Сегодня будем говорить про моделирование поведенческих аспектов программных систем. Существует множество способов описания алгоритмов, каждый из них имеет свои достоинства и недостатки, и предназначен для применения в различных ситуациях. Например, при описании алгоритмов, которые предназначены для выполнения компьютером, используются языки программирования, но для описания алгоритмов, выполняемых человеком, языки программирования неудобны и применяются другие способы.

Средства моделирования поведения в UML ввиду разнообразия областей применения языка должны удовлетворять набору различных и частично противоречивых требований.

- Модель должна быть достаточно детальной для того, чтобы послужить основой для составления компьютерной программы, либо если генерации кода не планируется, содержать в себе все особенности реализации, которые автор хотел передать читателям.
- Модель должна быть компактной и обозримой, чтобы служить средством общения между людьми в процессе разработки системы и для обмена идеями.
- Модель не должна зависеть от особенностей реализации конкретных компьютеров, средств программирования и технологий, чтобы не сужать область применения языка UML.
- Средства моделирования поведения в UML должны быть знакомыми и привычными для большинства пользователей языка и не должны противоречить требованиям популярных парадигм программирования.

Удовлетворить сразу всем требованиям в полной мере, видимо, практически невозможно -- средства моделирования поведения UML являются результатом многочисленных компромиссов.

В UML предусмотрено несколько различных средств для описания поведения. Выбор того или иного средства диктуется типом поведения, которое нужно описать. Можно разделить все средства моделирования поведения в UML на четыре группы:

- описание поведения с явным выделением состояний;
- описание поведения с явным выделением потоков данных и управления;
- описание поведения как последовательности сообщений во времени;
- описание параллельного поведения.

Заметим, что полного взаимно-однозначного соответствия между выделенными группами и каноническими диаграммами UML не наблюдается. Действительно, если первые две группы средств однозначно отображаются диаграммами автомата и диаграммами деятельности, то для описания последовательности сообщений применяется несколько различных типов диаграмм, а описание параллельного поведения "размазано" по всем типам канонических диаграмм. Рассмотрим эти четыре группы средств.

Моделирование состояний

При использовании объектно-ориентированного подхода к проектированию, программная система представляет собой множество объектов, взаимодействующих друг с другом. При этом возможна ситуация, когда поведение некоторых объектов разумно рассматривать в терминах их жизненного цикла, т.е. текущее поведение

объекта определяется его историей. Жизненный цикл -- по сути последовательность изменений состояния объекта.

Для описания такого поведения используется конечный автомат, который изображается посредством диаграммы конечных автоматов (или, как их ещё называют, диаграммы состояний). При этом состояния конечного автомата соответствуют возможным состояниям объекта, т. е. различным наборам значений атрибутов объекта, а переходы происходят в результате возникновения различных событий. Но также диаграммы конечных автоматов можно использовать для описания жизненного цикла не только объектов-экземпляров отдельных классов, но и для более крупных конструкций, в частности, для всей модели приложения или, напротив, для гораздо более мелких элементов -- отдельных операций.

Диаграммы конечных автоматов очень хорошо подходят для моделирования поведения реактивных объектов. Реактивным называется объект, поведение которого лучше всего характеризуется его реакцией на события, произошедшие вне его собственного контекста. У реактивного объекта есть четко выраженный жизненный цикл, когда текущее поведение обусловлено прошлым. Подавляющее большинство реальных устройств дискретного управления прекрасно описываются конечными автоматами. Другим распространённым примером реактивной системы может служить графический интерфейс пользователя.

Основными элементами диаграммы конечных автоматов являются «Состояние», «Событие» и «Переход».

Состояние -- стабильный "отрезок жизни" объекта, когда он готов к обработке событий и делает это в зависимости от предыдущей истории поведения. Состояние может содержать имя и/или дополнительно список внутренних действий. Список внутренних действий содержит перечень действий или деятельностей, которые выполняются во время нахождения объекта в данном состоянии:

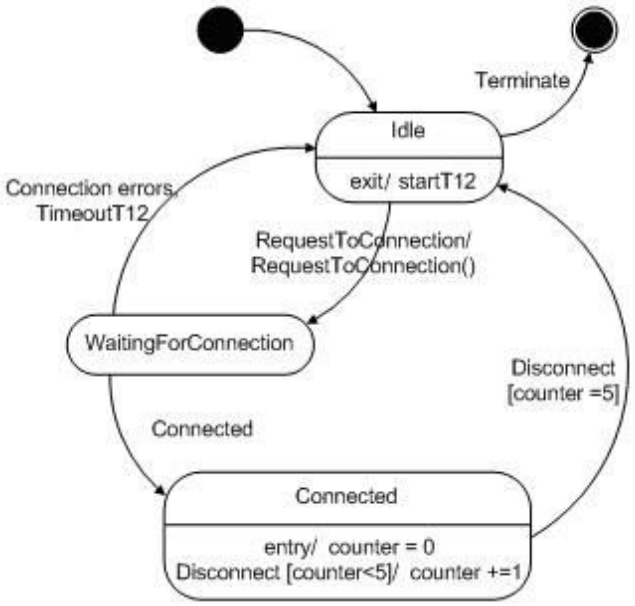
- *entry* -- действие, которое выполняется в момент входа в данное состояние (входное действие);
- *exit* -- действие, которое выполняется в момент выхода из данного состояния (выходное действие);
- *do* -- выполняющаяся деятельность ("do activity") в течение всего времени, пока объект находится в данном состоянии
- *внутренний переход* -- это переход, который происходит внутри состояния: компонента обрабатывает событие, не выходя из состояния.

Важной конструкцией конечного автомата является событие (event) -- происшествие, на которое компонента реагирует и которое может быть создано этой же компонентой (например, компонента посылает сообщение себе самой) или какой-то другой компонентой из окружения. События бывают следующих видов:

- изменение значения некоторого выражения (change event);
- срабатывание некоторого таймера, то есть прием специального сообщения (timeout);
- получение компонентой сообщения (signal event);
- вызов операции компоненты извне через ее интерфейс (call event);
- обращение извне к переменным компоненты через ее интерфейс.

Факт смены одного состояния другим изображается с помощью перехода. Переход осуществляется при наступлении некоторого события: окончания выполнения

деятельности (do activity), получении объектом сообщения или приемом сигнала, представляет собой цепочку действий по обработке данного события и завершается новым состоянием компоненты.



Переход может быть триггерным и нетриггерным. Если переход срабатывает, когда все операции исходного состояния завершены, он называется нетриггерным или переходом по завершении. Если переход инициируется каким-либо событием, он считается триггерным. Для триггерного перехода характерно наличие имени, которое может быть записано в следующем формате:

<имя события> (<список параметров, разделенных запятыми>) [<сторожевое условие>] <выражение действия>

Обязательным параметром является только имя события. После имени события могут следовать круглые скобки для явного задания параметров соответствующего события-триггера (например, пользователь, инициирующий действие).

Переход невозможно прервать, и если уж он запустился, то все действия, определенные в нем, должны отработать, прежде чем компонента сможет отреагировать на какое-либо следующее событие в системе. После окончания перехода компонента оказывается в новом состоянии: обработка текущего события, вызвавшего переход, считается завершенной и компонента может реагировать на следующие события.

Если при срабатывании перехода возможно ветвление, в имени перехода используется *сторожевое условие* (guard condition). Оно всегда записывается в прямых скобках после события-триггера и представляет собой некоторое булевское выражение. В общем случае из одного состояния может быть несколько переходов с одним и тем же событием-триггером, при этом целевое состояние будет зависеть от того какое из сторожевых условий примет значение «истина».

