаточный материал № 30



Большинство перечисленных выше диаграмм являются в своей основе графами специального вида, состоящими из вершин в форме геометрических фигур, которые связаны между собой ребрами или дугами. Поскольку информация, которую содержит в себе граф, носит топологический характер, ни геометрические размеры, ни расположение элементов диаграмм не имеют принципиального значения.

Для диаграмм языка UML существуют три типа визуальных графических обозначений, которые важны с точки зрения заключенной в них информации:

Геометрические фигуры на плоскости, играющие роль вершин графов соответствующих диаграмм. При этом сами геометрические фигуры выступают в роли графических примитивов языка UML, а форма этих фигур (прямоугольник, эллипс) должна строго соответствовать изображению отдельных элементов языка UML (класс, вариант использования, состояние, деятельность). Графические примитивы языка UML имеют фиксированную семантику, переопределять которую пользователям не допускается. Графические примитивы должны иметь собственные имена, а, возможно, и другой текст, который содержится внутри границ соответствующих геометрических фигур или, как исключение, вблизи этих фигур.

Графические взаимосвязи, которые представляются различными линиями на плоскости. Взаимосвязи в языке UML обобщают понятие дуг и ребер из теории графов, но имеют менее формальный характер и более развитую семантику.

Специальные графические символы, изображаемые вблизи от тех или иных визуальных элементов диаграмм и имеющие характер дополнительной спецификации (украшений).

Все диаграммы в языке UML изображаются с использованием фигур на плоскости. Отдельные элементы - с помощью геометрических фигур, которые могут иметь различную высоту и ширину с целью размещения внутри них других конструкций языка UML. Наиболее часто внутри таких символов помещаются строки текста, которые уточняют семантику или фиксируют отдельные свойства соответствующих элементов языка UML. Информация, содержащаяся внутри фигур, имеет значение для конкретной модели проектируемой системы, поскольку регламентирует реализацию соответствующих элементов в программном коде.

Пути представляют собой последовательности из отрезков линий, соединяющих отдельные графические символы. При этом концевые точки отрезков линий должны обязательно соприкасаться с геометрическими фигурами, служащими для обозначения вершин диаграмм, как принято в теории графов. С концептуальной точки зрения путям в языке UML придается особое значение, поскольку это простые топологические сущности. Отдельные части пути или сегменты могут не существовать вне содержащего их пути. Пути всегда соприкасаются с другими графическими символами на обеих границах соответствующих отрезков линий, т.е. пути не могут обрываться на диаграмме линией, которая не соприкасается ни с одним графическим символом. Как отмечалось выше, пути могут иметь в качестве окончания или терминатора специальную графическую фигуру – значок, который изображается на одном из концов линий.

Дополнительные значки или украшения представляют собой графические фигуры фиксированного размера и формы. Они не могут увеличивать свои размеры, чтобы разместить внутри себя дополнительные символы. Значки размещаются как внутри других графических конструкций, так и вне их. Примерами значков могут служить окончания связей элементов диаграмм или графические обозначения кванторов видимости атрибутов и операций классов.

Строки текста служат для представления различных видов информации в грамматической форме. Предполагается, что каждое использование строки текста должно соответствовать синтаксису в нотации языка UML. В отдельных случаях может быть реализован грамматический разбор этой строки, который необходим для получения дополнительной информации о модели. Например, строки текста в различных секциях обозначения класса могут соответствовать атрибутам этого класса или его операциям. На использование строк накладывается условие: требуется, чтобы семантика всех допустимых символов была заранее определена в языке UML или служила предметом его расширения в конкретной модели.

Раздаточный материал № 31

Рекомендации по графическому изображению диаграмм языка UML

При графическом изображении диаграмм следует придерживаться следующих основных рекомендаций:

Каждая диаграмма должна служить законченным представлением соответствующего фрагмента моделируемой предметной области. Речь идет о том, что в процессе разработки диаграммы необходимо учесть все сущности, важные с точки зрения контекста данной модели и диаграммы. Отсутствие тех или иных элементов на диаграмме служит признаком неполноты модели и может потребовать ее последующей доработки.

Все сущности на диаграмме модели должны быть одного уровня представления. Здесь имеется в виду согласованность не только имен одинаковых элементов, но и возможность вложения отдельных диаграмм друг в друга для достижения полноты представлений. В случае достаточно сложных моделей систем желательно придерживаться стратегии последовательного уточнения или детализации отдельных диаграмм.

Вся информация о сущностях должна быть явно представлена на диаграммах. В языке UML при отсутствии некоторых символов на диаграмме могут быть использованы их значения по умолчанию (например, в случае неявного указания видимости атрибутов и операций классов), тем не менее, необходимо стремиться к явному указанию свойств всех элементов диаграмм.

Диаграммы не должны содержать противоречивой информации. Противоречивость модели может служить причиной серьезных проблем при ее реализации и последующем использовании на практике. Например, наличие замкнутых путей при изображении отношений агрегирования или композиции приводит к ошибкам в программном коде, который будет реализовывать соответствующие классы. Наличие элементов с одинаковыми именами и различными атрибутами свойств в одном пространстве имен также приводит к неоднозначной интерпретации и может быть источником проблем.

Каждая диаграмма должна быть самодостаточной для правильной интерпретации всех ее элементов и понимания семантики всех используемых графических символов. Любые пояснительные тексты, которые не являются собственными элементами диаграммы (например, комментариями), не должны приниматься во внимание разработчиками. В то же время общие фрагменты диаграмм могут уточняться или детализироваться на других диаграммах этого же типа, образуя вложенные или подчиненные диаграммы. Таким образом, модель системы на языке UML представляет собой пакет иерархически вложенных диаграмм, детализация которых должна быть достаточной для последующей генерации программного кода, реализующего проект соответствующей системы.

Количество типов диаграмм для конкретной модели приложения строго не фиксировано. Для простых приложений нет необходимости строить все без исключения типы диаграмм. Некоторые из них могут просто отсутствовать в проекте системы, и это не будет считаться ошибкой разработчика. Например, модель системы может не содержать диаграмму развертывания для приложения, выполняемого локально на компьютере пользователя. Важно понимать, что перечень диаграмм зависит от специфики конкретного проекта системы.

