

Министерство образования и науки Российской Федерации

---

САНКТ–ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ  
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

---

**Приоритетный национальный проект «Образование»  
Национальный исследовательский университет**

*Д. Ю. ИВАНОВ Ф. А. НОВИКОВ*

# **УНИФИЦИРОВАННЫЙ ЯЗЫК МОДЕЛИРОВАНИЯ UML**

Учебное пособие

*Рекомендовано Учебно-методическим объединением  
по университетскому политехническому образованию в качестве  
учебного пособия для студентов высших учебных заведений  
обучающихся по направлению подготовки  
«Системный анализ и управление»*

Санкт-Петербург  
Издательство Политехнического университета  
2011

УДК  
ББК

Рецензенты:

Заведующий кафедрой «Прикладная математика» физико-механического  
факультета СПбГПУ канд. техн. наук В.Е. Клавдиев

Директор по разработке ООО «ПРОКДИ» Золотарева Н.Н.

*Иванов Д. Ю. Унифицированный язык моделирования UML: Учеб.  
Пособие / Д. Ю. Иванов, Ф. А. Новиков – СПб.: Изд-во Политехн. ун-та, 2011. –  
229 с.*

Системно излагаются основы моделирования на унифицированном языке моделирования UML. Даются рекомендации по моделированию использования, моделированию структуры и моделированию поведения. Пособие снабжено справочником элементов графической нотации UML. Приводятся сравнительные характеристики и тенденции развития всех версий языка UML.

Учебное пособие предназначено для студентов вузов, обучающихся по магистерской программе «Математическое и программное обеспечение компьютерных систем» направления подготовки магистров «Системный анализ и управление». Оно может быть также использовано при обучении в системах повышения квалификации и в учреждениях дополнительного профессионального образования.

Работа выполнена в рамках реализации программы развития национального исследовательского университета «Модернизация и развитие политехнического университета как университета нового типа, интегрирующего мультидисциплинарные научные исследования и надотраслевые технологии мирового уровня с целью повышения конкурентоспособности национальной экономики»

Печатается по решению редакционно-издательского совета  
Санкт-Петербургского государственного политехнического университета.

ISBN

© Иванов Д. Ю., Новиков Ф. А. , 2011  
© Санкт-Петербургский государственный  
политехнический университет, 2011

# ОГЛАВЛЕНИЕ

Введение	4
1. Введение в UML	7
1.1. Что такое UML	7
1.2. Назначение UML	10
1.3. Модель и ее элементы	15
1.4. Общие диаграммы	27
1.5. Специальные диаграммы	37
1.6. Модели и их представления	41
Выводы	47
2. Моделирование использования	49
2.1. Значение моделирования использования	49
2.2. Диаграммы использования	52
2.3. Реализация вариантов использования	65
Выводы	75
3. Моделирование структуры	76
3.1. Объектно-ориентированное моделирование структуры	76
3.2. Сущности на диаграмме классов	85
3.3. Отношения на диаграмме классов	92
3.4. Диаграммы реализации	109
3.5. Моделирование на уровне ролей и экземпляров классификаторов	116
Выводы	127
4. Моделирование поведения	129
4.1. Модели поведения	129
4.2. Диаграммы автомата	130
4.3. Диаграммы деятельности	163
4.4. Диаграммы взаимодействия	189
4.5. Моделирование параллелизма	212
Библиографический список	228

## ВВЕДЕНИЕ

Учебное пособие "Унифицированный язык моделирования UML" содержит подробное описание всех основных версий унифицированного языка моделирования UML и набор рекомендаций по применению языка для моделирования программных систем.

В первой главе дается общий обзор языка "с высоты птичьего полета". Это необходимо, чтобы в последующих главах использовать примеры с некоторым "забеганием вперед". Действительно, моделируя в деталях какой-то один аспект системы, приходится, может быть, на поверхностном уровне, привлекать и другие аспекты, иначе нет надежды получить концептуально целостную модель.

Во второй главе рассматривается самый важный, по нашему мнению, аспект моделирования — моделирование использования, которое обычно является первым этапом построения реальных моделей.

Третья и четвертая главы посвящены моделированию структуры и моделированию поведения, соответственно. Эти достаточно объемные главы содержат большое количество практических примеров.

Язык UML является графическим, но в нем интенсивно используются и тексты, которые располагаются внутри фигур, в виде надписей возле линий и т. д. Тексты имеют определенный формат, или, правильнее сказать, *синтаксис*.

В учебном пособии мы используем следующие упрощенные средства описания синтаксиса. Синтаксис каждого текстового фрагмента описывается отдельно, в виде правила (фразы) в отдельной строке и выделен моноширинным шрифтом. В этой фразе слова означают синтаксические понятия, которые либо считаются очевидными, либо уточняются далее. Те понятия, которые являются обязательными, записаны прописными буквами, а те, которые не

являются обязательными, записаны строчными буквами. Те символы, которые не являются буквами, являются разделителями и должны быть записаны так, как они показаны в правиле. Например, синтаксис понятия "непустой список не более чем из трех элементов в скобках через запятую" описывается следующей фразой:

(ЭЛЕМЕНТ1, элемент2, элемент3)

Определения выделяются рамочкой, *определяемые термины*, как в тексте, так и в определениях выделяются *курсивом*.

**Важные утверждения выделяются жирным шрифтом.**

Огромное значение в учебном пособии имеют *диаграммы*. На диаграммах используются разнообразные графические элементы: фигуры, линии, значки. Различных примитивных графических элементов не так много, и они по большей части не имеют раз и навсегда зафиксированного смысла: их смысл определяется контекстом и комбинацией с другими элементами. В тексте пособия приводятся названия, и описывается смысл всех этих графических элементов моделирования и их комбинаций, но при первом знакомстве читателю может быть трудно сразу догадаться, что вот **эта** линия в **этом** контексте означает такое-то понятие, а похожая линия в другом контексте означает совсем другое. Чтобы помочь читателю, мы помещаем в тексте в скобках указание на определенный элемент диаграммы, например, пишем: (1 на рис. 1.11), а на самой диаграмме специальной нотацией (серый треугольник с номером внутри) указываем, какой конкретно элемент имеется в виду. Если в тексте мы делаем ссылку на элемент, представленный на ближайшей диаграмме, то номер рисунка не указывается. Например, "на рис. В.1 изображено действие (1), при выполнении которого может быть сгенерировано исключение, которое обозначается специальным значком (2) и приводит к переходу (3) в заключительное состояние (4)."

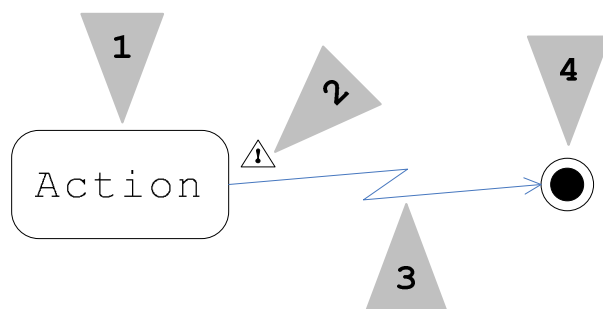


Рис. В.1. Форматы пояснений на диаграммах

Дополнительную информацию, относящуюся к материалу этого пособия, можно найти на сайтах по адресам <http://uml3.ru>, <http://umlmanual.ru> и в книге [1].

# 1. ВВЕДЕНИЕ В UML

## 1.1. ЧТО ТАКОЕ UML

Предметом этого пособия является UML в целом. Прежде чем обсуждать что-либо "в целом", резонно сначала точно определить, о чем идет речь в частности. Обсуждаемый предмет обозначается идентификатором UML, который является аббревиатурой полного названия Unified Modeling Language. Правильный перевод этого названия на русский язык — унифицированный язык моделирования. Таким образом, обсуждаемый предмет характеризуется тремя словами, каждое из которых является точным термином.

### 1.1.1. UML — это язык

Главным словом в этом сочетании является слово "язык".

***Язык** — это знаковая система для хранения и передачи информации.*

UML можно охарактеризовать как формальный искусственный язык, хотя и не в такой степени, как многие распространенные языки программирования. Признаком искусственности служит наличие трех общепризнанных авторов — *Гради Буча, Ивара Якобсона и Джеймса Рамбо* [2].

Для описания формальных искусственных языков (в частности, для описания языков программирования) придумано и используется множество различных способов. Однако на практике сложилась общепринятая структура таких описаний. Считается, что формальный искусственный язык описан должным образом, если это описание содержит, по меньшей мере, следующие части:

- *синтаксис* (syntax), то есть определение правил составления конструкций языка;

- *семантика* (semantics), то есть определение правил приписывания смысла конструкциям языка;

- *прагматика* (pragmatics), то есть определение правил использования конструкций языка для достижения определенных целей.

Как формальный искусственный язык UML имеет синтаксис, семантику и прагматику, хотя эти части названы в некоторых случаях иначе и описаны по другому, нежели это принято в текстовых языках программирования, поскольку, во-первых, UML язык графический, а не текстовый, а во-вторых, UML язык моделирования, а не программирования.

### **1.1.2. UML — это язык моделирования**

Слово "моделирование", входящее в название UML, имеет множество смысловых оттенков и сложившихся способов употребления.

В отношении разработки программного обеспечения так сложилось, что результаты фаз анализа и проектирования, оформленные средствами определенного языка, принято называть *моделью*. Деятельность по составлению моделей естественно назвать *моделированием*. Именно в этом смысле UML является языком моделирования.

Таким образом, модель UML — это, прежде всего, описание объекта или явления, а также и кое-что другое, а именно все, что авторам UML удалось включить в язык, не нарушая принципа унификации, к изложению которого мы переходим в следующем разделе.

### **1.1.3. UML — это унифицированный язык моделирования**

Приложив заслуживающие уважения усилия, авторы UML при поддержке и содействии всей международной программистской общественности смогли свести воедино (унифицировать) большую часть того, что было известно и до них. Если попытаться проследить историю возникновения и развития элементов UML, как на уровне основополагающих идей, так и на уровне технических деталей, то



пришлось бы назвать сотни имен и десятки организаций. Мы не будем этого делать потому, что история развития UML отнюдь не завершена — язык постоянно совершенствуется, обогащается и расширяется. Мы полагаем достаточным привести на рис. 1.1 картинку, иллюстрирующую историю развития UML.

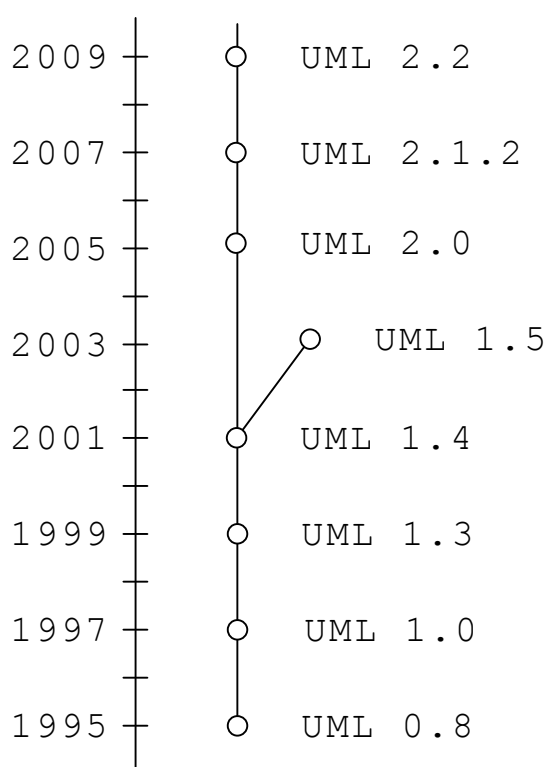


Рис. 1.1. История развития UML

Как видно из рис. 1.1, на особом положении оказалась версия 1.5. Дело в том, что версия 1.5 была выпущена в тот момент, когда "моделирующая общественность" предвкушала появление обещанной версии 2.0. На самом деле версия 1.5 содержит некоторые элементы версии 2.0, в частности, набор элементарных действий, достаточно широкий для того, чтобы применять UML не только как язык моделирования, но и как язык программирования. Но "генеральная линия" развития инструментальных средств прошла мимо этого явления. Все крупные поставщики инструментов предпочли заявить о поддержке версии 2.0 (иногда даже не имея для этого достаточных оснований), и оставили без поддержки версию 1.5.

## 1.2. НАЗНАЧЕНИЕ UML

UML предназначен для моделирования. Сами авторы UML определяют свое детище следующим образом.

***Язык UML** — это графический язык моделирования общего назначения, предназначенный для спецификации, визуализации, проектирования и документирования всех артефактов, создаваемых при разработке программных систем [2].*

### 1.2.1. Спецификация

В типичных случаях в процессе разработки приложений участвуют по меньшей мере два действующих лица: *заказчик* (конкретный человек или группа лиц, или организация) и *разработчик* (это может быть программист-одиночка, временная команда проекта или целая организация, специализирующаяся на разработке программного обеспечения). Из-за того, что действующих лиц двое, очень многое зависит от степени их взаимопонимания.

Одним из ключевых этапов разработки приложения является определение того, каким требованиям должно удовлетворять разрабатываемое приложение. В результате этого этапа появляется формальный или неформальный документ (артефакт), который называют по-разному, имея в виду примерно одно и то же: постановка задачи, требования, техническое задание, внешние спецификации и др.

***Спецификация** — это декларативное описание того, как нечто устроено или работает.*

Необходимо принимать во внимание три толкования спецификаций.

1. То, которое имеет в виду действующее лицо, являющееся источником спецификации (например, заказчик).

2. То, которое имеет в виду действующее лицо, являющееся потребителем спецификации (например, разработчик).

3. То, которое объективно обусловлено природой специфицируемого объекта.

Эти три трактовки спецификаций могут не совпадать, и, к сожалению, как показывает практика, сплошь и рядом не совпадают, причем значительно.

**Основное назначение UML — предоставить, с одной стороны, достаточно формальное, с другой стороны, достаточно удобное, и, с третьей стороны, достаточно универсальное средство, позволяющее до некоторой степени снизить риск расхождений в толковании спецификаций.**

### 1.2.2. Визуализация

Известная поговорка гласит, что лучше один раз увидеть, чем сто раз услышать. Мы добавим: тем паче тысячу раз прочитать пересказ. Особенности человеческого восприятия таковы, что текст с картинками воспринимается легче, чем голый текст. А картинки с текстом — еще легче. Модели UML допускают представление в форме картинок, причем эти картинки наглядны, интуитивно понятны, практически однозначно интерпретируются и легко составляются. Фактически, развитие и детализация этого тезиса составляет большую часть содержания остальной части книги. Мы не будем забегать вперед, и просто приведем пример без всяких объяснений (рис. 1.2). Разве что-нибудь непонятно?

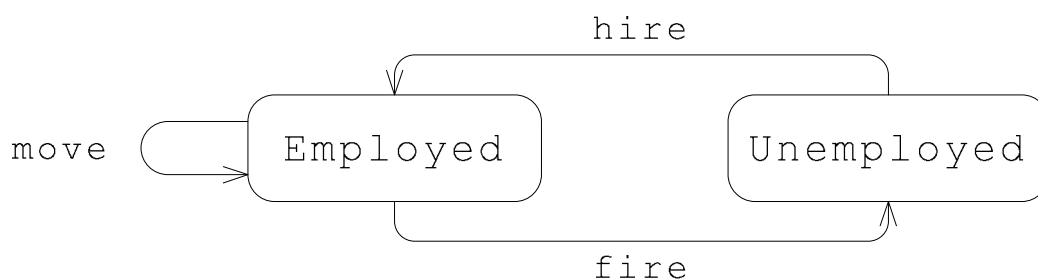


Рис. 1.2. Жизненный цикл работника на предприятии

Таким образом, второе по важности назначение UML состоит в том, чтобы служить адекватным средством коммуникации между людьми. Разумеется, наглядность визуализации моделей UML имеет значение, только если они должны составляться или восприниматься

человеком — это назначение UML не имеет отношения к компьютерам.

### 1.2.3. Проектирование

В оригинале данное назначение UML определено с помощью слова *construct*, которое мы передаем осторожным термином "проектирование". Речь идет о том, что UML предназначен не только для описания абстрактных моделей приложений, но и для непосредственного манипулирования артефактами, входящими в состав этих приложений, в том числе такими, как программный код. Другими словами, одним из назначений UML является, например, создание таких моделей, для которых возможна автоматическая генерация программного кода (или фрагментов кода) соответствующих приложений. Более того, природа моделей UML такова, что возможен и обратный процесс: автоматическое построение модели по коду готового приложения.

**Автоматическое (или автоматизированное) проектирование и конструирование приложений по спецификациям дело трудное, но не безнадежное.** Инструменты, поддерживающие UML, все время совершенствуются, так что в перспективе третье предназначение UML может выйти и на первое место.

### 1.2.4. Документирование

Модели UML являются артефактами, которые можно хранить и использовать как в форме электронных документов, так и в виде твердой копии. В последних версиях UML с целью достижения более полного соответствия этому назначению сделано довольно много. В частности, специфицировано представление моделей UML в форме документов в формате XMI<sup>1</sup>, что обеспечивает практическую интероперабельность при работе с моделями. Другими словами,

---

<sup>1</sup> XMI (XML Metadata Interchange) — внешний формат данных, основанный на языке XML (схема и набор правил использования тэгов), предназначенное для сериализации моделей и обмена ими.

модели UML являются документами, которые можно использовать самыми разными способами, начиная с печати картинок и заканчивая автоматической генерацией человекочитаемых текстовых описаний.

