



МИНОБРНАУКИ РОССИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технологический университет «СТАНКИН»
(ФГБОУ ВО МГТУ «СТАНКИН»)

Кафедра "Информационные технологии и вычислительные системы"

Новоселова Ольга Вячеславовна

Современные технологии и средства разработки программного обеспечения

Курс лекций
по дисциплине «Современные технологии и средства разработки программного обеспечения»

для студентов МГТУ «СТАНКИН», обучающихся

по направлению: 09.03.01 "Информатика и вычислительная техника" ,

по профилю: "Программное обеспечение средств вычислительной техники и автоматизированных систем"

Москва 2020г.

Оглавление.
7 семестр

Раздел 4. Современные технологии формирования архитектуры программного продукта.

<u>Лекции 10-12.</u>	<u>3</u>
----------------------	----------

Тема 10. Архитектура, управляемая моделями. Платформенно-независимая модель (проект) программного продукта (PIM-модель).

Технология построения платформенно-независимого проекта программного продукта с помощью языка моделирования UML.

Тема 11. Анализ сценариев использования и построение диаграммы классов программного продукта с помощью языка моделирования UML. Построение диаграммы последовательности (взаимодействия).

Тема 12. Анализ взаимодействия классов программного продукта и построения для них диаграмм состояний (конечного автомата) с помощью языка моделирования UML.

<u>Контрольные вопросы</u>	<u>34</u>
----------------------------	-----------

<u>Список литературы</u>	<u>35</u>
--------------------------	-----------

7 семестр.

Раздел 4. Современные технологии формирования архитектуры программного продукта.

Лекции 10-12.

Тема 10. Архитектура, управляемая моделями. Платформенно-независимая модель (проект) программного продукта (PIM-модель).

Технология построения платформенно-независимого проекта программного продукта с помощью языка моделирования UML.

Тема 11. Анализ сценариев использования и построение диаграммы классов программного продукта с помощью языка моделирования UML.

Построение диаграммы последовательности (взаимодействия).

Тема 12. Анализ взаимодействия классов программного продукта и построения для них диаграмм состояний (конечного автомата) с помощью языка моделирования UML.

План лекции.

1. Архитектура, управляемая моделями.
2. Платформенно-независимая модель (проект) программного продукта (PIM-модель).
3. Технология построения платформенно-независимого проекта программного продукта с помощью языка моделирования UML.
4. UML: диаграмма классов.
5. UML: диаграмма последовательности.
6. UML: диаграмма состояний (конечного автомата).

Основная часть.

Архитектура, управляемая моделями. Платформенно-независимая модель (проект) программного продукта (PIM-модель).

Аббревиатура MDA расшифровывается как Model Driven Architecture - архитектура, управляемая моделью. MDA - это архитектура, описывающая новый способ разработки программного обеспечения.

В основе новой архитектуры лежит идея о полном разделении этапов общего проектирования (моделирования) и последующей реализации приложения на конкретной программной платформе. Идея эта не нова: сначала при помощи специальных средств проектирования создается общая и независимая от способов реализации модель приложения, а затем осуществляется реализация программы в какой-либо среде разработки. При этом процесс разработки полностью основан на модели, которая должна содержать всю необходимую для программирования информацию. Очевидны преимущества, которые дает такой подход:

- Независимость модели от средств разработки обеспечивает возможность реализации на любой программной платформе.
- Приложение, реализованное в архитектуре MDA, может быть легко перенесено из одной операционной системы в другую.
- Существенна экономия ресурсов при реализации приложения для нескольких программных платформ одновременно.
- Архитектура позволяет до известной степени автоматизировать процесс программирования.

Наличие подробной модели обеспечивает автоматическое создание типовых частей приложения, разработка которых поддается автоматизации. Например, создание пользовательского интерфейса, программирование типовых операций, создание базы данных и организация доступа к данным.

Появление MDA и возможность реализации определило необходимость ряда стандартов и технологий, на практике доказавших свою полезность. Концептуальной основой появления MDA стали спецификации OMA, ORB, CORBA. Перевести замысел в практическую плоскость позволили технологии объектно-ориентированного программирования (ООП), стандарт CWM, языки UML, XML, MOF. Работами по созданию новой архитектуры программирования занялся консорциум OMG (Object Management Group).

По мнению создателей, архитектура MDA является новым витком эволюции технологий программирования, так как описывает процесс разработки в целом. Новизна MDA заключается в том, что описание процесса разработки в ней выполнено с использованием современных средств представления и позволяет автоматизировать создание приложений. И весьма вероятно, что через некоторое время архитектура MDA станет общим промышленным стандартом в разработке программного обеспечения.

Основные понятия

Модель - описание или спецификация системы и ее окружения, созданная для определенных целей. Часто является комбинацией текстовой и графической информации. Текст может быть описан специализированным или естественным языком.

Управление на основе модели процесс разработки системы, использующий модель для понимания, конструирования, распространения и других операций.

Платформа набор подсистем и технологий, которые представляют собой единый набор функциональности, используемой любым приложением без уточнения деталей реализации.

Вычислительная независимость качество модели, обозначающее отсутствие любых деталей структуры и процессов.

Платформенная независимость качество модели, обозначающее ее независимость от свойств любой платформы.

Вычислительно-независимая модель - модель, скрывающая любые детали реализации и процессов системы; описывает только требования к системе и ее окружению.

Платформенно-независимая модель - модель, скрывающая детали реализации системы, зависящие от платформы, и содержащая элементы, не изменяющиеся при взаимодействии системы с любой платформой.

Платформенно-зависимая модель - модель системы с учетом деталей реализации и процессов, зависящих от конкретной платформы.

Модель платформы - набор технических характеристик и описаний технологий и интерфейсов, составляющих платформу.

Преобразование модели - процесс преобразования одной модели системы в другую модель той же системы.

Архитектура разработки приложений на основе моделей

Архитектура MDA описывает и структурирует поэтапный процесс разработки любых программных систем на основе создания и использования моделей. При этом используется несколько типов моделей, создаваемых и преобразуемых на различных этапах разработки. Процесс разработки по MDA это последовательное (поэтапное) продвижение от одной модели системы к другой. При этом каждая последующая модель преобразуется из предыдущей и дополняется новыми деталями. Модели, общая схема разработки и процесс преобразования моделей - ключевые составные части архитектуры.

Типы моделей

Рассмотрим типы моделей, используемые в архитектуре MDA.

Вычислительно-независимая модель (Computation Independent Model, CIM) описывает общие требования к системе, словарь используемых понятий и условия функционирования (окружение). Модель не должна содержать никаких сведений технического характера, описаний структуры и функционала системы. CIM максимально общая и независимая от реализации системы модель. Спецификация MDA подчеркивает, что CIM

должна быть построена так, чтобы ее можно было преобразовать в платформенно-независимую модель. Поэтому CIM рекомендуется выполнять с использованием унифицированного языка моделирования UML.

Платформенно-независимая модель (Platform Independent Model, PIM) описывает состав, структуру, функционал системы. Модель может содержать сколь угодно подробные сведения, но они не должны касаться вопросов реализации системы на конкретных платформах. Модель PIM создается на основе CIM. Для создания модели используется унифицированный язык моделирования UML.

Платформенно-зависимая модель (Platform Specific Model, PSM) описывает состав, структуру, функционал системы применительно к вопросам ее реализации на конкретной платформе. В зависимости от назначения модель может быть более или менее детализированной. Модель создается на основе двух моделей. Модель PIM служит основой модели PSM. Модель платформы используется для доработки PSM в соответствии с требованиями платформы.

Модель платформы описывает технические характеристики, интерфейсы, функции платформы. Зачастую модель платформы представлена в виде технических описаний и руководств. Модель платформы используется при преобразовании модели PIM в модель PSM. Для целей MDA описание модели платформы должно быть представлено на унифицированном языке моделирования UML.

Уровни модели

В зависимости от уровня детализации платформы, модели (кроме модели платформы) могут содержать сведения о различных функциональных частях системы. В этом случае говорят об уровнях модели. Обычно различают следующие основные уровни модели.

Уровень бизнес-логики содержит описание основного функционала приложения, обеспечивающего исполнение его назначения. Как правило, уровень бизнес-логики хуже всего поддается автоматизации. Он составляет огромную долю кода приложения, который приходится писать вручную.

Уровень данных описывает структуру данных приложения, используемые источники, форматы данных, технологии и механизмы доступа к данным.

Уровень пользовательского интерфейса описывает возможности приложения по взаимодействию с пользователями, а также состав форм приложения, функционал элементов управления (например, контроль ввода данных). Легкость автоматизации этого уровня зависит от того, насколько унифицированы пользовательские операции. Если удастся создать типовые шаблоны элементов управления для основных операций, появляется возможность автоматической генерации форм и их содержимого при создании приложения из модели.

Этапы разработки

Процесс разработки разбивается на три этапа (рис.4.1).

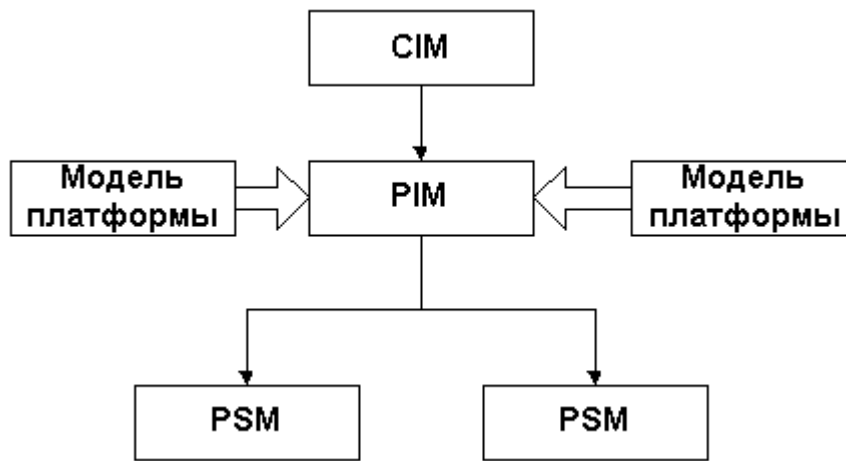


Рисунок 4.1. Этапы разработки программной системы в соответствии с MDA-архитектурой

1. На первом этапе разрабатывается вычислительно-независимая модель (CIM). Часто модель, создаваемую на этом этапе, также называют доменной или бизнес-моделью. Цель данного этапа разработка общих требований к системе, создание общего словаря понятий, описание окружения, в котором система будет функционировать.

Сущности, описываемые в модели CIM этого этапа, должны тщательно анализироваться и отрабатываться. Право на включение в модель должны иметь только те элементы, которые будут использованы и развиты на последующих этапах разработки. Для создания модели CIM на данном этапе можно использовать любые средства. Однако для совместимости с последующими этапами весьма желательно иметь описание модели на языке UML. Следует учитывать, что модель CIM первого этапа представляет собой скорее общую концепцию системы и не является насущно необходимой для процесса разработки приложения. При создании небольших программных систем этот этап можно опустить, однако при работе со сложными проектами он становится почти обязательным. К примеру, разработка текстового технического задания, казалось бы, никак не помогает собственно процессу программирования, зато существенно способствует пониманию задачи в целом и позволяет избежать грубых ошибок проектирования в дальнейшем.

2. На втором этапе разрабатывается платформенно-независимая модель (PIM). Она может разрабатываться с нуля в случае отсутствия модели первого этапа или основываться на CIM. Преобразование CIM в PIM осуществляется на основе описания на языке UML, созданного на первом этапе. Здесь в него добавляются элементы, описывающие бизнес-логику, общую структуру системы, состав и взаимодействие подсистем, распределение функционала по элементам, общее описание и требования к пользовательскому интерфейсу. Модель PIM этого этапа обязательно включается во все автоматизированные среды разработки приложений на основе MDA.

3. На третьем этапе создаются платформенно-зависимые модели (PSM). Их число соответствует числу программных платформ, на которых будет функционировать приложение. Кроме этого, возможны случаи, когда приложение (или его составные части) должны работать на нескольких платформах одновременно. Модель PSM создается путем преобразования модели PIM с учетом требований модели платформы. На этапе создания модели PSM разработка приложения согласно MDA архитектуре заканчивается.

Считается, что правильно построенная PSM содержит техническую информацию, достаточную для генерации исходного кода (там, где это возможно) и необходимых ресурсов приложения. Здесь эстафетную палочку должна подхватить среда разработки, реализующая MDA. Архитектура MDA описывает еще один вариант прохождения третьего этапа, который называется *прямым преобразованием в код*. Спецификация MDA для этого случая сообщает,

что могут существовать инструментарии, напрямую преобразующие модель PIM в исполняемый код приложения. Модель PSM при этом может создаваться как контрольное описание, позволяющее проверить результат прямого преобразования.