Любая модель системы должна содержать только те элементы, которые определены в нотации языка UML. Имеется в виду требование начинать разработку проекта, используя только те конструкции, которые уже определены в метамодели UML. Как показывает практика, этих конструкций вполне достаточно для представления большинства типовых проектов программных систем. И только при отсутствии необходимых базовых элементов языка UML следует использовать механизмы их расширения для адекватного представления конкретной модели системы. Не допускается переопределение семантики тех элементов, которые отнесены к базовой нотации метамодели языка UML.

Вопросы:

- назначение языка UML
- перечислить канонические диаграммы.
- перечислить типа визуальных графических обозначений.
- дать короткие рекомендации по графическому изображению диаграмм языка UML.

Диаграмма вариантов использования

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

Разработка диаграммы вариантов использования преследует такие цели:

- □ определить общие границы и контекст моделируемой предметной области на начальных этапах проектирования системы;
- □ сформулировать общие требования к функциональному поведению проектируемой системы;
- □ разработать исходную концептуальную модель системы для ее последующей детализации в форме логических и физических моделей;
- □ подготовить исходную документацию для взаимодействия разработчиков системы с ее заказчиками и пользователями.

Суть данной диаграммы состоит в следующем: проектируемая система представляется в форме так называемых вариантов использования, с которыми взаимодействуют некоторые внешние сущности или актеры. При этом актером или действующим лицом называется любой объект, субъект или система, взаимодействующая с моделируемой системой извне. Это может быть человек, техническое устройство, программа или любая другая система, которая может служить источником воздействия на моделируемую систему так, как определит сам разработчик. В свою очередь вариант использования служит для описания сервисов, которые система предоставляет актеру. Другими словами, каждый вариант использования определяет некоторый набор действий, совершаемый системой при диалоге с актером. При этом ничего не говорится о том, каким образом будет реализовано взаимодействие актеров с системой и собственно выполнение вариантов использования.

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

Базовые элементы диаграммы вариантов использования — собственно вариант использования и актер.

Вариант использования

Вариант использования (use case) представляет собой спецификацию общих особенностей поведения или функционирования моделируемой системы без рассмотрения внутренней структуры этой системы.

Цель определения варианта использования заключается в том, чтобы зафиксировать некоторый аспект или фрагмент поведения проектируемой системы без указания особенностей реализации данной функциональности. В этом смысле каждый

вариант использования соответствует отдельному сервису, который предоставляет моделируемая система по запросу актера, т. е. определяет один из способов применения системы. Сервис, который инициализируется по запросу актера, должен представлять собой законченную последовательность действий. Это означает, что после того как система закончит обработку запроса актера, она должна возвратиться в исходное состояние, в котором она готова к выполнению следующих запросов.

Диаграмма вариантов использования должна содержать конечное множество вариантов использования, которые в целом определяют все возможные стороны ожидаемого поведения системы.

Примерами вариантов использования могут являться следующие действия: проверка состояния текущего счета клиента, оформление заказа на покупку товара, получение дополнительной информации о кредитоспособности клиента, отображение графической формы на экране монитора и другие действия.

Актеры

Актер (actor) представляет собой любую внешнюю по отношению к моделируемой системе сущность, которая взаимодействует с системой и использует ее функциональные возможности для достижения определенных целей или решения частных задач. При этом актеры служат для обозначения согласованного множества ролей, которые могут играть пользователи в процессе взаимодействия с проектируемой системой. Стандартным графическим обозначением актера на диаграммах является фигурка "человечка", под которой записывается конкретное имя актера с заглавной буквы.

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

В качестве актеров могут выступать другие системы, в том числе подсистемы проектируемой системы или ее отдельные классы.

Для моделируемой системы актерами могут быть как субъекты-пользователи, так и другие системы.

В общем случае, актер всегда находится вне системы, его внутренняя структура никак не определяется. Для актера имеет значение только его внешнее представление, т. е. то, как он воспринимается со стороны системы. Актеры взаимодействуют с системой посредством передачи и приема сообщений от вариантов использования. Сообщение представляет собой запрос актером сервиса от системы и получение этого сервиса.

Раздаточный материал № 32 (самостоятельно)

Для идентификации актеров в процессе проектирования системы могут быть рекомендованы вопросы, ответы на которые должны помочь разработчикам на начальных этапах выполнения проекта.

1. Какие организации или лица будут использовать проектируемую систему?
2. Кто будет получать пользу от использования системы?
3. Кто будет использовать информацию от системы?
4. Будет ли использовать система внешние ресурсы?
5. Может ли один пользователь играть несколько ролей при взаимодействии с системой?
6. Могут ли различные пользователи играть одну роль при взаимодействии с системой?

7. Будет ли система взаимодействовать с законодательными, исполнительными, налоговыми или другими органами?

Диаграммы классов

Диаграмма классов (class diagram) служит для представления статической структуры модели системы в терминологии классов объектно-ориентированного программирования. Диаграмма классов может отражать, в частности, различные взаимосвязи между отдельными сущностями предметной области, такими как объекты и подсистемы, а также описывает их внутреннюю структуру и типы отношений.

Класс (class) в языке UML является абстрактным описанием или представлением свойств множества объектов, которые обладают одинаковой структурой, поведением и отношениями с объектами из других классов. Графически класс в нотации языка UML изображается в виде прямоугольника, который дополнительно может быть разделен горизонтальными линиями на разделы или секции. В этих секциях могут указываться имя класса, атрибуты (переменные) и операции (методы).

Обязательным элементом обозначения класса является его имя. На начальных этапах разработки диаграммы отдельные классы могут обозначаться простым прямоугольником с указанием только имени соответствующего класса (рис. 5.1, *а).* По мере проработки отдельных компонентов диаграммы описания классов дополняются атрибутами (рис. 5.1, *б)* и операциями. Предполагается, что окончательный вариант диаграммы содержит наиболее полное описание классов, которые состоят из трех разделов или секций.

Имя класса должно быть уникальным в пределах пакета, который может содержать несколько диаграмм классов или, возможно, только одну диаграмму. Имя указывается в самой верхней секции прямоугольника, поэтому эта секция часто называется секцией имени класса. В дополнение к общему правилу именования элементов языка UML, имя класса записывается по центру секции имени полужирным шрифтом и должно начинаться с заглавной буквы. Рекомендуется в качестве имен классов использовать существительные, записанные по практическим соображениям без пробелов.