Поясним последнюю фразу предыдущего абзаца. Стандарт требует, чтобы во внутреннем представлении модели для каждого элемента моделирования было отведено место, где можно хранить неформальное текстовое описание этого элемента. Большинство инструментов это требование выполняют: буквально для каждой линии или фигуры на диаграмме можно ввести текст, который поясняет смысл и назначение именно этой линии или фигуры. Более того, многие инструменты умеют из этих текстовых описаний собирать цельные, вполне осмысленные и хорошо отформатированные текстовые документы, которые можно использовать именно как привычные текстовые описания моделируемой системы. К сожалению, это замечательная возможность на практике используется меньше, чем она того заслуживает. Дело в том, что так же, как программисты не любят и ленятся писать осмысленные комментарии к программному коду, так и архитекторы не любят и ленятся писать текстовые пояснения к своим диаграммам.

### **1.2.5. Чем НЕ является UML**

Не следует думать, что UML — это панацея от всех детских болезней программирования. Для ясного понимания назначения и области применения UML полезно сопоставить UML с другими родственными явлениями.

Во-первых, **UML не является языком программирования**. Дело не в том, что UML язык графический, а подавляющее большинство практических языков программирования являются текстовыми языками. Гораздо важнее то, что для моделей UML не определена *операционная семантика*, то есть, не определен способ выполнения моделей на компьютере. Это сделано вполне сознательно, в противном случае UML оказался бы зависимым от

некоторой модели вычислимости, уровень абстрактности его концепций пришлось бы существенно снизить, и он не отвечал бы своему основному назначению: служить средством спецификации приложений и других систем на любом уровне абстракции и в различных предметных областях.

Во-вторых, **UML не является спецификацией инструмента** (хотя инструменты подразумеваются и имеются, например, Rational Rose, Borland Together, Telelogic Rhapsody, Visual Paradigm, Microsoft Visio, Enterprise Architect, StarUML, и др.). Сам язык никоим образом не навязывает то, как его нужно поддерживать инструментальными средствами. Решение всех вопросов, связанных с реализацией UML на компьютере полностью отдано на откуп разработчикам инструментов.

В-третьих, **UML не является моделью процесса разработки приложений** (хотя модель процесса разработки необходима и имеется множество различных моделей, предложенных разными авторами). Конечно, у авторов UML есть собственная модель процесса — Rational Unified Process (RUP), которую они не могли не иметь в голове, разрабатывая язык, но, тем не менее, ими сделано все для того, чтобы устранить прямое влияние RUP на UML и сделать UML пригодным для использования в **любой** модели процесса или даже без оной.

#### **1.2.6. Способы использования UML**

Из сказанного выше видно, что UML предназначен для решения различных задач, соответственно он может быть использован и практически используется по-разному. Далее мы перечисляем различные способы использования UML.

**Рисование картинок.** Графические средства UML можно и нужно использовать безотносительно ко всему остальному. Даже рисование диаграмм карандашом на бумаге позволяет упорядочить мысли и зафиксировать для себя существенную информацию о моделируемом приложении или иной системе.

**Обмен информацией.** Сообщество людей, применяющих и понимающих UML, стремительно растет. Если вы будете использовать UML, то вас будут понимать другие, и вы будете понимать других "с полувзгляда".

**Спецификация систем.** Это важнейший способ использования UML. И хотя не во всех случаях UML оказывается абсолютно адекватным средством спецификации, мы надеемся, что по мере развития языка все меньше будет оставаться таких исключений, где UML неприменим.

**Повторное использование архитектурных решений.** Повторное использование ранее разработанных решений — ключ к повышению эффективности. К сожалению, модели UML пока что повторно используются в весьма ограниченных масштабах.

**Генерация кода.** Генерировать код нужно и можно, но возможности имеющихся инструментов не стоит переоценивать.

**Имитационное моделирование.** Возможности построения моделей UML, из которых путем вычислительных экспериментов можно было бы извлекать информацию о моделируемом объекте, пока что уступают возможностям специализированных систем, сконструированных для этих целей.

**Верификация моделей.** Было бы замечательно, если бы по модели можно было бы делать формальные заключения об ее свойствах: модель непротиворечива, согласована, эффективна и т. п. Кое-что UML позволяет проверить, но, конечно, очень мало. Здесь уместно привести аналогию с традиционными системами программирования: они позволяют быстро и надежно избавиться от синтаксических ошибок, но с логическими ошибками дело обстоит гораздо хуже. Может быть, в будущем...

### 1.3. МОДЕЛЬ И ЕЕ ЭЛЕМЕНТЫ

*Модель UML (UML model) — это совокупность конечного множества конструкций языка, главные из которых — это сущности и отношения между ними.*

Сами сущности и отношения модели являются экземплярами метаклассов метамодели.

### 1.3.1. Сущности

Для удобства обзора сущности в UML можно подразделить на четыре группы:

- структурные;
- поведенческие;
- группирующие;
- аннотационные.

Структурные сущности, как нетрудно догадаться, предназначены для описания структуры. Обычно к структурным сущностям относят следующие.

**Объект** (object) — сущность, обладающая уникальностью и инкапсулирующая в себе состояние и поведение.

**Класс** (class) — описание множества объектов с общими атрибутами, определяющими состояние, и операциями, определяющими поведение.

**Интерфейс** (interface) — именованное множество операций, определяющее набор услуг, которые могут быть запрошены потребителем и предоставлены поставщиком услуг.

**Кооперация** (collaboration) — совокупность объектов, которые взаимодействуют для достижения некоторой цели.

**Действующее лицо** (actor) — сущность, находящаяся вне моделируемой системы и непосредственно взаимодействующая с ней.

**Компонент** (component) — модульная часть системы с четко определенным набором требуемых и предоставляемых интерфейсов.

**Артефакт** (artifact) — элемент информации, который используется или порождается в процессе разработки программного обеспечения. Другими словами, артефакт — это физическая единица реализации, получаемая из элемента модели (например, класса или компонента).



**Узел** (node) — вычислительный ресурс, на котором размещаются и при необходимости выполняются артефакты.

Поведенческие сущности предназначены для описания поведения. Основных поведенческих сущностей всего две: состояние и действие (точнее, две с половиной, потому что иногда употребляется еще и деятельность, которую можно рассматривать как особый случай состояния).

**Состояние** (state) — период в жизненном цикле объекта, находясь в котором объект удовлетворяет некоторому условию и осуществляет собственную *деятельность* или ожидает наступления некоторого события.

**Деятельность** (activity) можно считать частным случаем *состояния*, который характеризуется продолжительными (по времени) не атомарными вычислениями.

**Действие** (action) — примитивное атомарное вычисление.

Это только надводная часть айсберга поведенческих сущностей: состояния бывают самые разные. Кроме того, при моделировании поведения используется еще ряд вспомогательных сущностей, которые здесь не перечислены, потому что сосуществуют только вместе с указанными основными.

Несколько особняком стоит сущность — вариант использования, которой присущи как структурные, так и поведенческие аспекты.

**Вариант использования** (use case) — множество сценариев, объединенных по некоторому критерию и описывающих последовательности производимых системой действий, доставляющих значимый для некоторого *действующего лица* результат.

Приведенная классификация не является исчерпывающей. У каждой из этих сущностей есть различные частные случаи и вариации, рассматриваемые в последующих главах.

Группирующая сущность в UML одна — *пакет* — зато универсальная.

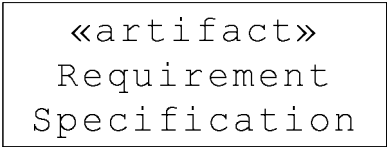
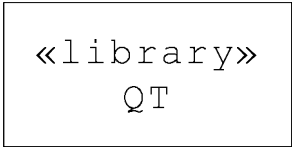
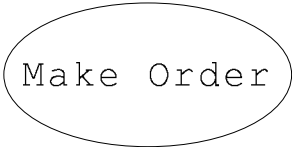




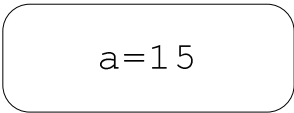



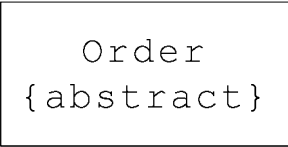


**Пакет** (package) — группа элементов модели (в том числе пакетов).



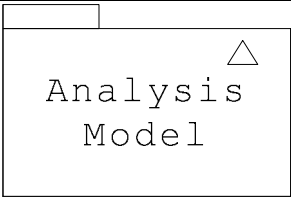

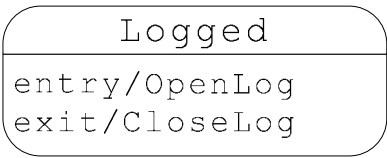

Аннотационная сущность тоже одна — **примечание** (comment) — зато в нее можно поместить все что угодно, так как содержание примечания UML не ограничивает.

В табл. 1.1 приведена стандартная нотация в минимальном варианте для упомянутых сущностей.

Таблица 1.1

**Нотация основных сущностей**

Название	Графическая нотация	
Артефакт		
Вариант использования		
Действующее лицо		
Деятельность и действие		
Интерфейс		
Класс		
Компонент		

Название	Графическая нотация
Кооперация	
Объект	
Пакет	
Примечание	
Состояние	
Узел	

### 1.3.2. Отношения

В UML используются четыре основных типа отношений:

- зависимость (dependency);
- ассоциация (association);
- обобщение (generalization);
- реализация (realization).

*Зависимость* — это наиболее *общий* тип отношения между двумя сущностями.

*Отношение зависимости* указывает на то, что изменение *независимой сущности* каким-то образом *влияет* на *зависимую сущность*.

Графически отношение зависимости изображается в виде пунктирной линии со стрелкой (1), направленной от зависимой сущности (2) к независимой (3) (рис. 1.3). Как правило, семантика

конкретной зависимости уточняется в модели с помощью дополнительной информации. Например, зависимость со стереотипом «use» означает, что зависимая сущность использует (скажем, вызывает операцию) независимую сущность.

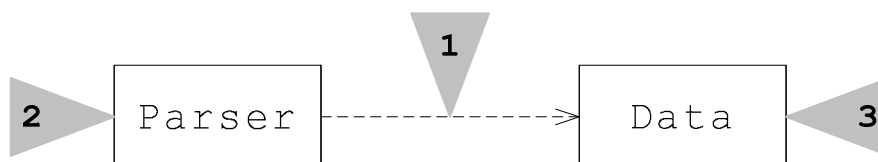


Рис. 1.3. Отношение зависимости

*Ассоциация* — это наиболее часто используемый тип отношения между сущностями.

*Отношение ассоциации имеет место, если одна сущность непосредственно связана с другой (или с другими — ассоциация может быть не только бинарной).*

Графически ассоциация изображается в виде сплошной линии (1) с различными дополнениями, соединяющей связанные сущности (рис. 1.4). На программном уровне непосредственная связь может быть реализована различным образом, главное, что ассоциированные сущности знают друг о друге. Например, отношение часть–целое является частным случаем ассоциации и называется отношением агрегации.

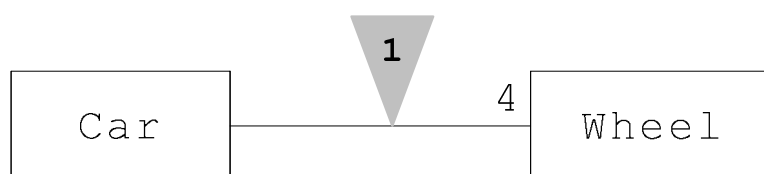


Рис. 1.4. Отношение ассоциации

*Обобщение* — это отношение между двумя сущностями, одна из которых является частным (специализированным) случаем другой.

Графически обобщение изображается в виде линии с треугольной незакрашенной стрелкой на конце (1), направленной от частного (2) (подкласса) к общему (3) (суперклассу) (рис. 1.5).

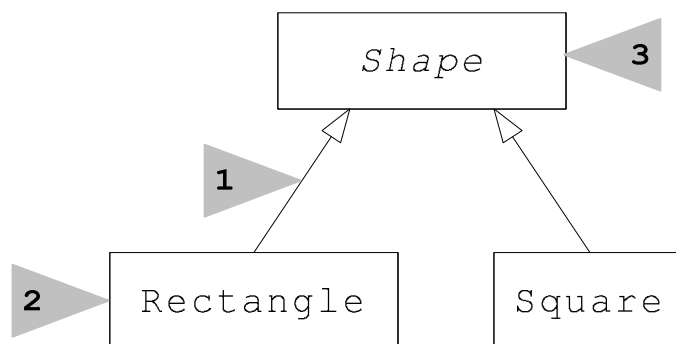


Рис. 1.5. Отношение обобщения

Отношение *реализации* используется несколько реже, чем предыдущие три типа отношений, поскольку часто подразумеваются по умолчанию.

*Отношение реализации указывает, что одна сущность является реализацией другой.*

Например, класс является реализацией интерфейса. Графически реализация изображается в виде пунктирной линии с треугольной незакрашенной стрелкой на конце (1), направленной от реализующей сущности (2) к реализуемой (3) (рис. 1.6).

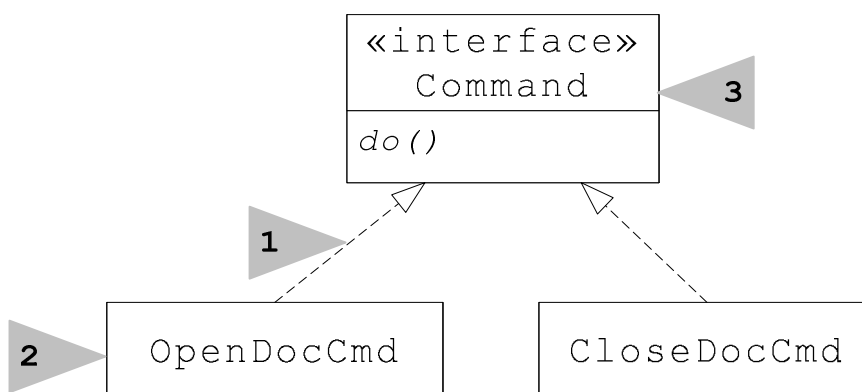


Рис. 1.6. Отношение реализации

Перечисленные типы отношений являются основными, различные их вариации и дополнительные отношения детально рассматриваются в последующих главах.

### 1.3.3. Диаграммы

Диаграммы UML есть та основная накладывается на модель структура, которая облегчает создание и использование модели.

*Диаграмм (diagram) — это графическое представление некоторой части графа модели.*

Вообще говоря, в диаграмму можно было бы включить любые (допустимые) комбинации сущностей и отношений, но произвол в этом вопросе затруднил бы понимание моделей. Поэтому авторы UML определили набор рекомендуемых к использованию типов диаграмм, которые получили название *канонических* типов диаграмм.

Заметим, что помимо сущностей и отношений на диаграмме присутствует другие элементы модели, которые мы также будем называть *конструкциями языка*. Это тексты, которые могут быть написаны внутри фигур сущностей или рядом с линиями отношений, рамки диаграмм и их фрагментов, значки, присоединяемые к линиям или помещаемые внутрь фигур. Эти элементы не только помогают представить модель в более наглядной форме, но подчас несут значительную смысловую нагрузку.

### 1.3.4. Классификация диаграмм

В UML 1 всего определено 9 канонических типов диаграмм. Ниже перечислены их названия, принятые в данном учебном пособии (в других источниках есть отличия).

- Диаграмма использования (Use Case diagram).
- Диаграмма классов (Class diagram).
- Диаграмма объектов (Object diagram).
- Диаграмма состояний (State chart diagram).

- Диаграмма деятельности (Activity diagram).
- Диаграмма последовательности (Sequence diagram).
- Диаграмма кооперации (Collaboration diagram).
- Диаграмма компонентов (Component diagram).
- Диаграмма размещения (Deployment diagram).

Сказанное можно проиллюстрировать условной классификацией диаграмм, приведенной на рис. 1.7.