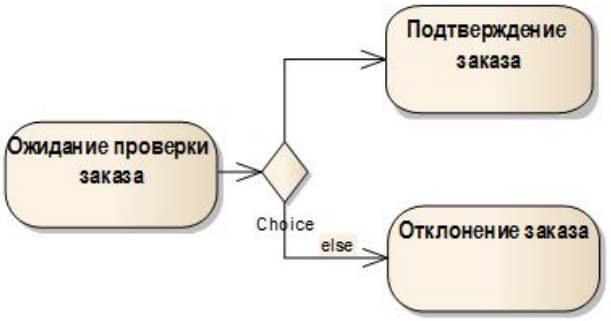
Также имя перехода может содержать *выражение действия* (action expression). В данном случае указанное действие выполняется сразу при срабатывании перехода и до начала каких бы то ни было действий в целевом состоянии. В общем случае

выражение действия может содержать целый список отдельных действий, разделенных точкой с запятой.

Помимо основных узлов, на диаграмме состояний могут использоваться так называемые псевдосостояния – вершины, которые не обладают поведением, и объект не находится в них, а «мгновенно» их проходит. К ним, например, относятся конечное и начальное состояние. Начальное состояние обычно не содержит никаких внутренних действий и определяет точку, в которой находится объект по умолчанию в начальный момент времени. Конечное состояние также не содержит никаких внутренних действий и служит для указания на диаграмме области, в которой завершается процесс изменения состояний в контексте конечного автомата.

Если необходимо отразить уничтожение объекта, используется узел завершения (terminate node), псевдосостояние, вход в которое означает завершение выполнения поведения конечного автомата в контексте его объекта.

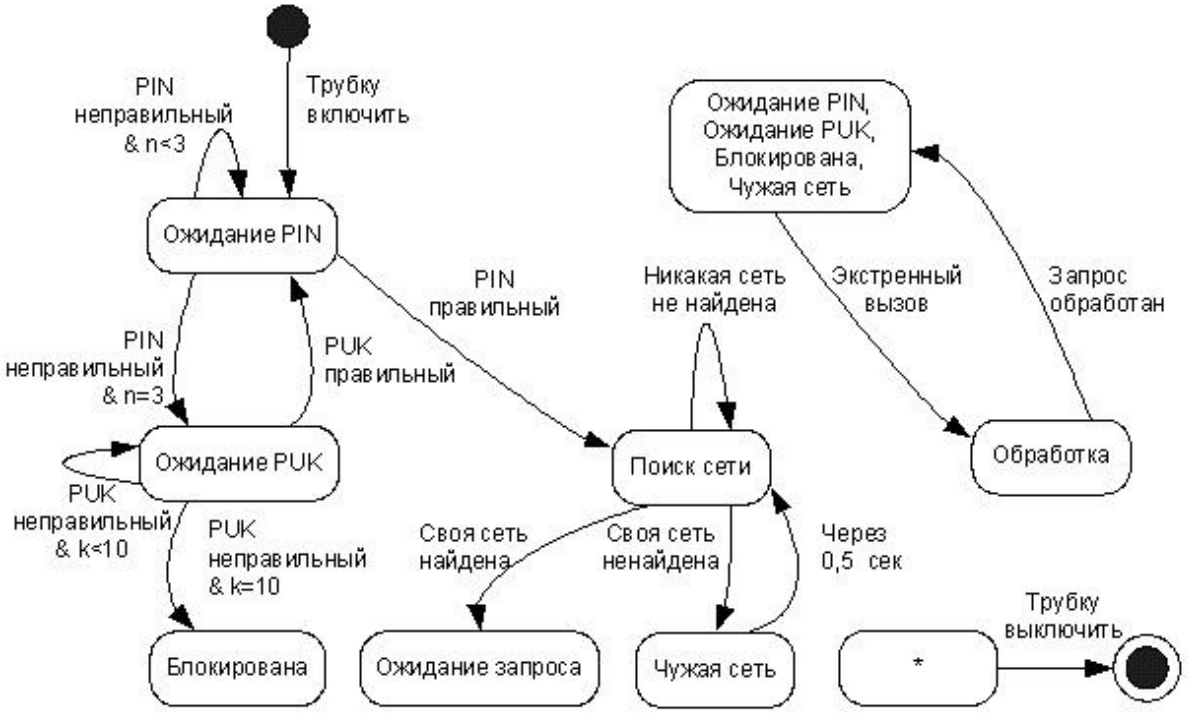
Узлы ветвления и распараллеливания аналогичны узлам на диаграмме активности.



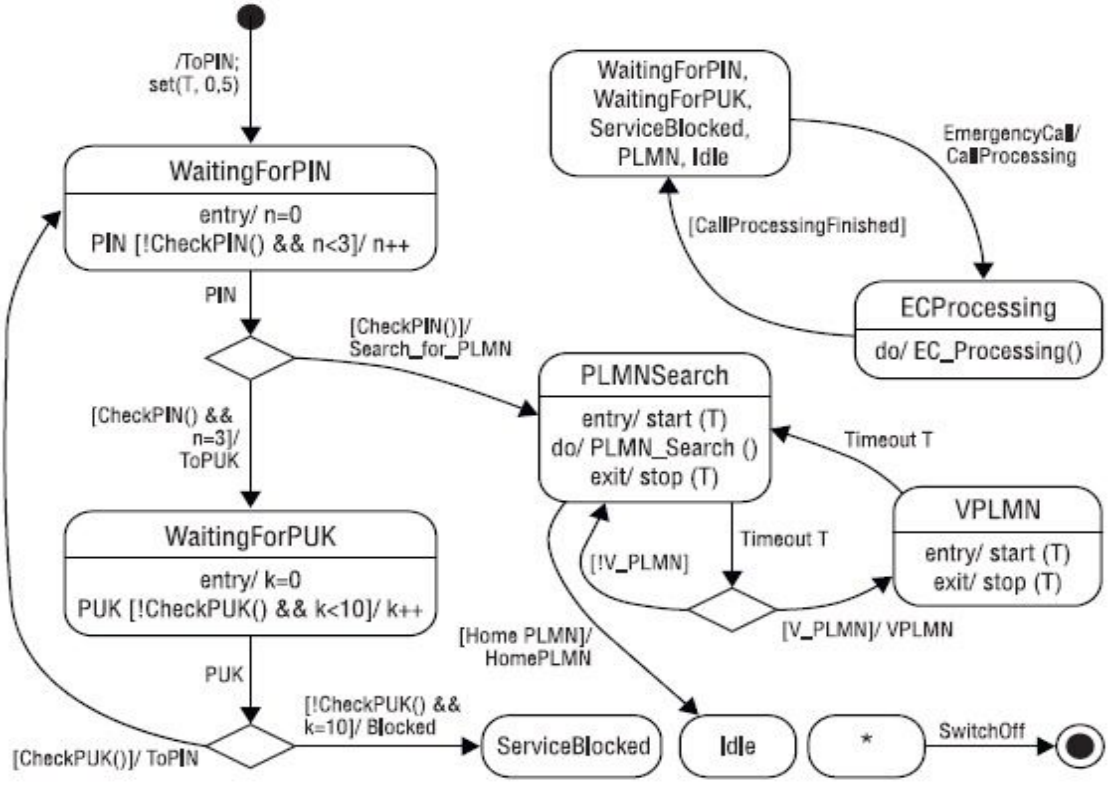
Переходы из псевдосостояния выбора в целевые состояния должны содержать сторожевые условия. Переход, который должен срабатывать, если ни одно из условий не примет значение «истина» должен содержать метку «else». В отличие от диаграммы активности, при отображении возможных вариантов перехода на диаграмме состояний узел выбора использовать не обязательно. Диаграмма состояний должна показывать возможное изменение состояния объекта, и не имеет своей целью выстраивать четкую последовательность переходов. Таким образом, из одного состояния могут выходить несколько переходов, конечной целью которых будут различные целевые состояния. Для отображения возможности выбора в данном случае достаточно в имени всех переходов добавить триггер и сторожевое условие.