Примерами имен классов могут быть такие существительные, как Сотрудник, Компания, Руководитель, Клиент, Продавец, Менеджер, Офис и многие другие, имеющие непосредственное отношение к моделируемой предметной области и функциональному назначению проектируемой системы.

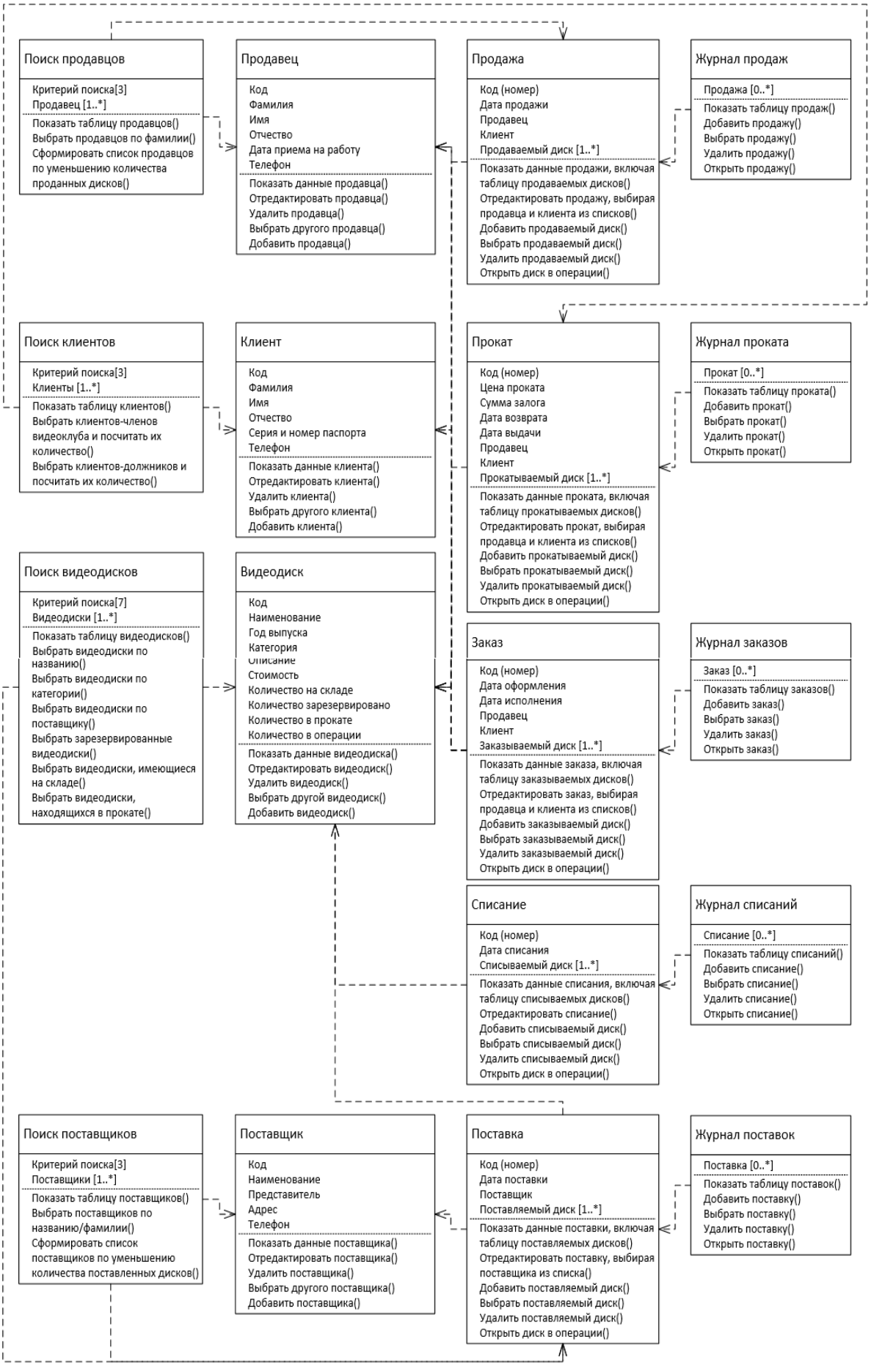
Класс может не иметь экземпляров или объектов. В этом случае он называется абстрактным классом, а для обозначения его имени используется наклонный шрифт (курсив). В языке UML принято общее соглашение о том, что любой текст, относящийся к абстрактному элементу, записывается курсивом.

Атрибут (attribute) класса служит для представления отдельного свойства или признака, который является общим для всех объектов данного класса. Атрибуты класса записываются во второй сверху секции прямоугольника класса, поэтому эту секцию часто называют *секцией атрибутов.*

Имя атрибута представляет собой строку текста, которая используется в качестве идентификатора соответствующего атрибута и поэтому должна быть уникальной в пределах данного класса. Имя атрибута является единственным обязательным элементом синтаксического обозначения атрибута, должно начинаться со срочной (малой) буквы и, как правило, не должно содержать пробелов.

Кратность атрибута характеризует общее количество конкретных атрибутов данного типа, входящих в состав отдельного класса. В общем случае кратность записывается в форме строки из цифр в квадратных скобках после имени соответствующего атрибута, при этом цифры разделяются двоеточием.

Раздаточный материал № 33



Раздаточный материал № 34 (справочно)

В качестве примера - следующие варианты задания кратности атрибутов:

[0. .1] — означает, что кратность атрибута может принимать значение 0 или 1. При этом 0 означает отсутствие данного атрибута у отдельных объектов рассматриваемого класса.

[0. .*] — означает, что кратность атрибута может принимать любое положительное целое значение большее или равное 0. Эта кратность может быть записана короче в виде простого символа — [*].

[1..*] — означает, что кратность атрибута может принимать любое положительное целое значение большее или равное 1.

[1. .5] — означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, 4, 5.

[1. .3,5,7] — означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, 5, 7.

[1. .3,7. .10] — означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, 7, 8, 9, 10.