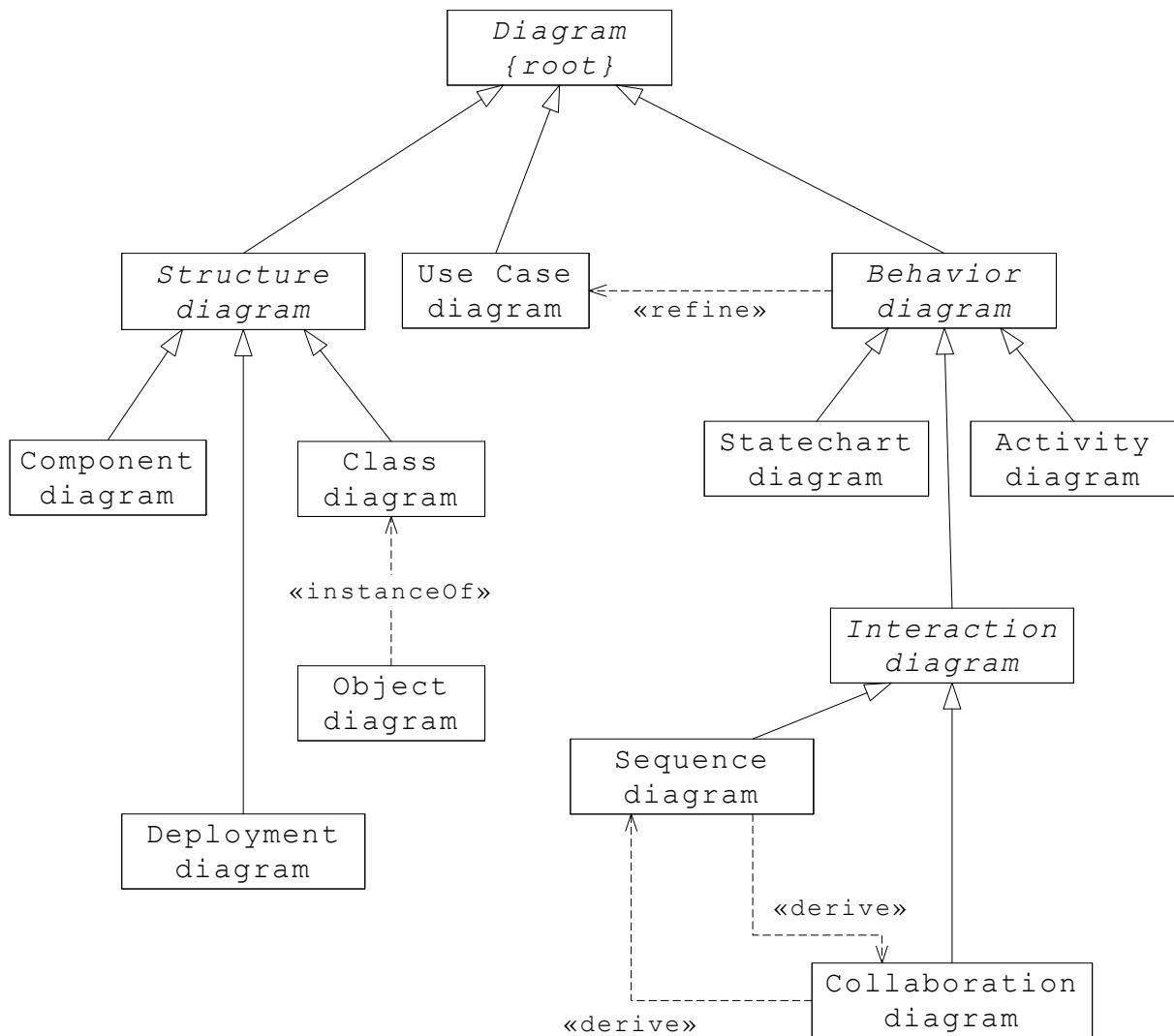


Рис. 1.7. Иерархия типов диаграмм для UML 1

В UML 2 внесены значительные коррективы как в список канонических диаграмм, а именно их число увеличилось до 13, так и в

список доступных конструкций языка, что значительно расширило область его применения. Кроме этого две диаграммы были переименованы: диаграмма кооперации была переименована в диаграмму коммуникации, а диаграмма состояний в диаграмму автомата.

Список новых диаграмм и их названий, принятых в учебном пособии, приведен ниже. На рис. 1.9 и рис. 1.10 приведены соответствующие диаграммы классов.

- Диаграмма внутренней структуры (Composite Structure diagram).
- Диаграмма пакетов (Package diagram).
- Диаграмма автомата (State machine diagram).
- Диаграмма коммуникации (Communication diagram).
- Обзорная диаграмма взаимодействия (Interaction Overview diagram).
- Диаграмма синхронизации (Timing diagram).

Но прежде чем перейти к следующему разделу, сделаем одно небольшое отступление относительно того, как стандарт требует оформлять диаграммы (рис. 1.8).

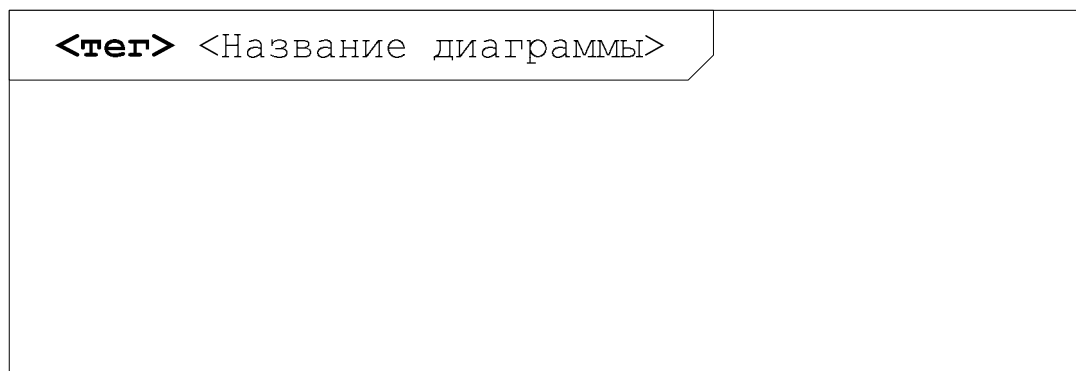


Рис. 1.8. Нотация для диаграмм

Основных элементов оформления два: наружная рамка и ярлычок с названием диаграммы. Если с рамкой все просто – это



прямоугольник, ограничивающий область в котором должны находиться элементы диаграммы, то название диаграммы записывается в специальном формате, приведенном на рис. 1.8.

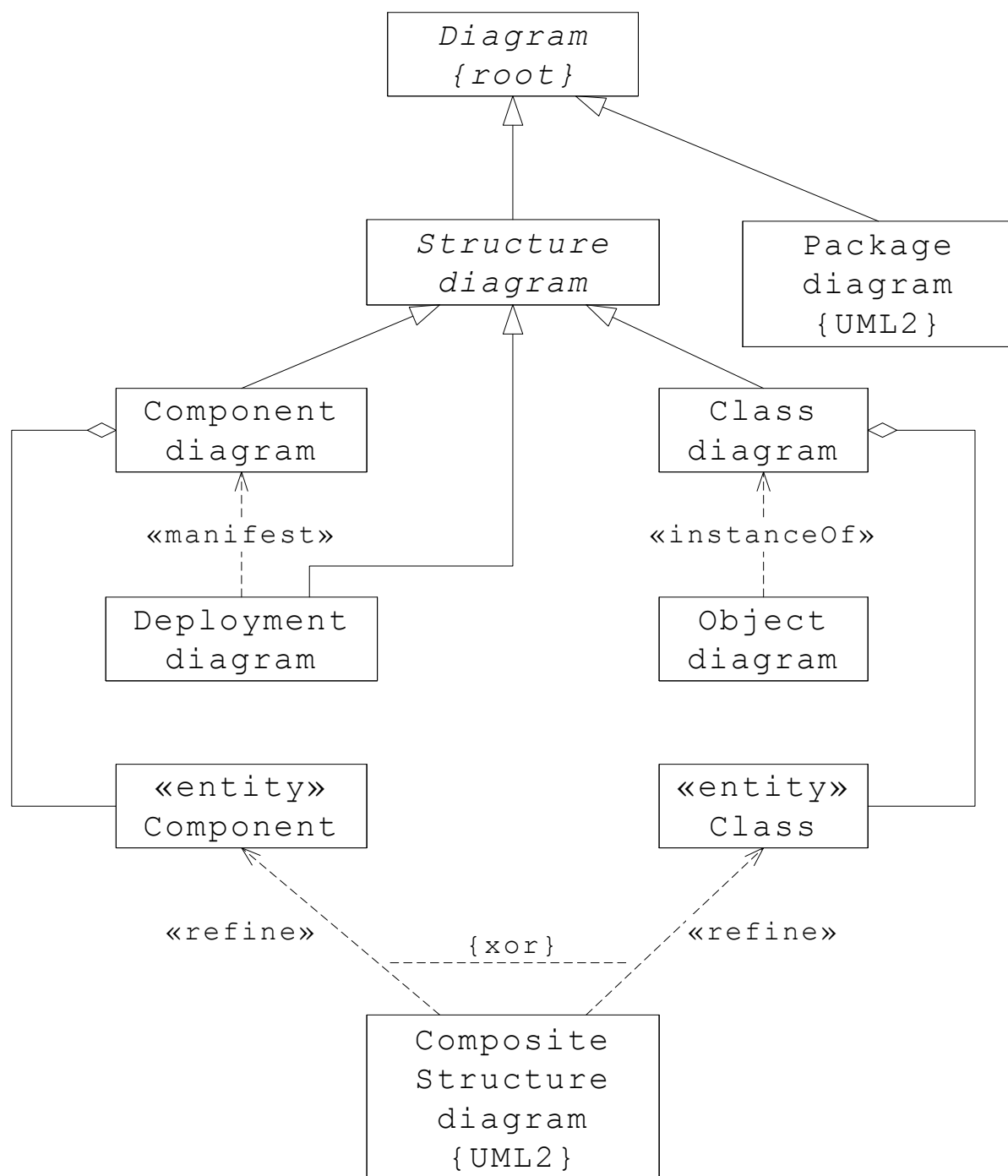


Рис. 1.9. Иерархия типов диаграмм для UML 2 (часть 1)

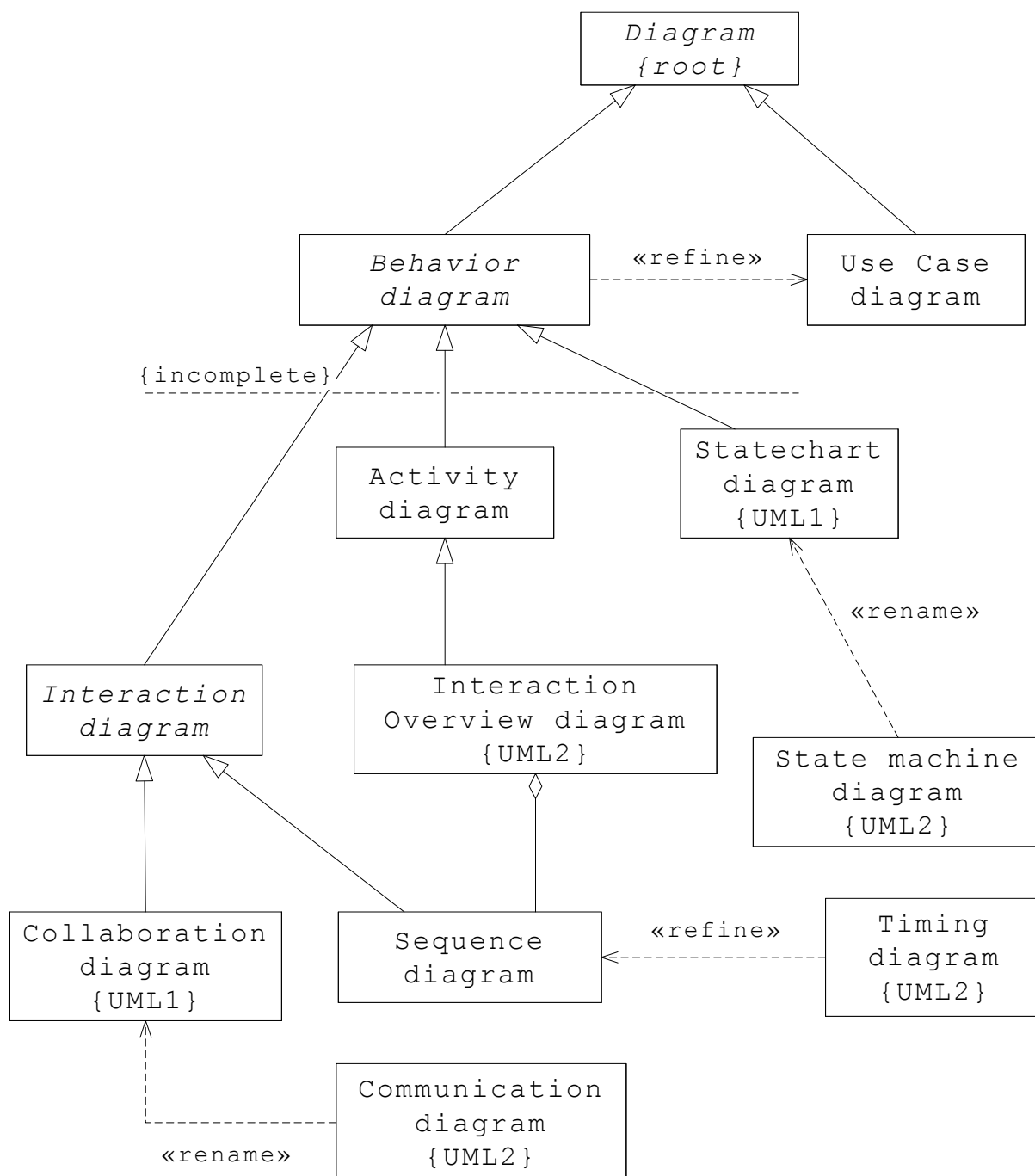


Рис. 1.10. Иерархия типов диаграмм для UML 2 (часть 2)

Возможные теги (типы) для диаграмм приведены в табл. 1.2. Теги, предлагаемые стандартом, записаны во второй столбец. Однако, как показала практика, предлагаемые стандартом правила не всегда удобны и логически обоснованы, поэтому третий столбец таблицы содержит взятую из сложившейся практики альтернативу.

Таблица 1.2

### Типы диаграмм для заголовка

Название диаграммы	Тег (стандартный)	Тег (предлагаемый)
Диаграмма использования	use case или uc	use case
Диаграмма классов	class	class
Диаграмма автомата	state machine или stm	state machine
Диаграмма деятельности	activity или act	activity
Диаграмма последовательности	interaction или sd	sd
Диаграмма коммуникации	interaction или sd	comm
Диаграмма компонентов	component или cmp	component
Диаграмма размещения	не определен	deployment
Диаграмма объектов	не определен	object
Диаграмма внутренней структуры	class	class или component
Обзорная диаграмма взаимодействия	interaction или sd	interaction
Диаграмма синхронизации	interaction или sd	timing
Диаграмма пакетов	package или pkg	package

## 1.4. ОБЩИЕ ДИАГРАММЫ

Все диаграммы UML можно условно разбить на две группы, первая из которых — общие диаграммы. Общие диаграммы практически не зависят от предмета моделирования и могут применяться в любом программном проекте без оглядки на предметную область, область решений и т. д.

### 1.4.1. Диаграмма использования

*Диаграмма использования (use case diagram) — это наиболее общее представление функционального назначения системы.*

Диаграмма использования призвана ответить на главный вопрос моделирования: **что** делает система во внешнем мире?

На диаграмме использования применяются два типа основных сущностей: варианты использования (1) и действующие лица (2), между которыми устанавливаются следующие основные типы отношений:

- ассоциация между действующим лицом и вариантом использования (3);
- обобщение между действующими лицами (4);
- обобщение между вариантами использования (5);
- зависимости между вариантами использования (6).

На диаграмме использования, как и на любой другой, могут присутствовать примечания (7). Основные элементы нотации, применяемые на диаграмме использования, показаны на рис. 1.11.

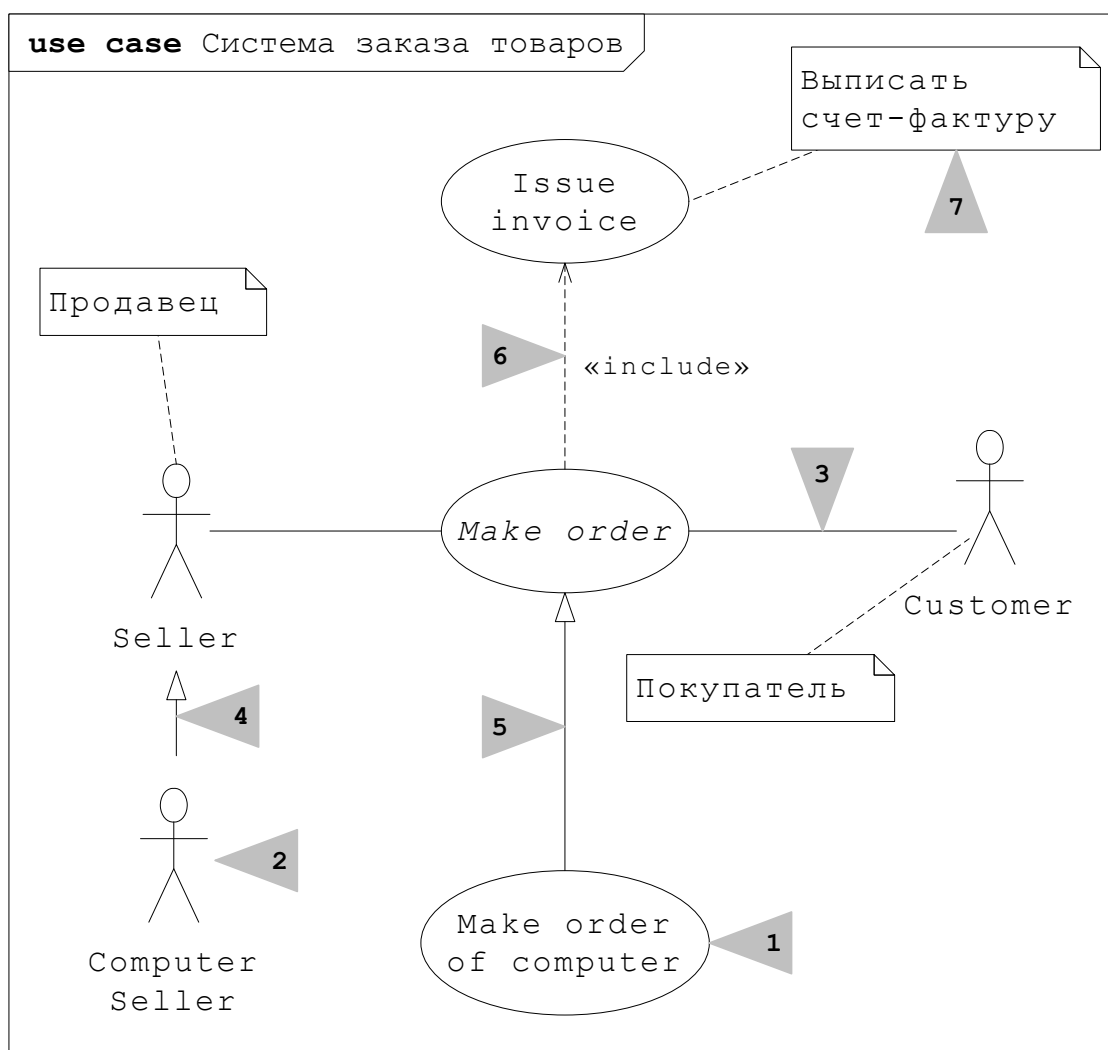


Рис. 1.11. Нотация диаграммы использования

### 1.4.2. Диаграмма классов

*Диаграмма классов (class diagram) — основной способ описания структуры системы.*

Это не удивительно, поскольку UML в первую очередь объектно-ориентированный язык, и классы являются основным (если не единственным) "строительным материалом".

