Технологии программирования. Компонентный подход

В. В. Кулямин

Лекция 6. Архитектура программного обеспечения

Аннотация

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

Ключевые слова

Архитектура ПО, компонент архитектуры, представление архитектуры, сценарий использования, методы оценки архитектуры, диаграммы классов, диаграммы взаимодействия, диаграммы сценариев, диаграммы компонентов, диаграммы развертывания.

Текст лекции

Анализ области решений

Допустим, мы разобрались в предметной области, поняли, что требуется от будущей программной системы, даже зафиксировали настолько полно, насколько смогли, требования и пожелания всех заинтересованных лиц ко всем аспектам качества программы. Что делать дальше?

На этом этапе (а точнее, гораздо раньше, обычно еще в ходе анализа предметной области) исследуются возможные способы решения тех задач, которые поставлены в требованиях. Не всегда бывает нужно специальное изучение этого вопроса — чаще всего 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.

Рассмотрим в качестве примера программное обеспечение авиасимулятора для командной тренировки пилотов. Задачей такой системы в целом является контроль и выработка необходимых для безопасного управления самолетом навыков у команд летчиков. Кроме того, отрабатываются навыки поведения в особых ситуациях, связанных с авариями, частичной потерей управления самолетом, тяжелыми условиями полета, и т.д.

Симулятор должен:

- Моделировать определенные условия полета и создавать некоторые события, к которым относятся следующие.
  - Скоростной и высотный режим полета, запас горючего, их изменения со временем.
  - Модель самолета и ее характеристики по управляемости, возможным режимам полета и скорости реакции на различные команды.
  - Погода за бортом и ее изменения со временем.
  - Рельеф и другие особенности местности в текущий момент, их изменения со временем.
  - Исходный и конечный пункты полета, расстояние и основные характеристики рельефа между ними.
  - Исправность или неисправность элементов системы контроля полета и управления самолетом, показатели системы мониторинга и их изменение со временем.
  - Наличие пролетающих вблизи курса самолета других самолетов, их геометрические и скоростные характеристики.
  - Чрезвычайные ситуации, например, террористы на борту, нарушение герметичности корпуса, внезапные заболевания и «смерть» отдельных членов экипажа.

При этом вся совокупность условий должна быть непротиворечивой, выглядеть и развиваться подобно реальным событиям. Некоторые условия, например, погода, должны изменяться достаточно медленно, другие события — происходить внезапно и приводить к связанным с ними последствиям (нарушение герметичности корпуса может сопровождаться поломками каких-то элементов системы мониторинга или «смертью» одного из пилотов). Если приборы показывают наличие грозы по курсу и они исправны, через некоторое время летчики должны увидеть грозу за бортом и она может начать оказывать влияние на условия полета.

- Принимать команды, подаваемые пилотами, и корректировать демонстрируемые характеристики полета и работы системы управления самолетом в зависимости от этих команд, симулируемой модели самолета и исправности системы управления. Например, при повороте на некоторый угол вправо, показываемый пилотам «вид из кабины» должен переместиться на соответствующий угол влево со скоростью, соответствующей скорости реакции симулируемой модели самолета и исправности задействованных элементов системы управления.

Понятно, что одним из элементов симулятора служит система визуализации обстановки за бортом — она показывает пилотам «вид за окнами». Пилоты в ходе полета ориентируются по показателям огромного количества датчиков, представленных на приборной панели самолета. Вся их работа тоже должна симулироваться. Наличие и характеристики работы таких датчиков могут зависеть от симулируемой модели, но их расположение, форма и цвет служат слишком важными элементами выработки навыков управления самолетом, поэтому требуется поддерживать эти характеристики близкими к реальным. Представлять их только в виде изображений на некоторой панели неудобно, поскольку они должны располагаться и выглядеть максимально похоже на реальные прототипы. Значит, симулировать можно только небольшое семейство самолетов с практически одним и тем же наборов приборов на приборной панели.

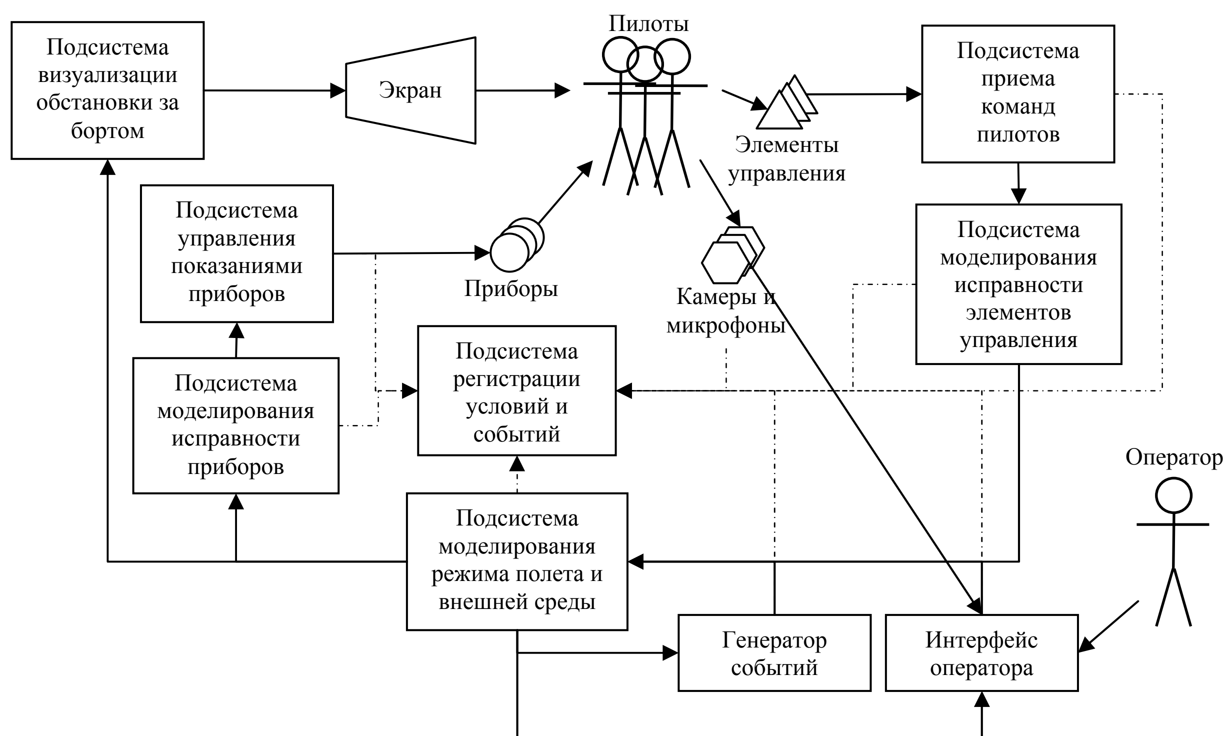

Рисунок 27. Примерная архитектура авиасимулятора.