[1. .3,7.. *] — означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, а также любое положительное целое значение большее или равное 7.

Если кратность атрибута не указана, то по умолчанию принимается ее значение равное [1.. 1], т. е. в точности 1.

Операция (operation) — это некоторый сервис, который предоставляет каждый экземпляр или объект класса по требованию своих клиентов (других объектов, в том числе и экземпляров данного класса). Операции класса записываются в третьей сверху секции прямоугольника класса, поэтому эту секцию часто называют секцией операций. Совокупность операций характеризует функциональный аспект поведения всех объектов данного класса.

Диаграмма деятельности

При моделировании поведения проектируемой возникает необходимость детализировать особенности алгоритмической и процедурной реализации выполняемых системой операций. Традиционно для этой цели использовались блок-схемы или структурные схемы алгоритмов. Каждая такая схема акцентирует внимание на последовательности выполнения определенных действий или элементарных операций, которые в совокупности приводят к получению желаемого результата.

С алгоритмическими и логическими операциями, требующими своего выполнения в определенной последовательности, мы сталкиваемся в самых различных бытовых и деловых ситуациях. Конечно, мы не всегда задумываемся о том, что подобные операции относятся к столь научным категориям. Например, чтобы позвонить по телефону, нам предварительно нужно снять трубку и убедиться, что телефон подключен к линии. Для приготовления кофе или заваривания чая необходимо вначале вскипятить воду. Чтобы выполнить ремонт двигателя автомобиля, требуется осуществить целый ряд нетривиальных операций, таких как разборка силового агрегата, снятие генератора и некоторых других.

Важно подчеркнуть то обстоятельство, что с увеличением сложности системы строгое соблюдение определенной последовательности выполняемых действий приобретает все более важное значение. Если попытаться заварить кофе холодной водой, то мы можем только испортить одну порцию напитка. Нарушение последовательности операций при ремонте двигателя может привести к его поломке или выходу из строя. Еще более катастрофические последствия могут произойти в случае отклонения от установленной последовательности действий при взлете или посадке авиалайнера, запуске ракеты, регламентных работах на АЭС.

Для моделирования процесса выполнения операций в языке UML используются так называемые диаграммы деятельности, в которой присутствуют обозначения состояний и переходов. Каждое состояние на диаграмме деятельности соответствует выполнению некоторой элементарной операции, а переход в следующее состояние срабатывает только при завершении операции в предыдущем состоянии. Графически диаграмма деятельности

представляется в форме графа деятельности, вершинами которого являются состояния действия, а дугами — переходы от одного состояния действия к другому.

Основным направлением использования диаграмм деятельности является визуализация особенностей реализации операций классов, когда необходимо представить алгоритмы их выполнения. При этом каждое состояние может являться выполнением операции некоторого класса либо ее части, позволяя использовать диаграммы деятельности для описания реакций на внутренние события системы.

В контексте языка UML деятельность (activity) представляет собой некоторую совокупность отдельных вычислений, выполняемых автоматом. При этом отдельные элементарные вычисления могут приводить к некоторому результату или действию (action). На диаграмме деятельности отображается логика или последовательность перехода от одной деятельности к другой, при этом внимание фиксируется на результате деятельности. Сам же результат может привести к изменению состояния системы или возвращению некоторого значения.

Состояние действия

Состояние действия (action state) является специальным случаем состояния с некоторым входным действием и, по крайней мере, одним выходящим из состояния переходом. Этот переход неявно предполагает, что входное действие уже завершилось. Состояние действия не может иметь внутренних переходов, поскольку оно является элементарным. Обычное использование состояния действия заключается в моделировании одного шага выполнения алгоритма (процедуры) или потока управления.

Графически состояние действия изображается фигурой, напоминающей прямоугольник со сферическими боковыми сторонами. Внутри этой фигуры записывается имя состояния действия в форме выражение действия (action-expression), которое должно быть уникальным в пределах одной диаграммы деятельности.

Раздаточный материал № 35

а *б*

простая деятельность *выражение*

Действие может быть записано на естественном языке, некотором псевдокоде или языке программирования. Никаких дополнительных или неявных ограничений при записи действий не накладывается. Рекомендуется в качестве имени простого действия использовать глагол с пояснительными словами (РМ № 35, а). Если же действие может быть представлено в некотором формальном виде, то целесообразно записать его на том языке программирования, на котором предполагается реализовывать разрабатываемый проект (РМ № 35, б).

Простой переход (simple transition) представляет собой отношение между двумя последовательными состояниями, которое указывает на факт смены одного состояния

другим. Переход осуществляется при наступлении некоторого события: окончания выполнения деятельности (do activity), получении объектом сообщения или приемом сигнала. На переходе указывается имя события. Кроме того, на переходе могут указываться действия, производимые объектом в ответ на внешние события при переходе из одного состояния в другое. Срабатывание перехода может зависеть не только от наступления некоторого события, но и от выполнения определенного условия (сторожевое условие).

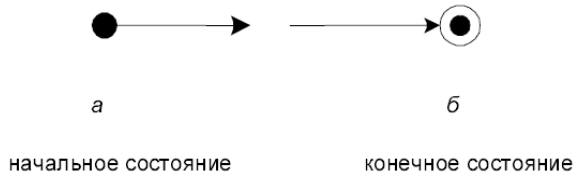
На диаграмме состояний переход изображается сплошной линией со стрелкой, которая выходит из исходного состояния и направлена в целевое состояние

Начальное и конечное состояния

В начальном состоянии объект находится по умолчанию в начальный момент. Оно служит для указания на диаграмме состояний графической области, от которой начинается процесс изменения состояний. Графически начальное состояние в языке UML обозначается в виде закрашенного кружка (РМ № 36, а), из которого может только выходить стрелка-переход.

В конечном состоянии должен находиться моделируемый объект или система по умолчанию после завершения работы конечного автомата. Оно служит для указания на диаграмме состояний графической области, в которой завершается процесс изменения состояний или жизненный цикл данного объекта. Графически конечное состояние в языке UML обозначается в виде закрашенного кружка, помещенного в окружность (РМ № 36, б), в которую может только входить стрелка- переход.