На диаграмме классов применяется один основной тип сущностей: классы (1) (включая многочисленные частные случаи классов: интерфейсы, примитивные типы, классы-ассоциации и многие другие), между которыми устанавливаются следующие основные типы отношений:

- ассоциация между классами (2) (с множеством дополнительных подробностей);
- обобщение между классами (3);
- зависимости (различных типов) между классами (4) и между классами и интерфейсами.

Некоторые элементы нотации, применяемые на диаграмме классов, показаны на рис. 1.12. Детальное описание приведено в главе 3.

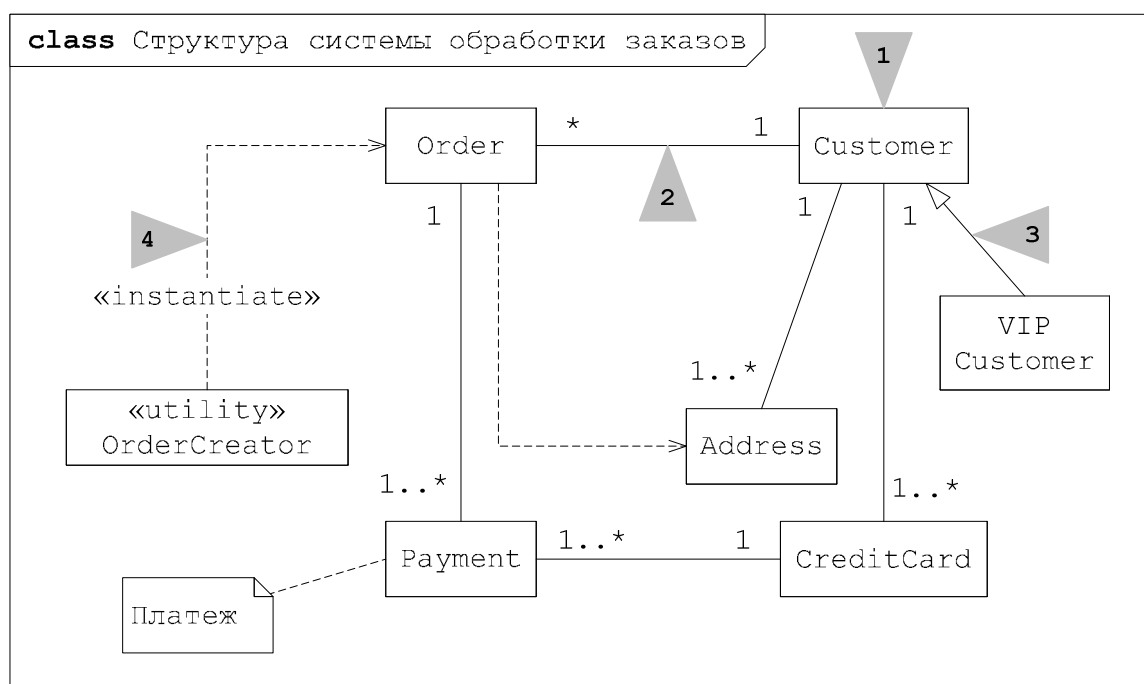


Рис. 1.12. Нотация диаграммы классов

### 1.4.3. Диаграмма автомата

*Диаграмма автомата (state machine diagram) или диаграмма состояний в UML 1 (state chart diagram) — это один из способов детального описания поведения в UML.*

В сущности, диаграммы автомата, как это следует из названия, представляют собой граф состояний и переходов конечного автомата (см. главу 4), нагруженный множеством дополнительных деталей и подробностей.

На диаграмме автомата применяют один основной тип сущностей — состояния (1), и один тип отношений — переходы (2), но и для тех и для других определено множество разновидностей, специальных случаев и дополнительных обозначений. Перечислять их все во вступительном обзоре не имеет смысла.

Детальное описание всех вариаций диаграмм автомата приведено в главе 4, а на рис. 1.13 показаны только основные элементы нотации, применяемые на диаграмме автомата.

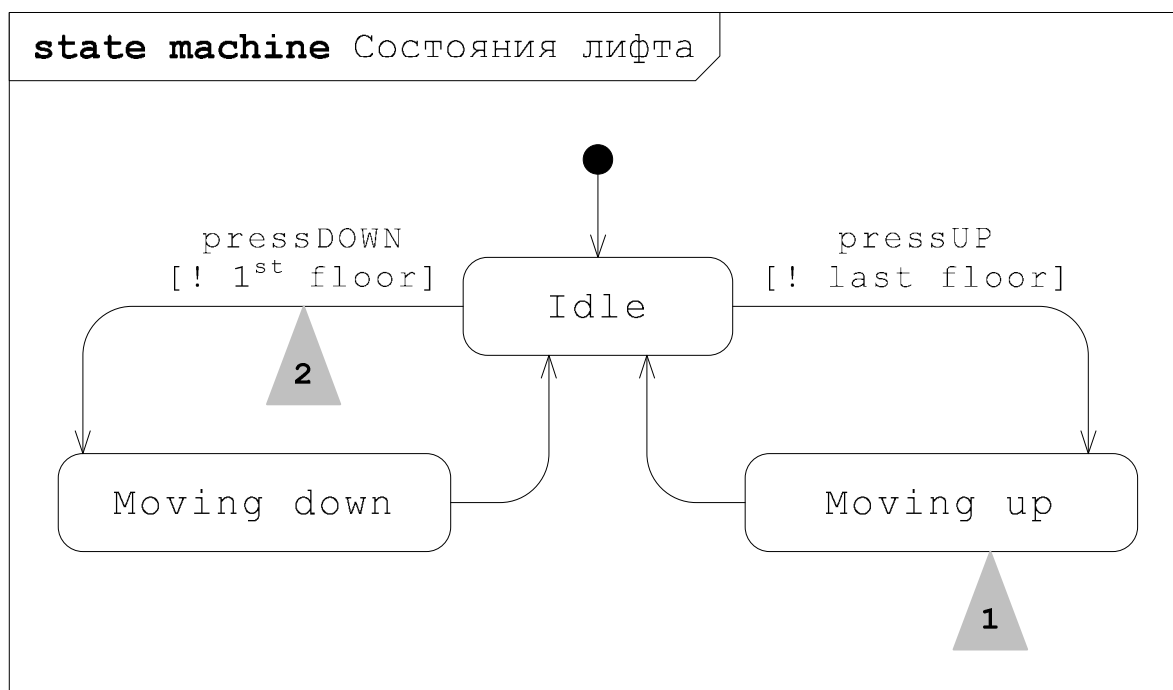


Рис. 1.13. Нотация диаграммы автомата

#### 1.4.4. Диаграмма деятельности

*Диаграмма деятельности (activity diagram) — еще один способ описания поведения, который визуально напоминает старую добрую блок-схему алгоритма.*

Однако за счет модернизированных обозначений, согласованных с объектно-ориентированным подходом, диаграмма деятельности UML является мощным средством для описания поведения системы.

На диаграмме деятельности применяют один основной тип сущностей — действие (1), и один тип отношений — переходы (2) (передачи управления). Также используются такие конструкции как развилки, слияния, соединения, ветвления (3), которые похожи на сущности, но таковыми на самом деле не являются, а представляют собой графический способ изображения некоторых частных случаев гипердуг в гиперграфе (см. врезку "Множества, отношения и графы"). Семантика элементов диаграмм деятельности подробно разобрана в главе 4.

Основные элементы нотации, применяемые на диаграмме деятельности, показаны на рис. 1.14. Детали описания приведены в главе 4.

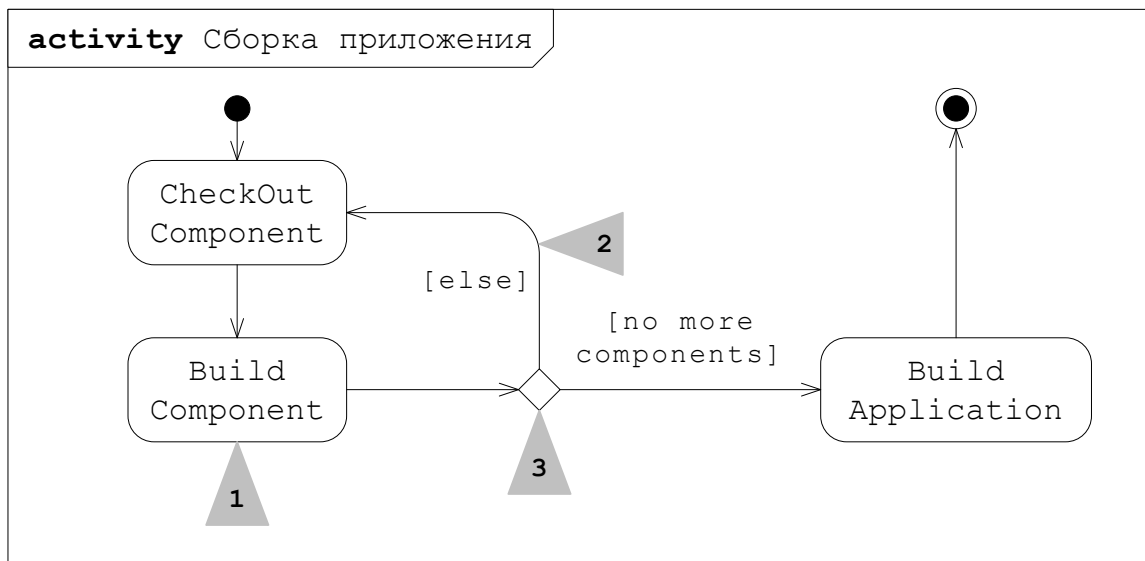


Рис. 1.14. Нотация диаграммы деятельности

### 1.4.5. Диаграмма последовательности

*Диаграмма последовательности (sequence diagram) — это способ описания поведения системы "на примерах".*

Фактически, диаграмма последовательности — это запись протокола конкретного сеанса работы системы (или фрагмента такого протокола). В объектно-ориентированном программировании самым существенным во время выполнения является пересылка сообщений между взаимодействующими объектами. Именно последовательность посылок сообщений отображается на данной диаграмме, отсюда и название.

На диаграмме последовательности применяют один основной тип сущностей — экземпляры взаимодействующих классификаторов (1) (в основном классов, компонентов и действующих лиц), и один тип отношений — связи (2), по которым происходит обмен сообщениями (3). Предусмотрено несколько способов посылки сообщений, которые в графической нотации различаются видом стрелки, соответствующей отношению.

Важным аспектом диаграммы последовательности является явное отображение течения времени. В отличие от других типов диаграмм, кроме разве что диаграмм синхронизации, на диаграмме последовательности имеет значение не только наличие графических связей между элементами, но и взаимное расположение элементов на диаграмме. А именно, считается, что имеется (невидимая) ось времени, по умолчанию направленная сверху вниз, и то сообщение, которое отправлено позже, нарисовано ниже.

На рис. 1.15 показаны основные элементы нотации, применяемые на диаграмме последовательности. Для обозначения самих взаимодействующих объектов применяется стандартная нотация — прямоугольник с именем экземпляра классификатора. Пунктирная линия, выходящая из него, называется *линией жизни* (lifeline) (4). Это не обозначение отношения в модели, а графический комментарий, призванный направить взгляд читателя диаграммы в



правильном направлении. Фигуры в виде узких полосок, наложенных на линию жизни, также не являются изображениями моделируемых сущностей. Это графический комментарий, показывающий отрезки времени, в течении которых объект *владеет потоком управления* (execution occurrence) (5) или другими словами имеет место *активация*<sup>2</sup> (activation) объекта. *Составные шаги взаимодействия* (combined fragment) (6) позволяют на диаграмме последовательности, отражать и алгоритмические аспекты протокола взаимодействия. Прочие детали нотации диаграммы последовательностей см. в главе 4.

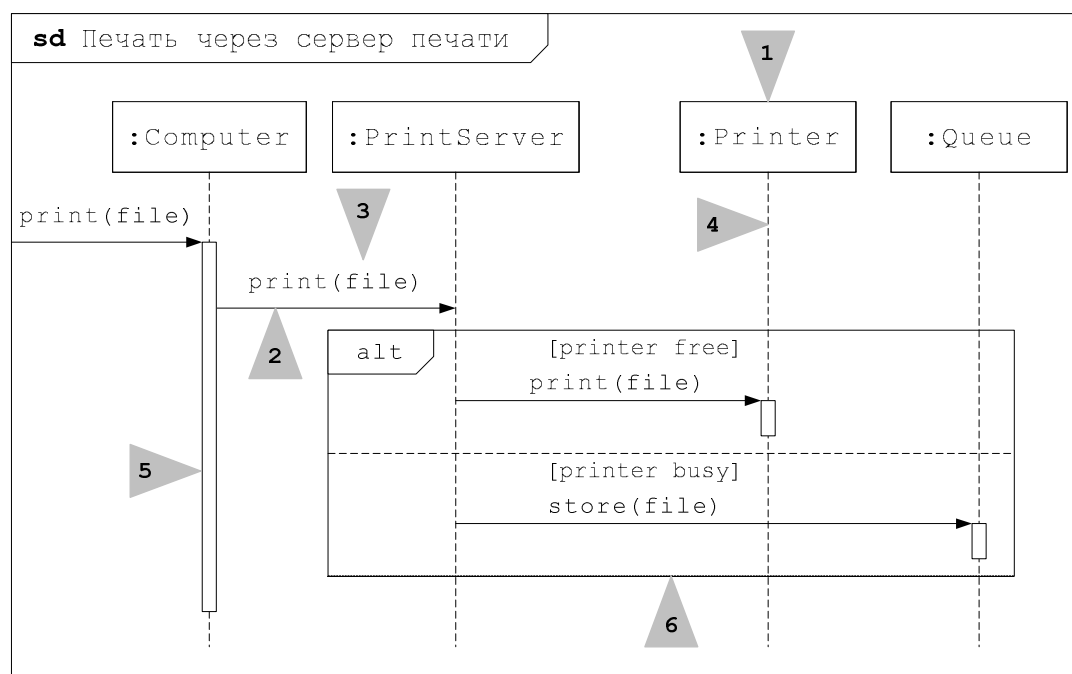


Рис. 1.15. Нотация диаграммы последовательности

#### 1.4.6. Диаграмма коммуникации

**Диаграмма коммуникации** (*communication diagram*) — способ описания поведения, семантически эквивалентный диаграмме последовательности.

<sup>2</sup> Термин "активация" использовался в UML 1 и на данный момент считается устаревшим.

Фактически, это такое же описание последовательности обмена сообщениями взаимодействующих экземпляров классификаторов, только выраженное другими графическими средствами. Таким образом, на диаграмме коммуникации также как и на диаграмме последовательности применяют один основной тип сущностей — экземпляры взаимодействующих классификаторов (1) и один тип отношений — связи (2). Однако здесь акцент делается не на времени, а на структуре связей между конкретными экземплярами. На рис. 1.16 показаны основные элементы нотации, применяемые на диаграмме коммуникации. Для обозначения самих взаимодействующих объектов применяется стандартная нотация — прямоугольник с именем экземпляра классификатора. Взаимное положение элементов на диаграмме кооперации не имеет значения — важны только связи (чаще всего экземпляры ассоциаций), вдоль которых передаются сообщения (3). Для отображения упорядоченности сообщений во времени применяется иерархическая десятичная нумерация. Сравните рис. 1.15 и рис. 1.16 (на них изображено одно и то же поведение), и вам все станет понятно. Прочие детали нотации диаграммы коммуникации см. в главе 4.