Ниже представлен упрощенный фрагмент автомата состояний телекоммуникационной системы. Он описан довольно неформально (имена состояний

и события обозначены по-русски, нет описания действий в переходах и т. д.) и мог создаваться, например, на стадии анализа предметной области.



На последующих шагах проектирования поведение системы могло бы быть уточнено и записано более формально:



Такие модели отлично подходят для генерации кода. Диаграмму состояний можно реализовать тремя основными способами: с помощью вложенного оператора switch, паттерна State и таблицы состояний. Самый прямой подход в работе с диаграммами состояний – это вложенный оператор switch. Например, код для диаграммы выше мог бы выглядеть как-то так:

```
void processStateMachine()
{
  bool finish = false;
  SendMessage(ToPin);
  set (T, 0.5);
  state = WatingForPIN;
  while (!finish)
    if (getEvent())
      finish = processEvents();
}


bool processEvents(){
   bool f = false;
   switch (state){
   case WatingForPIN:
       if (nextstate) n = 0;
       nextstate = true;
       if (input_message == PIN) {
           if (CheckPIN()) {
               SendMessage(Search_for_PLMN);
               state = PLMNSearch;
           } else if (n == 3) {
               SendMessage(ToPUK);
               state = WatingForPUK;
           } else {
               n++;
               nextstate = false;
           }
       } else if (input_message == EmergencyCall) {
           SendMessage(CallProcessing);
           prevstate = state;
           state = ECProcessing;
       } else if (input_message == SwitchOff) {
           f = true;
       }
       break;
   case WatingForPUK:
       if (nextstate) k = 0;
       nextstate = true;
```

```
            ...
        }

        return f;
    }
```

Более подробный листинг для всех состояний можно посмотреть тут.

Конечный автомат компоненты работает как цикл, запускающий обработчик события (в данном случае, процедуру processEvents()), когда компонента понимает, что произошло некоторое событие, предназначенное ей для обработки. Цикл останавливается, если процедура processEvents() возвращает значение false. Это означает, что произошедшее событие -- это получение компонентой сообщения SwitchOff, команды выключить трубку.

По-хорошему, у компоненты должна быть еще очередь событий, в которую некий диспетчер помещает информацию о тех событиях в системе, которые ей предназначаются. А сама компонента, когда готова обрабатывать следующее событие, обращается в эту очередь. Работа с очередью скрыта в операции getEvent(), а соответствующие структуры данных не показаны, чтобы не усложнять примера.

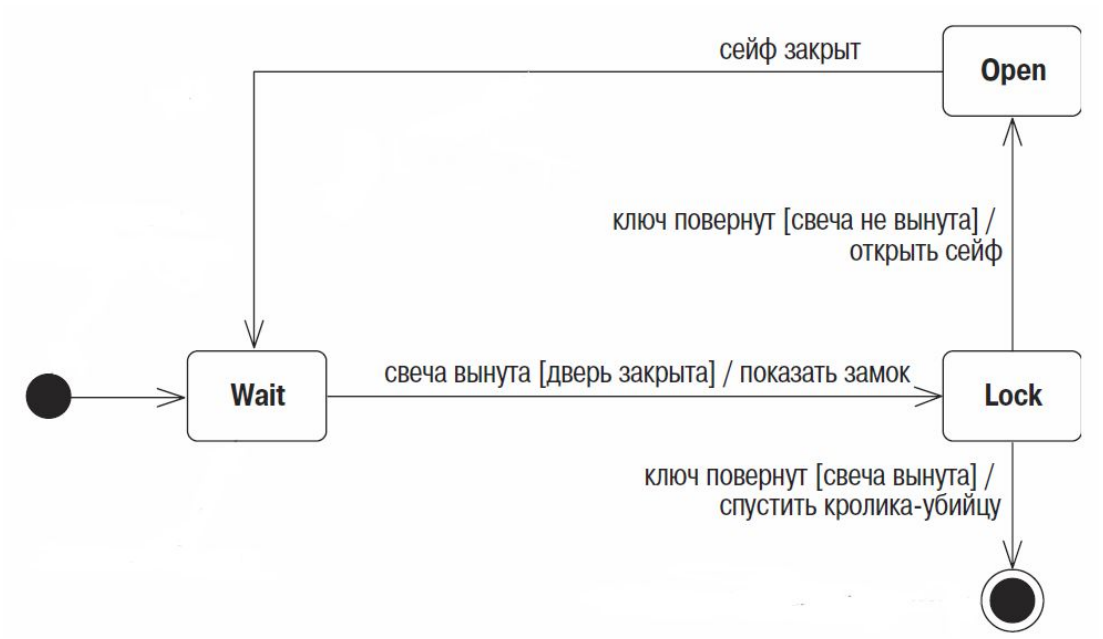
Теперь о процедуре processEvents(). Обработчик событий конечного автомата реализуется как оператор switch по состояниям компоненты, каждая его ветка соответствует определенному состоянию. Внутри этих веток происходит ветвление по возможным событиям. Это почти всегда либо еще один оператор switch, либо условный оператор по значениям обрабатываемых в данном состоянии сообщений. Исключением является обработка события "найдена своя станция" в состоянии PLMNSearch. Здесь событием является значение true переменной HomePLMN, а не присланное извне сообщение. Аналогичный способ обработки такого же типа события можно увидеть в состоянии ECProcessing.

Далее для некоторых состояний (WaitingForPIN, WaitingForPUK) ветки оператора switch начинаются с проверки значения логической переменной nextstate, и в случае, когда ее значение true, выполняются действия по входу для этих состояний. Это нужно, поскольку деятельность по входу при внутренних переходах не должна выполняться. Если переход внутренний, то перед его инициацией значение данной переменной устанавливается в false.

Хотя этот способ и прямой, но очень громоздкий. Тут много довольно примитивного кода, и в нём очень легко запутаться, поэтому этот подход надо применять с большой осторожностью.

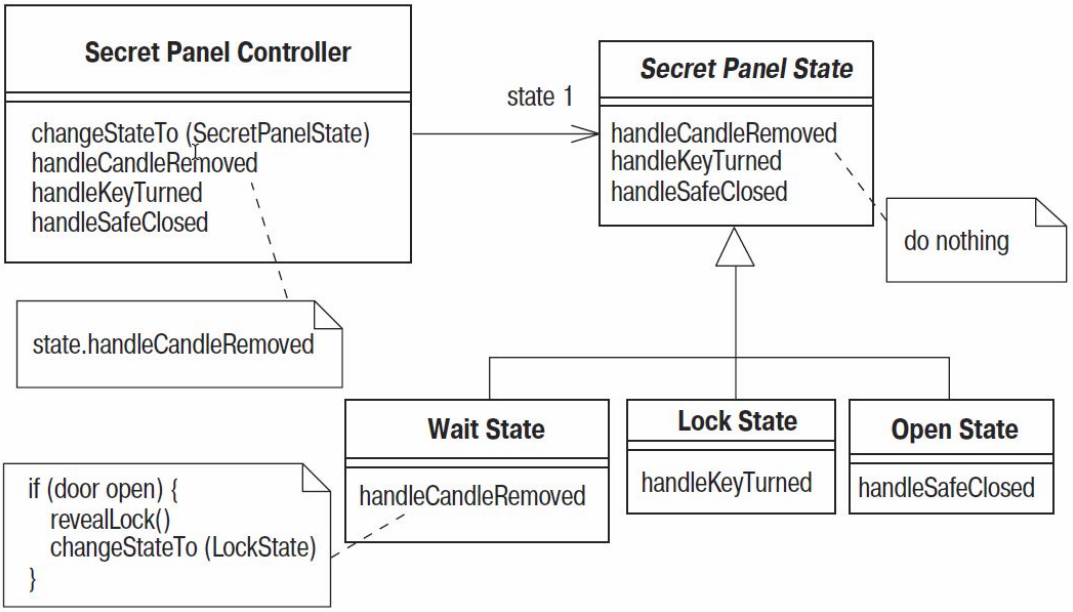
Таблица состояний представляет диаграмму состояний в виде данных. Затем мы строим интерпретатор, который использует таблицу состояний во время выполнения программы, или генератор кода, который порождает классы на основе этой таблицы.