Преобразование моделей PIM - PSM

Наиболее сложным и ответственным этапом при разработке приложений в рамках архитектуры MDA является преобразование модели PIM в модель PSM. Именно на этом этапе общее описание системы на языке UML приобретает вид, пригодный для воплощения приложения на конкретной платформе. Как уже говорилось, в процессе принимает участие модель платформы. Преобразование моделей проходит три последовательные стадии:

- Разработка схемы преобразования (mapping)
- Маркирование (marking)
- Собственно преобразование (transformation)

Первоначально необходимо разработать схему преобразования элементов модели PIM в элементы модели PSM (рис. 4.2). Для каждой платформы создается собственная схема преобразования, которая напрямую зависит от возможностей платформы. Схема преобразования затрагивает как содержание модели (совокупность элементов и их свойства), так и саму модель (метамодель, используемые типы). В схеме преобразования нужным типам модели, свойствам метамодели, элементам модели PIM ставятся в соответствие типы модели, свойства метамодели, элементы модели PSM. При преобразовании моделей может использоваться несколько схем преобразования. Для связывания используются марки (mark) самостоятельные структуры данных, принадлежащие не моделям, а схемам преобразования и содержащие информацию о созданных связях. Наборы марок могут быть объединены в тематические шаблоны, которые возможно использовать в различных схемах преобразования. Процесс задания марок называется маркированием. В простейшем случае один элемент модели PIM соединяется маркой с одним элементом модели PSM. В более сложных случаях один элемент модели PIM может иметь несколько марок из разных схем преобразования. Что касается преобразования метамодели, то в большинстве случаев марки могут расставляться автоматически. А вот для элементов модели часто требуется вмешательство разработчика. В процессе маркирования необходимо использовать сведения о платформе. Эти сведения содержатся в модели платформы.

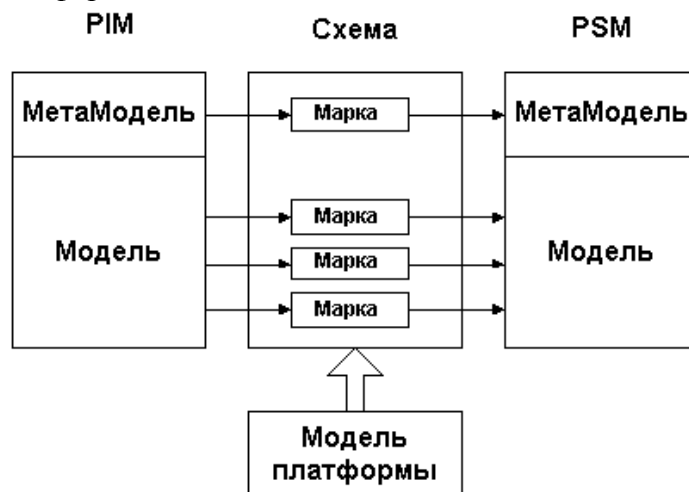


Рисунок 4.2. Схема преобразования PIM – PSM.

Процесс преобразования моделей заключается в переносе маркированных элементов модели и метамодели PIM в модель и метамодель PSM. Процесс преобразования должен документироваться в виде карты переноса элементов модели и метамодели. Способ преобразования моделей может быть:

- Ручной

- С использованием профилей
- С настроенной схемой преобразования
- Автоматический

Анализ сценариев использования и построение диаграммы классов программного продукта с помощью языка моделирования UML.

Диаграмма классов.

Диаграмма классов - основной способ отображения структуры разрабатываемой системы, показывающий классы и отношения между ними. Диаграмма классов - это графическое представление статической модели, в которой собраны декларативные (статичные) элементы, такие как классы, типы, а также их содержимое и отношения. Диаграмма классов содержит некоторые конкретные элементы поведения (например, операции), однако их динамика отражается на других видах диаграмм.

Следует заметить, что диаграмма классов может также содержать интерфейсы, отношения, а также объекты и связи. Когда говорят о данной диаграмме, то имеют в виду статическую структурную модель проектируемой системы. Поэтому диаграмму классов принято считать графическим представлением таких структурных взаимосвязей логической модели системы, которые не зависят от времени.

Объекты

Объект - предмет или понятие из реального мира. Это дискретная сущность с четко определенными границами и индивидуальностью, инкапсулирующая состояние и поведение. Экземпляр класса.

Особенности объекта:

- имя объекта;
- состояние объекта. Состояние включает в себя имена свойств (атрибутов! описывающих объект, и значения этих атрибутов в определенный момент времени;
- поведение объекта, определяющееся функциями (методами), которые используют или изменяют значения атрибутов объекта.

Классы

Класс - набор объектов, имеющих одинаковые характеристики (Обозначение класса см. рис. 4.3). Другими словами, является абстрактным описанием или представлением свойств множества объектов, которые обладают одинаковой структурой, поведением и отношениями с объектами из других классов. Класс используют для объявления переменных. Это поименованное описание структуры данных и поведения некоего множества объектов. Класс описывает некоторую концепцию в моделируемой системе. В зависимости от типа модели класс может отражать концепцию реального мира (например, для аналитической модели) или содержать концепции, относящиеся к алгоритмической или компьютерной реализации (для модели проектирования).

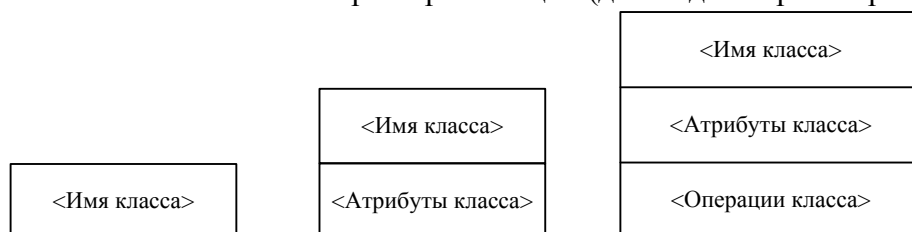


Рис. 4.3. Варианты обозначения класса.

Особенности класса:

- имя класса (имя объекта может включать в себя имя класса) – обязательный элемент;
- атрибуты каждого объекта класса;
- поведение в терминах операций, а не методов.

На начальных этапах разработки диаграммы отдельные классы могут обозначаться простым прямоугольником с указанием только имени соответствующего класса. По мере проработки отдельных компонентов диаграммы, описания классов дополняются атрибутами и операциями. Предполагается, что окончательный вариант диаграммы содержит наиболее полное описание классов, которые состоят из трех разделов или секций.

Имя класса – должно быть уникальным, указывается в самой верхней секции прямоугольника, записывается по центру секции имени полужирным шрифтом и должно начинаться с заглавной буквы, должны использоваться существительные без пробелов. Именно имена классов образуют словарь предметной области. Класс, не имеющий экземпляров или объектов, называется абстрактным классом, а для обозначения его имени используется наклонный шрифт.

Атрибут класса – служит для представления отдельного свойства или признака, который является общим для всех объектов данного класса. Каждый атрибут – отдельная строка, записывается во второй сверху секции прямоугольника класса, принята определенная стандартизация записи, которая подчинена синтаксическим правилам. Из всех элементов обязательно указывать имя атрибута (идентификатор соответствующего атрибута) – должно быть уникальным в пределах данного класса, должно начинаться со строчной буквы и не должно содержать пробелов, используют существительные.

Операция класса – это некоторый сервис, который предоставляет каждый экземпляр или объект класса по требованию своих клиентов (других объектов, в том числе и экземпляров данного класса). Совокупность операций характеризует функциональный аспект поведения всех объектов данного класса. Каждая операция – отдельная строка, записывается в третьей сверху секции прямоугольника, принята определенная стандартизация записи, которая подчинена синтаксическим правилам. Из всех элементов обязательно указывать имя операции (идентификатор операции) – должно быть уникальным в пределах данного класса, должно начинаться со строчной буквы и не должно содержать пробелов, используют глагол или глагольный оборот.

Операция - служба, которая может быть запрошена объектом для изменения поведения. Или это абстракция того, что позволено делать с объектом.

Метод - реализация этой службы.

Экземпляр класса - объект, принадлежащий определенному классу.

Отношения между классами

1. **Ассоциация** - структурная связь между классами (рис. 4.4). Соответствует наличию произвольного отношения или взаимосвязи между классами. Ассоциация двунаправлена - можно перемещаться от одного класса к другому и наоборот. Направление связи можно указать, используя стрелку (рис. 4.5). В качестве дополнительных специальных символов могут использоваться имя ассоциации, символ навигации, а также имена и кратность классов-ролей ассоциации (рис. 4.6).



Рис. 4.4. Отношение ассоциации: а) обозначение двунаправленной связи, б) пример.

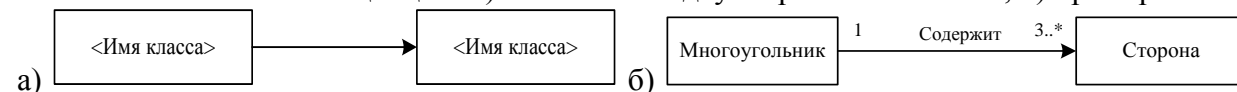


Рис. 4.5. Отношение ассоциации: а) обозначение однонаправленной связи, б) пример.

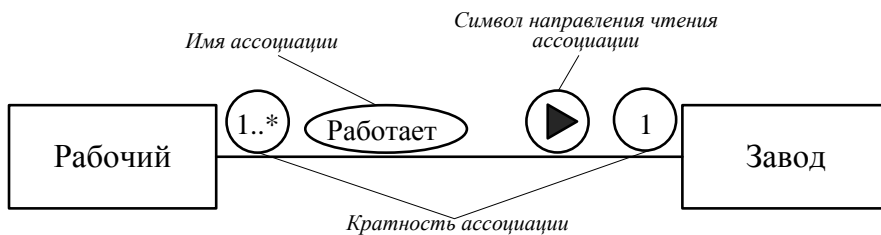


Рис. 4.6. Применение специальных символов.

2. **Обобщение** - отношение между более общим классом (суперклассом или родительским классом) и уточняющей версией этого класса (подклассом или потомком) (рис.4.7). Подкласс наследует атрибуты и операции своего суперкласса. Может использоваться для представления иерархических взаимосвязей между классами и т.д.



Рис. 4.7. Отношение обобщения: а) обозначение, б) пример.

3. **Агрегация** - специальный вид ассоциации, обозначающий отношение "целое/часть", при котором один или несколько классов являются "частями" одного "целого" (рис. 4.8). Имеет место между несколькими классами в том случае, если один из классов представляет собой некоторую сущность, которая включает в себя в качестве составных частей другие сущности. Раскрывая внутреннюю структуру системы отношение агрегации показывает, из каких элементов состоит система и как они связаны между собой. Порождаемая здесь иерархия принципиально отличается от порождаемой отношением обобщения – здесь части системы никак не обязаны наследовать ее свойства и поведение, поскольку являются вполне самостоятельными сущностями. Может указываться кратность.

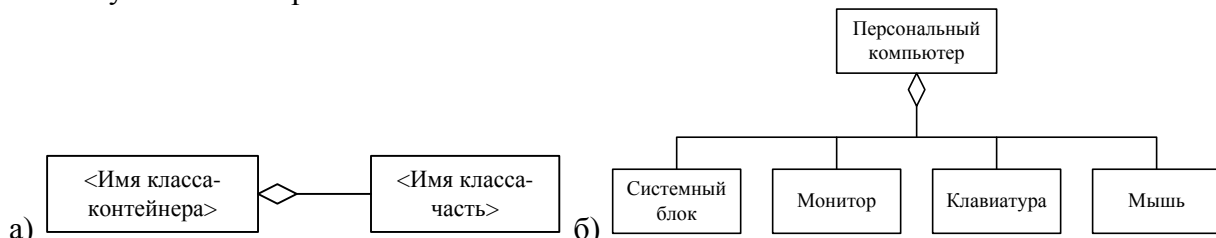


Рис. 4.8. Отношение агрегации: а) обозначение, б) пример.

4. **Композиция** – частный случай отношения агрегации. Служит для спецификации более сильной формы отношения «часть-целое», при которой составляющие части тесно взаимосвязаны с целым (рис. 4.9). Специфика заключается в том, что части не могут выступать в отрыве от целого, то есть с уничтожением целого уничтожаются и все его составные части. Может указываться кратность.

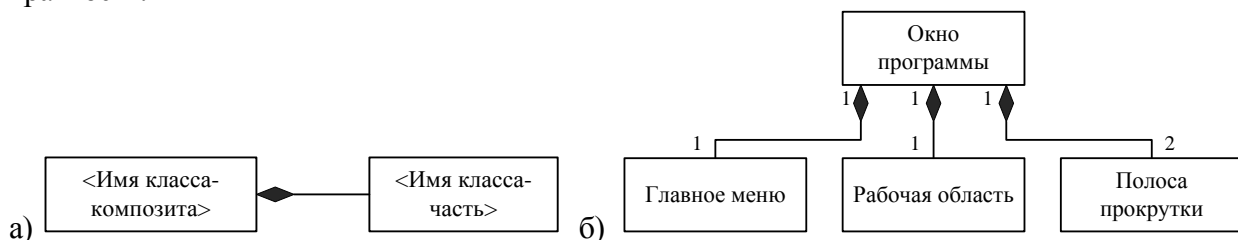


Рис. 4.9. Отношение композиции: а) обозначение, б) пример.

5. Зависимость – в общем случае указывает на некоторое семантическое отношение между двумя элементами модели или двумя множествами таких элементов, которые не являются отношением ассоциации, обобщения, агрегации или композиции. Оно касается только самих элементов модели. Используется в такой ситуации, когда некоторое изменение одного элемента модели может потребовать изменения другого зависящего от него элемента модели. (рис. 4.10).