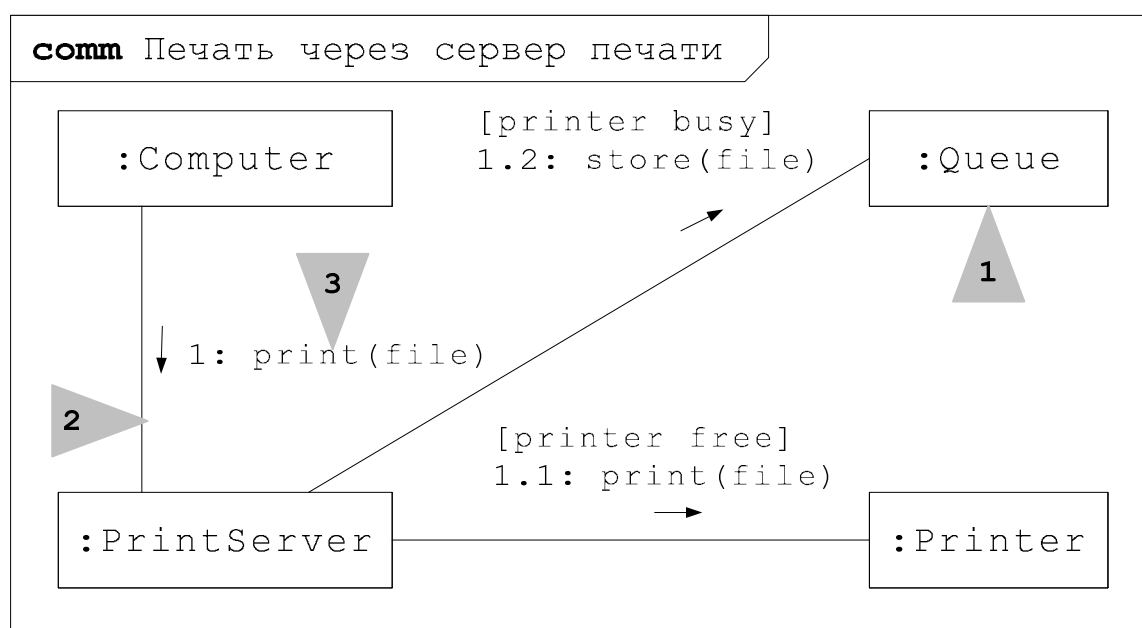


Рис. 1.16. Нотация диаграммы коммуникации

### 1.4.7. Диаграмма компонентов

*Диаграмма компонентов (component diagram) — показывает взаимосвязи между модулями (логическими или физическими), из которых состоит моделируемая система.*

Основной тип сущностей на диаграмме компонентов — это сами компоненты (1), а также интерфейсы (2), посредством которых указывается взаимосвязь между компонентами. На диаграмме компонентов применяются следующие отношения:

- реализации между компонентами и интерфейсами (компонент реализует интерфейс);
- зависимости между компонентами и интерфейсами (компонент использует интерфейс) (3).

На рис. 1.17 показаны основные элементы нотации, применяемые на диаграмме компонентов. Детальное описание приведено в главе 3.

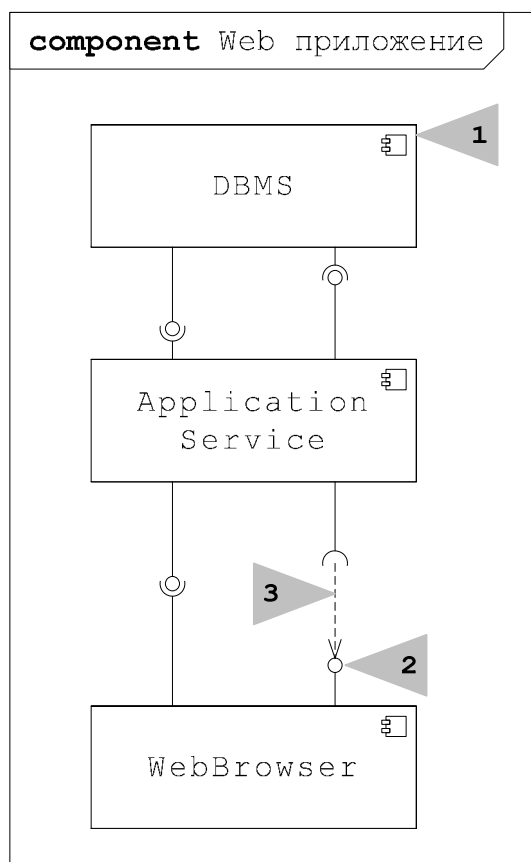


Рис. 1.17. Нотация диаграммы компонентов

#### 1.4.8. Диаграмма размещения

*Диаграмма размещения (deployment diagram) наряду с отображением состава и связей элементов системы показывает, как они физически размещены на вычислительных ресурсах во время выполнения.*

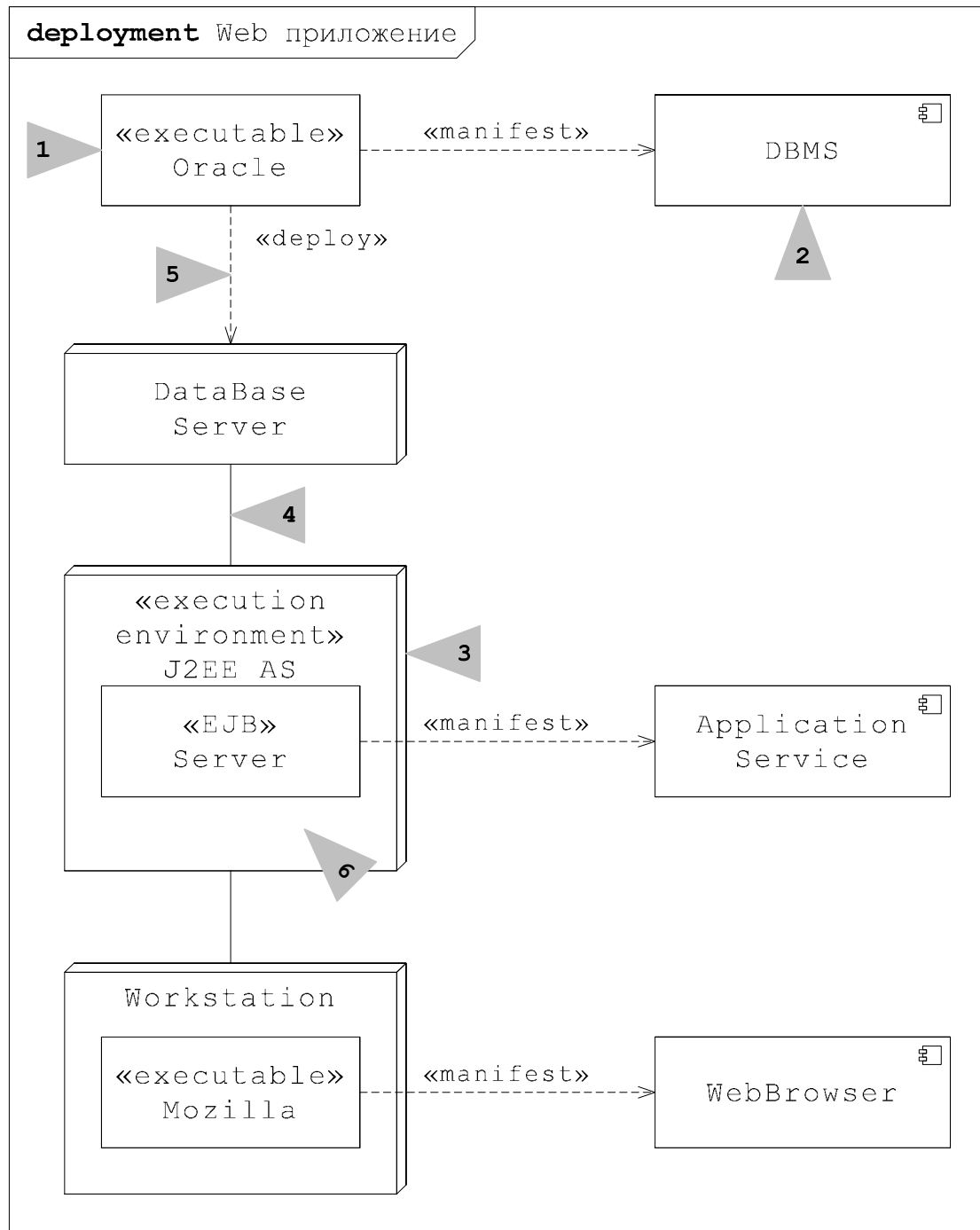


Рис. 1.18. Нотация диаграммы размещения

Таким образом, на диаграмме размещения, по сравнению с диаграммой компонентов, добавляется два типа сущностей: *артефакт* (1), который является реализацией компонента (2) и *узел* (3) (может быть как классификатор, описывающий тип узла, так и конкретный экземпляр), а также отношение ассоциации между узлами (4), показывающее, что узлы физически связаны во время выполнения.

На рис. 1.18 показаны основные элементы нотации, применяемые на диаграмме размещения. Для того чтобы показать, что одна сущность является частью другой, применяется либо отношение зависимости «*deploy*» (5), либо фигура одной сущности помещается внутрь фигуры другой сущности (6). Детальное описание диаграммы приведено в главе 3.

## 1.5. СПЕЦИАЛЬНЫЕ ДИАГРАММЫ

Специальные диаграммы характеризуются тем, что чаще всего служат для дополнения какой-либо общей диаграммы, например, являются ее частным случаем или же просто играют вспомогательную роль, уточняя некоторые детали.

### 1.5.1. Диаграмма объектов

*Диаграмма объектов (object diagram)* — является экземпляром диаграммы классов.

На диаграмме объектов применяют один основной тип сущностей: *объекты* (1) (экземпляры классов), между которыми указываются конкретные *связи* (2) (чаще всего экземпляры ассоциаций).

Диаграммы объектов имеют вспомогательный характер — по сути это примеры (можно сказать, дампы памяти), показывающие, какие имеются объекты и связи между ними в некоторый конкретный момент функционирования системы.

Основные элементы нотации, применяемые на диаграмме объектов, показаны на рис. 1.19. Детальное описание приведено в главе 3.

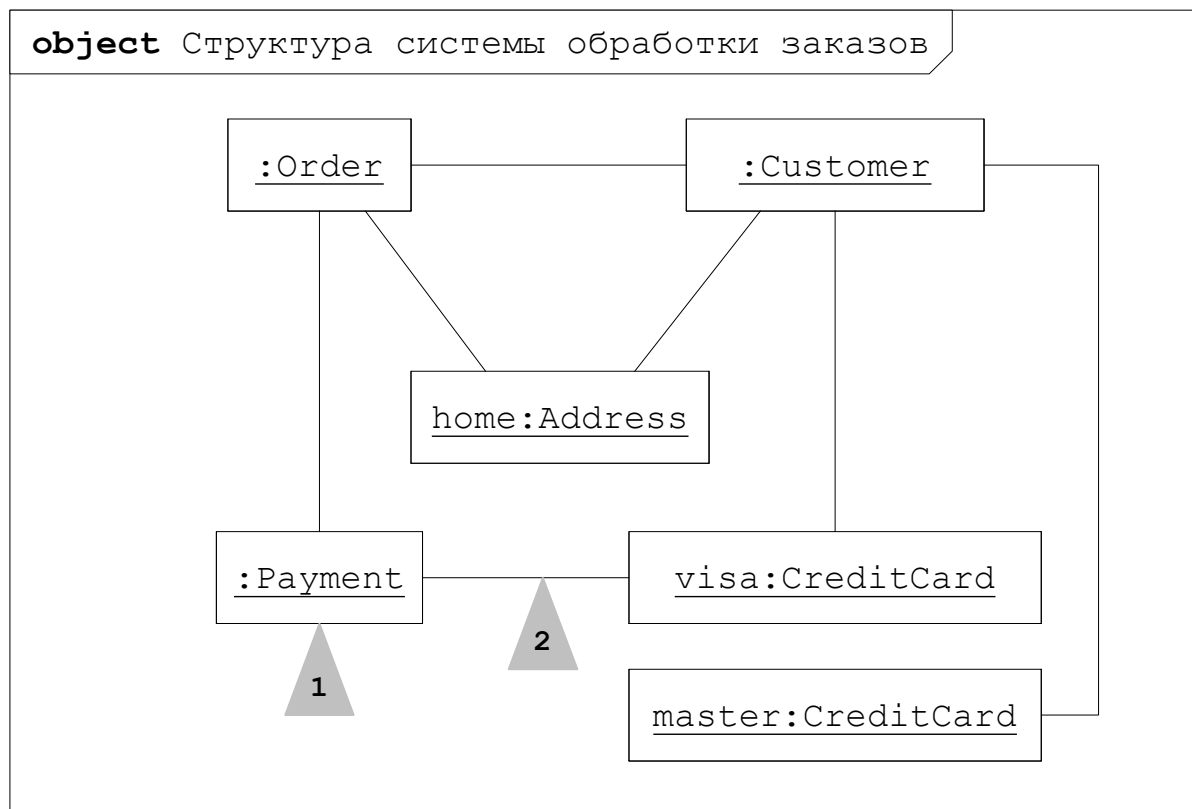


Рис. 1.19. Нотация диаграммы объектов

### 1.5.2. Диаграмма внутренней структуры

*Диаграмма внутренней структуры (composite structure diagram) используется для более подробного представления структурных классификаторов, прежде всего классов и компонентов.*

Структурный классификатор изображается в виде прямоугольника (1), в верхней части которого находится имя классификатора (2). Внутри находятся *части* (parts) (3). Частей может быть несколько. Части могут взаимодействовать друг с другом. Это

обозначается с помощью *соединителей* (connectors) (4) различных видов. Место на внешней границе части, к которому присоединяется соединитель, называется *портом* (port) (5). Порты располагаются также на внешней границе структурного классификатора (6). Основные элементы нотации, применяемые на диаграмме внутренней структуры, показаны на рис. 1.20. Детальное описание приведено в главе 3.

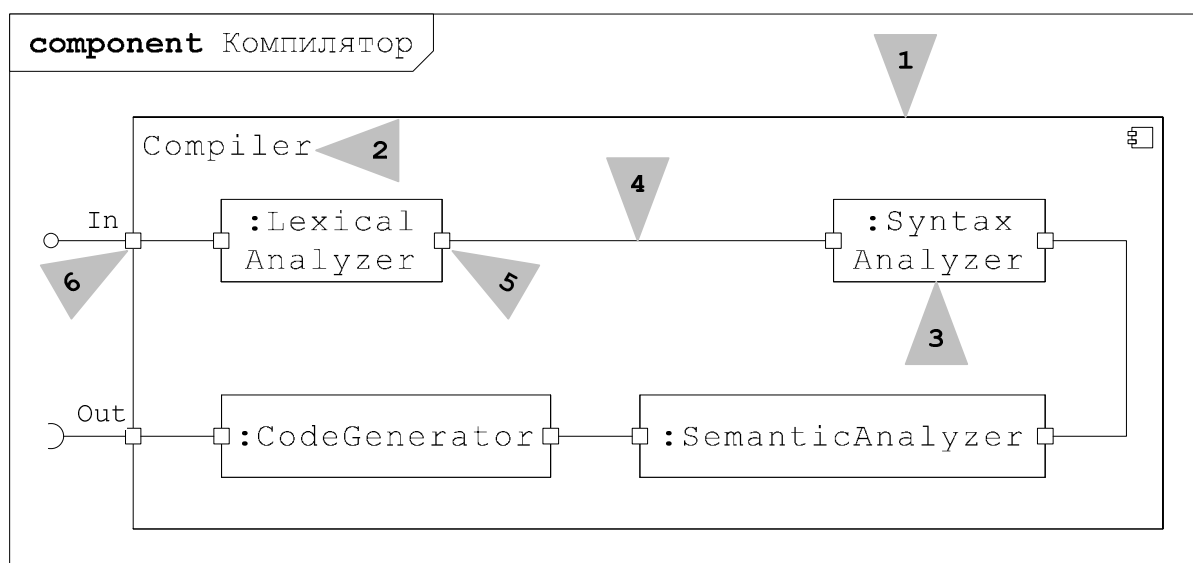


Рис. 1.20. Нотация диаграммы внутренней структуры

### 1.5.3. Обзорная диаграмма взаимодействия

**Обзорная диаграмма взаимодействия** (*interaction overview diagram*) является разновидностью диаграммы деятельности с расширенным синтаксисом: в качестве элементов обзорной диаграммы взаимодействия могут выступать **ссылки на взаимодействия** (*interaction use*) (1), определяемые диаграммами последовательности.

Основные элементы нотации показаны на рис. 1.21. Детальное описание приведено в главе 4.