Очевидно, большая часть работы над таблицей состояний проводится однажды, но затем ее можно использовать всякий раз, когда надо решить проблему, связанную с состояниями. Таблица состояний времени выполнения может быть модифицирована без перекомпиляции, что в некотором смысле удобно.



Исходное состояние	Целевое состояние	Событие	Защита	Процедура
Wait	Lock	Candle removed (свеча удалена)	Door open (дверца открыта)	Reveal lock (показать замок)
Lock	Open	Key turned (ключ повернут)	Candle in (свеча на месте)	Open safe (открыть сейф)
Lock	Final	Key turned (ключ повернут)	Candle out (свеча удалена)	Release killer rabbit (освободить убийцу-кролика)
Open	Wait	Safe closed (сейф закрыт)		

Шаблон проектирования «Состояние» представляет иерархию классов состояний для обработки поведения состояний. Каждое состояние на диаграмме имеет свой подкласс состояния. Контроллер имеет методы для каждого события, которые просто перенаправляют к классу состояния. Диаграмма состояний, показанная на рисунке выше, могла бы быть реализована с помощью следующих классов:



Вершиной иерархии является абстрактный класс, который содержит описание всех методов, обрабатывающих события, но без реализации. Для каждого конкретного состояния достаточно переписать метод-обработчик определенного события, инициирующего переход из состояния. Про шаблоны проектирования, в том числе и про “Состояние”, мы будем говорить подробнее через пару занятий.

В каждом случае реализация моделей состояний приводит к довольно стереотипной программе, поэтому обычно для этого прибегают к тому или иному способу генерации кода.

Моделирование потока управления и данных

Поток управления -- это последовательность выполнения операторов (действий) в программе. Он в том или ином виде присутствует в любом поведении, только не всегда он описывается явно. На поток управления оказывают влияние различные управляющие конструкции: операторы перехода, условные операторы, операторы цикла, а также вызовы подпрограмм и разное асинхронное взаимодействие между объектами.

Одной из главных диаграмм при моделировании потока управления является диаграмма активности. Мы с вами уже немного про них говорили, так что кратко повторим основные моменты.

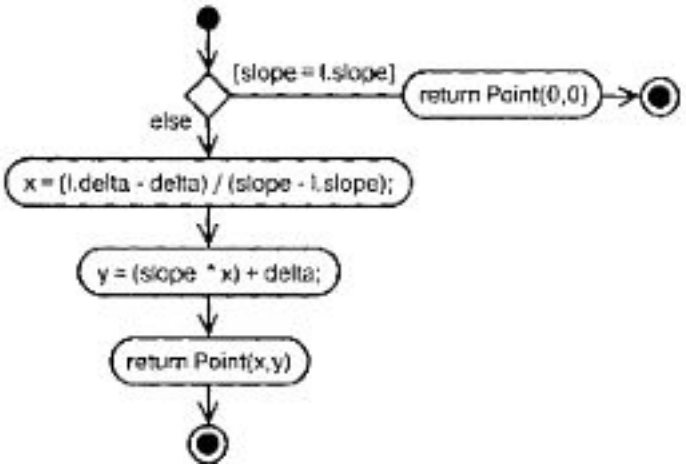
Применяемая в диаграммах активности графическая нотация во многом похожа на нотацию диаграммы конечных автоматов, поскольку на этих диаграммах также присутствуют обозначения состояний и переходов. Каждое состояние на диаграмме деятельности соответствует выполнению некоторой элементарной операции, а переход в следующее состояние выполняется только при завершении этой операции.

Как мы уже говорили два занятия назад, с диаграммами активности начинают работать ещё на стадии выявления требований, когда расписывают последовательность элементарных шагов, требуемых для реализации сценариев использования. Во многом диаграммы активностей не сильно отличаются от блок-схем

или даже текстового описания алгоритмов, однако позволяют вести декомпозицию задачи сверху-вниз в терминах визуальных моделей.

На стадии дальнейшего проектирования диаграммы активности могут использоваться для визуализации, специфицирования, конструирования и документирования поведения элементов других диаграмм. Чаще всего диаграммы активности раскрывают операции классов. При таком использовании диаграмма активности становится просто продвинутой версией блок-схемы выполняемых действий (в отличие от блок-схем тут поддерживается посылка-приём сигналов и параллельное выполнение). Основное преимущество такого использования диаграммы активности заключается в том, что все её элементы семантически связаны с элементами других моделей. Например, любая другая операция или сигнал, на которые есть ссылка из действия, могут быть проверены на соответствие типа классу целевого объекта.

Но тут надо не увлекаться и не превращать эти диаграммы в язык программирования. Иначе могут получиться что-то такое, а то и похуже:



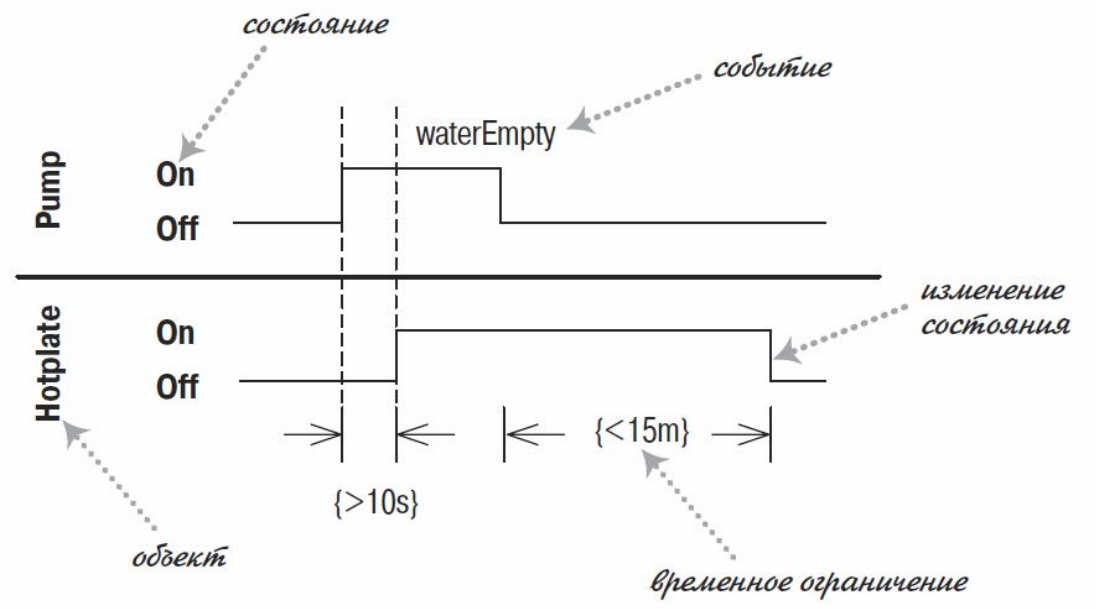
Использование диаграмм деятельности для моделирования операции становится разумным, когда эта операции сложна, так что разобраться в ней, глядя только на код, достаточно трудно. Взгляд же на диаграмму позволит понять такие аспекты алгоритма, которые нелегко было бы уловить, изучая один лишь код. А если делать диаграммы активности слишком подробными, они сами уже становятся плохо читаемыми. Если хочется зафиксировать в диаграмме какой-то фрагмент кода, лучше добавить элемент-комментарий и поместить код туда.