Кроме того, все команды пилотов должны восприниматься соответствующими компонентами симулятора и встраиваться в моделируемые условия. В симулятор должен быть включен генератор событий, зависящий от текущей ситуации, а также интерфейс мониторинга и управления, с помощью которого внешний оператор мог бы создавать определенные события во

время симуляции полета наблюдать за всем, что происходит. Все события и действия пилотов должны протоколироваться с помощью камер и микрофонов для дальнейшего разбора полетов.

Рис. 27 показывает набросок архитектуры такого авиасимулятора. Каждый из указанных компонентов решает свои задачи, которые необходимы для работы всей системы. В совокупности они решают все задачи системы в целом. Стрелками показаны потоки данных и управления между компонентами. Пунктирные стрелки изображают потоки данных, передаваемых для протоколирования.

Архитектура определяет большинство характеристик качества ПО в целом. Архитектура служит также основным средством общения между разработчиками, а также между разработчиками и всеми остальными лицами, заинтересованными в данном ПО.

Выбор архитектуры задает способ реализации требований на высоком уровне абстракции. Именно архитектура почти полностью определяет такие характеристики ПО как надежность, переносимость и удобство сопровождения. Она также значительно влияет на удобство использования и эффективность ПО, которые, однако, сильно зависят и от реализации отдельных компонентов. Значительно меньше влияние архитектуры на функциональность — обычно заданную функциональность можно реализовать, использовав совершенно различные архитектуры.

Поэтому выбор между той или иной архитектурой определяется в большей степени именно нефункциональными требованиями и необходимыми свойствами ПО с точки зрения удобства сопровождения и переносимости. При этом для построения хорошей архитектуры надо учитывать возможные противоречия между требованиями к различным характеристикам и уметь выбирать компромиссные решения, дающие приемлемые значения по всем показателям.

Так, для повышения эффективности в общем случае выгоднее использовать монолитные архитектуры, в которых выделено небольшое число компонентов (в пределе — единственный компонент). Этим обеспечивается экономия как памяти, поскольку каждый компонент обычно имеет свои данные, а здесь число компонентов минимально, так и времени работы, поскольку возможность оптимизировать работу алгоритмов обработки данных имеется также только в рамках одного компонента.

С другой стороны, для повышения удобства сопровождения, наоборот, лучше разбить систему на большое число отдельных маленьких компонентов, с тем чтобы каждый из них решал свою небольшую, но четко определенную часть общей задачи. При этом, если возникают изменения в требованиях или проекте, их обычно можно свести к изменению в постановке одной, реже двух или трех таких подзадач и, соответственно, изменять только отвечающие за решение этих подзадач компоненты.

С третьей стороны, для повышения надежности лучше использовать либо небольшой набор простых компонентов, либо дублирование функций, т.е. сделать несколько компонентов ответственными за решение одной подзадачи. Заметим, однако, что ошибки в ПО чаще всего носят неслучайный характер. Они повторяемы, в отличие от аппаратного обеспечения, где ошибки связаны часто со случайными изменениями характеристик среды и могут быть преодолены простым дублированием компонентов без изменения их внутренней реализации. Поэтому при таком обеспечении надежности надо использовать достаточно сильно отличающиеся способы решения одной и той же задачи в разных компонентах.

Другим примером противоречивых требований служат характеристики удобства использования и защищенности. Чем сильнее защищена система, тем больше проверок, процедур идентификации и пр. нужно проходить пользователям. Соответственно, тем менее удобна для них работа с такой системой. При разработке реальных систем приходится искать некоторый разумный компромисс, чтобы сделать систему достаточно защищенной и способной поставить ощутимую преграду для несанкционированного доступа к ее данным и, в то же время, не отпугнуть пользователей сложностью работы с ней.

Список стандартов, регламентирующих описание архитектуры, которое является основной составляющей проектной документации на ПО, выглядит так.

- **IEEE 1016-1998 Recommended Practice for Software Design Descriptions [2]** (рекомендуемые методы описаний проектных решений для ПО).
- **IEEE 1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems [3]** (рекомендуемые методы описания архитектуры программных систем).
  Основное содержание этого стандарта сводится к определению набора понятий, связанных с архитектурой программной системы.
  Это, прежде всего, само понятие архитектуры как набора основополагающих принципов организации системы, воплощенных в наборе ее компонентов, связях их друг с другом и между ними и окружением системы, а также принципов проектирования и развития системы.
  Это определение, в отличие от данного в начале этой лекции, делает акцент не на наборе структур в основе архитектуры, а на принципах ее построения.
  Стандарт IEEE 1471 определяет также ***представление архитектуры (architectural description)*** как согласованный набор документов, описывающий архитектуру с точки зрения определенной группы заинтересованных лиц с помощью набора моделей. Архитектура может иметь несколько представлений, отражающих интересы различных групп заинтересованных лиц.
  Стандарт рекомендует для каждого представления фиксировать отраженные в нем взгляды и интересы, роли лиц, которые заинтересованы в таком взгляде на систему, причины, обуславливающие необходимость такого рассмотрения системы, несоответствия между элементами одного представления или между различными представлениями, а также различную служебную информацию об источниках информации, датах создания документов и пр.
  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]. Основные его шаги следующие.