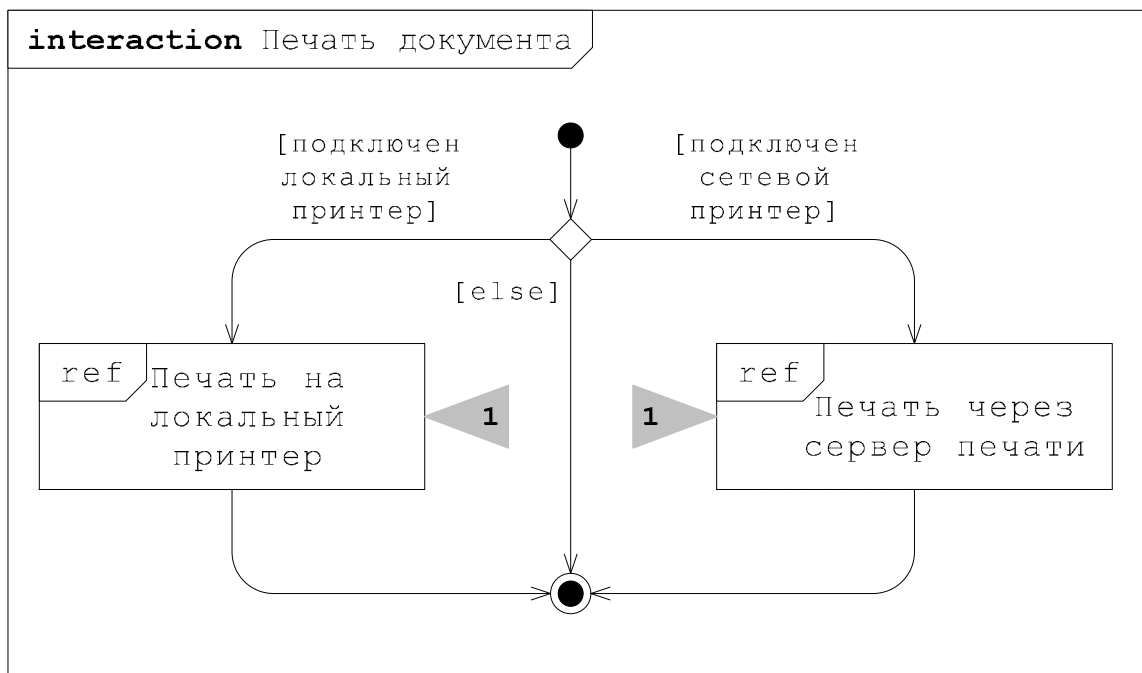


Рис. 1.21. Нотация обзорной диаграммы взаимодействия

#### 1.5.4. Диаграмма синхронизации

*Диаграмма синхронизации (timing diagram) представляет собой особую форму диаграммы последовательности, на которой особое внимание уделяется изменению **состояний** (1) различных экземпляров классификаторов и их временной синхронизации (2).*

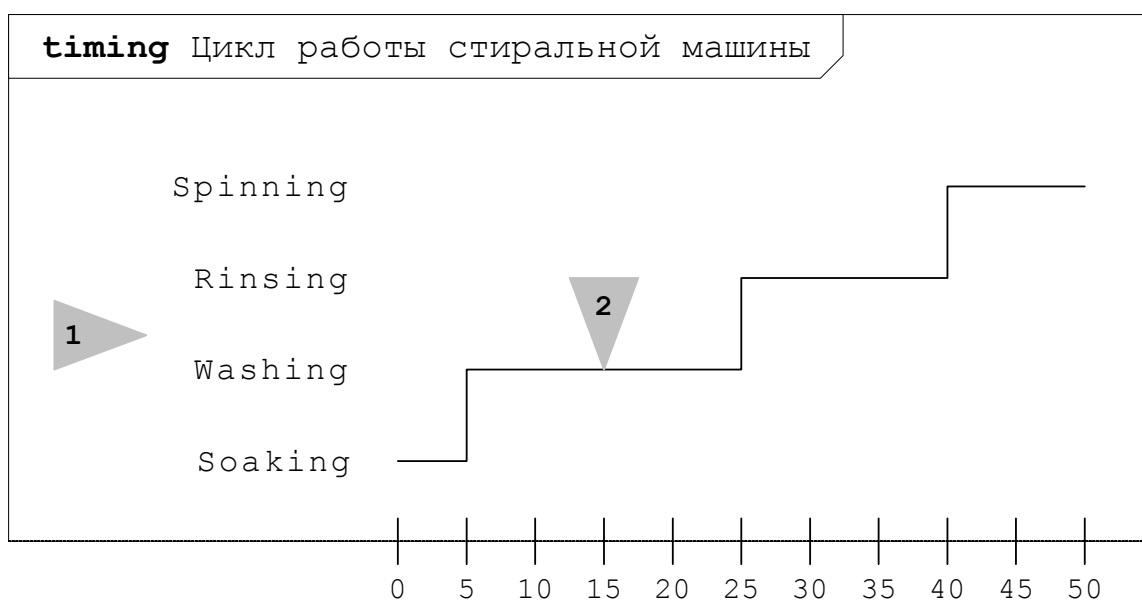


Рис. 1.22. Нотация диаграммы синхронизации



Основные элементы нотации показаны на рис. 1.22. Детальное описание приведено в главе 4.

### 1.5.5. Диаграмма пакетов

*Диаграмма пакетов (package diagram) — единственное средство, позволяющее управлять сложностью самой модели.*

Основные элементы нотации — пакеты (1) и зависимости с различными стереотипами (2), применяемые на диаграмме, показаны на рис. 1.23.

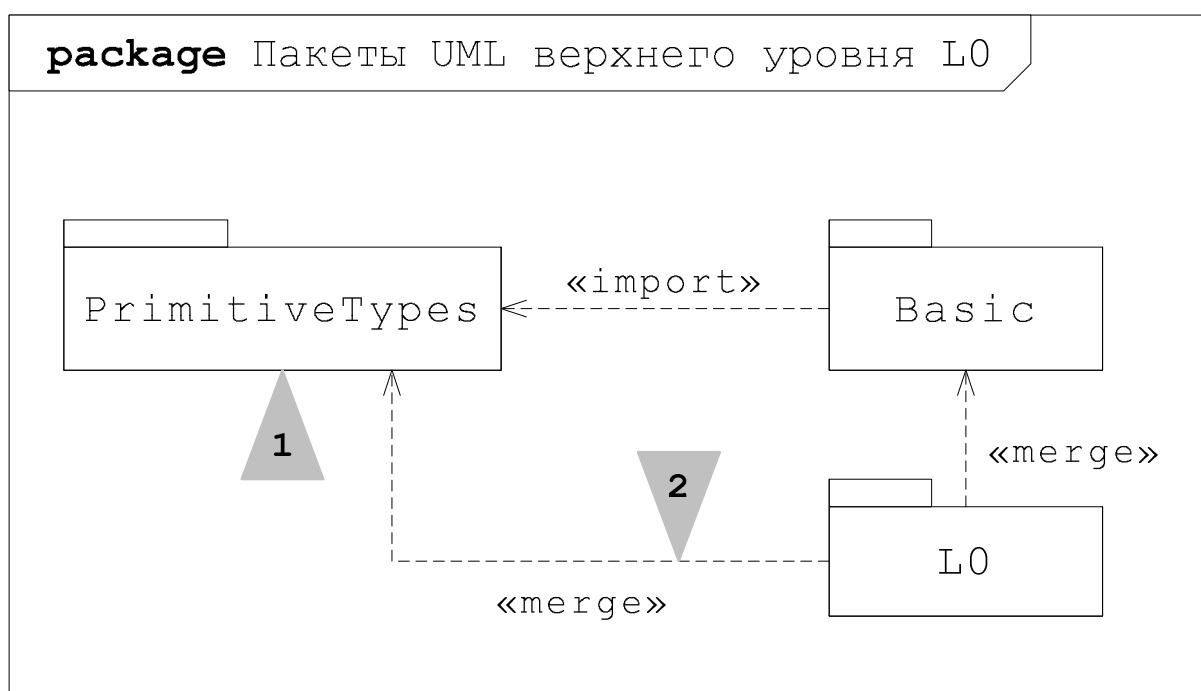


Рис. 1.23. Нотация диаграммы пакетов

## 1.6. МОДЕЛИ И ИХ ПРЕДСТАВЛЕНИЯ

Было бы очень соблазнительно иметь возможность строить модели любых систем для любых целей единообразно, придерживаясь, так сказать, одной универсальной точки зрения. Во многих ранних методологиях моделирования программных систем такие попытки (более или менее удачные) предпринимались.

Как показывает практический опыт, не удастся описать с единой точки зрения все без исключения аспекты моделируемой системы. Действительно, в модели нужно отразить множество вещей: интерфейсы для взаимодействия с внешним миром, внутреннюю логическую структуру программы, структуру хранимых данных, алгоритмы функционирования, состав артефактов, включаемых в поставку, и многое другое. Было бы самонадеянно утверждать, что единое средство описания всех аспектов сразу в принципе невозможно, — просто пока мы не знаем такого средства. Отсюда следует вывод: моделировать сложную систему следует с нескольких различных точек зрения, каждый раз принимая во внимание один аспект моделируемой системы и абстрагируясь от остальных. Этот тезис является одним из основополагающих принципов UML, может быть самым важным принципом, предопределившим практический успех UML.

Идея состоит в том, что абстрактный граф модели, состоящий из множества разнотипных сущностей и отношений, не подлежит конструированию или изучению в целом. Каждый раз для визуализации, изменения или иных манипуляций из этого общего графа вычленяются только сущности и отношения, релевантные для определенного аспекта моделируемой системы, а все остальные игнорируются. Такой вид с определенной точки зрения, можно сказать, проекцию модели, мы называем *представлением* (view). Можно сказать, что представление — это средство логического структурирования модели.

### **1.6.1. Классические представления из UML 1 и 2**

Набор используемых представлений модели является еще менее формальным и догматическим, чем набор канонических диаграмм.

Одним из самых популярных является набор представлений, описанных авторами языка в UML 1 [2] и показанных на рис. 1.24.

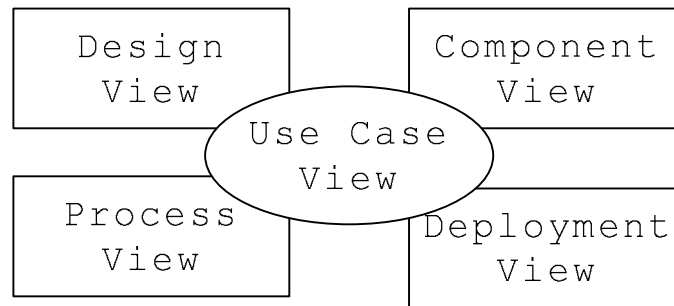


Рис. 1.24. Представления из UML 1

*Представление использования (Use Case View)* — это описание поведения системы в терминах вариантов использования с точки зрения внешних по отношению к системе действующих лиц. Данное представление описывает не то, как организована система, а те функциональные требования, которым она должна удовлетворять. При этом структурные аспекты передаются диаграммами использования, а поведенческие аспекты — диаграммами взаимодействия, состояний и деятельности.

*Представление проектирования (Design View)* предназначено для описания словаря предметной области, то есть, в парадигме объектно-ориентированного программирования, классов, а также таких вспомогательных сущностей как, например, интерфейсы или кооперации. Структурные аспекты передаются диаграммами классов и объектов, а поведенческие аспекты — диаграммами взаимодействия, состояний и деятельности.

*Представление процессов (Process view)* — это описание взаимодействия элементов управления (процессов, потоков) во время работы системы. Оно отражает такие нефункциональные требования, как, например, обеспечение параллелизма. Структурные аспекты передаются с помощью концепции активных классов, представляющих процессы и потоки, а поведенческие аспекты — диаграммами взаимодействия, состояний и деятельности.

*Представление компонентов (Component view)* — это описание системы на уровне артефактов (компонентов, файлов и т. д.), используемых для сборки, выпуска, конфигурации программного

продукта. Структурные аспекты передаются диаграммами компонентов, а поведенческие аспекты — диаграммами взаимодействия, состояний и деятельности.

*Представление размещения* (Deployment view) отражает топологию связей аппаратных средств и размещения на них компонентов. Структурные аспекты передаются диаграммами размещения, а поведенческие аспекты — диаграммами взаимодействия, состояний и деятельности.

В табл. 1.3 приведены наборы представлений, описанные авторами языка в UML 2<sup>3</sup>. Первоначальные пять представлений, ассоциирующиеся с UML 1, при переходе к UML 2 были дополнены и в результате образовали набор уже из восьми представлений.

Таблица 1.3

#### Представления модели и диаграммы в языке UML

<b>Представления</b>	<b>Диаграммы</b>	<b>Комментарий</b>
<i>Статическое представление</i> (Static view)	Диаграмма классов	В UML 1 – часть представления проектирования (Design view)
<i>Представление проектирования</i> (Design view)	Диаграмма внутренней структуры Диаграмма кооперации Диаграмма компонентов	В UML 1 частично отображается в представлении процессов (Process view) и представлении компонентов (Component view)
<i>Представление использования</i> (Use Case view)	Диаграмма использования	В UML 1 описывается одноименным представлением
<i>Представление конечных автоматов</i> (State machine view)	Диаграмма автомата	В UML 1 данное представление используется всеми представлениями по мере необходимости

<sup>3</sup> Таблица отражает видение, характерное для UML 2. Комментарии поясняют различие между представлениями в UML 1 и UML 2.

<b>Представления</b>	<b>Диаграммы</b>	<b>Комментарий</b>
<i>Представление деятельности</i> (Activity view)	Диаграмма деятельности Обзорная диаграмма взаимодействия	В UML 1 данное представление используется всеми представлениями по мере необходимости.
<i>Представление взаимодействия</i> (Interaction view)	Диаграмма последовательности Диаграмма коммуникации Диаграмма синхронизации	В UML 1 данное представление используется всеми представлениями по мере необходимости.
<i>Представление развертывания (размещения)</i> (Deployment view)	Диаграмма развертывания	В UML 1 описывается одноименным представлением
<i>Представление управления моделью</i> (Model Management view)	Диаграмма пакетов	Отсутствует в UML 1

### 1.6.2. Три представления — взгляд авторов

Учитывая неформальный характер самого понятия представления, и опираясь на собственный опыт использования UML, в этом пособии мы предлагаем свой вариант набора представлений. Их всего три.

*Представление использования.* По сути это то же самое представление, что было указано выше. Представление использования призвано отвечать на вопрос, что делает система полезного. Определяющим признаком для отнесения элементов модели к представлению использования является, по нашему мнению, явное сосредоточение внимания на факте наличия у системы внешних границ, то есть выделение внешних действующих лиц, взаимодействующих с системой, и внутренних вариантов

использования, описывающих различные сценарии такого взаимодействия. Таким образом, единственным выразительным средством представления использования оказываются диаграммы использования.

*Представление структуры.* Представление структуры призвано отвечать (с разной степенью детализации) на вопрос: из чего состоит система. Определяющим признаком для отнесения элементов модели к представлению структуры является явное выделение структурных элементов — составных частей системы — и описания взаимосвязей между ними. Принципиальным является чисто статический характер описания, то есть отсутствие понятия времени в любой форме, в частности, в форме последовательности событий и/или действий. Представление структуры описывается, прежде всего, и главным образом диаграммами классов, а также, если нужно, диаграммами компонентов, размещения, внутренней структуры и, в редких случаях, диаграммами объектов.

*Представление поведения.* Представление поведения призвано отвечать на вопрос: как работает система. Определяющим признаком для отнесения элементов модели к представлению поведения является явное использование понятия времени, в частности, в форме описания последовательности событий/действий, то есть в форме алгоритма. Представление поведения описывается диаграммами автомата и деятельности, а также обзорной диаграммой взаимодействия, диаграммами коммуникации и последовательности. В редких случаях можно воспользоваться диаграммой синхронизации.

Такой набор представлений является ортогональным и согласованным с классификацией диаграмм (см. рис. 1.9 и рис. 1.10). Более того, он во многом инспирирован опытом моделирования.

По нашему мнению, **процесс моделирования (независимо от назначения модели) является не линейным последовательным, а итеративным и параллельным**, примерно таким, как показано на рис. 1.25.

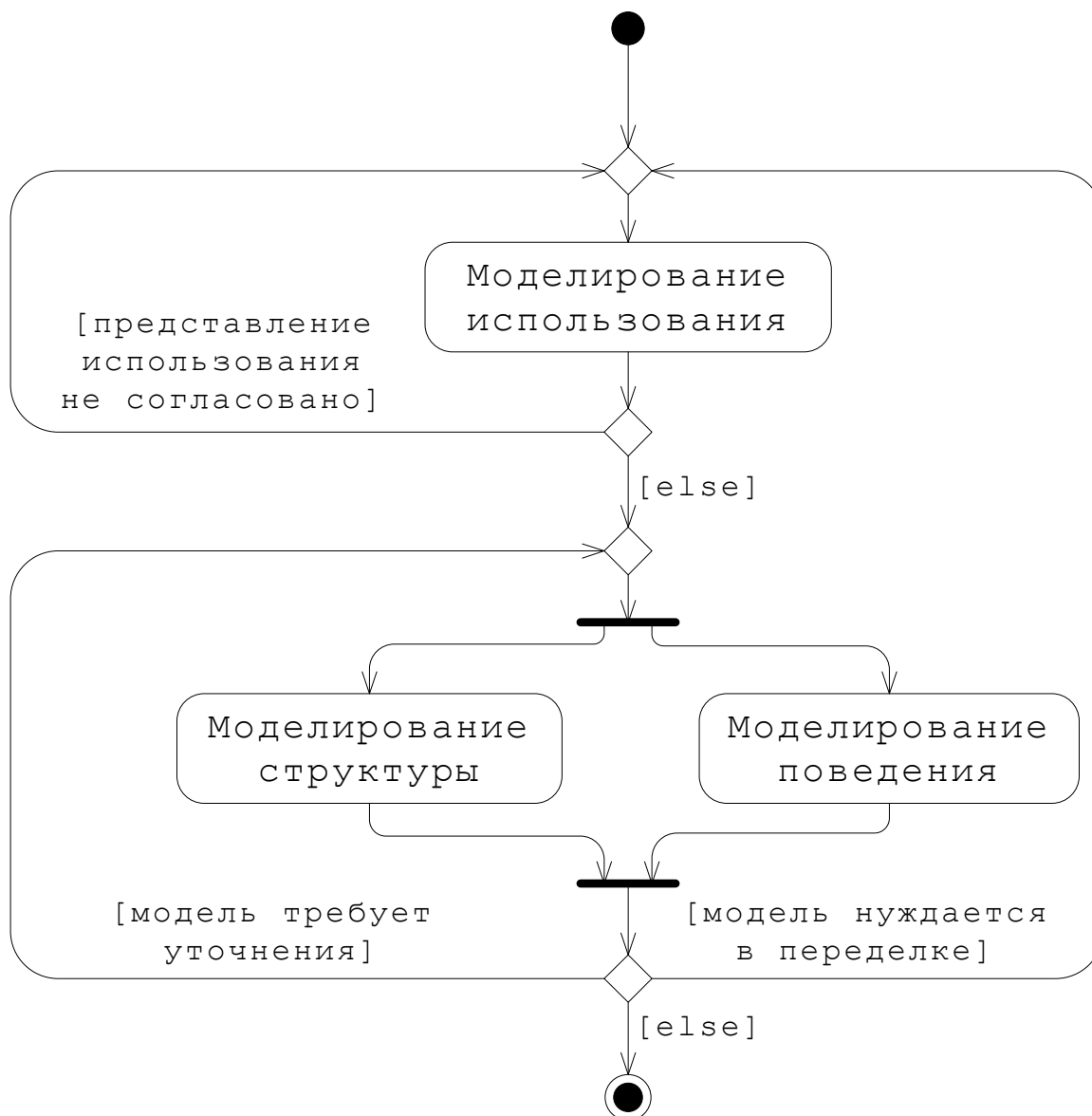


Рис. 1.25. Процесс моделирования

Другими словами, процесс моделирования циклический, на каждом шаге может присутствовать уточнение представления использования, за которым следует параллельное моделирование структуры и поведения.