При создании диаграмм активности важно следить за тем, чтобы каждая диаграмма была сконцентрирована на описании одного аспекта динамики системы и содержала бы только те элементы, которые существенны для понимания этого аспекта. Детали других уровней абстракции лучше размещать на других диаграммах, более или менее высокоуровневых.

А ещё стоит отметить, что если диаграммы состояний основывались на формализме конечных автоматов, то диаграммы активности формализуются с помощью сетей Петри.

Ещё один вид диаграмм, который ориентирован на моделирование взаимодействия -- временные диаграммы. Они полезны для обозначения временных интервалов между изменениями состояний различных объектов. Временные

диаграммы активно использовались в разработке электроники, но в разработке ПО используются довольно редко. Ниже представлен пример диаграммы для насоса (Pump) и нагревательного элемента (Hotplate) в кофеварке (coffee pot). Представим себе правило, которое гласит, что между включением насоса и включением нагревательного элемента должно пройти по крайней мере 10 секунд. Когда емкость с водой становится пустой (waterEmpty), насос выключается, а нагревательный элемент не может оставаться включенным более 15 минут.



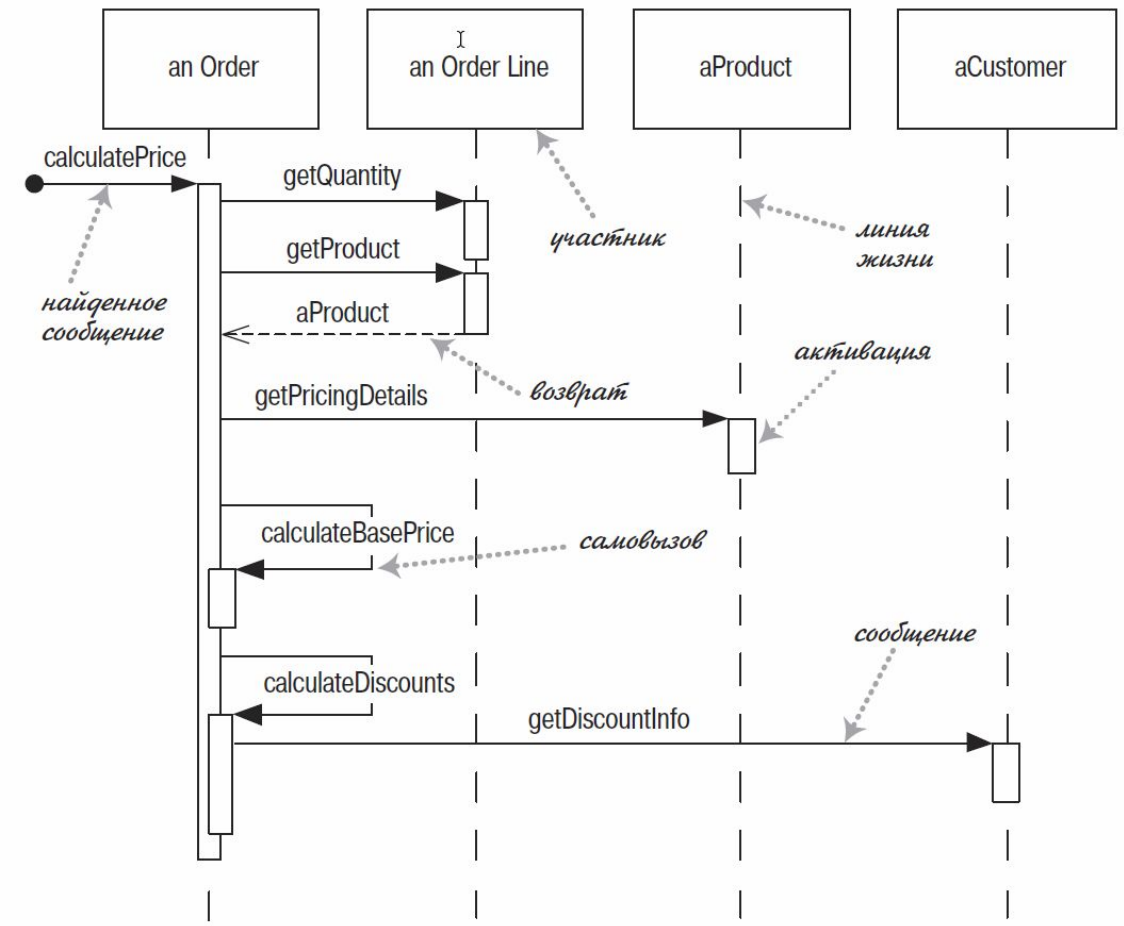
Моделирование последовательности сообщений

Взаимодействие нескольких программных объектов между собой описывается диаграммами взаимодействия в одной из двух практически эквивалентных форм: диаграммой коммуникации и диаграммой последовательности. Для объектно-ориентированной программы поведение прежде всего определяется взаимодействием объектов посредством передачи сообщений, поэтому диаграммы данного типа довольно хорошо заходят для моделирования взаимодействия частей объектно-ориентированных систем. В частности, их активно используют для документирования разного рода протоколов или простого взаимодействия объектов или подсистем.

Диаграммы последовательности, как и диаграммы активности, могут начинать использоваться ещё на стадии работы с требованиями для раскрытия сценариев использования системы.

Предположим, что у нас есть заказ, и мы собираемся вызвать команду для определения его стоимости. При этом объекту заказа (Order) необходимо просмотреть все позиции заказа и определить их цены, основанные на правилах построения цены продукции в строке заказа (Order Line). Проделав это для всех позиций заказа, объект заказа должен вычислить общую скидку, которая определяется индивидуально для каждого клиента.

На рисунке ниже приведена диаграмма, представляющая реализацию данного сценария. Диаграммы последовательности показывают взаимодействие, представляя каждого участника вместе с его линией жизни (lifeline), которая идет вертикально вниз и упорядочивает сообщения на диаграмме; сообщения также следует читать сверху вниз.

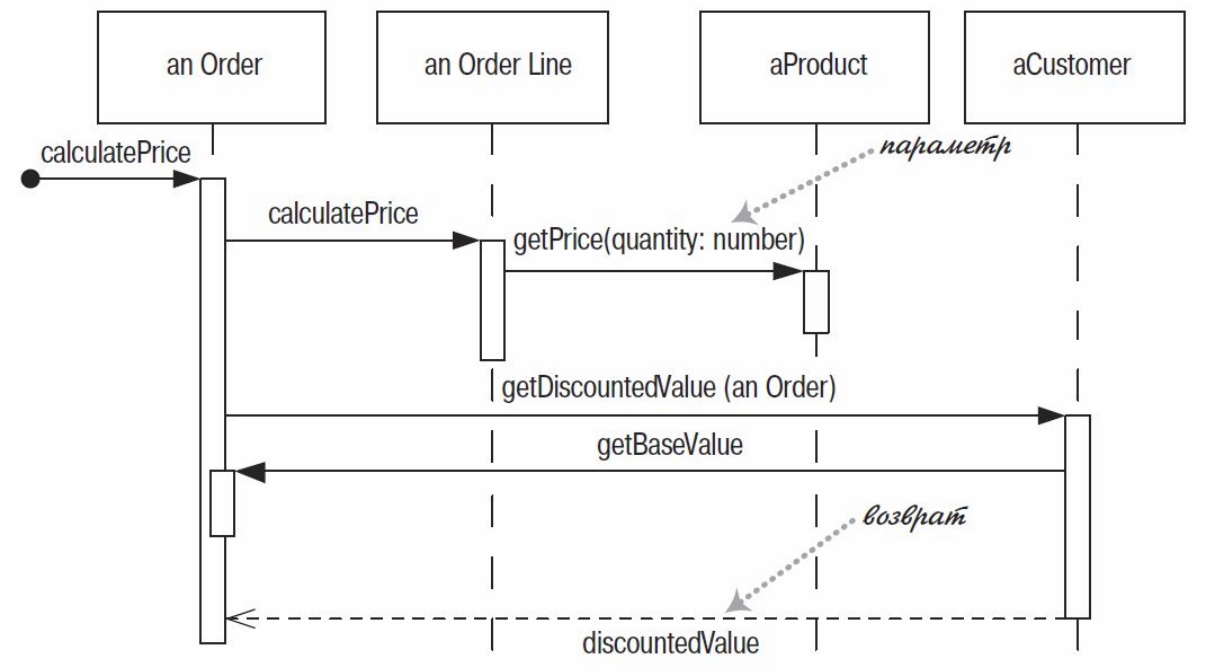


Нотация диаграмм последовательностей очень простая и понятная. Каждая линия жизни имеет полосу активности, которая показывает интервал активности участника при взаимодействии. Она соответствует времени нахождения в стеке одного из методов участника. Вообще в UML полосы активности не обязательны, но они весьма удобны (если только не приходится рисовать диаграммы на доске или на бумаге).