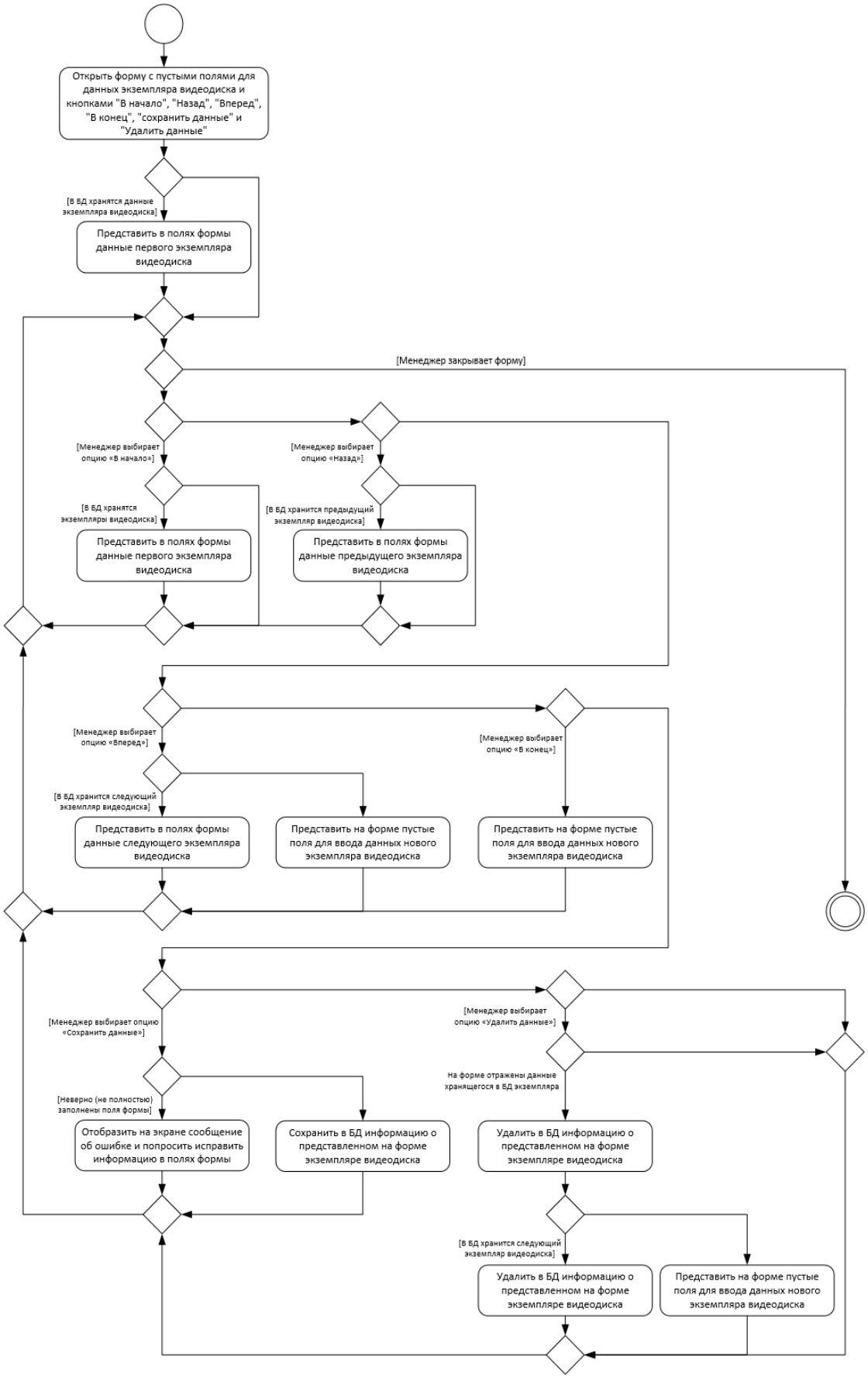
Раздаточный материал № 36



а *б*

начальное состояние конечное состояние

Раздаточный материал № 37



Архитектура ПО

Архитектуру информационной системы можно представить в виде следующих компонентов:

Слой представления — клиентская часть с графическим интерфейсом пользователя (GUI, Graphical User Interface), которая выполняет роль терминала — средства представления данных и отправки команд. Часто этот слой называют frontend.

Слой бизнес-логики, где происходит обработка команд, полученных от клиента и выполняются основные вычисления. Чаще всего это реализуется в виде серверного приложения (backend). Здесь же располагается система управления базой данных (СУБД) как надстройка над базой данной (БД), которая позволяет обратиться к данным и манипулировать ими, о чем мы писали здесь.

Слой доступа к данным, т.е. сама БД как хранилище данных в структурированном виде, что в конечном итоге на низком уровне сводится к файлам с записанными данными.

Трехслойная архитектура информационной системы

Раздаточный материал № 38

Такая послойная модель компонентного строения ИС получила название 3-х слойной архитектуры и сегодня реализуется везде. Однако, реализация этих 3-х слоев может быть выполнена по-разному. Если все компоненты всех 3-х слоев находятся на одном устройстве (компьютер, мобильный телефон), такая архитектура называется настольной (desktop). Например, локальный (не облачный) текстовый редактор, калькулятор. А если компоненты 3-х слоев ИС распределены по нескольким устройствам, ее архитектура называется распределенной (distributed). Именно такая архитектура сегодня встречается в большинстве современных ИС.

Основные виды распределенных архитектур

Поскольку компоненты распределенной системы распределены по разным устройствам (узлам), им необходимо средство взаимодействия друг с другом, т.е. сеть

передачи информации. Передача данных по сети выполняется по правилам сетевых протоколов, которые регламентируют 7-ми уровневая модель OSI (Open Systems Interconnection).

В зависимости от расположения компонентов 3-х слоев ИС на клиенте и сервере, различают следующие виды распределенных архитектур:

Файл-серверная – самая примитивная распределенная архитектура, когда слои представления и бизнес-логики находятся на клиенте, а также там реализуется часть вычислений, т.е. операторов по обработке данных. Сервер отвечает только за хранение и управление файлами. Это простое и дешевое с точки зрения реализации решение подходит только ИС, небольших по количеству пользователей, имеет низкую производительность и предполагает передачу по сети огромного объема данных.