1. Определить набор сценариев действий пользователей или внешних систем, использующих некоторые возможности, которые могут уже планироваться для реализации в системе или быть новыми. Сценарии должны быть значимы для конкретных заинтересованных лиц, будь то пользователь, разработчик, ответственный за сопровождение, представитель контролирующей организации и пр. Чем полнее набор сценариев, тем выше будет качество анализа. Можно также оценить частоту появления и важность сценариев, возможный ущерб от невозможности их выполнить.
2. Определить архитектуру (или несколько сравниваемых архитектур). Это должно быть сделано в форме, понятной всем участникам оценки.
3. Классифицировать сценарии. Для каждого сценария из набора должно быть определено, поддерживается ли он уже данной архитектурой или для его поддержки нужно вносить в нее изменения. Сценарий может поддерживаться, т.е. его выполнение не потребует внесения изменений ни в один из компонентов, или же не поддерживаться, если его

выполнение требует изменений в описании поведения одного или нескольких компонентов или изменений в их интерфейсах. Поддержка сценария означает, что лицо, заинтересованное в его выполнении, оценивает степень поддержки как достаточную, а необходимые при этом действия — как достаточно удобные.

4. Оценить сценарии. Определить, какие из сценариев полностью поддерживаются рассматриваемыми архитектурами. Для каждого неподдерживаемого сценария надо определить необходимые изменения в архитектуре — внесение новых компонентов, изменения в существующих, изменения связей и способов взаимодействия. Если есть возможность, стоит оценить трудоемкость внесения таких изменений.
5. Выявить взаимодействие сценариев. Определить какие компоненты требуется изменять для неподдерживаемых сценариев; если требуется изменять один компонент для поддержки нескольких сценариев — такие сценарии называют взаимодействующими. Нужно оценить смысловые связи между взаимодействующими сценариями.
   Малая связанность по смыслу между взаимодействующими сценариями означает, что компоненты, в которых они взаимодействуют, выполняют слабо связанные между собой задачи и их стоит декомпозировать.
   Компоненты, в которых взаимодействуют много (> 2-х) сценариев, также являются возможными проблемными местами.
6. Оценить архитектуру в целом (или сравнить несколько заданных архитектур). Для этого надо использовать оценки важности сценариев и степень их поддержки архитектурой.

Рассмотрим сравнительный анализ двух архитектур на примере индексатора — программы для построения индекса некоторого текста, т.е. упорядоченного по алфавиту списка его слов без повторений.

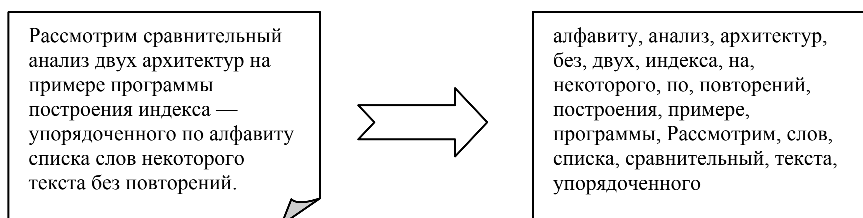

Рисунок 28. Пример работы индексатора текста.

1. Выделим следующие сценарии работы или модификации программы.
   a. Надо сделать так, чтобы индексатор мог работать в инкрементальном режиме, читая на входе одну фразу за другой и пополняя получаемый в процессе работы индекс.
   b. Надо сделать так, чтобы индексатор мог игнорировать предлоги, союзы, местоимения, междометия, частицы и другие служебные слова.
   c. Надо сделать так, чтобы индексатор мог обрабатывать тексты, подаваемые ему на вход в виде архивов.
   d. Надо сделать так, чтобы в индексе оставались только слова в основной грамматической форме — существительные в единственном числе и именительном падеже, глаголы в неопределенной форме и пр.
2. Определим две возможных архитектуры индексатора для сравнительного анализа.
   a. В качестве первой архитектуры рассмотрим разбиение индексатора на два компонента. Один компонент принимает на свой вход входной текст, полностью прочитывает его и выдает на выходе список слов, из которых он состоит. Второй компонент принимает на вход список слов, а на выходе выдает его упорядоченный вариант без повторений. Этот вариант архитектуры построен в стиле «каналы и фильтры» (см. следующую лекцию).

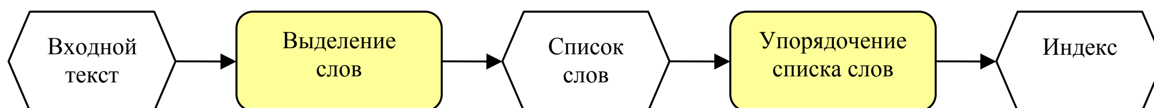

Рисунок 29. Архитектура индексатора в стиле каналов и фильтров.

b. Другой вариант архитектуры индексатора устроен следующим образом. Имеется внутренняя структура данных, хранящая подготовленный на настоящий момент вариант индекса. Он представляет собой упорядоченный список без повторений всех слов, прочитанных до настоящего момента. Кроме того, имеются две переменные — строка, хранящая последнее (быть может, не до конца) прочитанное слово, и ссылка на то слово в подготовленном списке, которое лексикографически следует за последним словом (соответственно, предшествующее этому слово в списке лексикографически предшествует последнему прочитанному слову).
В дополнение к этим данным имеются следующие компоненты.
   i. Первый читает очередной символ на входе и передает его на обработку одному из остальных.
   Если это разделитель слов (пробел, табуляция, перевод строки), управление получает второй компонент.
   Если это буква — третий.
   Если входной текст кончается — четвертый.
   ii. Второй компонент закачивает ввод последнего слова — оно помещается в список перед тем местом, на которое указывает ссылка; после чего последнее слово становится пустым, а ссылка начинает указывать на первое слово в списке.
   iii. Третий компонент добавляет прочитанную букву в конец последнего слова, после чего, быть может, перемещает ссылку на следующее за полученным слово в списке.
   iv. Четвертый компонент выдает полученный индекс на выход.
Эта архитектура построена в стиле «репозиторий» (см. следующую лекцию).