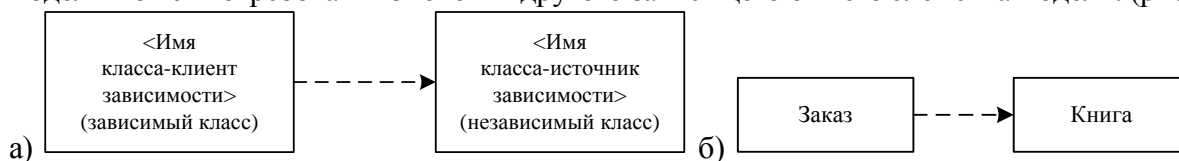


Рис. 4.10. Отношение зависимости: а) обозначение, б) пример.

Стрелка зависимости может помечаться необязательными, но стандартными ключевыми словами в кавычках, например: «access» (доступ), «bind» (связывание), «call» (вызов), «derive» (вызов), «friend» (дружественность), «import» (импорт), «instantiate» (создание экземпляра), «parameter» (параметр), «realize» (реализация), «refine» (уточнение), «send» (отправка), «trace» (трассировка), «use» (использование).

Интерфейсы

Интерфейс является специальным случаем класса, у которого имеются только операции и отсутствуют атрибуты (рис. 4.11). Применительно к диаграммам классов, интерфейсы определяют совокупность операций, который обеспечивают необходимый набор сервисов или функциональности для актеров. Интерфейсы не могут содержать ни атрибутов, ни состояний, ни направленных ассоциаций. Они содержат только операции без указания особенностей их реализации.

Важность интерфейсов заключается в том, что они определяют стыковочные узлы в проектируемой системе, что совершенно необходимо для организации коллективной работы над проектом.

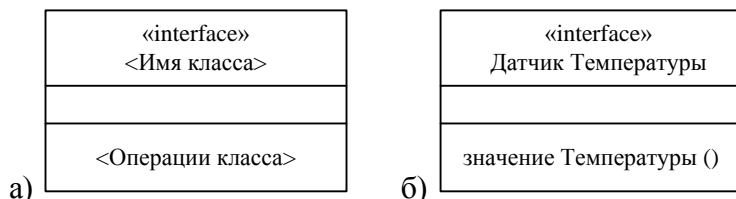


Рис. 4.11. Интерфейс: а) обозначение, б) пример.

Элементы, связи, ограничения, манипуляции UML

Элементы и связи

Элементы и связи между элементами UML:

- диаграмм классов - таблица 4.1.

Табл. 4.1

Обозначение элемента	Название элемента	Что отражает
<Имя класса>	Класс	Объект предметной области
<Имя класса> <Атрибуты класса>	Класс с атрибутами	Объект предметной области с определением его состояния

<div><Имя класса></div> <div><Атрибуты класса></div> <div><Операции класса></div>	Класс с атрибутами и операциями		Объект предметной области с определением его состояния и поведения
	двунаправленная ассоциация	Связь между объектами предметной области (отношение)	Структурная связь между классами
	однонаправленная ассоциация		Структурная связь между классами
	обобщение		Отношение «класс-подкласс»
	агрегация		Отношение «целое-часть»
	композиция		Отношение «целое-часть»
	зависимость		Семантическое отношение между элементами модели

Ограничения и манипуляции

1. Необходимо дать диаграмме имя, связанное с ее назначением.
2. На диаграмме необходимо располагать элементы так, чтобы свести к минимуму число пересекающихся линий.
3. Элементы должны быть пространственно организованы так, чтобы семантически близкие сущности располагались рядом.
4. Для привлечения внимания к важным особенностям диаграммы можно использовать примечания и цвет.
5. Необходимо стараться не показывать слишком много разных видов отношений, как правило, в каждой диаграмме классов должны доминировать отношения какого-либо одного вида.

Диаграмма классов - модель предметной области с атрибутами.

На данном этапе описания классов, выделенных в модели предметной области, дополняются атрибутами. Для каждого класса формируется свой набор атрибутов, описывающих свойства данного класса. Также существует возможность дополнить диаграмму не учтенными ранее классами.

Построение диаграммы последовательности (взаимодействия).

Диаграммы последовательности действий: базовые понятия

Диаграмма последовательности действий – одна из двух диаграмм взаимодействий.

На этих диаграммах показывают связи, включающие множество объектов и отношений между ними, в том числе сообщения, которыми объекты обмениваются. При этом диаграмма последовательности акцентирует внимание на временной упорядоченности сообщений, а диаграмма кооперации – на структурной организации посылающих и принимающих сообщения объектах. Данные диаграммы семантически эквивалентны и преобразуются друг в друга без потери информации.

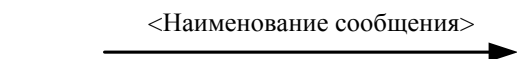
Диаграмма последовательности - диаграмма, отражающая временной порядок сообщений, передаваемых между объектами; используется для распределения операций между классами.

Графически такая диаграмма представляет собой таблицу, объекты в которой располагаются вдоль оси X, а сообщения в порядке возрастания времени – вдоль оси Y. Обычно инициирующий взаимодействие объект размещают слева, а остальные – правее (тем дальше, чем более подчиненным является объект). Причем никакие статические связи между объектами

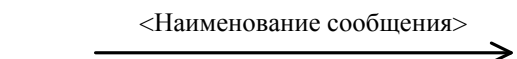
не визуализируются. Затем вдоль оси Y размещаются сообщения, которые объекты посылают и принимают, причем начальному сообщению соответствует самая верхняя часть диаграммы, а более поздние оказываются ниже. При этом масштаб на оси времени не указывается, поскольку моделируется лишь временная упорядоченность взаимодействий типа «раньше - позже». Это дает картину, позволяющую понять развитие потока управления во времени.

Сообщения и действия

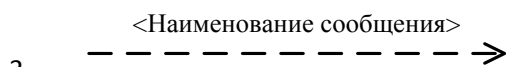
Сообщение - связь между двумя объектами или в пределах одного объекта, означающая какой-либо вид деятельности. Спецификация обмена данными между объектами, при котором передается некая информация в расчете на то, что в ответ последует определенное действие.



1. Эта разновидность сообщений наиболее распространена: используется для выполнения операций, вызова процедур, обозначения вложенных потоков управления. Начало стрелки соприкасается с фокусом управления того объекта-клиента, который инициирует это сообщение. Конец стрелки соприкасается с линией жизни того объекта, который принимает это сообщение и выполняет в ответ определенные действия. При этом принимающий объект может получить фокус управления, становясь в этом случае активным. Передающий объект может потерять фокус управления или остаться активным.



2. Данный вид сообщения используется для обозначения простого асинхронного сообщения, которое передается в произвольный момент времени. Передача такого сообщения обычно не сопровождается получением фокуса управления объектом-получателем.

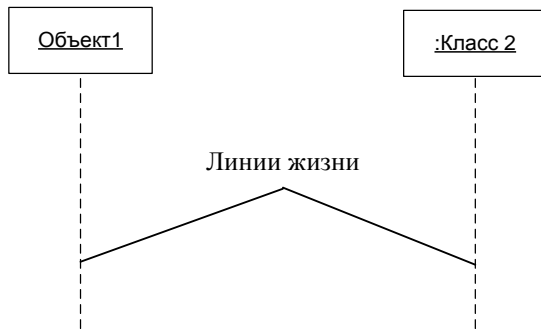


3. Данный вид сообщения используется для возврата из вызова процедуры. В процедурных потоках эта стрелка может быть опущена, поскольку ее наличие неявно предполагается в конце активизации объекта. Для непроцедурных потоков управления, включая параллельные и асинхронные сообщения, стрелка возврата должна указываться явным образом.

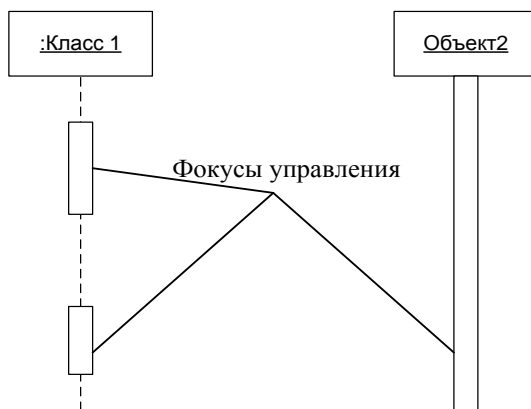
Вид деятельности - набор действий, выполняемых объектом на регулярной основе. Вид деятельности неатомарен - он разбивается на более мелкие виды деятельности.

Действие, являющееся результатом получения сообщения – это исполняемое выражение, приводящее к изменению одного или нескольких атрибутов объекта или к возврату некоторого значения (значений) объекту, отправившему сообщение. Действие атомарно - его нельзя прервать.

Линия жизни объекта - обозначение периода времени, в течение которого объект существует в системе и, следовательно, может потенциально участвовать во всех ее взаимодействиях. Линия жизни изображается пунктирной вертикальной линией на диаграмме последовательности.

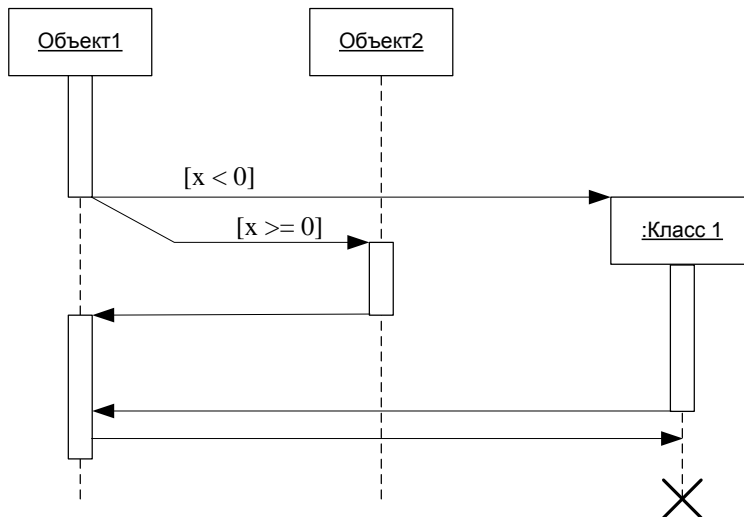


Фокус управления - период времени, в течение которого объект контролирует поток событий (выполняет некоторое действие, находясь в активном состоянии). Фокус управления на диаграмме последовательности указывать не обязательно.



Ветвление потока управления – на данных диаграммах можно визуализировать простое ветвление процесса – изображается необходимое количество стрелок, выходящих из одной точки фокуса управления объекта. При этом рядом с каждой из стрелок явно указывается соответствующее условие ветви в форме булевского выражения.

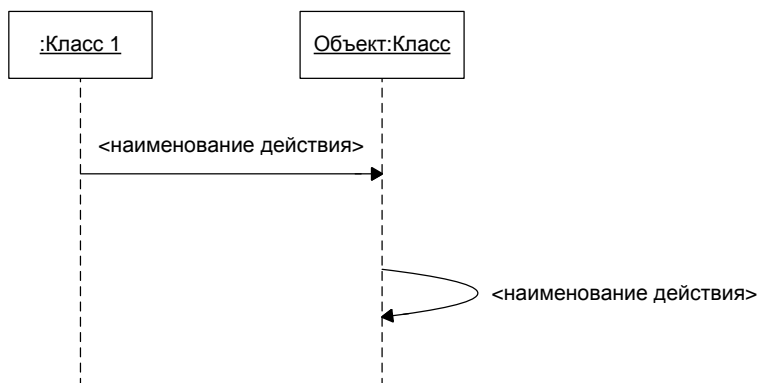
Количество ветвей может быть произвольным, однако наличие большого количества ветвлений может усложнить интерпретацию диаграммы последовательности. Условие должно быть явно указано для каждой ветви и записываться в форме обычного текста, псевдокода или выражения языка программирования. Это выражение всегда должно возвращать некоторое булевское выражение. Запись условий должна исключать одновременную передачу альтернативных сообщений по нескольким ветвям (двум и более), иначе может возникнуть конфликт ветвления.



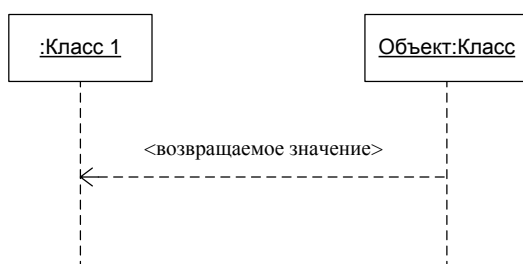
Типы действий

1. Вызов и возврат.

Действие вызова - вызов метода объекта (вызывает операцию, применяемую к объекту). Объект может производить действие вызова по отношению к другому объекту или по отношению к самому себе. Действие вызова обозначается стрелкой, направленной от вызывающего объекта к вызываемому.