У вызовов можно указывать или не указывать возврат. Если возврат указывается, над ним можно задать имя возвращаемого значения, это позволит связать его с происходящим далее (например, getProduct() возвращает объект aProduct, к которому потом идёт следующий запрос).

Другой подход к решению задачи можно увидеть на следующей диаграмме. Основная задача остается той же самой, но способ взаимодействия участников для ее решения совершенно другой. Заказ спрашивает каждую строку заказа о его собственной цене (Price). Сама строка заказа передает вычисление дальше – объекту продукта (Product). Подобным же образом для вычисления скидки объект заказа

вызывает метод для клиента (Customer). Поскольку для выполнения этой задачи клиенту требуется информация от объекта заказа, то он делает повторный вызов в отношении заказа для получения этих данных.

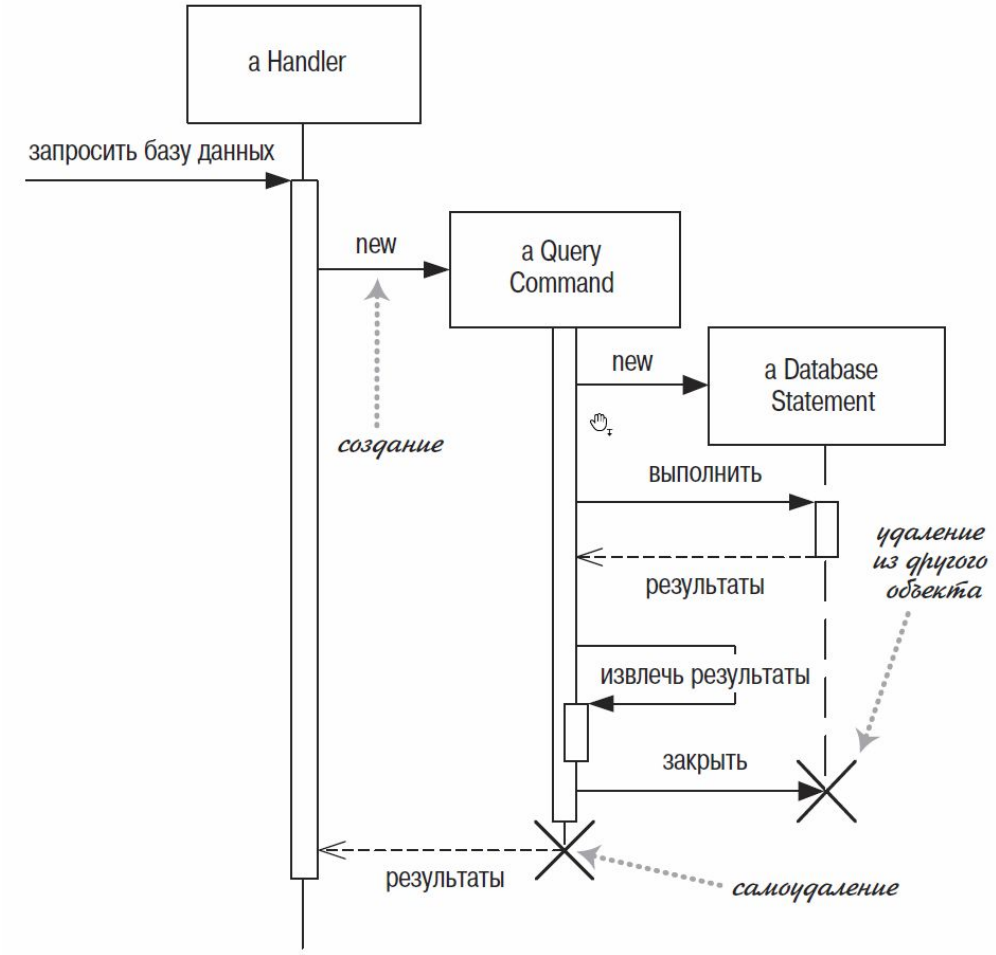


На этих двух диаграммах надо обратить внимание на то, насколько ясно диаграмма последовательности показывает различия во взаимодействии участников. В этом проявляется мощь диаграмм взаимодействий. Они не очень хорошо представляют детали алгоритмов, такие как циклы или условное поведение, но делают абсолютно прозрачными вызовы между участниками и дают действительно ясную картину того, какую обработку выполняют конкретные участники. На первой диаграмме представлено централизованное управление, когда один из участников в значительной степени выполняет всю обработку, а другие предоставляют данные. На второй диаграмме изображено распределенное управление, при котором обработка распределяется между многими участниками, каждый их которых выполняет небольшую часть алгоритма.

Оба стиля обладают преимуществами и недостатками. Большинство разработчиков, особенно новички в ООП, чаще всего применяют централизованное управление. Во многих случаях это проще, так как вся обработка сосредоточена в одном месте. Напротив, в случае распределенного управления при попытке понять программу создается ощущение погони за объектами. Но надо учитывать, что одна из главных задач хорошего проектирования заключается в локализации изменений. Данные и программный код, получающий доступ к этим данным, часто изменяются вместе. Поэтому размещение данных и обращающейся к ним программы в одном месте -- первое правило объектно-ориентированного проектирования. Кроме того, распределенное управление позволяет создать больше возможностей для применения полиморфизма, чем в случае применения условной логики. Если алгоритмы определения цены отличаются для различных типов продуктов, то механизм распределенного управления позволяет нам использовать подклассы класса продукта для обработки этих вариантов.

Вообще, объектно-ориентированный стиль предназначен для работы с большим количеством небольших объектов, обладающих множеством небольших методов, что дает широкие возможности для переопределения и изменения. Этот стиль сбивает с толку людей, применяющих длинные процедуры. Научиться этому может быть трудно, потому как требуется перестроение мышления.