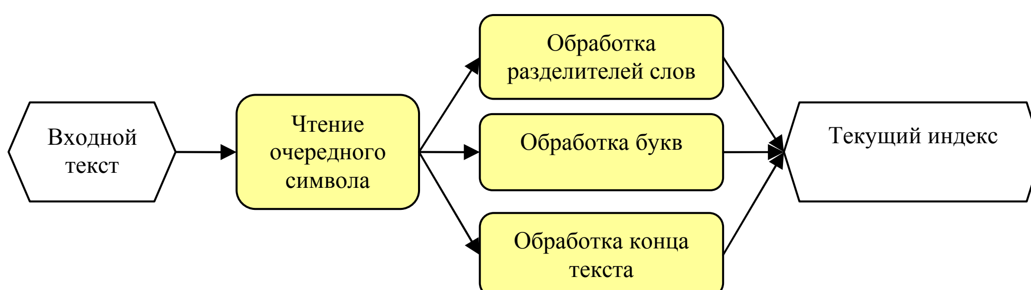

Рисунок 30. Архитектура индексатора в стиле репозитория.

3. Определим поддерживаемые сценарии из выделенного набора.
   a. Сценарий a.
   Этот сценарий прямо поддерживается второй архитектурой.
   Чтобы поддержать его в первой, необходимо внести изменения в оба компонента так, чтобы первый компонент мог бы пополнять промежуточный список, читая входной текст фраза за фразой, а второй — аналогичным способом пополнять результирующий упорядоченный список, вставляя туда поступающие ему на вход слова.
   b. Сценарий b.
   Обе архитектуры не поддерживают этот сценарий.
   Для его поддержки в первой архитектуре необходимо изменить первый компонент или, лучше, вставить после него дополнительный фильтр, отбрасывающий вспомогательные

части речи.
Для поддержки этого сценария второй архитектурой нужно ввести дополнительный компонент, который перехватывает буквы, выдаваемые модулем их обработки (соответственно, этот модуль больше не должен перемещать указатель по итоговому списку) и сигналы о конце слова от первого компонента, после чего он должен отсеивать служебные слова.

   c. Сценарий c.
   Этот сценарий также требует изменений в обеих архитектурах.
   Однако в обоих случаях эти изменения одинаковы — достаточно добавить дополнительный компонент, декодирующий архивы, если они подаются на вход.
   d. Сценарий d.
   Этот сценарий также не поддерживается обеими архитектурами.
   Требуемые им изменения аналогичны требованиям второго сценария, только в этом случае дополнительный компонент-фильтр должен еще и преобразовывать слова в их основную форму и только после этого пытаться добавить результат к итоговому индексу.
   Таким образом, требуется, как и во втором случае, изменить или добавить один компонент в первой архитектуре и изменить один и добавить новый во второй.
4. Мы уже выполнили оценку сценариев на предыдущем шаге. Итоги этой оценки приведены в Таблице 6.
5. Мы видели, что при использовании первого варианта архитектуры только для поддержки первого сценария пришлось бы вносить изменения в ее компоненты. В остальных случаях достаточно было добавить новый компонент, что несколько проще.
   При использовании второго варианта нам в двух разных сценариях, помимо добавления нового компонента, потребовалось изменить компонент, обрабатывающий буквы.

Архитектура	Сценарий a	Сценарий b	Сценарий c	Сценарий d
Каналы и фильтры	- -	+ + *	+ + *	+ + *
Репозиторий	+ + + +	+ + - + *	+ + + + *	+ + - + *

Таблица 6. Итоги оценки двух вариантов архитектуры индексатора.
+ обозначает возможность не изменять компонент, - — необходимость изменения компонента,
*** — необходимость добавления одного компонента**

6. В целом первая архитектура на предложенных сценариях выглядит лучше второй.
   Единственный ее недостаток — отсутствие возможности инкрементально поставлять данные на вход компонентам. Если его устранить, сделав компоненты способными потреблять данные постепенно, эта архитектура станет почти идеальным вариантом, поскольку она легко расширяется — для решения многих дополнительных задач потребуется только добавлять компоненты в общий конвейер.
   Вторая архитектура, несмотря на выигрыш в инкрементальности, проигрывает в целом.
   Основная ее проблема — слишком специфически построенный компонент-обработчик букв. Необходимость изменить его в нескольких сценариях показывает, что нужно объединить обработчик букв и обработчик конца слов в единый компонент, выдающий слова целиком, после чего полученная архитектура не будет ничем уступать исправленной первой.

UML. Виды диаграмм UML

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) [5-7], 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. Пример диаграммы классов изображен на Рис. 31.

  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.
  На Рис. 31 изображены классы `Account`, `Person`, `Organization`, `Address`, `CreditAccount` и абстрактный класс `Client`.
  Класс `CreditAccount` имеет **`private`** поле `maximumCredit` типа **`double`**, а также **`public`** метод `getCredit()` и **`protected`** метод `setCredit()`.

  Интерфейсы, т.е. типы, имеющие только набор операций и не определяющие способов их реализации, часто показываются в виде небольших кружков, хотя могут изображаться и как обычные классы. На Рис. 31 представлен интерфейс `AccountInterface`.

  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).
  Композицией на Рис. 31 является связь между классами `Organization` и `Address`.

  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. 31 между классами `Account` и `Client`.