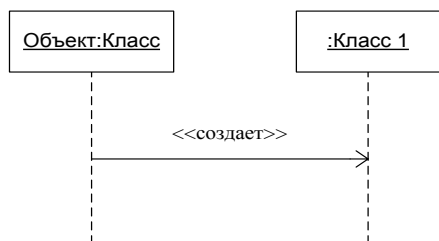
Действие возврата - возврат значения в ответ на действие вызова. Действие возврата обозначается пунктирной линией со стрелкой, направленной от объекта, отправляющего значение, к объекту, получающему значение.



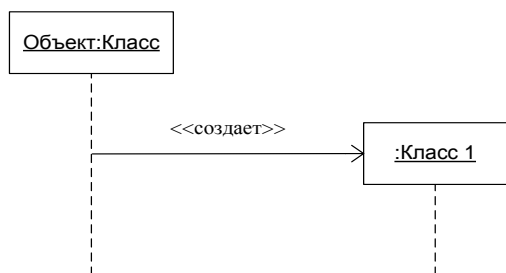
Действие возврата может не отображаться вместе с действием вызова, если возвращаемое значение очевидно из контекста.

2. Создание и уничтожение.

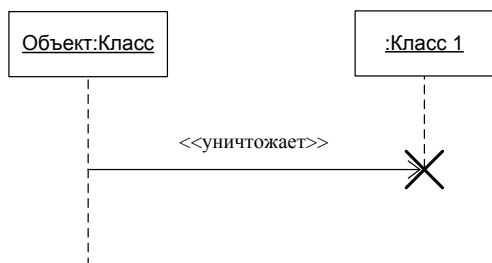
Действие создания – создание нового объекта (отправка запроса классу на создание его экземпляра).



Создаваемый объект можно помещать справа от линии "создания". Линия жизни этого объекта начинается с момента совершения действия создания.

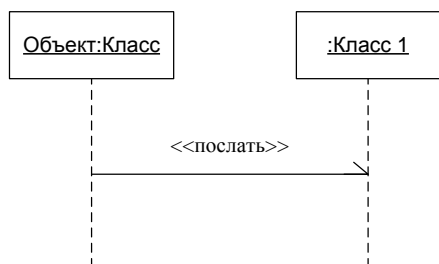


Действие уничтожения - уничтожение объекта. Символ X на конце линии - завершение линии жизни объекта. Объект может совершать действие уничтожения по отношению к другому объекту или по отношению к самому себе.



3. Послание

Действие «послать» - посылает объекту сигнал. Сигнал – это именованный объект, который асинхронно возбуждается одним объектом и принимается (перехватывается) другим. (создание и уничтожение объектов – тоже разновидности сигналов.)



Стереотипы сообщений


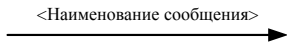
Сообщение	Наименование сообщения	Пояснения
<<call>>	вызвать	Сообщение, требующее вызова операции или процедуры объекта-получателя. Если сообщение с этим стереотипом рефлексивное, то оно инициирует локальный вызов операции у пославшего это сообщение объекта.
<<return>>	возвратить	Сообщение, возвращающее значение выполненной операции или процедуры вызвавшему ее объекту. Значение результата может инициировать ветвление потока управления.
<<create>>	создать	Сообщение, требующее создания другого объекта для выполнения определенных действий. Созданный объект может стать активным (ему передается поток управления), а может остаться пассивным.
<<destroy>>	уничтожить	Сообщение, требующее уничтожения соответствующего объекта. Посылается в случае, когда объект больше не нужен, когда необходимо прекратить действия со стороны существующего в системе объекта.
<<send>>	послать	Обозначает посылку другому объекту сигнала, который асинхронно инициируется одним объектом и принимается другим. Отличие сигнала от сообщения в том, что сигнал должен быть явно описан в том классе, объект которого инициирует его передачу.

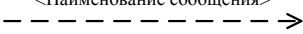

Элементы, связи, ограничения, манипуляции UML

Элементы и связи

Элементы и связи между элементами UML для диаграммы последовательности - таблица 4.2.

Табл. 4.2

Обозначение элемента	Название элемента	Что отражает
Главная последовательность	Текст последовательности действий в прецеденте	Текстовое описание последовательности действий
Альтернативная последовательность		
	Класс или объект	Объект предметной области или класс
	Линия жизни	Период существования объекта, класса
	Фокус управления	Период времени, в течении которого объект выполняет некоторое действие, является активным
	Сообщения	Используется для вызова процедур, выполнения операций и ли обозначения вложенных потоков управления.
		Используется для обозначения простого асинхронного сообщения, которое передается в произвольный момент времени.

<p style="text-align: center;"><Наименование сообщения> </p>		Используется для возврата из вызова процедуры.
 <p style="text-align: center;"><текст></p>	Передача управления	Передача управления другому прецеденту.

Ограничения и манипуляции

На диаграмме последовательности изображаются исключительно те объекты, которые непосредственно участвуют во взаимодействии и не показываются возможные статические ассоциации с другими объектами.

Крайним слева на диаграмме изображается объект, который является инициатором взаимодействия. Правее изображается объект, который непосредственно взаимодействует с первым. Таким образом, все объекты на диаграмме образуют некоторый порядок, определяемый степенью активности этих объектов при взаимодействии друг с другом.

Сообщения, расположенные на диаграмме последовательности выше, иницируются раньше тех, которые расположены ниже.

Построение диаграммы последовательности действий.

Диаграмма последовательности действий формируется для каждого прецедента (варианта использования).

При построении диаграммы последовательности действий формируются два заголовка: *вертикальный*, отражающий последовательность действий в прецеденте и *горизонтальный*, отражающий классы, участвующие в прецеденте.

На поле, сформированном заголовками, отражаются сообщения, которыми классы обмениваются между собой. Сообщение представляет собой законченный фрагмент информации, который отправляется одним объектом другому. Прием сообщения инициирует выполнение определенных действий, направленных на решение отдельной задачи тем объектом, которому это сообщение отправлено.

Из диаграммы последовательности сразу видно время жизни и активности всех объектов. Дополнительно диаграммы последовательности на передачу отдельных сообщений могут иметь временные ограничения.

Диаграмма классов уровня проектирования.

Для того, чтобы построить диаграмму классов уровня проектирования необходимо дополнить диаграмму классов, сформированную на предыдущем этапе, интерфейсными классами и для всех классов определить операции. Операции добавляются в описания классов на основании диаграммы последовательности действий.

Анализ взаимодействия классов программного продукта и построения для них диаграмм состояний (конечного автомата) с помощью языка моделирования UML.

Диаграммы состояний (statechart diagram)

Поскольку одной из характерных особенностей систем различного назначения является взаимодействие элементов системы между собой. Для представления динамических особенностей взаимодействия элементов модели при анализе реализации прецедентов (вариантов использования) предназначены диаграммы кооперации и последовательности. Тем не менее, для многих сложных систем, особенно физических систем реального времени, данных представлений может оказаться недостаточно как для моделирования таких систем в целом, так

и их отдельных подсистем. А вся концепция динамической системы основывается на понятии состояния системы.

Например, любое техническое устройство может характеризоваться двумя состояниями – «исправно» и «неисправно». При этом использовать данное устройство можно только тогда, когда он находится в исправном состоянии, иначе сначала необходимо восстановить его работоспособность, а потом использовать.

Необходимо разобраться со смыслом (семантикой) понятия «состояние». Характеристика состояний системы не зависит или слабо зависит от логической структуры, описанной на диаграмме классов. На данной диаграмме фиксируется статичное состояние системы. Динамический аспект на данной диаграмме не представлен. Поэтому, когда рассматривается система, необходимо описать динамический аспект ее поведения. А это влечет за собой использование специальных понятий.

Для общего представления функциональности системы используют диаграммы прецедентов, которые описывают поведение системы в целом на концептуальном уровне. Для моделирования поведения системы на логическом уровне используют несколько диаграмм: состояний, деятельности, последовательности и кооперации. Причем каждая из них обращает внимание на определенные аспекты функционирования системы.

Диаграмма состояний описывает процесс изменения состояний системы или ее подсистемы при реализации всех прецедентов. При этом изменение состояний отдельных элементов системы может быть вызвано внешними воздействиями со стороны других элементов или извне системы. Для описания реакции системы на такие внешние воздействия используют диаграммы состояний.

Диаграмма состояний предназначена для описания возможных последовательностей состояний и переходов, которые в совокупности характеризуют поведение моделируемой системы в течение всего ее жизненного цикла. Диаграмма состояний представляет динамическое поведение сущностей, на основе спецификации их реакции на восприятие некоторых конкретных событий.

Системы, которые реагируют на внешние действия от пользователей или от других систем, иногда называют реактивными. Если такие действия происходят в произвольные случайные моменты, то говорят об асинхронном поведении модели.

Иногда данные диаграммы состояний используют для описания функциональности экземпляров отдельных классов (объектов), то есть для изменения всех возможных изменений состояний конкретных объектов.

Диаграмма состояний является графом специального вида, который служит для представления некоторого конечного автомата. Понятие «конечный автомат» обладает в контексте языка UML дополнительной семантикой, несмотря на то что основано на общей теории автоматов. Вершинами графа конечного автомата являются состояния и псевдосостояния, изображаемые соответствующими графическими символами. Дуги графа служат для обозначения переходов из состояния в состояние. Диаграммы состояний могут быть вложены друг в друга, образуя вложенные диаграммы для более детального представления состояний отдельных элементов модели. Для понятия семантики конкретной диаграммы состояний необходимо представлять не только особенности поведения моделируемой сущности, но и знать общие сведения из теории конечных автоматов.

Конечные автоматы

Конечный автомат (state machine) в языке UML представляет собой некоторый формализм для моделирования поведения отдельных элементов модели или системы в целом. Конечный автомат описывает поведение отдельного объекта в форме последовательности состояний, которые охватывают все этапы его жизненного цикла, начиная от создания объекта и заканчивая его уничтожением. Каждая диаграмма состояний представляет собой некоторый конечный автомат.

Основные понятия, входящие в формализм конечного автомата, являются *состояние* и *переход*. Основное различие между ними заключается в том, что длительность нахождения

системы в отдельном состоянии существенно превышает время, которое затрачивается на переход из одного состояния в другое. Предполагается, что в пределе время перехода из одного состояния в другое равно 0 (если дополнительно ничего не оговорено), т.е. переход объекта из состояния в состояние происходит мгновенно.

В общем случае конечный автомат представляет динамические аспекты моделируемой системы в виде ориентированного графа, вершины которого соответствуют состояниям, а дуги – переходам. При этом моделирование поведения соответствует перемещению по графу состояний от вершины к вершине, по связывающим их дугам с учетом их ориентации.

Для графа состояний системы можно ввести в рассмотрение специальные свойства.

1. Одним из таких свойств является выделение из всей совокупности двух специальных состояний: начального и конечного. Также необходимо отметить, что предположительно последовательность изменения состояний упорядочена во времени, то есть каждое последующее состояние наступает позже предшествующего ему состоянию (хотя ни в графе состояний, ни на диаграмме состояний время нахождения системы в том или ином состоянии явно не учитывается).

2. Второе свойство – достижимость состояний. То есть навигация или ориентированный путь в графе состояний определяет специальное бинарное отношение на множестве всех состояний системы. Это отношение характеризует потенциальную возможность перехода системы из рассматриваемого состояния в некоторое другое состояние. Очевидно, что для достижимости состояний необходимо наличие связывающего их ориентированного пути в графе состояний.

3. Вложение одних конечных автоматов в другие для уточнения внутренней структуры отдельных более общих состояний (макросостояний). В этом случае вложенные конечные автоматы получили название конечных подавтоматов. Подавтоматы могут использоваться для внутренней спецификации процедур и функций, реализация которых обуславливает поведение моделируемой системы или объекта. (Например, состояние неисправности устройства можно детализировать на отдельные подсостояния, каждое из которых может характеризовать неисправность отдельных подсистем, входящих в состав данного устройства).

Формализм обычного конечного автомата основан на выполнении следующих обязательных условий:

1. Конечный автомат не запоминает историю перемещений из состояния в состояние. То есть КА «забывает» все состояния, которые предшествовали текущему в данный момент. Принципиальное значение имеет факт нахождения моделируемого элемента в том или ином состоянии, а не последовательность состояний, в результате которой КА оказался в текущем состоянии.
2. В каждый момент времени КА может находиться только в одном из своих состояний. То есть формализм предполагает моделирование последовательного поведения, когда моделируемый элемент в течение своего жизненного цикла последовательно проходит через все свои состояния. При этом, КА может находиться в отдельном состоянии как угодно долго, если не происходит никаких событий.
3. Длительность нахождения КА в том или ином состоянии, а также время достижения какого-либо состояния не специфицируются. То есть время на диаграмме присутствует в неявном виде, хотя для отдельных событий может быть указан интервал времени и в явном виде.
4. Количество состояний КА должно быть обязательно конечным и все они должны быть специфицированы явным образом. Это необходимо, так как некоторые псевдосостояния (начальное и конечное состояния) могут не иметь спецификаций. В этом случае их семантика и назначение полностью определяются из контекста модели и рассматриваемой диаграммы состояний.
5. Граф КА не должен содержать изолированных состояний и переходов. Это означает, что для каждого состояния (кроме начального), должно быть определено хотя бы одно предшествующее состояние. Каждый переход обязательно должен соединять два состояния

конечного автомата. Допускается переход из состояния в себя, который называют рефлексивным переходом или «петлей».