Клиент-серверная, которая делится на 2 подвида:

двузвенная – своего рода развитие файл-серверной версии. Она также обеспечивает многопользовательскую работу с данными, но имеет более высокую надежность, т.к. теперь на клиенте находится лишь слой представления и часть слоя бизнес-логики, такая как операторы обращения к СУБД. А другая часть манипулирования с данными, зашитая в СУБД, например, хранимые процедуры (объект БД в виде набора SQL-инструкций, компилируется лишь однажды и хранится на сервере), непосредственное выполнение запросов, обработка транзакций, а также само хранение файлов с данными и управление ими – реализуется на мощном сервере. Это снижает требования к клиентскому узлу, но не устраняет необходимость передачи большого объема данных по сети. Клиент становится тоньше по сравнению с файл-серверной моделью, но это все еще «толстый клиент».

трехзвенная — устраняет недостатки двухзвенной архитектуры, располагая каждый из слоев на отдельном узле. Теперь на клиенте находится только пользовательский интерфейс со средствами вывода данных и ввода команд. По сути, клиент превращается в терминал и называется «тонкий клиент». За выполнение вычислений и формирование запросов к СУБД отвечает сервер приложений, а слой доступа к данным в виде СУБД и БД находится на отдельном сервере данных. Таким образом, трехзвенная архитектура устраняет почти все недостатки файл-серверной и клиент-серверной на 2-х звеньях ценой увеличения расходов на администрирование и разработку серверных частей.

Сравнение распределенных архитектур ИС

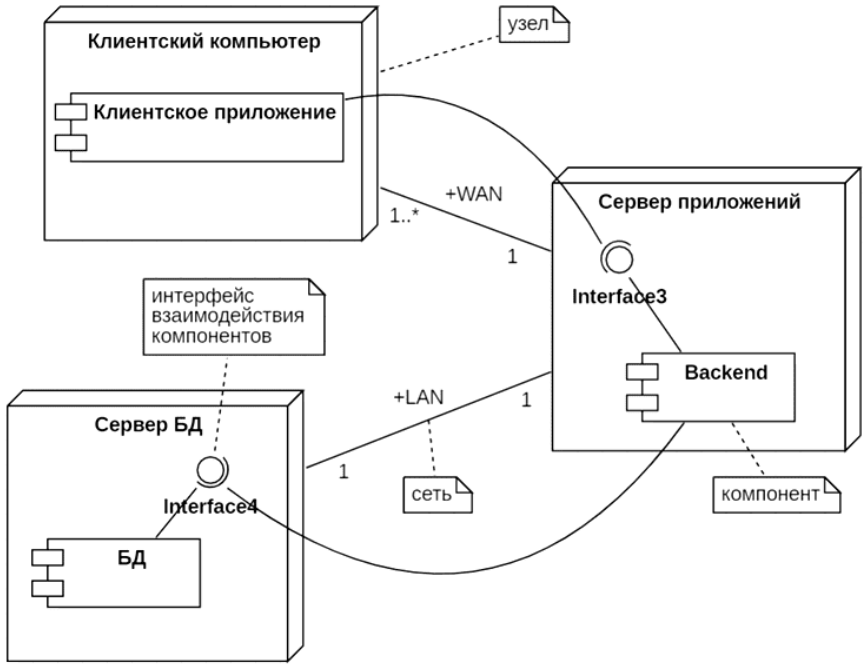
Раздаточный материал № 39 (справочно)

<table>
<tr><th rowspan="3">Компонент и критерий оценки</th><th colspan="3">Архитектура</th></tr>
<tr><th rowspan="2">Файл-сервер</th><th colspan="2">Клиент-сервер</th></tr>
<tr><th>Двухзвенная</th><th>Трехзвенная</th></tr>
<tr><td>Слой представления</td><td>На клиенте</td><td>На клиенте</td><td>На клиенте</td></tr>
<tr><td>Слой бизнес-логики</td><td>На клиенте</td><td>Частично на клиенте, частично на сервере</td><td>На сервере приложений</td></tr>
<tr><td>Слой доступа к данным</td><td>Частично на клиенте, частично на сервере</td><td>На сервере</td><td>На сервере данных</td></tr>
<tr><td>Достоинства</td><td>Низкая стоимость и высокая скорость разработки</td><td>Гарантия целостности данных</td><td>• Тонкий клиент
• минимальная передача данных между клиентом и сервером по сети (только аргументы функций и результаты вычислений)
• высокая масштабируемость
• небольшой трафик между серверами
• снижение нагрузки на сервер данных
• простота расширения функциональных возможностей и обновления</td></tr>
<tr><td rowspan="3">Недостатки</td><td rowspan="2">• Низкая производительность
• Низкая надежность
• Низкая масштабируемость и расширяемость</td><td>• сложность изменения бизнес-логики
• слабая защита данных от взлома</td><td></td></tr>
<tr><td colspan="2">• сложность администрирования и разработки
• высокие расходы на администрирование и разработку серверной части</td></tr>
<tr><td colspan="2">• высокие требования к вычислительной мощности клиента (ЦП, ОЗУ, диск)
• большой объем данных, передаваемых от клиента к серверу
• высокие требования к пропускной способности сети и клиентам</td><td></td></tr>
</table>

Можно графически изобразить развертывание компонентов всех 3-х слоев трехзвенной архитектуры по узлам в виде соответствующей UML-диаграммы (deployment).

Раздаточный материал № 40

Диаграмма развертывания UML пример для трехзвенной монолитной архитектуры ИС



Если все функциональные возможности ИС реализованы в виде одного, а не нескольких серверных компонентов, и поддержаны единой базой данных, это соответствует монолитной архитектуре. Если же весь набор функциональных возможностей ИС представлен не одним, а несколькими backend'ами, которые взаимодействуют между собой и каждый из них имеет свою ограниченную по контексту область действия, такая архитектура называется микросервисной. Именно по такому принципу реализуется большинство современных ИС. Например, интернет-банк – это целый набор сервисов: сервис по работе с банковскими картами, сервис кредитов, сервис депозитов и пр. Такая реализация принципа единой ответственности ускоряет время и снижает скорость разработки отдельного сервиса, однако, усложняет проектирование взаимодействия разных модулей этой распределенной системы.