В диаграммах последовательности для создания и удаления участников применяются некоторые дополнительные обозначения:

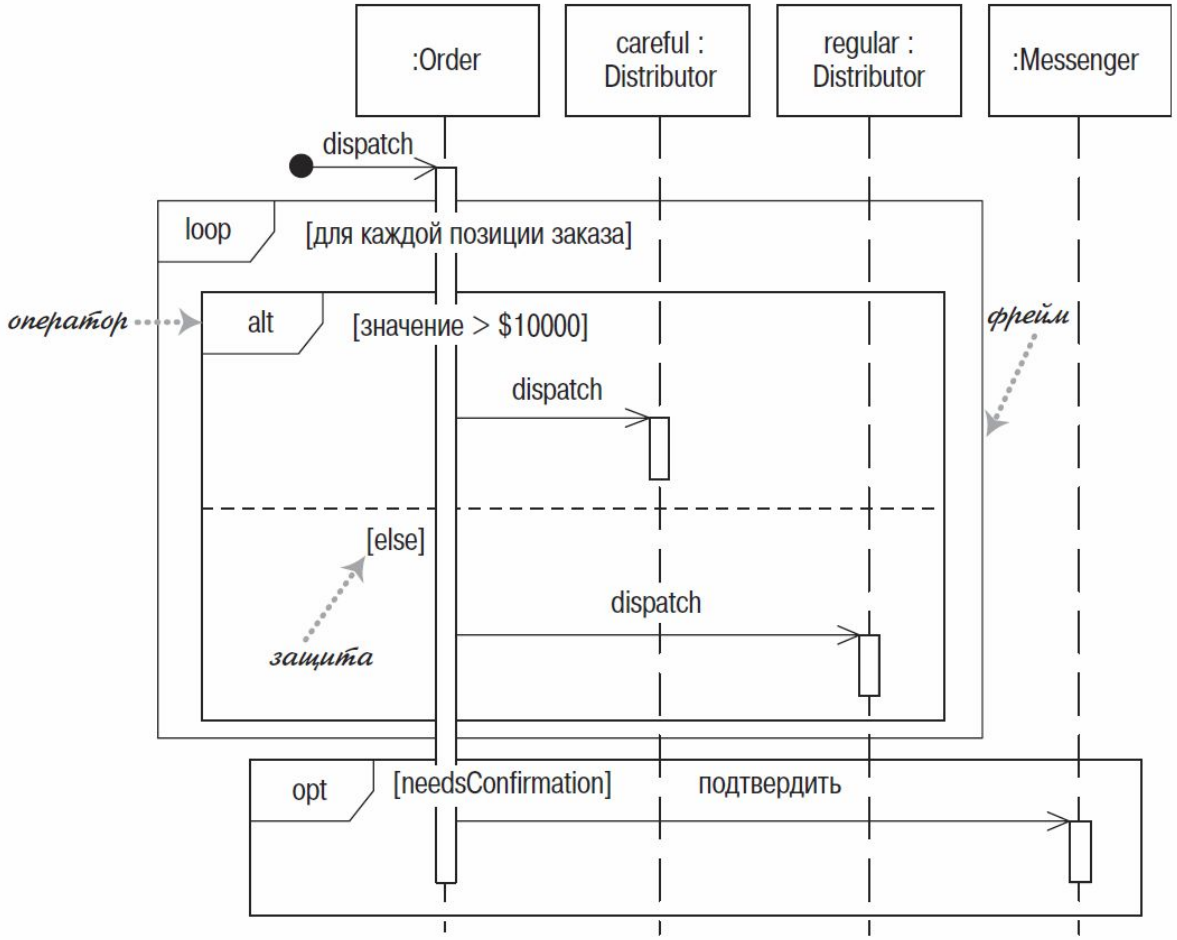


Общая проблема диаграмм последовательности заключается в том, как отображать циклы и условные конструкции. Прежде всего надо напомнить себе, что диаграммы последовательности для этого не предназначены. Подобные управляющие структуры лучше показывать с помощью диаграммы деятельности или собственно кода. Диаграммы последовательности применяются для визуализации процесса взаимодействия объектов, а не как средство моделирования алгоритма управления.

И для циклов, и для условий используются специальные обозначения, фреймы взаимодействий (interaction frames), представляющие собой средство разметки диаграммы взаимодействия. На диаграмме ниже показан простой алгоритм, основанный на следующем псевдокоде.

```
foreach (lineitem)
      if (product.value > $10K)
            careful.dispatch
```

```
        else
              regular.dispatch
        end if
  end for
  if (needsConfirmation)
        messenger.confirm
```



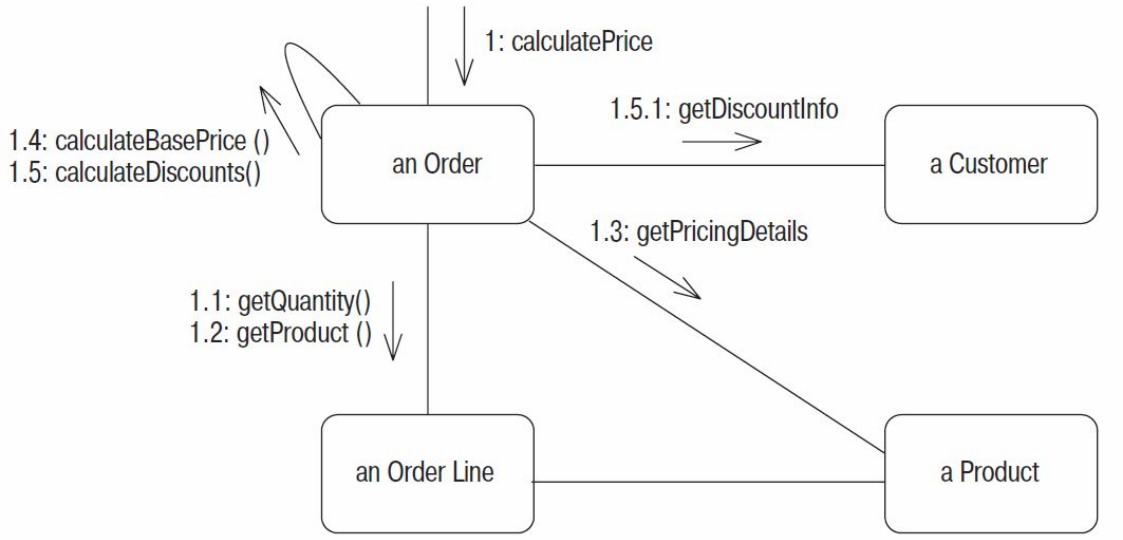
В основном фреймы состоят из некоторой области диаграммы последовательности, разделенной на несколько фрагментов. Каждый фрейм имеет оператор, а каждый фрагмент может иметь охранное условие. Так, для отображения цикла применяется оператор loop с единственным фрагментом, а тело итерации помещается в охранное условие. Для условной логики можно использовать оператор alt и помещать условие в каждый фрагмент. Будет выполнен только тот фрагмент, охранное условие которого имеет истинное значение. Для единственной области существует оператор opt.

Также на этой диаграмме используются обычные стрелки в отличие от предыдущей диаграммы, где стрелки были закрашенные. Закрашенные стрелки показывают синхронное сообщение, а простые стрелки -- асинхронное.

В заключение заметим, что диаграммы последовательности удобно использовать тогда, когда требуется посмотреть на взаимодействие нескольких объектов в рамках одного сценария использования. Если вы хотите посмотреть на поведение одного

объекта в нескольких случаях использования, то этого хорошо подойдёт диаграмма конечных автоматов. Если же надо изучить поведение нескольких объектов в нескольких сценариях или потоках, поможет диаграмма активности.

Ну и рассмотрим кратко коммуникационные диаграммы, которые решают ту же самую задачу, что и диаграммы последовательностей. Вместо того, чтобы рисовать каждого участника в виде линии жизни и показывать последовательность сообщений, располагая их по вертикали, как это делается в диаграммах последовательности, коммуникационные диаграммы допускают произвольное размещение участников, позволяя рисовать связи, показывающие отношения участников, и использовать нумерацию для представления последовательности сообщений.



Вложенная десятичная нумерация выглядит довольно странно, но нужна, потому что требуется исключить неопределенность при самовызовах. Например, в этом примере четко показано, что метод getDiscountInfo() вызывается из метода calculateDiscount(), чего не было бы видно при сплошной нумерации.