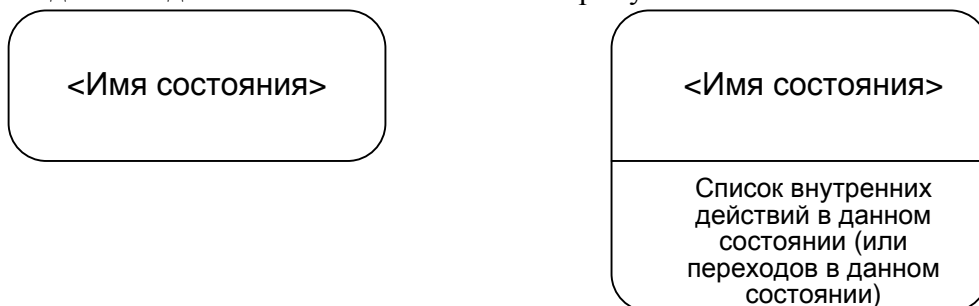
6. КА не должен содержать конфликтующих переходов, то есть таких переходов из одного и того же состояния, когда при наступлении одного и того же события моделируемый элемент одновременно может перейти в два и более последующих состояния (кроме случая параллельных конечных подавтоматов). В языке UML исключение конфликтов возможно на основе введения сторожевых условий.

Диаграммы состояний: базовые понятия

1. Состояние

Это абстрактный метакласс, который используется для моделирования отдельной ситуации, в течении которой имеет место выполнение некоторого условия. Состояние может быть задано в виде набора конкретных значений атрибутов объекта некоторого класса, при этом изменение отдельных значений этих атрибутов будет отражать изменение состояния моделируемого объекта или системы в целом.

Необходимо отметить, что не каждый атрибут класс может характеризовать состояние его объектов. Как правило, имеют значение только такие свойства элементов системы, которые отражают динамический или функциональный аспект ее поведения. В этом случае состояние будет характеризоваться некоторым инвариантным условием, которое включает в себя только значимые для поведения объекта или системы атрибуты классов и их значения.



Под действием понимают некоторую атомарную операцию, выполнение которой приводит к изменению состояния или возврату некоторого значения (например, «истина» или «ложь»).

1.1. Имя состояния

Имя состояния – это строка текста, которая раскрывает содержательный смысл или семантику данного состояния. Имя должно представлять собой законченное предложение и всегда записываться с заглавной буквы. В качестве имени рекомендуют использовать глаголы в настоящем времени или соответствующие причастия. Как исключение, имя может отсутствовать, т.е. оно не является обязательным для некоторых состояний. В этом случае состояние является анонимным и если на диаграмме состояний их несколько, то все они должны различаться между собой.

1.2. Список внутренних действий

Для некоторых состояний требуется указать некоторые действия, которые должны быть выполнены моделируемым элементом при его нахождении в каком-либо состоянии. Для этого служит дополнительная секция в обозначении состояния, которая содержит перечень внутренних действий или деятельность, которые выполняются в процессе нахождения моделируемого элемента в данном состоянии.

Каждое из действий записывается в виде отдельной строки и имеет следующий формат:

<метка действия '/' выражение действия>

Метка действия указывает на обстоятельства или условия, при которых будет выполняться деятельность, определенная выражением действия. При этом выражение действия может использовать любые атрибуты и связи, которые принадлежат области имен или

контексту моделируемого объекта. Если список выражений действия пустой, то метка действия с разделителем в виде наклонной черты '/' не указываются.

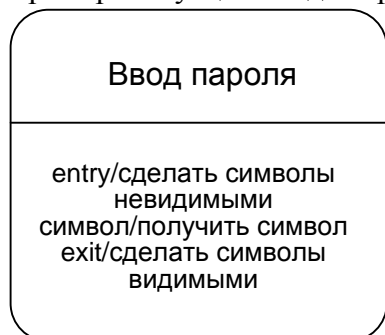
Перечень меток действий фиксирован (причем эти метки не могут быть использованы в качестве имен событий):

- entry – указывает на то, что следующее за ней выражение действия должно быть выполнено в момент входа в данное состояние (входное действие);
- exit – указывает на то, что следующее за ней выражение действия должно быть выполнено в момент выхода из данного состояния (выходное действие);
- do – специфицирует некоторую деятельность (do activity) или так называемую деятельность, которая выполняется в течении всего времени, пока объект находится в данном состоянии, или до тех пор, пока не будет выполнено условие ее окончания, специфицированное в соответствующей операции класса или вычислительной процедуре. В последнем случае при завершении деятельности генерируется соответствующее сообщение;
- include – используется для обращения к конечному подавтомату, при этом следующее за ней выражение действия содержит имя этого подавтомата.

Во всех остальных случаях метка действия идентифицирует событие, которое запускает соответствующее выражение действия. Эти события называются внутренними переходами. Семантически они эквивалентны рефлексивным переходам для данного состояния, за исключением той особенности, что и выход из этого состояния или повторный вход в него не происходит, то есть действия входа и выхода не выполняются.

(Внутренние действия атомарны и не могут быть прерваны никакими внешними событиями, в отличие от внутренней деятельности, выполнение которой длится некоторое время).

Пример: ситуация ввода пароля пользователем при аутентификации входа в систему

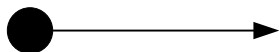


1 и 3 действия стандартные, а действие с меткой символ обеспечивает выполнение операции получения символа с клавиатуры.

1.3. Начальное состояние

Начальное состояние (initial state) представляет собой частный случай состояния, которое не содержит никаких внутренних действий и поэтому относится к категории псевдосостояния. В этом состоянии находится объект по умолчанию в начальный момент. Оно служит для указания на диаграмме состояний графической области, от которой начинается процесс изменения состояний. Графически начальное состояние обозначается в виде закрашенного кружка, из которого может выходить только стрелка-переход.

На самом верхнем уровне представления объекта переход из начального состояния может быть помечен событием создания (инициализации) данного объекта. В противном случае это переход никак не помечается. Если он никак не помечен, то он является первым на диаграмме состояний в следующее за ним состояние. Каждая диаграмма или поддиаграмма состояний должна иметь единственное начальное состояние.

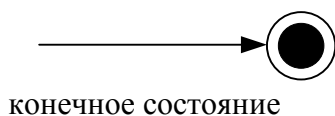


Начальное состояние

1.4. Конечное состояние

Конечное (финальное) состояние (final state) представляет собой частный случай состояния, которое не содержит никаких внутренних действий и поэтому относится к категории псевдосостояния. В этом 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 начальное состояние обозначается в виде закрашенного кружка, помещенного в окружность, в которую может только входить стрелка-переход.

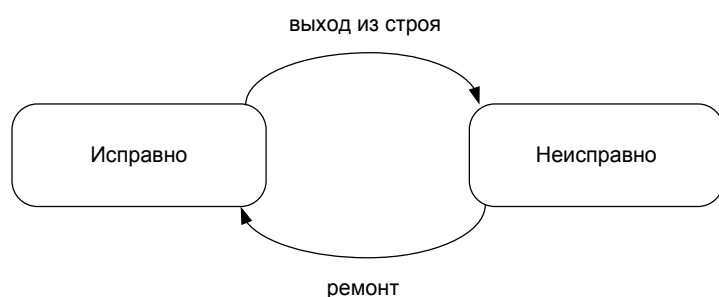
Каждая диаграмма состояний или подсостояний может иметь несколько конечных состояний, при этом все они считаются эквивалентными на одном уровне вложенности состояний.



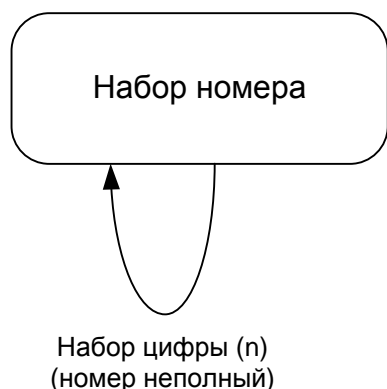
2. Переход

Простой переход – это отношение между двумя последовательными состояниями, которое указывает на факт смены одного состояния другим. Пребывание моделируемого объекта или системы в первом состоянии может сопровождаться выполнением некоторых внутренних действий (деятельности), при этом переход в другое состояние будет возможен либо после завершения этих действий (деятельности), либо при возникновении некоторых событий. В обоих случаях говорят, что переход срабатывает, или происходит срабатывание перехода. До срабатывания перехода объект находится в предыдущем от него состоянии, называемом исходным состоянием, или в источнике (это не начальное состояние), а после срабатывания – в последующем от него состоянии (целевом состоянии).

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, называемого сторожевым условием. Объект перейдет из одного состояния в другое только если произошло указанное событие и сторожевое условие приняло значение «истина».



Переход может быть направлен в то же состояние, из которого он выходит. В этом случае его называют переходом в себя. Исходное и целевое состояния перехода в себя совпадают. Этот переход изображается петлей со стрелкой и отличается от внутреннего перехода. При переходе в себя объект покидает исходное состояние, а затем снова входит в него. При этом всякий раз выполняются внутренние действия, специфицированные метками entry и exit.



На диаграмме состояний переход изображается сплошной линией со стрелкой, которая выходит из исходного состояния и направлена в целевое состояние. Каждый переход может быть помечен строкой текста, которая имеет следующий общий формат:

`<сигнатура события>['<сторожевое условие>']<выражение действия>.`

При этом сигнатура события описывает некоторое событие с необходимыми аргументами:

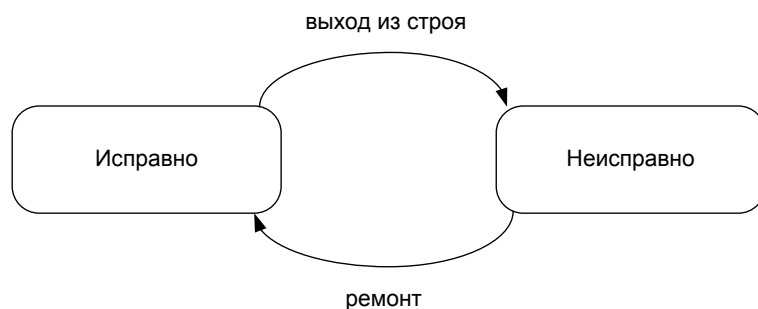
`<имя события>'(<список параметров, разделенных запятыми>')'.`

2.1. Событие

Термин событие (event) является самостоятельным элементом. Формально, событие представляет собой спецификацию некоторого факта, имеющего место в пространстве и во времени. Про события говорят, что они «происходят», при этом отдельные события должны быть упорядочены во времени. После наступления некоторого события нельзя уже вернуться к предыдущим событиям, если такая возможность не предусмотрена явно в модели.

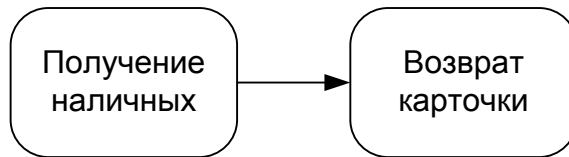
Семантика понятия события фиксирует внимание на внешних проявлениях качественных изменений, происходящих при переходе моделируемого объекта из состояния в состояние. (Например, после успешного ремонта компьютера происходит событие - восстанавливается его работоспособность). События играют роль стимулов, которые инициируют переходы из одних состояний в другие. В качестве событий можно рассматривать сигналы, вызовы, окончание фиксированных промежутков времени или моменты окончания выполнения определенных действий. В зависимости от вида происходящих событий-стимулов различают два типа переходов: триггерные и нетриггерные.

А) переход называется триггерным, если его специфицирует некоторое событие-триггер. В этом случае рядом со стрелкой триггерного перехода обязательно указывается имя события в форме строки текста, начинающейся со строчной буквы. Чаще всего в качестве имен таких переходов задают имена операций, вызываемых у тех или иных объектов системы. После имени такого события следуют круглые скобки для явного задания параметров соответствующей операции. Если таких параметров нет, то список со скобками может отсутствовать.



Здесь переходы триггерные, т.к. с каждым из них связано некоторое событие-триггер, происходящее асинхронно в момент выхода из строя технического устройства или в момент окончания его ремонта.

Б) переход называется нетриггерным, если он происходит по завершении выполнения действия (деятельности) в исходном состоянии. Нетриггерные переходы часто называют переходами по завершении ду-деятельности. Для них рядом со стрелкой перехода не указывается никакого имени события, а в исходном состоянии должна быть описана внутренняя деятельность, по завершении которой произойдет тот или иной нетриггерный переход.