Тестирование ПО

Раздаточный материал № 41

Наличие ошибок в любой программе весьма справедливо описывается следующими шуточными аксиомами:

1. Любая программа содержит ошибки.
2. Если программа не содержит ошибок, их содержит алгоритм, который реализует эта программа.
3. Если ни программа, ни алгоритм ошибок не содержат, такая программа даром никому не нужна.

В общем-то, практика показывает, что так оно и есть. Helloworld можно написать и без ошибок, однако практическая ценность такой программы сомнительна. А любая достаточно сложная программная система ошибки в том или ином виде содержит. Помочь обнаружить ошибки и призвано тестирование.

Важно понимать, что тестирование не может доказать отсутствие ошибок в программе, оно может лишь помочь обнаружить ошибки, которые в программе присутствуют.

Сами по себе тесты делятся на разные виды. Например, по объекту тестирования:

- функциональные: тестирование основной функциональности программы;
- тестирование производительности при разных условиях:
    - нагрузочное позволяет проверить работу в штатном режиме;
    - стресс-тестирование позволяет нагрузить систему больше, чем положено, и посмотреть, что с ней будет: она может и выдержать, а если не выдержит, то бывает полезно посмотреть, как именно система упала, и найти тонкие места в проекте;
    - тестирование стабильности — проверка работы системы на протяжении длительных промежутков времени.
- тестирование удобства использования: тестирование с использованием эргономических метрик;
- тестирование интерфейса пользователя: тестирование времени отклика, корректной работы и т.д.;
- тестирование безопасности: проверка механизмов аутентификации и авторизации, проверка алгоритмов шифрования;
- тестирование локализации;
- тестирование совместимости;
- и т.д.

По знанию о системе:

- white box: тестировщик обладает знаниями о внутреннем устройстве системы и ее программном коде;
- black box: тестировщик не обладает знаниями о внутреннем устройстве системы и тестирует лишь определяемый ей контракт взаимодействия. Такие тестировщики дешевле, зато потенциально могут найти не так много ошибок, как те, которые смотрят при тестировании в код.

Аналогично тестирование можно разделять по степени автоматизации (ручное и автоматизированное), по степени подготовленности (по документации и ad-hoc), по времени проведения и т.д.

> Свободное тестирование (ad-hoc testing) - это вид тестирования, который выполняется без подготовки к тестированию продукта, без определения ожидаемых результатов, проектирования тестовых сценариев. Это неформальное, импровизационное тестирование.

Ещё одним крайне важным делением представляется деление по изолированности компонент:

- модульное тестирование (юнит-тесты) — тестируются отдельные модули системы (функции, методы, классы, и т.д.);
- интеграционное тестирование — тестируется взаимодействие набора модулей системы;
- системное тестирование — тестируется вся система целиком, в рабочих условиях.

Интеграционное тестирование и системное тестирование обычно выполняются специально обученными людьми, а вот модульное тестирование — самими программистами.

Модульное тестирование заключается в том, что для каждой нетривиальной функции или метода пишутся свои тесты, которые проверяют, что метод работает в соответствии со своим контрактом. Предполагается, что модульные тесты будут запускаться после каждой сборки программы, поэтому они должны работать как можно быстрее. В большинстве случае модульные тесты используются как регрессионные, то есть они гарантируют, что изменения в системе не нарушили контракты ее интерфейсов и не привели к нарушению ее функциональности.

Наличие хорошей тестовой базы имеет целый ряд преимуществ для проекта.

- **Тесты помогают искать ошибки в коде.** Если тесты запускаются достаточно часто, вы сможете оперативно отследить, что вы такое поменяли, после чего тесты перестали проходить.
- **Тесты облегчают изменение программы.** Вы что-то поменяли, что не должно отразиться на наблюдаемом поведении вашего класса или метода, запускаете тесты и смотрите, что они действительно все прошли. Подобные изменения, направленные на улучшение сопровождаемости без изменений в поведении программы с точки зрения пользователя, называются рефакторингом, вот юнит-тесты для рефакторинга чрезвычайно полезны. Они дают некую уверенность, что ничего не сломалось ненароком.
- **Тесты можно рассматривать как документацию к коду.** Любой, кто захочет воспользоваться вашим классом, может посмотреть в ваши юнит-тесты к этому классу и посмотреть, как его использовать.
- **Тесты помогают улучшить структуру программы.** Яростную мешанину в коде невозможно толком оттестировать, так что волей-неволей придётся задуматься об аккуратной архитектуре.

Что бы понять место модульного тестирования смотрим пирамиду тестирования

Раздаточный материал № 42

Участки пирамиды подразумевают количество необходимых тестов данной категории. Чем больше площадь, тем больше необходимо тестов. Чем ниже находятся на пирамиде тесты, тем:

- проще и быстрее они должны разрабатываться.
- ниже затраты на поддержку тестов.
- быстрее скорость прохождения отдельного теста.

Перечисленные факторы влияют на выбор между устойчивостью к багам и быстрой обратной связью.

Сквозное тестирование (End-to-end, E2E, Chain testing) — это вид тестирования, используемый для проверки программного обеспечения от начала до конца, а также его интеграцию с внешними интерфейсами. Цель сквозного тестирования состоит в проверке всего программного обеспечения на предмет зависимостей, целостности данных и связи с другими системами, интерфейсами и базами данных для проверки успешного выполнения полного производственного сценария.

Раздаточный материал № 43

Сквозное тестирование vs системное тестирование

Сквозное тестирование	Системное тестирование
Проверяет программную систему, а также взаимосвязанные подсистемы.	Проверяет только программную систему в соответствии со спецификациями требований.
Проверяет весь сквозной поток процессов.	Проверяет функциональные возможности и функции системы.
Для тестирования рассматриваются все интерфейсы и серверные системы.	Рассматриваются функциональное и нефункциональное тестирование
Выполняется после завершения тестирования системы.	Выполняется после интеграционного тестирования.
Сквозное тестирование включает в себя проверку внешних интерфейсов, которую сложно автоматизировать. Следовательно, ручное тестирование предпочтительнее.	Для тестирования системы можно выполнять как ручное, так и автоматизированное тестирование.