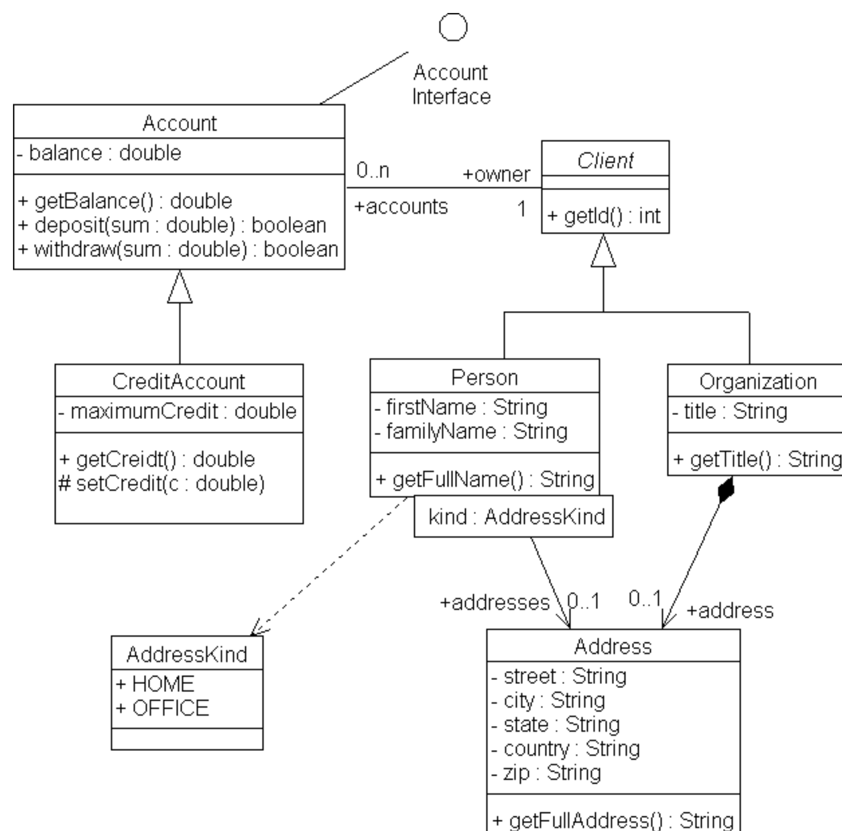

Рисунок 31. Диаграмма классов.

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. 31 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.

Наследование классов изображается стрелкой с пустым наконечником, ведущей от наследника к предку. На Рис. 31 класс `CreditAccount` наследует классу `Account`, а классы `Person` и `Organization` — классу `Client`.

Реализация интерфейсов показывается в виде пунктирной стрелки с пустым наконечником, ведущей от класса к реализуемому им интерфейсу, если тот показан в виде прямоугольника. Если же интерфейс изображен в виде кружка, то связь по реализации показывается обычной сплошной линией (в этом случае неоднозначности в ее толковании не возникает). Такая связь изображена на Рис. 31 между классом `Account` и интерфейсом `AccountInterface`.

Один класс *использует* другой, если этот другой класс является типом параметра или результата операции первого класса. Иногда связи по использованию показываются в виде пунктирных стрелок. Пример такой связи между классом `Person` и перечислимым типом `AddressKind` можно видеть на Рис. 31.

Ссылочные связи, реализованные в виде ассоциативных массивов или отображений (map) — такая связь в зависимости от некоторого набора ключей определяет набор ссылок-значений — показываются при помощи стрелок, имеющих прямоугольник с перечислением типов и имен ключей, примыкающий к изображению класса, от которого идет стрелка. Множественность на конце стрелки при этом обозначает количество ссылок, соответствующее одному набору значений ключей.

На Рис. 31 такая связь ведет от класса `Person` к классу `Address`, показывая, что объект класса `Person` может иметь один адрес для каждого значения ключа `kind`, т.е. один домашний и один рабочий адреса.

Диаграммы классов используются чаще других видов диаграмм.

- ***Диаграммы объектов (object diagrams)*** показывают часть объектов системы и связи между ними в некотором конкретном состоянии или суммарно, за некоторый интервал времени. Объекты изображаются прямоугольниками с идентификаторами ролей объектов (в контексте тех состояний, которые изображены на диаграмме) и типами. Однородные коллекции объектов могут изображаться накладывающимися друг на друга прямоугольниками.
  Такие диаграммы используются довольно редко.

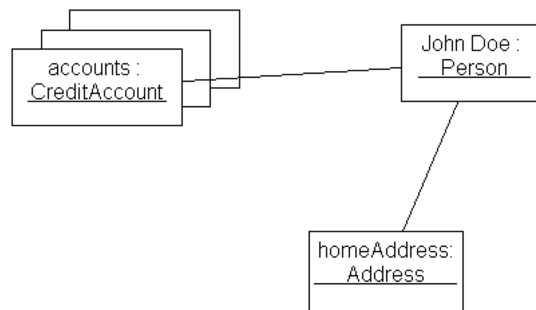

Рисунок 32. Диаграмма объектов.

- ***Диаграммы компонентов (component diagrams)*** представляют компоненты в нескольких смыслах — атомарные составляющие системы с точки зрения ее сборки, конфигурационного управления и развертывания. Компоненты сборки и конфигурационного управления обычно представляют собой файлы с исходным кодом, динамически подгружаемые библиотеки, HTML-странички и пр., компоненты развертывания — это компоненты JavaBeans, CORBA, COM и т.д. Подробнее о таких компонентах см. Лекцию 12.
  Компонент изображается в виде прямоугольника с несколькими прямоугольными или другой формы «зубами» на левой стороне.

  Связи, показывающие зависимости между компонентами, изображаются пунктирными стрелками. Один компонент зависит от другого, если он не может быть использован в отсутствии этого другого компонента в конфигурации системы. Компоненты могут также реализовывать интерфейсы.
  Диаграммы этого вида используются редко.

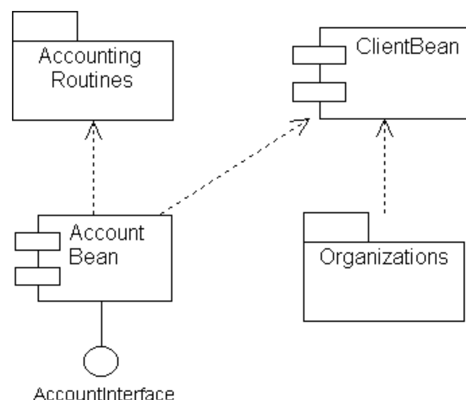

Рисунок 33. Диаграмма компонентов.