## ВЫВОДЫ

1. UML — это формальный графический объектно-ориентированный язык, который необходимо освоить.

2. Знание UML является необходимым, однако не является достаточным условием построения разумных моделей сложных программных систем.

3. UML имеет синтаксис, семантику и прагматику, которые нужно знать и использовать с учетом особенностей реальной задачи и инструмента.

4. Модель UML состоит из описания сущностей и отношений между ними.

5. Элементы модели группируются в диаграммы и представления для наилучшего описания моделируемой системы с различных точек зрения.



## 2. МОДЕЛИРОВАНИЕ ИСПОЛЬЗОВАНИЯ

### 2.1. ЗНАЧЕНИЕ МОДЕЛИРОВАНИЯ ИСПОЛЬЗОВАНИЯ

Для большинства средств UML нетрудно отыскать аналоги среди широко используемых практических методов конструирования программных систем. И это не удивительно, ведь именно эти методы были унифицированы посредством UML. А вот для диаграмм использования известный аналог указать труднее. Мы попытаемся объяснить прагматику моделирования использования на конкретном примере.

#### 2.1.1. Сквозной пример

В остальных частях учебного пособия рассматривается один сквозной пример моделирования сравнительно несложного приложения — **информационной системы отдела кадров**. Выбор примера обусловлен следующими соображениями.

Во-первых, **предметная область до некоторой степени знакома всем**. Таким образом, суть задачи заранее ясна, и можно сосредоточить внимание на тонкостях применения UML, а не на объяснении особенностей предметной области.

Во-вторых, **информационная система отдела кадров — это типичное офисное приложение** из самого распространенного класса систем автоматизации делопроизводства. UML как нельзя лучше подходит для моделирования именно таких систем и все средства языка можно проиллюстрировать естественным образом.

В-третьих, **авторам случалось разрабатывать похожие системы** на самом деле, а не только в книге.

Итак, поставим себя на место разработчика и предположим, что в нашем распоряжении имеется следующий текст, поступивший от заказчика.

## ТЕХНИЧЕСКОЕ ЗАДАНИЕ

*Информационная система «Отдел кадров» (сокращенно ИС ОК) предназначена для ввода, хранения и обработки информации о сотрудниках и движении кадров. Система должна обеспечивать выполнение следующих основных функций.*

- 1. Прием, перевод и увольнение сотрудников.*
- 2. Создание и ликвидация подразделений.*
- 2. Создание вакансий и сокращение должностей.*

Конечно, техническое задание из одного абзаца текста и трех нумерованных пунктов — это не более чем учебный пример<sup>4</sup>. Однако даже на этом примере видны многие характерные "особенности" подобных документов, которые, увы, слишком часто встречаются в реальной жизни. С одной стороны, что-то написано, а с другой стороны не очень понятно, что делать дальше. Безо всяких объяснений заказчик использует термины своей предметной области — разработчик должен их знать и понимать. Требований к реализации нет вовсе. Функции не упорядочены по приоритетам: не ясно, что является критически важным, а чем можно поступиться в случае необходимости.

Мы постараемся показать, как, применяя UML, можно постепенно превратить расплывчатое описание приложения во вполне четкую модель, пригодную для реализации.

### **2.1.2. Преимущества моделирования использования**

Отвлечемся пока от технических деталей нотации диаграмм использования и рассмотрим, что предлагается делать на первом шаге моделирования использования.

---

<sup>4</sup> В реальности довольно часто задание формулируется заказчиком очень кратко и в процессе разработки "обрастает" деталями. Мы используем похожий прием, постепенно усложняя начальную постановку задачи.

Перечислим те преимущества, которые дает этот подход по сравнению с другими.

**Простые утверждения.** Моделирование использования фактически позволяет переписать исходное техническое задание (или просто записать, если никакой исходной формулировки требований не было) в строгой и формальной, но, в тоже время, очень простой и наглядной графической форме, как совокупность простых утверждений относительно того, что делает система для пользователей. Конечно, использование такой формы не гарантирует от ошибок (вряд ли гарантия от ошибок вообще возможна), но благодаря простоте и наглядности формы их легче заметить.

**Абстрагирование от реализации.** Моделирование использования предполагает формулирование требований к системе абсолютно независимо от ее реализации. Другими словами, представление использования описывает только, **что** делает система (но не **как** это делается и не **зачем** это нужно делать). Заметим, что другие подходы, используя на первых шагах термины и понятия реализации (структура программы, структура данных, структура взаимодействующих объектов) накладывают невольные ограничения на реализацию, которые не вытекают из существа задачи, а значит, могут служить источником неэффективности и ошибок.

**Декларативное описание.** Каждый вариант использования описывает (а вернее сказать, именуется) некоторое множество последовательностей действий, доставляющих значимый для пользователя результат. Однако никакого императивного описания представление использования не содержит, в модели нет указаний на то, какой вариант использования должен выполняться раньше, а какой позже, то есть, нет описания алгоритма, а значит, нет алгоритмических ошибок.

**Выявление границ.** Представление использования определяет границы системы и постулирует существование во внешнем мире использующих ее агентов (действующих лиц). Описание системы в

виде черного ящика с определенными интерфейсами кажется очень похожим на представление использования, но здесь есть важное различие, которое часто упускается из вида. Если ограничиться только описанием интерфейсов, то очень легко допустить ошибки следующего типа: предусмотреть интерфейс, который не нужен, потому что им никто не пользуется. Или, аналогично, забыть интерфейс, который необходим определенной категории пользователей. На диаграмме использования одинокие и покинутые действующие лица и варианты использования обнаруживаются с первого взгляда.

Наш вывод таков: **моделирование использования безопаснее и надежнее альтернативных методов**, то есть при прочих равных условиях позволяет совершить меньше грубых проектных ошибок на ранних стадиях проектирования. В этом заключается основное преимущество данного метода.

## 2.2. ДИАГРАММЫ ИСПОЛЬЗОВАНИЯ

Диаграммы использования были предложены Иваром Якобсоном в их нынешней графической форме еще в 1986 году. Диаграммы использования являются, безусловно, самым стабильным элементом UML — они не менялись уже двадцать лет с лишним, фактически, приняли законченную форму задолго до появления языка. Одновременно эти диаграммы имеют самую простую нотацию: всего два основных типа сущностей (действующие лица и варианты использования) и три типа отношений (зависимости, ассоциации, обобщения)!

### 2.2.1. Действующие лица

Вопрос о выделении (или идентификации) действующих лиц при составлении модели — один из самых болезненных. Неудачный выбор действующих лиц может отрицательно повлиять на всю модель

в целом. Здесь легко впасть в крайность: объявить, что имеется одно действующее лицо (внешний мир), взаимодействующее со всеми вариантами использования или, наоборот, придумать искусственных действующих лиц для каждого варианта использования. Оба экстремальных варианта являются, по существу, моделью черного ящика и сводят к нулю преимущества моделирования использования, рассмотренные в предыдущем разделе. Формального метода идентификации действующих лиц не существует. Здесь мы перечислим некоторые приемы, которые полезно иметь в виду при выделении действующих лиц и покажем применение этих приемов на нашем примере информационной системы отдела кадров. Для начала укажем более детальное определение действующего лица.

*С синтаксической точки зрения **действующее лицо** — это стереотип классификатора, который обозначается специальным значком. Для действующего лица указывается только имя, идентифицирующее его в системе. Семантически действующее лицо — это множество логически взаимосвязанных ролей.*

С прагматической точки зрения главным является то, что действующие лица находятся **вне** проектируемой системы (или рассматриваемой части системы).

В типовых случаях различные действующие лица назначаются для категорий пользователей (если их удастся выделить естественным образом), внешних программных и аппаратных средств (если система взаимодействует с таковыми).

Рассмотрим наш пример с информационной системой отдела кадров. Выделение категорий пользователей происходит, как правило, неформально: из соображений здравого смысла и собственного опыта. Тем не менее, несколько советов мы можем дать. Имеет смысл отнести пользователей к разным категориям, если наблюдаются следующие признаки:

- пользователи участвуют в разных (независимых) бизнес-процессах;

- пользователи имеют различные права на выполнение действий и доступ к информации;

- пользователи взаимодействуют с системой в разных режимах: от случая к случаю, регулярно, постоянно.

Опираясь на собственные советы, применительно к нашему примеру мы в первом приближении выделяем две категории пользователей:

- менеджер персонала, который работает с конкретными людьми;

- менеджер штатного расписания, который работает с абстрактными должностями и подразделениями.

Бизнес-процесс пользователя первой категории включает в себя не только работу с приложением, но и беседы с конкретными людьми, интервью и тому подобное, чем явно отличается от других бизнес-процессов предприятия.

Пользователи второй категории, очевидно, должны иметь специальные права доступа, поскольку вряд ли допустимо, чтобы кто угодно мог создавать и уничтожать подразделения на предприятии.

На рис. 2.1 мы начинаем формировать представление использования информационной системы отдела кадров. Менеджер персонала имеет имя `Personnel Manager`, а менеджер штатного расписания — `Staff Manager`, в соответствии с используемой дисциплиной имен.

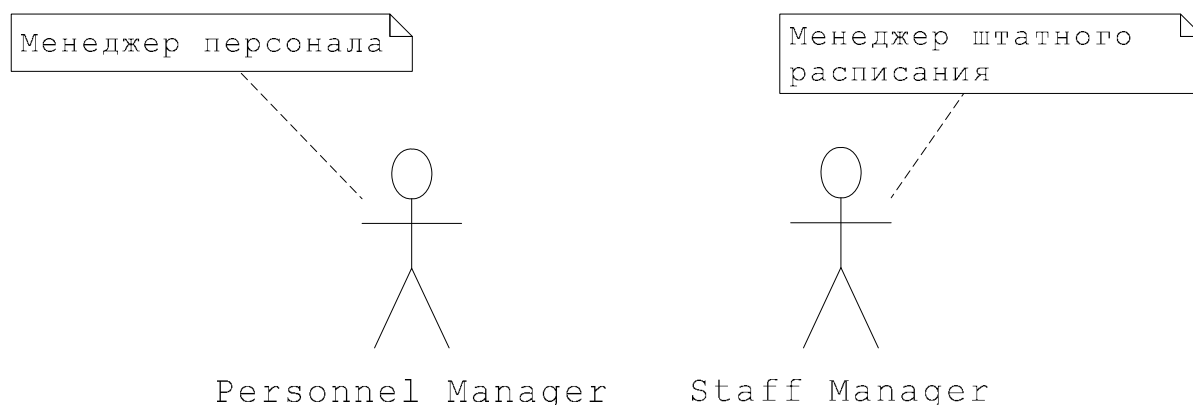


Рис. 2.1. Действующие лица ИС ОК

Для UML пока что нет достаточно устоявшейся дисциплины имен, но некоторый набор рекомендаций можно найти в литературе. Мы, по возможности, следуем этим рекомендациям. В частности, в качестве имен действующих лиц рекомендуется использовать существительное (возможно с определяющим словом), а в качестве имен вариантов использования — глагол (возможно, с дополнением). Эти правила основаны на семантике моделирования использования.

### 2.2.2. Варианты использования

Выделение вариантов использования — ключ ко всему дальнейшему моделированию. На этом этапе определяется функциональность системы, то есть, **что** она должна делать.

Нотация для варианта использования очень скудная — это просто имя, помещенное в овал (или помещенное под овалом — такой вариант тоже допустим). Другими словами, функции, выполняемые системой, на уровне моделирования использования никак не раскрываются — им только даются имена.

*Семантически **вариант использования** (use case) — это описание множества возможных последовательностей действий (событий), приводящих к значимому для действующего лица результату.*

*Каждая конкретная последовательность действий называется **сценарием**.*

В нашем примере простой анализ текста технического задания выявляет семь вариантов использования:

- прием сотрудника;
- перевод сотрудника;
- увольнение сотрудника;
- создание подразделения;
- ликвидация подразделения;
- создание вакансии;

- сокращение должности.

Опираясь на знание предметной области, получим набор вариантов использования, представленный на рис. 2.2.

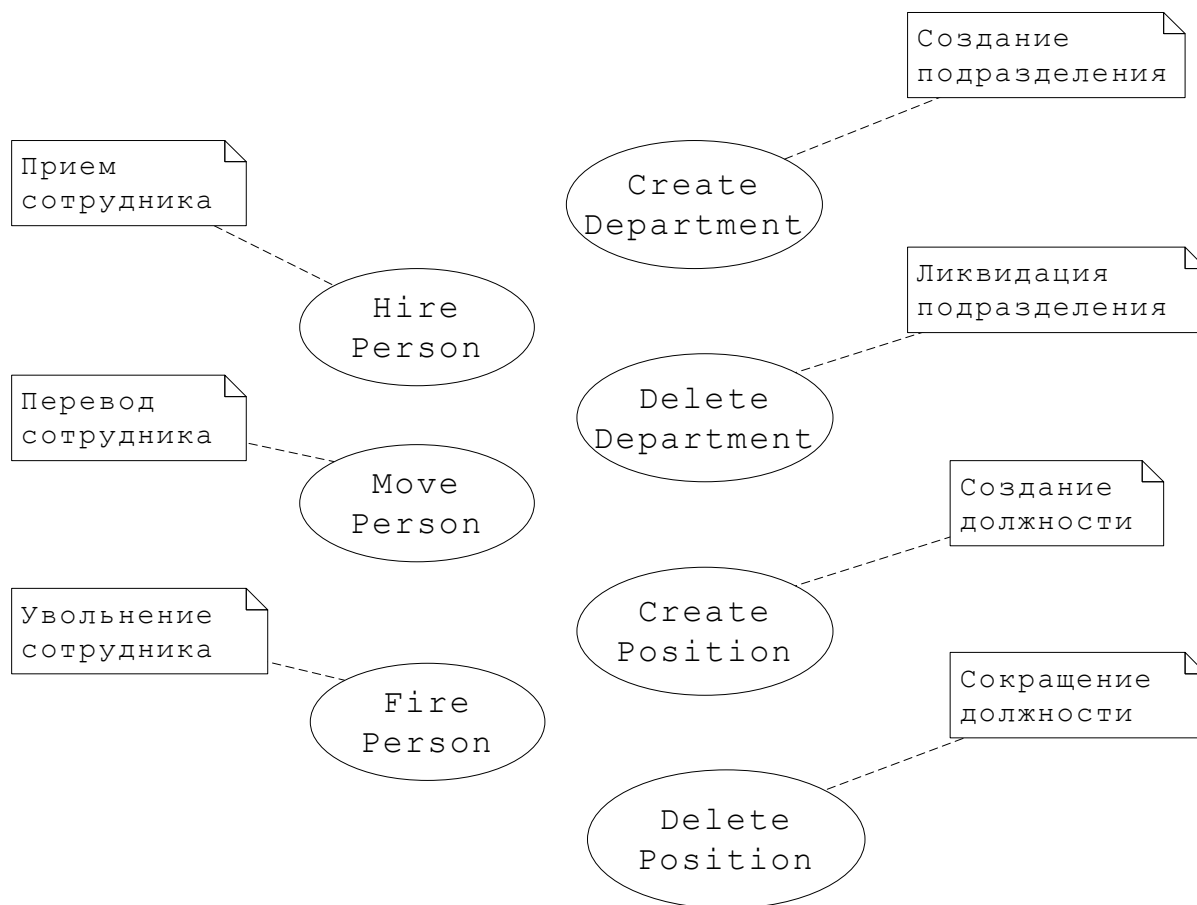


Рис. 2.2. Варианты использования ИС ОК

### 2.2.3. Примечания

Третьим типом сущности, применяемым на диаграмме использования, является примечание. Заметим, что примечания являются очень важным средством UML, значение которого часто недооценивается начинающими пользователями. **Примечания можно и нужно употреблять на всех типах диаграмм, а не только на диаграммах использования.** UML является унифицированным, но



никак не универсальным языком — при моделировании проектировщик часто может сказать о моделируемой системе больше, чем это позволяет сделать строгая, но ограниченная нотация UML. В таких случаях наиболее подходящим средством для внесения в модель дополнительной информации является примечание.