Тест-кейсы

В тестировании, чтобы проверить, корректно ли работает программное обеспечение (ПО), делают определенные действия и сверяют полученный результат с ожидаемым. Другими словами — моделируют ситуацию работы ПО. Чтобы описать шаги, создают тест-кейсы.

Тест-кейсы - это четкое описание входных данных, условий и процедуры тестирования, ожидаемых результатов. Они определяют один сценарий — конкретную цель тестирования программного обеспечения. Целью может быть проверка ПО: соответствует ли оно требованиям.

Четко определенные тест-кейсы позволяют многократно запускать одни и те же тесты, применять для последовательно изменяющихся версий программного обеспечения. А еще отслеживать регрессивные ошибки ПО — то есть те, которые повторяются и ухудшают качество продукта.

Виды тест-кейсов

Классификация зависит от типа входных данных, действий и ожидаемого поведения ПО:

- Положительные. Подтверждают, что ПО соответствует требованиям. Показывают, что при корректных входных данных и действиях пользователя ПО выполняет функции.
- Отрицательные. Показывают, что ПО способно обрабатывать некорректные входные данные или неверные действия пользователя. Например, выводить соответствующие сообщения, подсказывать, как исправить ситуацию.
- Деструктивные. Демонстрируют, что никакие внешние воздействия или высокие нагрузки не приводят к потере данных пользователя, ПО можно использовать. Условие: нагрузки не разрушают аппаратную часть.

Жизненный цикл тест-кейса выражается в наборе состояний:

Раздаточный материал № 44

- Не запускался. Тест-кейс создали, но тестирование по нему не проводили.
- Пройден успешно. Ожидаемые и фактические результаты работы ПО совпадают.
- Провален. Обнаружили дефект: ожидаемый результат минимум по одному шагу тест-кейса не совпадает с фактическим.
- Пропущен. Тест-кейс отменили. Например, потому что изменили требования к ПО.

Обязательные атрибуты тест-кейса:

Раздаточный материал № 45

✓ Уникальный идентификатор — некое уникальное значение. По нему на тест-кейс ссылаются из других документов или тест-кейсов. Бывает буквенным, числовым, буквенно-числовым. Чаще всего применяют простую сквозную нумерацию.

✓ Краткое описание — лаконичное описание сути тест-кейса. Может содержать ссылку на требование к ПО.

✓ Входные данные — сведения о первоначальном состоянии системы, которое важно для тест-кейса. А еще значения для ввода или передачи ПО.

✓ Шаги — полная последовательность действий. Ее выполняют, чтобы провести описываемую тест-кейсом проверку.

✓ Ожидаемый результат — описание планируемого поведения или результата ПО. Может базироваться на требовании к программному обеспечению, общей логике работы.

✓ Фактический результат — описание итогового поведения или результата ПО. Если они совпадают, это указывают. Когда не совпадают, подробно описывают расхождения. Пометка «не совпадает», «отличается» — это грубая ошибка.

✓ Статус — текущее состояние тест-кейса.

Раздаточный материал № 46

Шаблон и пример тест-кейса

Идентификатор	Описание	Шаги	Входные данные	Ожидаемые результаты	Фактические результаты	Статус
TU01	Проверка входа пользователя с существующими логином и паролем	Откройте сайт http://blahblahblah.ru ↓ Введите логин ↓ Введите пароль ↓ Нажмите кнопку «Войти»	Логин = user99 Пароль = pass99	Пользователь должен попасть на главную страницу	Как ожидали	Пройден успешно
TU02	Проверка входа пользователя с несуществующими логином и паролем	Откройте сайт http://blahblahblah.ru ↓ Введите логин ↓ Введите пароль ↓ Нажмите кнопку «Войти»	Логин = user99 Пароль = badlass99	Пользователь должен остаться на странице логина. Появится сообщение «Неверные логин или пароль»	Как ожидали	Пройден успешно

Раздаточный материал № 47 (самостоятельно)

Правила составления тест-кейса

☞ Создавайте простые тест-кейсы. То есть лаконичные и понятные не только вам. Используйте повелительное наклонение, например: «перейдите на домашнюю страницу», «введите данные», «нажмите здесь». Шаги должны быть четкие, без лишних деталей. Так проще понять шаги теста и ускорить работу.

☞ Учитывайте интересы конечного пользователя. Конечная цель любого программного проекта — простое и понятное приложение, отвечающее запросу клиентов. Тестировщик создает тест-кейсы с учетом мнения конечного пользователя.

☞ Избегайте повторов. Если тест-кейс нужен, чтобы выполнить другой тест-кейс, оставьте ссылку по идентификатору в столбце предварительного условия.

☞ Не предполагайте. Не додумывайте функциональность и возможности ПО. Строго придерживайтесь спецификации.

☞ Пишите тестовые примеры. Они должны покрывать все требования к ПО из спецификации. Используйте чек-листы и автоматизированные средства учета покрытия тестами. Это гарантия того, что ни одна функция или условие не останутся непроверенными.

☞ Задавайте идентификатор тест-кейса. Так, чтобы его было легко идентифицировать. Например, когда отслеживают ошибки или определяют требования к ПО на более позднем этапе.

☞ Внедряйте методы тестирования. Эти техники помогают спланировать несколько тест-кейсов и находить ошибки:

анализ граничных значений — проверяйте верхние и нижние границы для допустимого диапазона значений;

эквивалентное разделение — разбивайте диапазон всевозможных тест-кейсов на равные части/группы с одинаковым поведением;

техника перехода состояний — создавайте тест-кейсы, которые покроют поведение ПО при переходе из одного состояния в другое.

☞ Внедряйте самоочистку. Тест-кейс должен возвращать среду в предтестовое состояние. Особенно это касается тестирования конфигураций.

☞ Создавайте повторяемые и самостоятельные текст-кейсы. Они должны всегда генерировать одинаковые результаты: независимо от того, кто их тестирует.

☞ Проводите экспертную оценку. Отправляйте текст-кейсы на проверку коллегам. Они могут обнаружить ошибки в дизайне тест-кейса, которые вы пропустили.

https://sky.pro/media/kak-napisat-test-keys/