На диаграмме компонентов, изображенной на Рис. 33, можно также увидеть *пакеты*, изображаемые в виде «папок», точнее — прямоугольников с прямоугольными «наростами»

над левым верхним углом. Пакеты являются пространствами имен и средством группировки диаграмм и других модельных элементов UML — классов, компонентов и пр. Они могут появляться на диаграммах классов и компонентов для указания зависимостей между ними и отдельными классами и компонентами. Иногда на такой диаграмме могут присутствовать только пакеты с зависимостями между ними.

- ***Диаграммы развертывания (deployment diagrams)*** показывают декомпозицию системы на физические устройства различных видов — серверы, рабочие станции, терминалы, принтеры, маршрутизаторы и пр. — и связи между ними, представленные различного рода сетевыми и индивидуальными соединениями.
  Физические устройства, называемые ***узлами*** системы (***nodes***), изображаются в виде кубов или параллелепипедов, а физические соединения между ними — в виде линий.
  На диаграммах развертывания может быть показана привязка (в некоторый момент времени или постоянная) компонентов развертывания системы к физическим устройствам — например, для указания того, что компонент EJB AccountEJB исполняется на сервере приложений, а аплет AccountInfoEditor — на рабочей станции оператора банка.

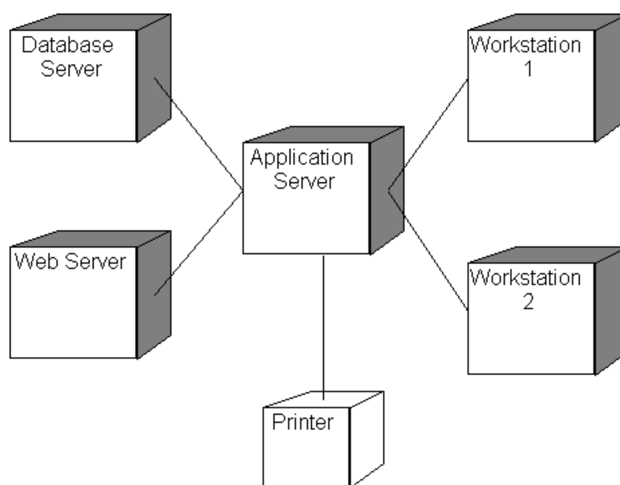

Рисунок 34. Диаграмма развертывания.

Эти диаграммы используются достаточно редко. Пример диаграммы развертывания изображен на Рис. 34.

Динамические диаграммы

Динамические диаграммы описывают происходящие в системе процессы. К ним относятся *диаграммы деятельности*, *сценариев*, *диаграммы взаимодействия* и *диаграммы состояний*.

- ***Диаграммы деятельности (activity diagrams)*** иллюстрируют набор процессов-деятельностей и потоки данных между ними, а также возможные их синхронизации друг с другом.

  Деятельность изображается в виде прямоугольника с закругленными сторонами, слева и справа, помеченного именем деятельности.

  Потоки данных показываются в виде стрелок. Синхронизации двух видов — *развилки (forks)* и *слияния (joins)* — показываются жирными короткими линиями (кто-то может посчитать их и тонкими закрашенными прямоугольниками), к которым сходятся или от которых расходятся потоки данных. Кроме синхронизаций, на диаграммах деятельности могут быть показаны разветвления потоков данных, связанных с выбором того или иного направления в зависимости от некоторого условия. Такие разветвления показываются в виде небольших ромбов.

  Диаграмма может быть поделена на несколько горизонтальных или вертикальных областей, называемых *дорожками (swimlanes)*. Дорожки служат для группировки

деятельностей в соответствии с выполняющими их подразделением организации, ролью, приложением, подсистемой и пр.
Диаграммы деятельности могут заменять часто используемые диаграммы потоков данных (см. Лекцию 4), поэтому применяются достаточно широко. Пример такой диаграммы показан на Рис. 35.

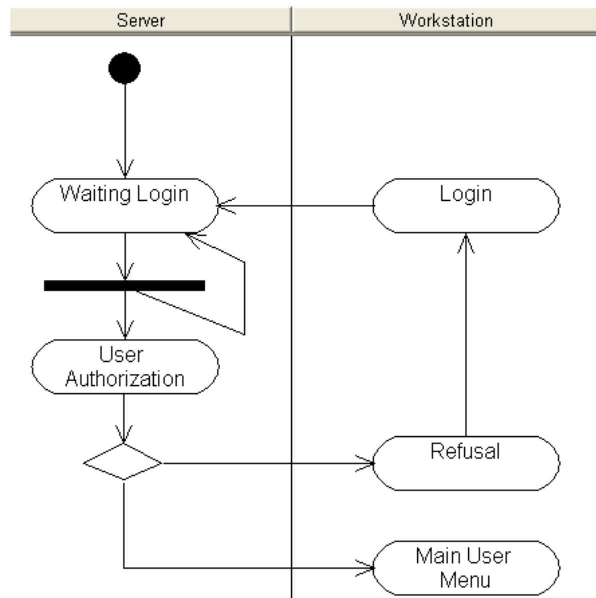

Рисунок 35. Диаграмма деятельности.

- ***Диаграммы сценариев*** (или ***диаграммы последовательности**, **sequence diagrams***) показывают возможные сценарии обмена сообщениями или вызовами во времени между различными компонентами системы (здесь имеются в виду архитектурные компоненты, компоненты в широком смысле — это могут быть компоненты развертывания, обычные объекты, подсистемы и пр.). Эти диаграммы являются подмножеством специального графического языка — языка *диаграмм последовательностей сообщений (Message Sequence Charts, MSC)*, который был придуман раньше UML и достаточно долго развивается параллельно ему.