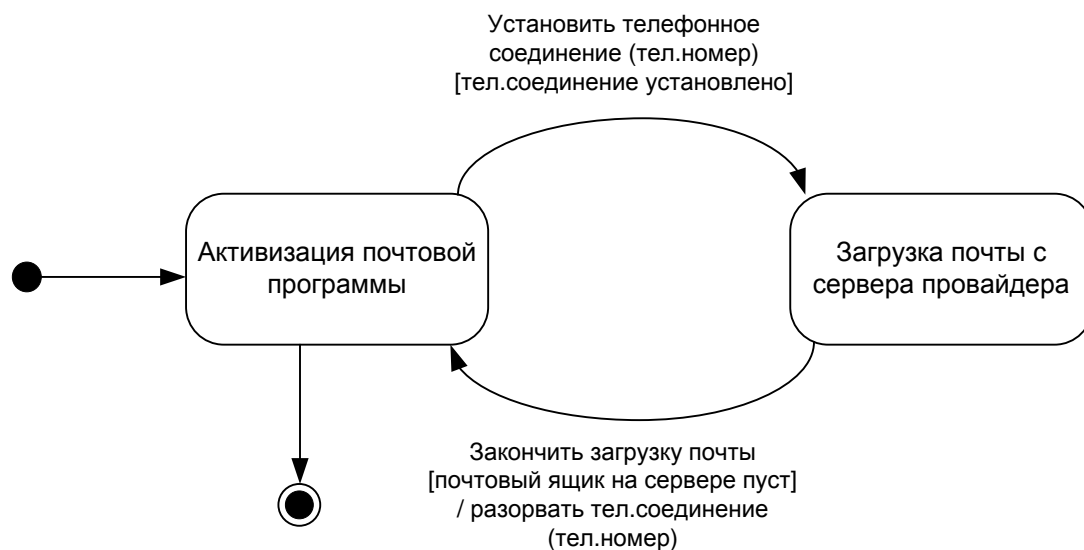


2.2. Сторожевое условие

Сторожевое условие (guard condition), если оно есть – то всегда записывается в прямых скобках и представляет собой некоторое булевское выражение (должно принимать одно из двух значений – «истина» или «ложь»). Из контекста диаграммы состояний должна явно следовать семантика этого выражения, а для записи выражения может использоваться обычный язык, псевдокод или язык программирования.

Дополнение триггерных и нетриггерных переходов сторожевыми условиями позволяет явно определить семантику их срабатывания. Если СУ принимает значение «истина», то соответствующий переход при наступлении события-триггера или завершения деятельности может сработать, в результате чего объект перейдет в целевое состояние. Если СУ принимает значение «ложь», то переход не может сработать, даже если произошло событие-триггер или завершилась деятельность в исходном переходе. В случае невыполнения СУ моделируемый объект или система останется в исходном состоянии. Однако вычисление истинности СУ в модели происходит только после возникновения ассоциированного с ним события-триггера или завершения деятельности, которые инициируют соответствующий переход.

Так как общее количество выходящих из состояния переходов не ограничено, но конечно, то может быть ситуация, когда из одного состояния могут выходить несколько переходов с одним и тем же событием-триггером. Каждый из таких переходов должен содержать свое СУ, но при этом никакие два и более СУ не должны одновременно принимать значение «истина». Иначе будет на диаграмме конфликт триггерных переходов, что будет свидетельствовать о несостоятельности модели системы в целом.



2.3. Выражение действия

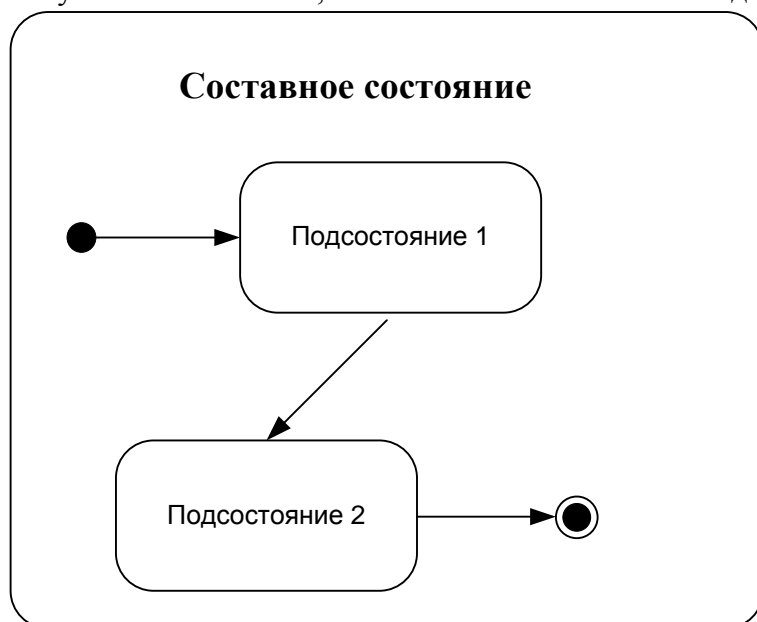
Выражение действия (action expression) выполняется только в том случае, когда переход срабатывает. Это вызов операции или передача некоторого сообщения, имеет атомарный

характер и выполняется после срабатывания соответствующего перехода до начала каких бы то ни было действий в целевом состоянии. Это действие может влиять как на сам объект, так и на его окружение, если это с очевидностью следует из контекста модели. Данное выражение записывается после знака «/» в строке текста, присоединенной к соответствующему переходу.

В общем случае, выражение действия может содержать список отдельных действий, разделенных «;». Обязательное требование – все действия из списка должны четко различаться между собой и следовать в порядке их записи. На синтаксис записи не накладывается никаких ограничений – запись должна быть понятна всем пользователям модели. (Поэтому записывают на каком-либо языке программирования).

3. Составное состояние и подсостояние

Составное состояние (composite state) – такое сложное состояние, которое состоит из других вложенных в него состояний. Последние выступают по отношению к первому как *подсостояния* (substate). Хотя между ними имеет место отношение композиции, графически все вершины диаграммы, которые соответствуют вложенным состояниям, изображаются внутри символа составного состояния. В этом случае размеры графического символа составного состояния увеличиваются так, чтобы вместить в себя все подсостояния.



Составное состояние, которое называют также *суперсостоянием* или *состоянием-композитом*, может содержать или несколько последовательных подсостояний, или несколько параллельных конечных подавтоматов. Каждое суперсостояние может уточняться только одним из указанных способов. При этом любое из подсостояний, в свою очередь, может являться составным состоянием и содержать внутри себя другие вложенные подсостояния. Количество уровней вложенности составных состояний в языке UML не фиксировано.

3.1. Последовательные подсостояния

Последовательные подсостояния (sequential substates) используются для моделирования такого поведения объекта, во время которого в каждый момент объект может находиться в одном и только одном подсостоянии. Поведение объекта в этом случае представляет собой последовательную смену подсостояний, начиная от начального и заканчивая конечным подсостоянием. Хотя моделируемый объект или система продолжает находиться в составном состоянии, введение в рассмотрение последовательных подсостояний позволяет учесть более тонкие логические аспекты его внутреннего поведения.

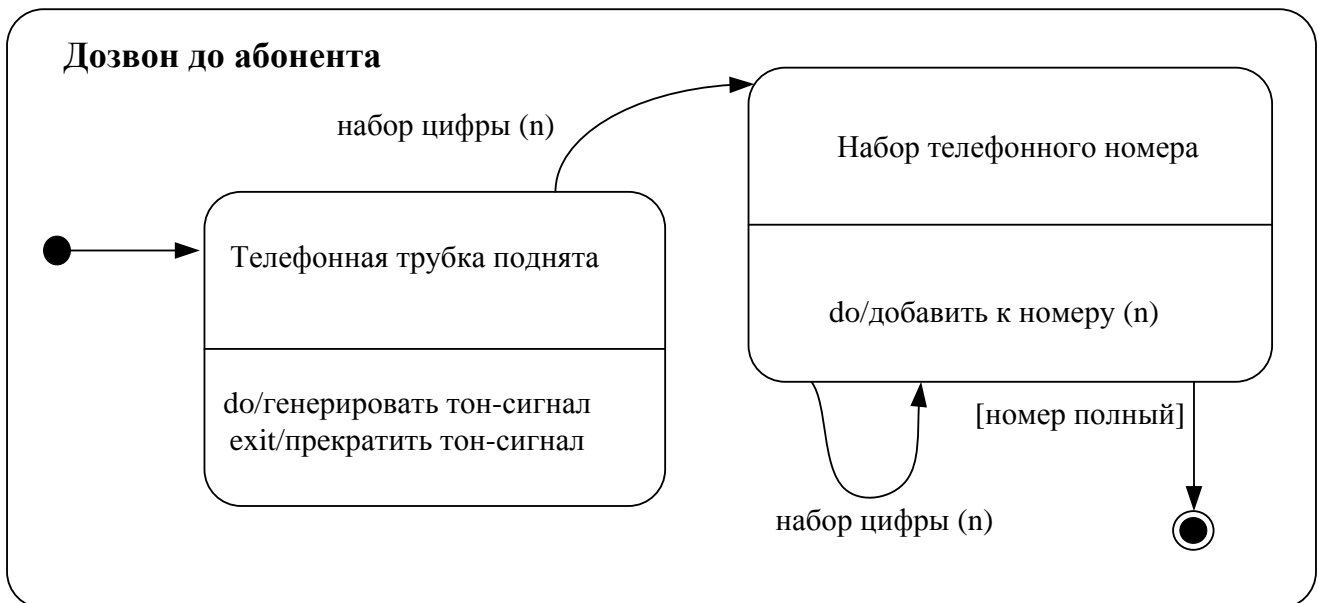
Как пример рассмотрим в качестве моделируемой системы обычный телефонный аппарат. Он может находиться в различных состояниях, одним из которых является состояние дозвона до абонента. Очевидно, для того чтобы позвонить, необходимо снять телефонную

трубку, услышать тоновый сигнал, после чего набрать нужный телефонный номер. Таким образом, состояние дозвона до абонента является составным и состоит из двух последовательных подсостояний: *Поднять телефонную трубку* и *Набрать телефонный номер*. Фрагмент диаграммы состояний для этого примера содержит одно составное состояние и два последовательных подсостояния.

Некоторых пояснений могут потребовать переходы. Два из них специфицируют событие-триггер, которое имеет имя: *набор цифры (n)* с параметром *n*. В качестве параметра, как нетрудно предположить, выступает отдельная цифра на диске телефонного аппарата. Переход из начального подсостояния нетриггерный, поскольку он не содержит никакой строки текста. Последний переход в конечное подсостояние также не имеет события-триггера, но имеет сторожевое условие, проверяющее правильность набранного номера абонента. Только в случае истинности этого условия телефонный аппарат может перейти в конечное подсостояние для суперсостояния *Дозвон до абонента*.

Каждое составное состояние должно содержать в качестве вложенных подсостояний начальное и конечное. При этом начальное подсостояние является исходным, когда происходит переход объекта в данное составное состояние. Если составное состояние содержит внутри себя конечное (финальное) подсостояние, то переход в это вложенное конечное состояние означает завершение нахождения объекта в данном суперсостоянии. Важно помнить, что для последовательных подсостояний начальное и конечное состояния должны быть единственными в каждом составном состоянии.

Это можно объяснить следующим образом. Каждая совокупность вложенных последовательных подсостояний представляет собой конечный подавтомат того конечного автомата, которому принадлежит рассматриваемое составное состояние. Поскольку каждый конечный автомат может иметь по определению единственное начальное и единственное конечное состояния, то для любого его конечного подавтомата это условие также должно выполняться.

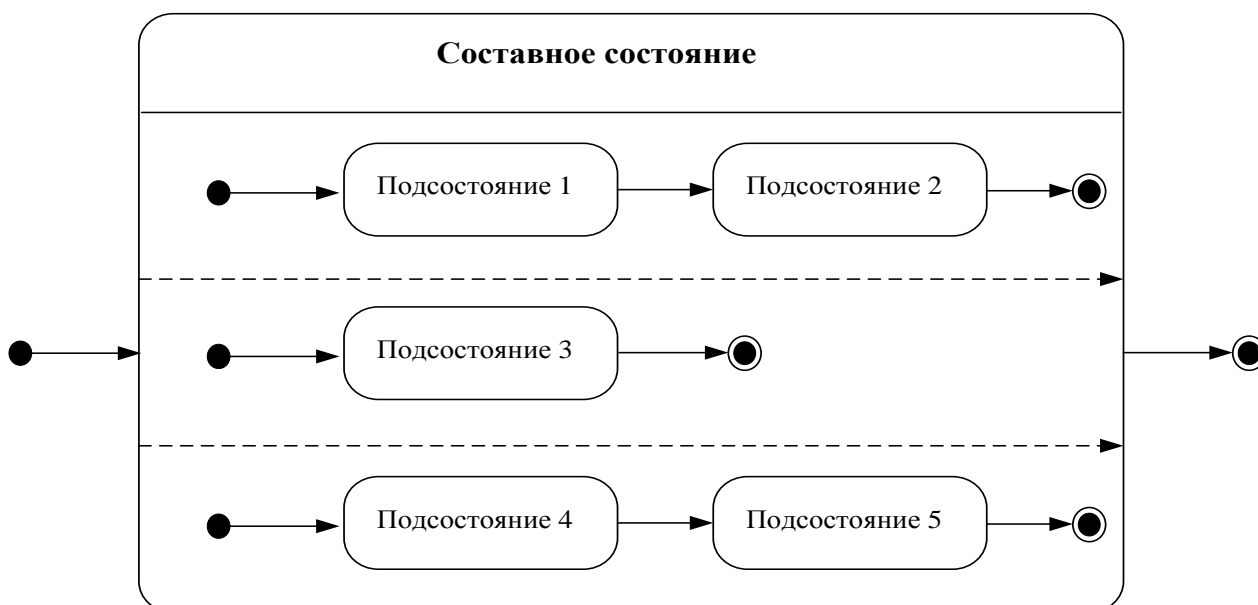


3.2. Параллельные подсостояния

Параллельные подсостояния (concurrent substates) позволяют специфицировать два и более конечных подавтомата, которые могут выполняться параллельно внутри составного события. Каждый из конечных подавтоматов занимает некоторую область (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.