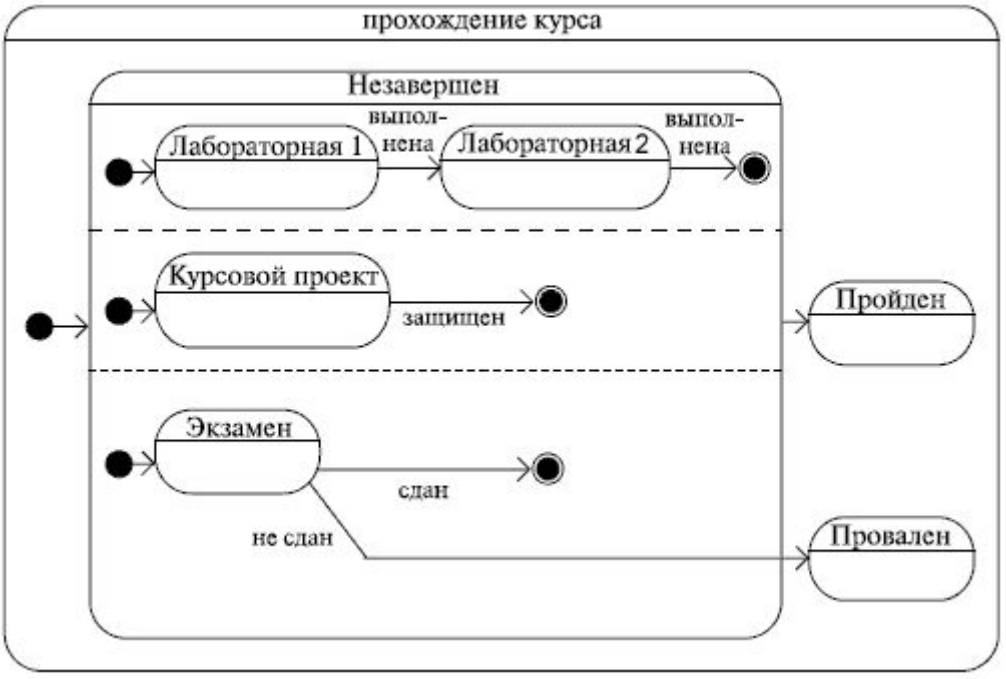
Если сравнивать эти два вида диаграмм, то можно предположить, что что диаграммы последовательности удобнее, если вы хотите подчеркнуть последовательность вызовов, а коммуникационные диаграммы лучше выбрать, когда надо акцентировать внимание на связях.

Моделирование параллельного поведения

Касательно представления потоков управления в UML остается сказать, что в реальных программных системах потоков управления может быть несколько. Все типы поведенческих диаграмм умеют в той или иной степени отражать данный факт.

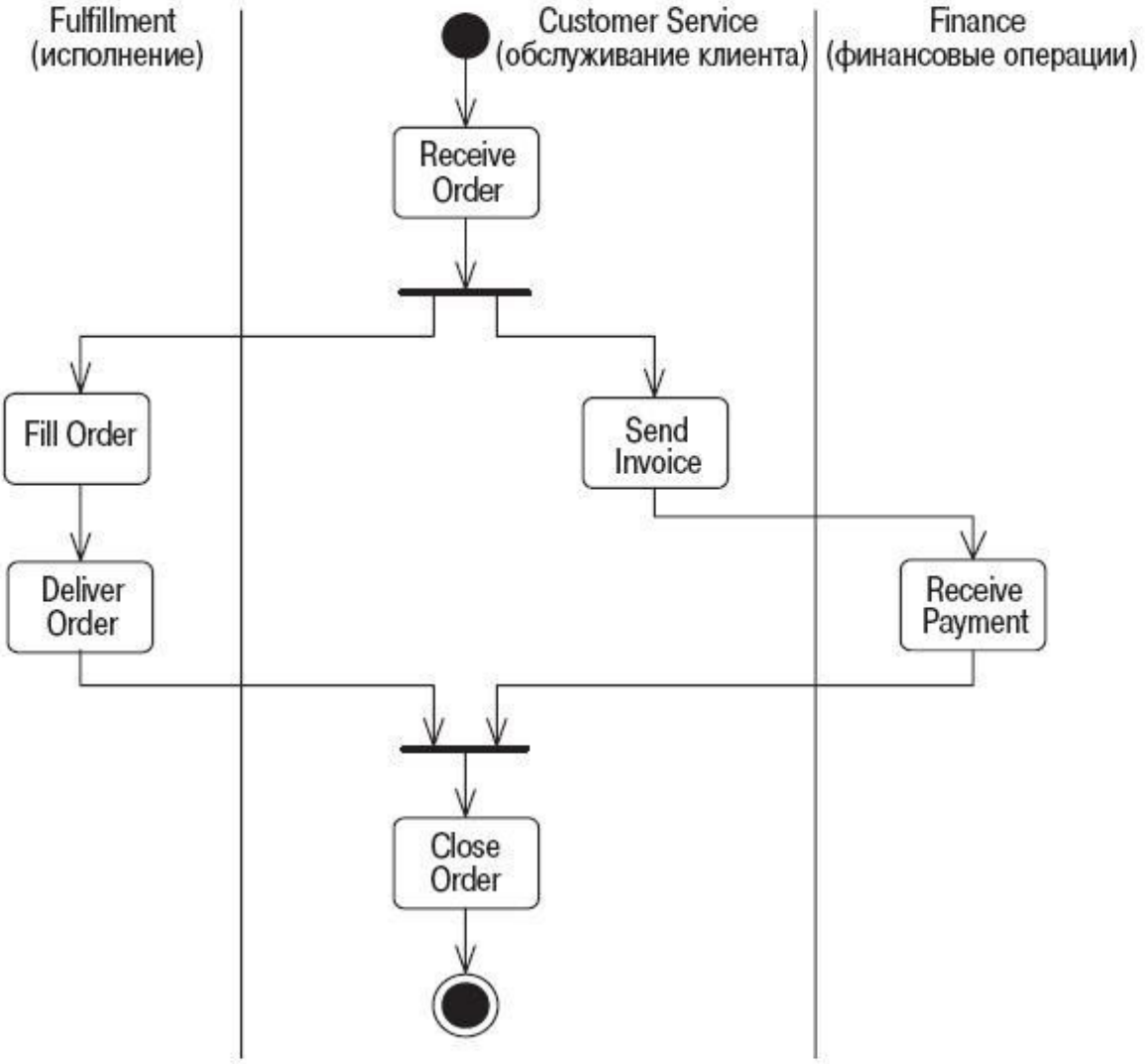
Диаграммы конечных автоматов в качестве событий для инициализации процесса перехода объекта из одного состояния в другое могут использовать события, являющиеся асинхронными, т.е. возникающими в другом потоке управления, нежели тот, в котором "живет" моделируемый объект. К тому же, есть возможность задавать параллельные подсостояния, которые специфицируют два и более подавтомата, которые могут выполняться параллельно внутри составного события. Каждый из подавтоматов занимает некоторую область или регион внутри составного состояния, которая отделяется от остальных горизонтальной пунктирной линией. Если на

диаграмме состояний имеется составное состояние с вложенными параллельными подсостояниями, то объект может одновременно находиться в каждом из этих подсостояний.



Поскольку каждый регион вложенного состояния специфицирует некоторый подавтомат, то для каждого из вложенных подавтоматов могут быть определены собственные начальное и конечные подсостояния. При переходе в данное составное состояние каждый из подавтоматов оказывается в своем начальном подсостоянии. Далее происходит параллельное выполнение каждого из этих подавтоматов, причем выход из составного состояния будет возможен лишь в том случае, когда все подавтоматы будут находиться в своих конечных подсостояниях. Если какой-либо из подавтоматов пришел в свое конечное состояние раньше других, то он должен ожидать, пока и другие подавтоматы не придут в свои конечные состояния.

На диаграммах активности же для отображения параллельного выполнения могут использовать специальные конструкции, который показывают точки, в которых поток управления может быть разделен на несколько параллельно исполняющихся потоков и точки, в которых, наоборот, несколько потоков управления опять сливаются вместе.


Fulfillment
(исполнение)
Customer Service
(обслуживание клиента)
Finance
(финансовые операции)
Receive
Order
Fill Order
Send
Invoice
Deliver
Order
Receive
Payment
Close
Order