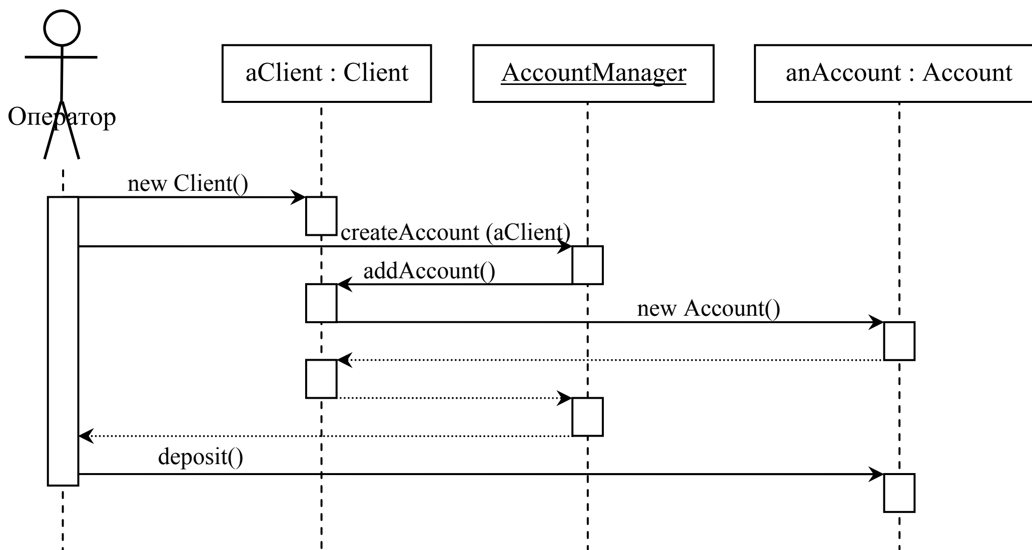

Рисунок 36. Пример диаграммы сценария открытия счета.

Компоненты, участвующие во взаимодействии, изображаются прямоугольниками вверху диаграммы. От каждого компонента вниз идет вертикальная линия, называемая его *линией*

жизни. Считается, что ось времени направлена вертикально вниз. Интервалы времени, в которые компонент активен, т.е. управление находится в одной из его операций, представлены тонким прямоугольником, для которого линия жизни компонента является осью симметрии.

Передача сообщения или вызов изображаются стрелкой от компонента-источника к компоненту-приемнику. Возврат управления показан пунктирной стрелкой, обратной к соответствующему вызову.

Эти диаграммы используются достаточно часто, например, при детализации сценариев, входящих в варианты использования. Пример такой диаграммы изображен на Рис. 36.

- ***Диаграммы взаимодействия (collaboration diagrams)*** показывают ту же информацию, что и диаграммы сценариев, но привязывают обмен сообщениями/вызовами не к времени, а к связями между компонентами. Пример такой диаграммы представлен на Рис. 37.

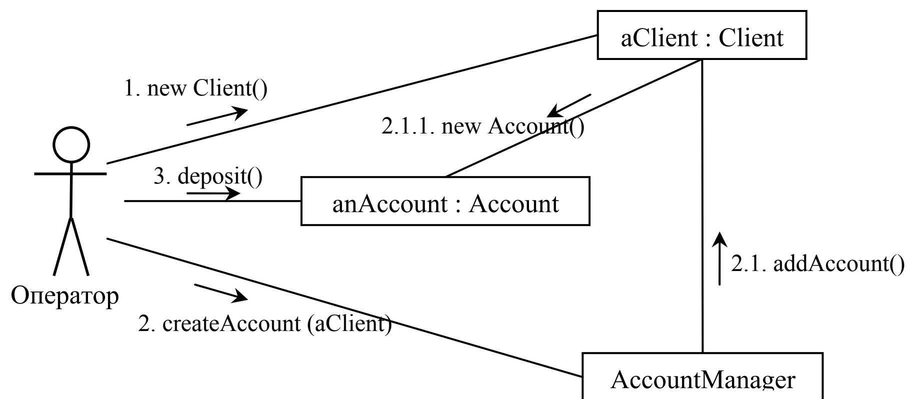

Рисунок 37. Диаграмма взаимодействия, соответствующая диаграмме сценария на Рис. 36.

На диаграмме изображаются компоненты в виде прямоугольников и связи между ними. Вдоль связей могут передаваться сообщения, показываемые в виде небольших стрелок, параллельных связи. Стрелки нумеруются в соответствии с порядком происходящих событий. Нумерация может быть иерархической, чтобы показать вложенность действий друг в друга (т.е. если вызов некоторой операции имеет номер 1, то вызовы, осуществляемые при выполнении этой операции, будут нумероваться как 1.1, 1.2, и т.д.). Диаграммы взаимодействия используются довольно редко.

- ***Диаграммы состояний (statechart diagrams)*** показывают возможные *состояния* отдельных компонентов или системы в целом, *переходы* между ними в ответ на какие-либо *события* и выполняемые при этом *действия*.

Состояния показываются в виде прямоугольников с закругленными углами, переходы — в виде стрелок. Начальное состояние представляется как небольшой темный кружок, конечное — как пустой кружок с концентрически вложенным темным кружком. Вы могли обратить внимание на темный кружок на диаграмме деятельности на Рис. 35 — он тоже изображает начальное состояние: дело в том, что диаграммы деятельности являются диаграммами состояний специального рода, а деятельности — частный случай состояний. Пример диаграммы состояний приведен на Рис. 38.

Состояния могут быть устроены иерархически: они могут включать в себя другие состояния, даже целые отдельные диаграммы вложенных состояний и переходов между ними. Пребывая в таком состоянии, система находится ровно в одном из его *подсостояний*. На Рис. 38 почти все изображенные состояния являются подсостояниями состояния `Site`. Кроме того, в нижней части диаграммы три состояния объединены, чтобы показать, что переход по действию `cancel` возможен в каждом из них и приводит в одно и то же состояние `Basket`.