Отдельные параллельные подсостояния могут, в свою очередь, состоять из нескольких последовательных подсостояний (конечные подавтоматы 1 и 2 на рис. 8.8). В этом случае по определению объект может находиться только в одном из последовательных подсостояний конечного подавтомата. Таким образом, для абстрактного примера (рис.8.8) допустимо одновременное нахождение объекта в подсостояниях (1,3,4), (2,3,4), (1,3,5), (2,3,5). Недопустимо нахождение объекта одновременно в подсостояниях (1,2,3) или (3, 4, 5).

Поскольку каждый регион вложенного состояния специфицирует некоторый конечный подавтомат, то для каждого из вложенных конечных подавтоматов могут быть определены собственные начальное и конечное состояния. При переходе в данное составное состояние каждый из конечных подавтоматов оказывается в своем начальном состоянии. Далее происходит параллельное выполнение каждого из этих конечных подавтоматов, причем выход из составного состояния будет возможен лишь в том случае. Когда все конечные подавтоматы будут находиться в своих конечных состояниях.



Если какой-либо из конечных подавтоматов пришел в свое финальное состояние раньше других, то он должен ожидать, пока и другие не придут в свои финальные состояния.

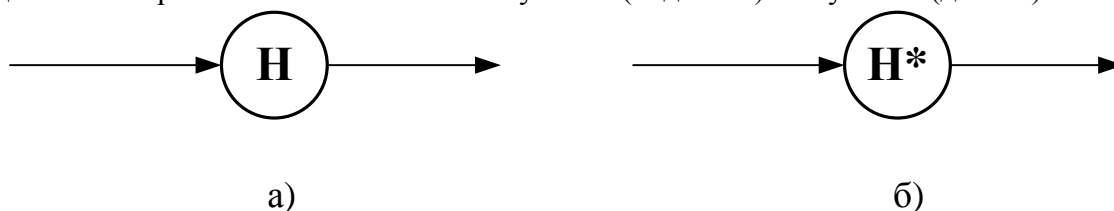
В некоторых случаях бывает желательно скрыть внутреннюю структуру составного состояния. Например, отдельный конечный подавтомат, специфицирующий составное состояние, может быть настолько большим по масштабу, что его визуализация затруднит общее представление диаграммы состояний. В подобной ситуации допускается не раскрывать на исходной диаграмме состояний данное составное состояние, а указать в правом нижнем углу специальный символ-пиктограмму (символ составного состояния). В последующем диаграмма состояний для соответствующего конечного подавтомата может быть изображена отдельно от основной с необходимыми комментариями.



4. Исторические состояния

Как было отмечено ранее, формализм обычного конечного автомата не позволяет учитывать предысторию в процессе моделирования поведения систем и объектов. Однако функционирование целого ряда систем основано на возможности выхода из отдельного суперсостояния с последующим возвращением в это же состояние. При этом может оказаться необходимым учесть ту часть деятельности, которая была выполнена на момент выхода из этого состояния, чтобы не начинать её выполнение сначала. Для этой цели в языке UML существует историческое состояние.

Историческое состояние (history state) применяется только в контексте составного состояния. Оно используется для запоминания того из последовательных подсостояний, которое было текущим в момент выхода из составного состояния. При этом существует две разновидности исторического состояния: неглубокое (недавнее) и глубокое (давнее).



Неглубокое историческое состояние (shallow history state) обозначается в форме небольшой окружности, в которую помещена латинская буква «Н» а). Это состояние обладает следующей семантикой. Во-первых, оно является первым подсостоянием в составном состоянии, и переход извне в рассматриваемое составное состояние должен вести непосредственно в данное историческое состояние. Во-вторых, при первом попадании в неглубокое историческое состояние оно не хранит никакой истории (история пуста). Другими словами, при первом переходе в недавнее историческое состояние оно заменяет собой начальное состояние соответствующего конечного подавтомата.

Далее может следовать последовательное изменение вложенных подсостояний. Если в некоторый момент происходит выход из составного состояния (например, в случае наступления некоторого события), то рассматриваемое историческое состояние запоминает (сохраняет) то из подсостояний, которое являлось текущим на момент выхода из данного составного состояния. При последующем входе в это составное состояние неглубокое историческое подсостояние имеет непустую историю и сразу отправляет конечный подавтомат в сохраненное подсостояние, минуя все предшествующие ему подсостояния.

Историческое состояние теряет свою историю в тот момент, когда конечный подавтомат доходит до своего конечного состояния. При этом неглубокое историческое состояние запоминает историю только того конечного подавтомата, к которому оно относится. Другими словами, этот тип псевдосостояния способен запомнить историю только одного с ним уровня вложенности.

Если сохраненное подсостояние также является составным состоянием, а при выходе из исходного составного состояния необходимо запомнить некоторое подсостояние второго уровня вложенности, то в этом случае следует воспользоваться более сильным псевдосостоянием – глубоким историческим состоянием.

Глубокое историческое состояние (deep history state) также обозначается в форме небольшой окружности, в которую помещена латинская буква «Н» с дополнительным символом «*» (звездочка) (**б**), и служит для запоминания всех подсостояний любого уровня вложенности для исходного составного состояния.

Простым примером, иллюстрирующим применением неглубокого исторического состояния, может служить логика работы почтовой программы-клиента. Предположим, при запуске этой программы мы находимся в состоянии написания нового сообщения, причем набран уже значительный фрагмент текста. Почтовая программа может быть сконфигурирована таким образом, что в фиксированные моменты (например: Каждые 30 минут) она проверяет наличие новых сообщений на сервере провайдера при удаленном доступе. Очевидно, что

очередной звонок, хотя и прерывает работу редактора (пользователя), не должен привести к потере набранного фрагмента текста.

В этом случае составное состояние *Работа редактора* должно содержать вложенное историческое состояние, которое запоминает выполненную работу. После окончания звонка и загрузки новой почты (в случае её наличия) мы должны вернуться к сохранённому фрагменту нашего текста и продолжить работу с редактором.

Рассмотренный ранее фрагмент диаграммы состояний почтовой программы-клиента может быть дополнен с учетом указанного аспекта её поведения. Читателю предлагается это проделать самостоятельно в качестве упражнения.

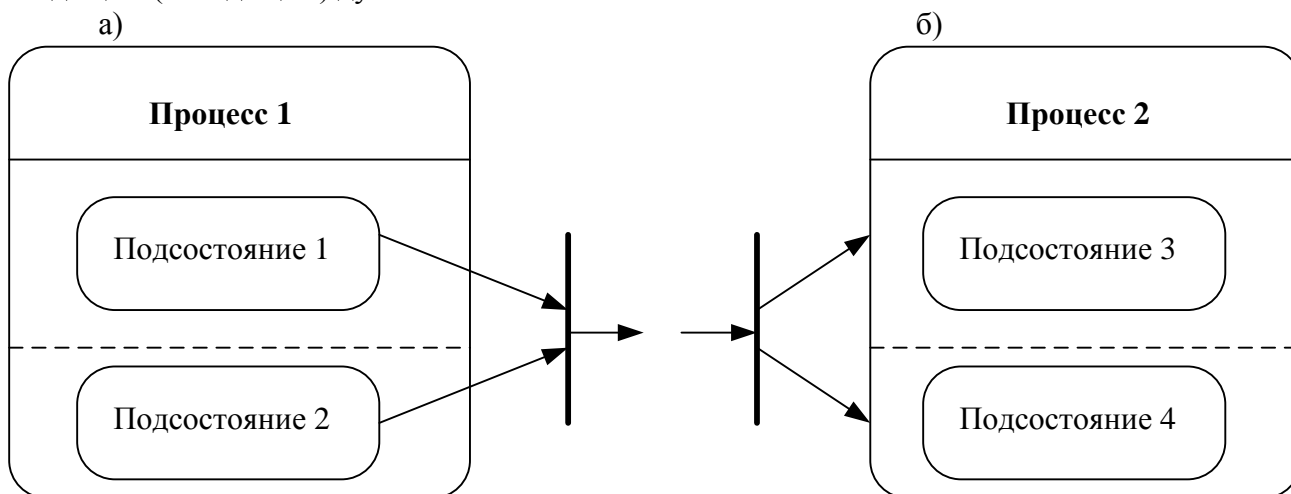
5. Сложные переходы

Рассмотренное выше понятие перехода является вполне достаточным для большинства типичных расчетно-вычислительных задач. Однако современные программные системы могут реализовывать очень сложную логику поведения отдельных своих компонентов. Может оказаться, что для адекватного представления процесса изменения состояний семантика обычного перехода для них недостаточна. С этой целью в языке UML специфицированы дополнительные обозначения и свойства, которыми могут обладать отдельные переходы на диаграмме состояний.

5.1. Переходы между параллельными состояниями

В отдельных случаях возникает необходимость явно показать ситуацию, когда некоторый переход может иметь несколько исходных состояний или несколько целевых состояний. Такой переход получил специальное название *параллельный* переход. Введение в рассмотрение параллельных переходов может быть обусловлено необходимостью синхронизировать и/или разделить отдельные процессы управления на параллельные нити без спецификации дополнительной синхронизации в параллельных конечных автоматах.

Графически такой переход изображается вертикальной черточкой, аналогично обозначению перехода в известном формализме сетей Петри. Если параллельный переход имеет две или более входящие дуги (*а*), то его называют *слиянием* (join). Если же он имеет две или более исходящих из него дуг (*б*), то его называют *разделением* (fork). Текстовая строка спецификации параллельного перехода записывается рядом с черточкой и относится ко всем входящим (исходящим) дугам.



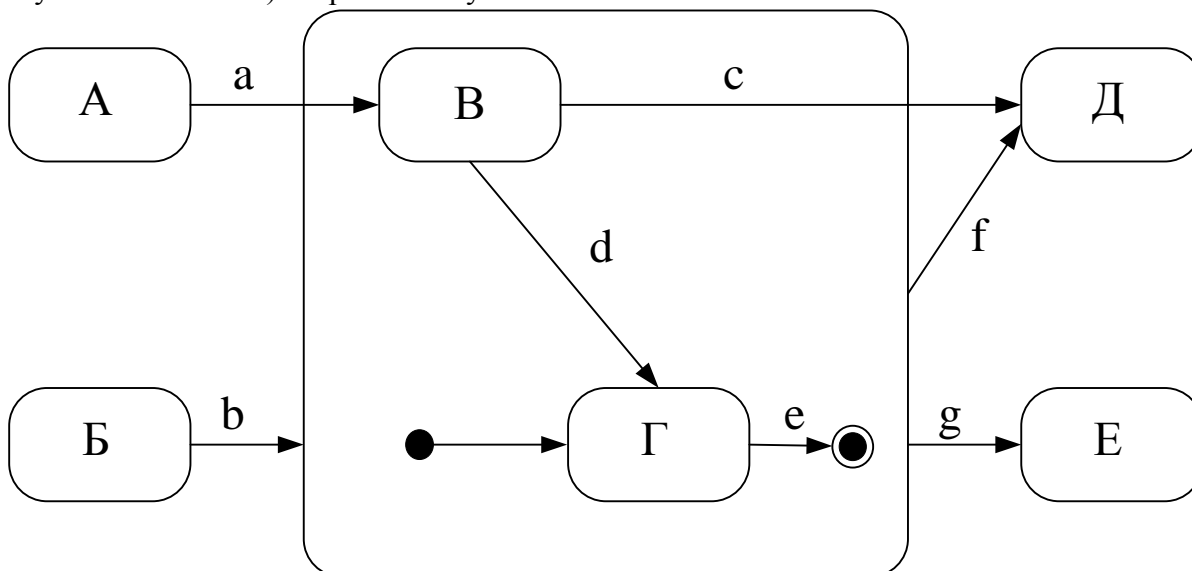
Срабатывание параллельного перехода происходит следующим образом. А первом случае переход-слияние срабатывает, если имеет место событие-триггер для всех исходных состояний этого перехода и выполнено (при его наличии) сторожевое условие. При срабатывании перехода-слияния одновременно покидаются все исходные подсостояния

перехода (подсостояния 1 и 2) и происходит переход в целевое состояние. При этом каждое из исходных подсостояний перехода должно принадлежать отдельному конечному подавтомату, входящему в состав составного конечного автомата (процессу 1).

Во втором случае происходит разделение составного конечного автомата на два конечных подавтомата, образующих параллельные ветви вложенных подпроцессов. При этом после срабатывания перехода-разделения моделируемая система или объект одновременно будет находиться во всех целевых подсостояниях этого параллельного перехода (состояния 3 и 4). Далее процесс изменения состояний будет протекать согласно ранее рассмотренным правилам для составных состояний.