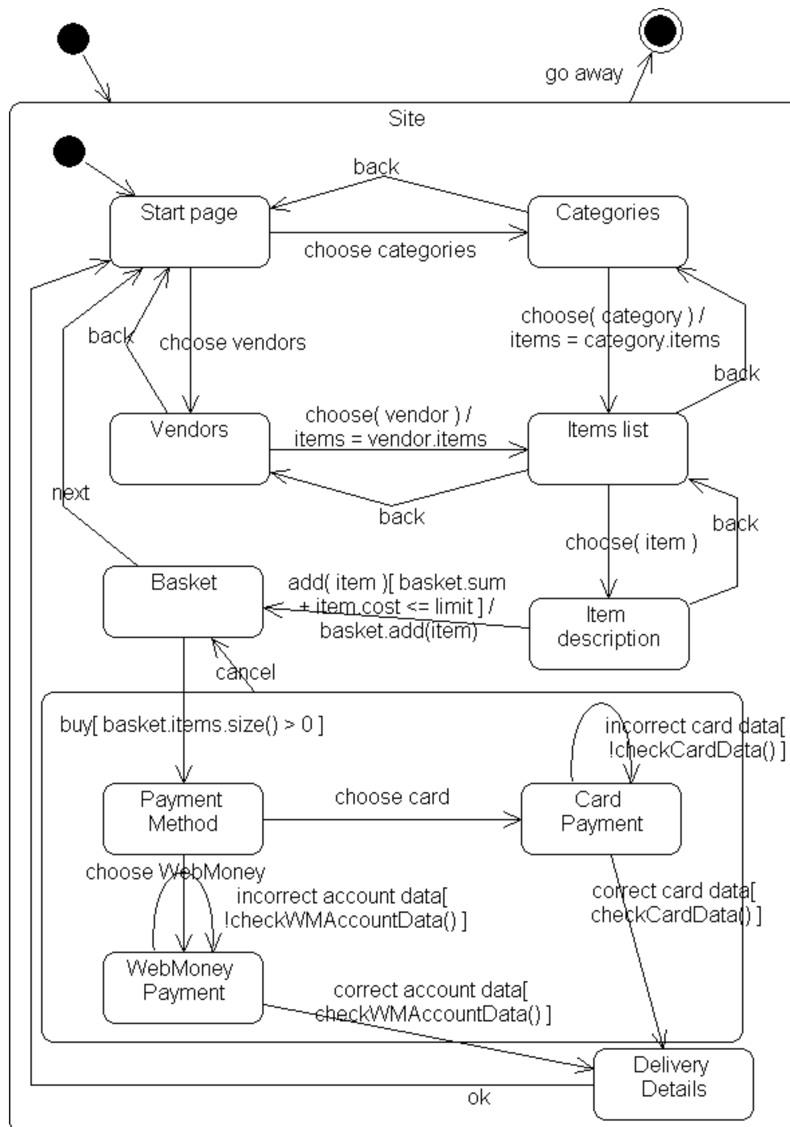

Рисунок 38. Пример диаграммы состояний, моделирующей сайт Интернет-магазина.

Состояние может декомпозироваться и на *параллельные подсостояния.* Они изображаются как области внутри объемлющего состояния, разделенные пунктирными линиями, их аналогом на диаграммах деятельности являются дорожки. Пребывая в объемлющем состоянии, система должна находиться одновременно в каждом из его параллельных подсостояний.

Помимо показанных на диаграмме состояний изображаемая подсистема может иметь глобальные (в ее рамках) переменные, хранящие какие-то данные. Значения этих переменных являются общими частями всех изображаемых состояний.

На Рис. 38 примерами переменных являются список видимых пользователем товаров, `items`, и набор уже отобранных товаров с количеством для каждого, корзина, `basket`.

Переходы могут происходить между состояниями одного уровня, но могут также вести из некоторого состояния в подсостояние соседнего или, наоборот, из подсостояния в некоторое состояние, находящее на том же уровне, что и объемлющее состояние.

На переходе между состояниями указываются следующие данные:

- *Событие*, приводящее к выполнению этого перехода. Обычно событие — это вызов некоторой операции в одном из объектов или приход некоторого сообщения, хотя могут указываться и абстрактные события.
  Например, из состояния `Categories` на Рис. 38 можно выйти, выполнив команду браузера «Назад». Она соответствует событию `back`, инициирующему переход в

состояние `Start page`. Другой переход из состояния `Categories` происходит при выборе категории товаров пользователем. Соответствующее событие имеет параметр — выбранную категорию. Оно изображено как `choose(category)`.

- *Условие выполнения* (*охранное условие, guardian*). Это условие, зависящее от параметров события и текущих значений глобальных переменных, выполнение которого необходимо для выполнения перехода. При наступлении нужного события переход выполняется, только если его условие тоже выполнено.
  Условие перехода показывается в его метке в квадратных скобках.
  На Рис. 38 примером условного перехода является переход из состояния `Basket` в состояние `Payment Method`. Он выполняется, только если пользователь выполняет команду «Оплатить» (событие `buy`) и при этом в его корзине есть хотя бы один товар.
- *Действие*, выполняемое в дополнение к переходу между состояниями. Обычно это вызовы каких-то операций и изменения значения глобальных переменных.
  Действие показывается в метке перехода после слеша (символа '/'). При этом изменения значений переменных перечисляются в начале, затем, после знака '^', указывается вызов операции.
  Например, на Рис. 38 при выборе пользователем категории товаров происходит переход из состояния `Categories` в `Items list`. При этом список товаров, видимый пользователю, инициализируется списком товаров выбранной категории.

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

Литература к Лекции 6

[1] Л. Басс, П. Клементс, Р. Кацман. Архитектура программного обеспечения на практике. СПб.: Питер, 2006.
[2] IEEE Std 1016-1998 Recommended Practice for Software Design Descriptions.
[3] IEEE 1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems.
[4] R. Kazman et al. SAAM: A Method for Analyzing the Properties of Software Architectures. Proceedings of the 16-th International Conference on Software Engineering, 1994.
[5] Г. Буч, Дж. Рамбо, А. Джекобсон. Язык UML. Руководство пользователя. М.: ДМК, 2000.
[6] Дж. Рамбо, А. Якобсон, Г. Буч. UML: Специальный справочник. СПб.: Питер, 2002.
[7] М. Фаулер, К. Скотт. UML в кратком изложении. М.: Мир, 1999.
[8] И. Соммервилл. Инженерия программного обеспечения. М.: Вильямс, 2002.
[9] Э. Дж. Брауде. Технология разработки программного обеспечения. СПб.: Питер, 2004.