5.2. Переходы между составными состояниями

Переход, стрелка которого соединена с границей некоторого составного состояния, обозначает переход в это составное состояние (переход *б*). Он эквивалентен переходу в начальное состояние каждого из конечных подавтоматов возможно, составного состояния (переходы *f* и *g*), относится к каждому из вложенных подсостояний. Это означает, что система или объект может покинуть данное составное состояние, находясь в любом из его подсостояний. Для этого вполне достаточно наступления триггерного события и выполнения (в случае его наличия) сторожевого условия.



Иногда желательно реализовать ситуацию, когда выход из отдельного вложенного подсостояния соответствовал бы выходу и из составного состояния тоже. В этом случае изображают переход, который непосредственно выходит из вложенного подсостояния и пересекает границу суперсостояния (переход *c*). Аналогично, допускается изображение переходов, входящих извне составного состояния в отдельное вложенное состояние (переход *a*).

5.3. Синхронизирующие состояния

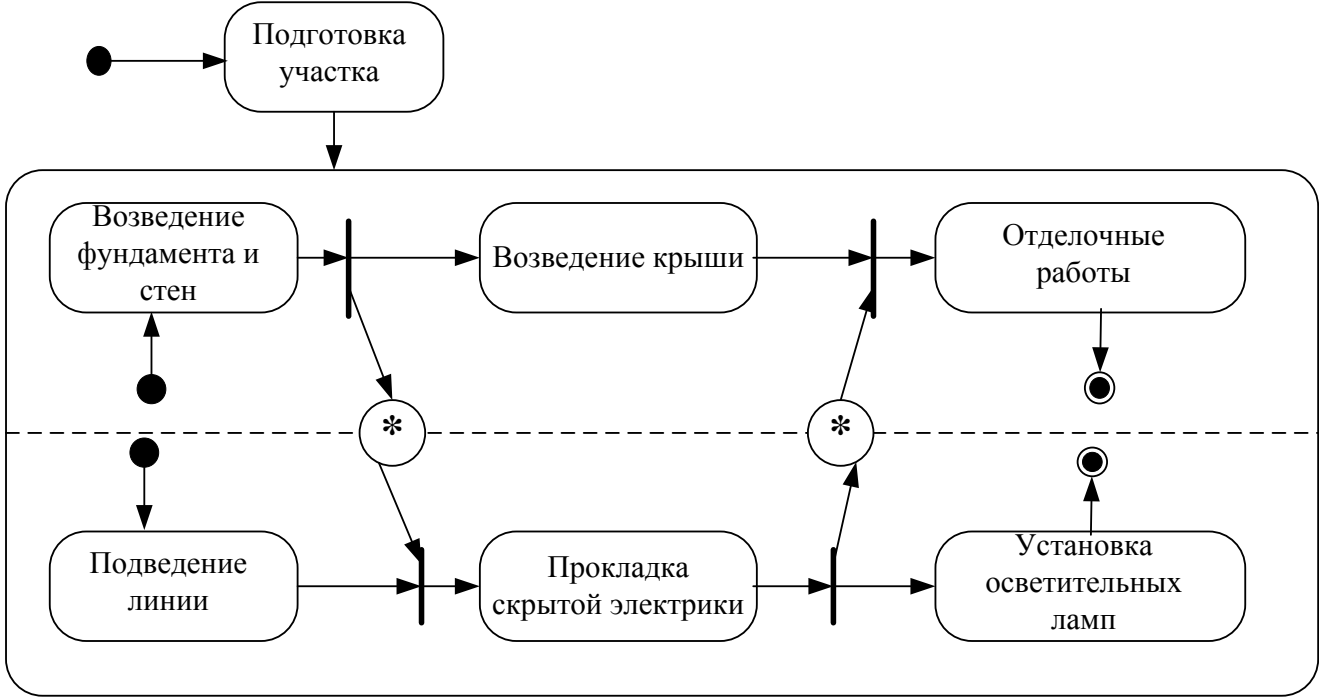
Как уже было отмечено, поведение параллельных конечных подавтоматов происходит независимо друг от друга, что позволяет, например, моделировать многозадачность в программных системах. Однако в отдельных ситуациях может возникнуть необходимость учесть в модели синхронизацию наступления отдельных событий и срабатывание соответствующих переходов. Для этой цели в языке UML имеется специальное псевдосостояние, которое называется синхронизирующим состоянием.

Синхронизирующее состояние (synch state) обозначается небольшой окружностью, внутри которой помещен символ звездочки «*». Оно используется совместно с переходом-

слиянием или переходом-разделением для того, чтобы явно указать событие в других конечных подавтоматах, оказывающие непосредственное влияние на поведение данного подавтомата.

Для иллюстрации использования синхронизирующих состояний рассмотрим упрощенную ситуацию с моделированием процесса постройки дома. Предположим, что постройка дома включает в себя строительные работы (возведение фундамента и стен, возведение крыши и отделочные работы) и работы по электрификации дома (подведение электрической линии, прокладка скрытой электропроводки и установка осветительных ламп). Очевидно, два этих комплекса работ могут выполняться параллельно, однако между ними есть некоторая взаимосвязь.

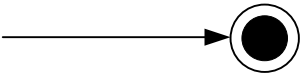



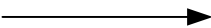
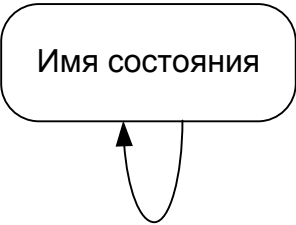
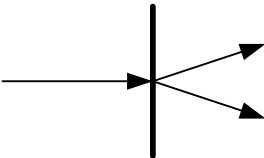
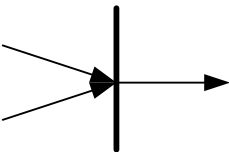
В частности, прокладка скрытой электропроводки может начаться лишь после того, как будет завершено возведение фундамента и стен. Отделочные работы следует начать лишь после того, как будет закончена прокладка скрытой электропроводки. В противном случае отделочные работы придется проводить повторно. Рассмотренные особенности синхронизации этих параллельных процессов учтены на соответствующей диаграмме состояний с помощью двух синхронизирующих состояний.



6. Элементы, связи, ограничения и манипуляции
Элементы, связи представлены в таблице 4.3.

Табл. 4.3

Обозначение элемента	Название элемента	Что отражает
	Состояние (state)	Описание состояний системы или объекта
	Состояние с внутренними действиями и деятельностью	Описание состояний системы или объекта с дополнительными свойствами
	Начальное состояние	Графическое представление начала процесса

	Конечное состояние	Графическое представление окончания процесса
	Неглубокое историческое состояние	Заменяет начальное состояние при переходе для соответствующего подавтомата
	Глубокое историческое состояние	Служит для запоминания всех подсостояний любого уровня вложенности для исходного составного состояния
	Синхронизирующее состояние	Служит для явного указания событий в других конечных подавтоматах, оказывающих непосредственное влияние на поведение данного подавтомата. Используется совместно с переходом –разделением или переходом-слиянием.
	Переход (transition)	Момент перехода системы или объекта из одного состояния в другое
	Рефлексивный переход или переход в себя	Момент перехода системы или объекта в себя
	Разделение (fork)	Отражает разделение на несколько параллельных подсостояний
	Слияние (join)	Отражает слияние параллельных переходов от параллельных подсостояний

Ограничения и правила построения диаграмм

Основные особенности построения диаграмм состояний были рассмотрены при описании соответствующих модельных элементов. Однако некоторые моменты не нашли отражения, о чем необходимо сказать в заключение этой главы.

По своему назначению диаграмма состояний не является обязательным представлением в модели и как бы «присоединяется» к тому элементу, который, по замыслу разработчиков, имеет нетривиальное поведение в течение своего жизненного цикла. Наличие у системы нескольких состояний, отличающихся от простой дихотомии «исправен – неисправен», «активен- неактивен», «ожидание – реакция на внешние действия», уже служит признаком необходимости построения диаграммы состояний. В качестве начального варианта диаграммы состояний, если нет очевидных соображений по поводу состояний объекта, можно воспользоваться подобными состояниями, рассматривая их как составные и уточняя их (детализируя их внутреннюю структуру) по мере рассмотрения логики поведения моделируемой системы или объекта.

При выделении состояний и переходов следует помнить, что длительность срабатывания отдельных переходов должна быть существенно меньше, чем нахождение моделируемых элементов в соответствующих состояниях.

Каждое из состояний должно характеризоваться определенной устойчивостью во времени. Другими словами, из каждого состояния на диаграмме не может быть самопроизвольного перехода в какое бы то ни было другое состояние. Все переходы должны быть явно специфицированы, в противном случае построенная диаграмма состояний является либо неполной (неадекватной), либо ошибочной с точки зрения нотации языка UML (ill formed).

При разработке диаграммы состояний нужно постоянно следить, чтобы объект в каждый момент находился только в единственном состоянии. Если это не так, то данное обстоятельство может быть следствием ошибки или неявным признаком наличия параллельности у поведения моделируемого объекта. В последнем случае следует явно специфицировать необходимое число конечных подавтоматов, вложив их в то составное состояние, которое характеризуется нарушением условия одновременности.

Следует выполнять обязательную проверку того, чтобы никак два перехода из одного состояния не могли сработать одновременно (требование отсутствия конфликтов у переходов). Наличие такого конфликта может служить признаком ошибки или неявной параллельности типа ветвления рассматриваемого процесса на два и более подконечных автомата. Если параллельность по замыслу разработчика отсутствует, то следует ввести дополнительные сторожевые условия либо изменить существующие, чтобы исключить конфликт переходов. При наличии параллельности следует заменить конфликтующие переходы одним параллельным переходом типа ветвления.

Использование исторических состояний оправдано в том случае, когда необходимо организовать обработку исключительных ситуаций (прерываний) без потери данных или выполненной работы. При этом применять исторические состояния, особенно глубокие, необходимо с известной долей осторожности. Нужно помнить, что каждый из конечных подавтоматов может иметь только одно историческое состояние. В противном случае возможны ошибки, особенно, когда подавтоматы изображаются на отдельных диаграммах состояний.

И, наконец, следует отметить, что некоторые дополнительные конструкции конечных автоматов, такие как точки динамического выбора (dynamic choice points) или точки соединения (junction points), в книге не нашли отражения. Это сделано по той причине, что данные модельные элементы хотя и позволяют моделировать более сложные аспекты динамического управления поведением объекта, но не являются базовыми. Соответствующая информация содержится в оригинальной документации по языку UML.

Контрольные вопросы:

1. Какую роль играет платформенно-независимая модель программного продукта в процессе его разработки и почему?
2. На каких базовых положениях основана концепция архитектуры программного продукта, управляемой моделями?
3. Какие виды моделей предусмотрены в языке моделирования UML 2.x, и с чем это связано?
4. UML: диаграмма классов.
5. Какие факторы и как надо учитывать при построении модели классов программного продукта на основе анализа сценариев его использования?
6. UML: диаграмма последовательности действий.
7. Какие факторы и как надо учитывать при моделировании взаимодействия объектов в рамках сценариев использования программного продукта?
8. UML: диаграмма состояний.
9. Какие факторы и как надо учитывать при моделировании поведения субъектов и объектов, участвующих в использовании программного продукта?

Список литературы:

1. Леоненков А.В. «Самоучитель UML 2» - СПб.: БХВ-Петербург, 2007. – 576 с.: ил.
2. Зубкова, Т.М. Технология разработки программного обеспечения: учебное пособие / Т.М. Зубкова; Министерство образования и науки Российской Федерации, Федеральное государственное бюджетное образовательное учреждение высшего образования «Оренбургский государственный университет», Кафедра программного обеспечения вычислительной техники и автоматизированных систем. - Оренбург: ОГУ, 2017. - 469 с.: ил. - Библиогр.: с. 454-459. - ISBN 978-5-7410-1785-2; То же [Электронный ресурс]. - URL: <http://biblioclub.ru/index.php?page=book&id=485553>.
3. Антамошкин, О.А. Программная инженерия. Теория и практика : учебник / О.А. Антамошкин ; Министерство образования и науки Российской Федерации, Сибирский федеральный университет. - Красноярск : Сибирский федеральный университет, 2012. - 247 с. : ил., табл., схем. - Библиогр.: с. 240. - ISBN 978-5-7638-2511-4 ; То же [Электронный ресурс]. - URL: <http://biblioclub.ru/index.php?page=book&id=363975>.
4. Программная инженерия: учебное пособие / сост. Т.В. Киселева ; Министерство образования и науки РФ, Федеральное государственное автономное образовательное учреждение высшего образования «Северо-Кавказский федеральный университет». - Ставрополь : СКФУ, 2017. - Ч. 1. - 137 с. : ил. - Библиогр. в кн. ; То же [Электронный ресурс]. - URL: <http://biblioclub.ru/index.php?page=book&id=467203>.

Дополнительно

1. Техника разработки программ: учебник для студентов вузов: в 2 кн. / Е.В. Крылов, В.А. Острейковский, Н.Г. Типикин. - М.: Высшая школа, 2007 - 2008 - (Для высших учебных заведений. Информатика и вычислительная техника)
2. UML 2.0. Объектно-ориентированное моделирование и разработка. 2-е изд. / Дж. Рамбо, М.Блаха. - СПб.: Питер, 2007. - 544 с.: ил.