В отличие от большинства языков программирования примечания в UML синтаксически оформлены с помощью специальной нотации и выступают на тех же правах, что и остальные сущности. А именно, примечание имеет свою графическую нотацию — прямоугольник с загнутым уголком ("собачье ухо"), в котором находится текст примечания. Примечания могут находиться в отношении соответствия с другими сущностями — эти отношения изображаются пунктирной линией без стрелок. Если пунктирная линия отсутствует, то примечание относится ко всей диаграмме.

Примечания содержат текст, который вводит пользователь — создатель модели. Это может быть текст в произвольном формате: на естественном языке, на языке программирования, на формальном логическом языке, например, OCL и т. д. Более того, если возможности инструмента это позволяют, в примечаниях можно хранить гиперссылки, вложенные файлы и другие артефакты, внешние по отношению к модели.

Примечания могут иметь стереотипы. В UML определены два стандартных стереотипа для примечаний:

- «requirement» — описывает общее требование к системе;
- «responsibility» — описывает ответственность сущности (классификатора).

Примечания первого типа часто присутствуют на диаграммах использования, а примечания второго типа — на диаграммах классов.

Возвращаясь к нашему примеру, будет совсем не лишним указать, что информацию о состоянии кадров нужно хранить постоянно, т. е. она не должна исчезать после завершения сеанса работы с системой (рис. 2.3).

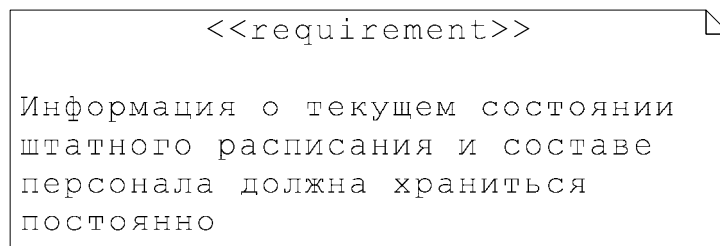


Рис. 2.3. Нефункциональное требование к ИС ОК

#### 2.2.4. Отношения на диаграммах использования

Как уже было отмечено в первой главе, на диаграммах использования применяются следующие основные типы отношений:

- ассоциация между действующим лицом и вариантом использования;
- обобщение между действующими лицами;
- обобщение между вариантами использования;
- зависимости между вариантами использования.

*Ассоциация между действующим лицом и вариантом использования показывает, что действующее лицо тем или иным способом взаимодействует (предоставляет исходные данные, получает результат) с вариантом использования.*

Другими словами, эта ассоциация обозначает, что действующее лицо так или иначе, но обязательно непосредственно участвует в выполнении каждого из сценариев, описываемых вариантом использования. Ассоциация является наиболее важным и, фактически, обязательным отношением на диаграмме использования. Действительно, если на диаграмме использования нет ассоциаций между действующими лицами и вариантами использования, то это означает, что система не взаимодействует с внешним миром. Такие системы, равно как и их модели, не имеют практического смысла.

Применительно к нашему примеру в первом приближении можно обозначить ассоциации, представленные на рис. 2.4.

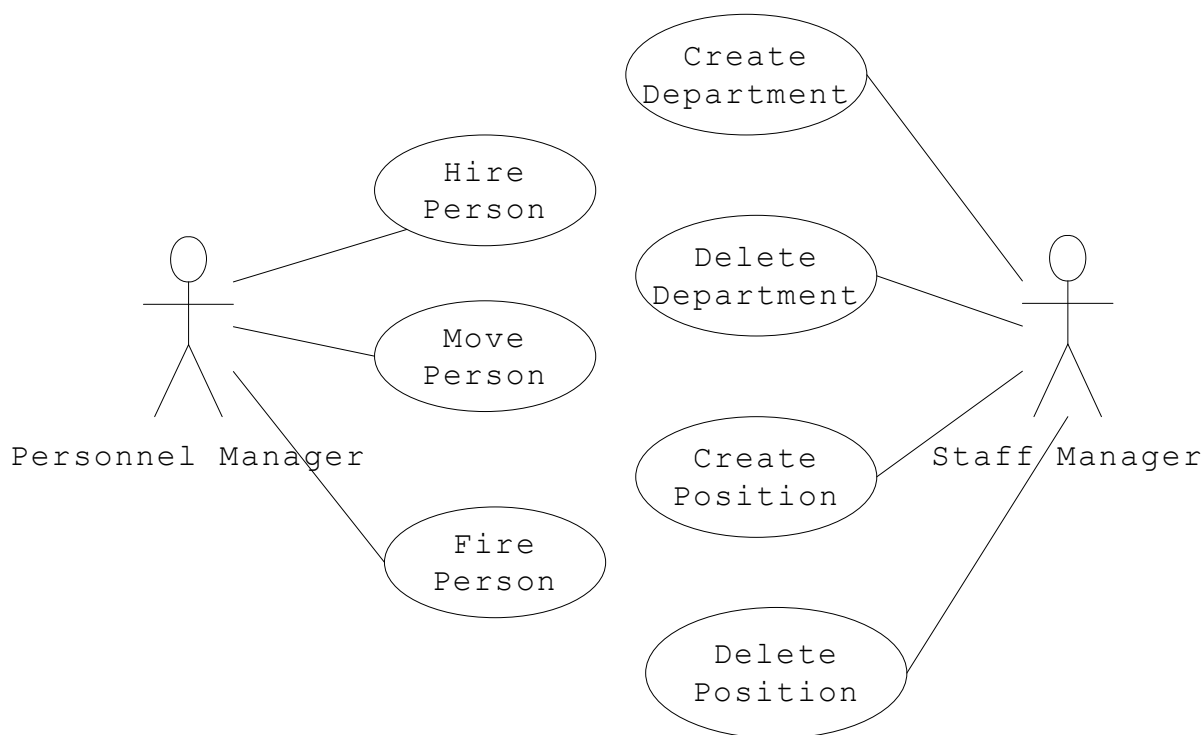


Рис. 2.4. Ассоциации между действующими лицами и вариантами использования

*Обобщение между действующими лицами* показывает, что одно действующее лицо наследует все свойства (в частности, участие в ассоциациях) другого действующего лица.

С помощью обобщения между действующими лицами легко показать иерархию категорий пользователей системы, в частности, иерархию прав доступа к выполняемым функциям и хранимым данным.

## ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

*Среди всех пользователей информационной системы следует выделить особую категорию пользователей (высшее руководство), которой разрешен доступ к любым данным и операциям.*

Это изменение в требованиях можно отразить в модели системы так, как показано на рис. 2.5.



Рис. 2.5. Иерархия категорий пользователей ИС ОК

Действующее лицо, будучи классификатором, может быть абстрактным классификатором, то есть таким классификатором, который не может иметь непосредственных экземпляров. Введение абстрактных действующих лиц позволяет без потери информации сократить количество непосредственных ассоциаций в модели, сделав ее более лаконичной, а значит более наглядной и полезной.

## ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

*Информационная система должна предоставлять возможность просматривать данные без внесения в них каких-либо изменений.*

Данное требование следует оформить в виде дополнительного варианта использования — Browse. Разумно предположить, что просматривать данные могут все категории пользователей. В этом случае можно поступить так, как показано на рис. 2.6, т. е. ввести обобщенного абстрактного пользователя User (1), который будет связан ассоциацией с вариантом использования Browse (2). При этом все специализации (3 и 4) обобщенного пользователя автоматически будут связаны ассоциацией с вариантом использования Browse.

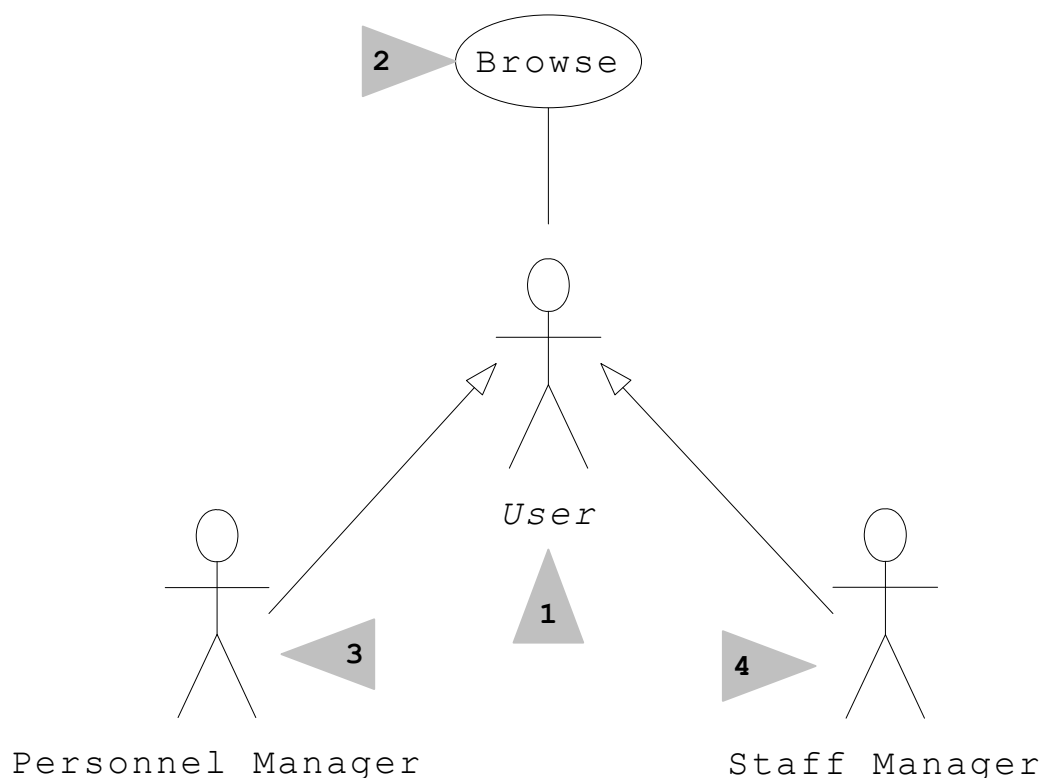


Рис. 2.6. Абстрактное действующее лицо

*Обобщение между вариантами использования показывает, что один вариант использования является частным случаем (подмножеством множества сценариев) другого варианта использования.*

Обобщающий вариант использования, будучи классификатором, может быть абстрактным классификатором. Например, такой важный для сотрудника вариант использования, как увольнение, на самом деле является абстракцией: в каждом конкретном случае имеет место ровно один из возможных частных случаев увольнения, которые приводят к одному и тому же результату с точки зрения менеджера персонала, но весьма различны с точки зрения сотрудника.

## ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

*Система должна поддерживать два способа увольнения сотрудника: по инициативе администрации и по собственному желанию.*

Данное обстоятельство можно отразить в модели так, как показано на рис. 2.7.

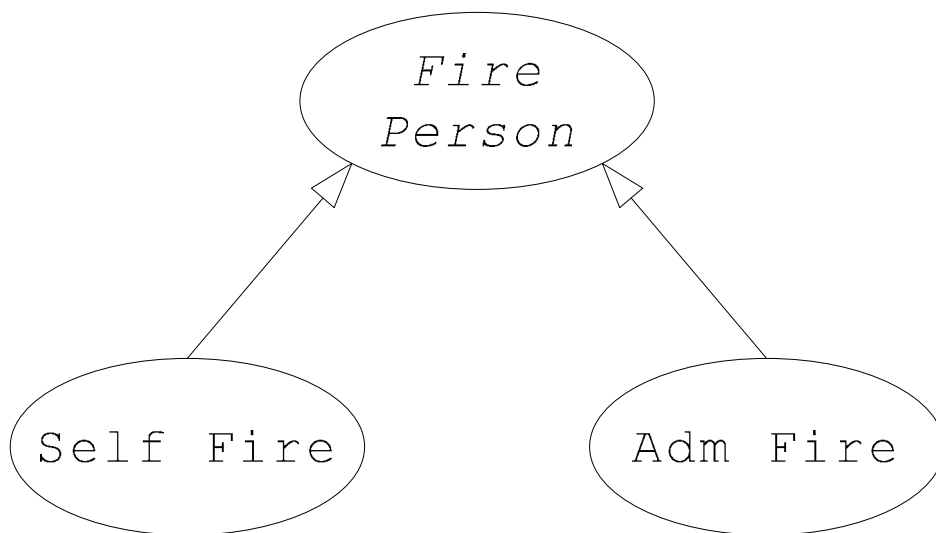


Рис. 2.7. Обобщение вариантов использования

Обобщенный абстрактный (имя написано курсивом) вариант использования *Fire Person* имеет две специализации, которые соответствуют увольнению работника по собственному желанию (*Self Fire*) и по инициативе администрации (*Adm Fire*).

*Зависимость между вариантами использования* показывает, что один вариант использования зависит от другого варианта использования.

В UML имеются два стандартных стереотипа зависимости между вариантами использования, которые в некотором смысле двойственны друг другу:

- «include» — показывает, что в **каждый** сценарий зависимого варианта использования в определенном месте вставляется в качестве подпоследовательности действий в сценарий независимого варианта использования;

- «extend» — показывает, что в **некоторый** сценарий независимого варианта использования может быть в определенном

месте вставлен в качестве подпоследовательности действий сценарий зависимого варианта использования.

## ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

*При увольнении сотрудника должна быть осуществлена выплата денежной компенсации за неиспользованный отпуск. В случае вынужденного сокращения возможна выплата выходного пособия. Учетная запись сотрудника при увольнении должна быть заблокирована.*

Блокировка учетной записи и выплата компенсации — это примеры вариантов использования, которые вполне могут быть востребованы как при увольнении, так и помимо него. Отношения зависимости между этими вариантами использования могут быть показаны на диаграмме использования, например, так, как это сделано на рис. 2.8.

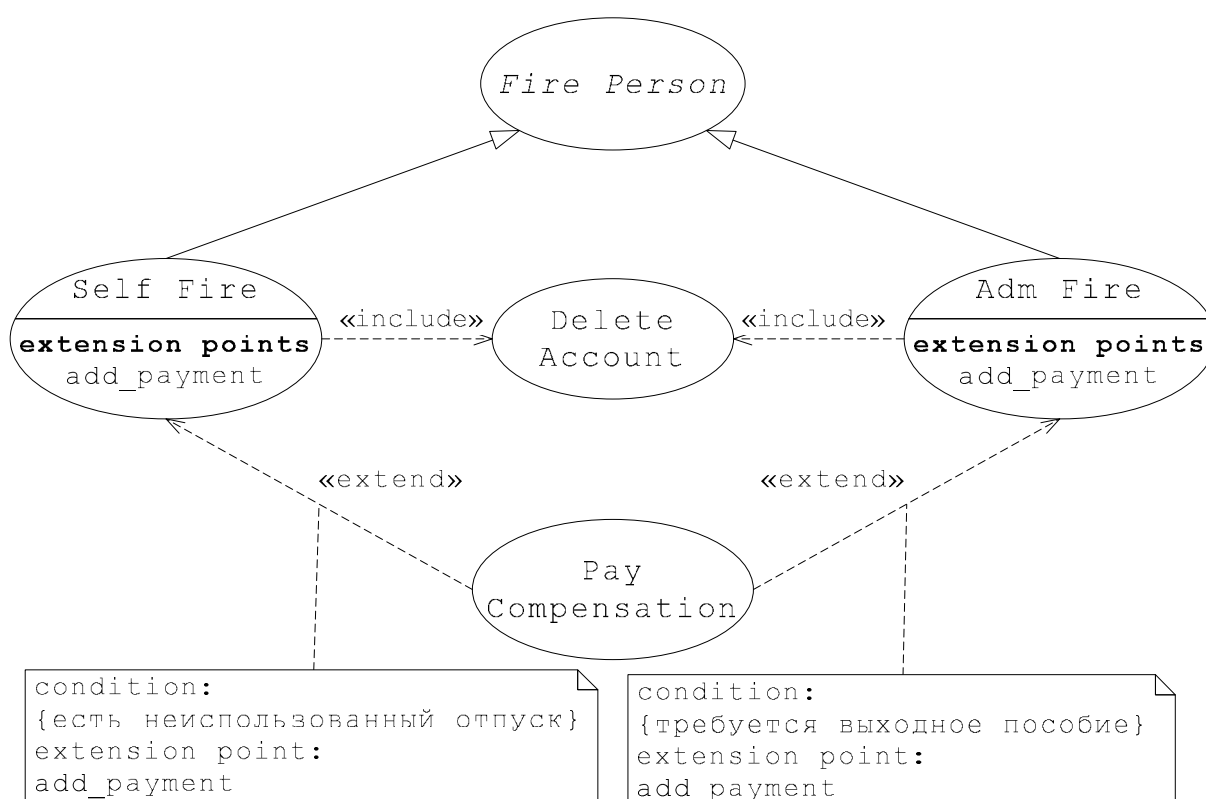


Рис. 2.8. Зависимости между вариантами использования

### 2.2.5. Способы применения моделей использования

Если посмотреть на модель использования с самой общей точки зрения, то нетрудно заметить, что в модели присутствуют:

- внутренняя моделируемая система, в форме набора вариантов использования, возможно связанных зависимостями и обобщениями;
- внешнее окружение, в форме набора действующих лиц, возможно связанных обобщениями;
- связь между моделируемой системой и внешним окружением в форме ассоциаций между действующими лицами и вариантами использования.

Обычно совершенно ясно, что находится внутри моделируемой системы, а что снаружи. Если это почему-либо неясно, или же требуется увеличить наглядность диаграмм, то можно воспользоваться специальной конструкцией, которая называется "границы системы".

***Границы системы** (system boundary) — это графический комментарий в форме прямоугольной рамки, применяемый на диаграммах использования и отделяющий внутреннюю часть системы от ее внешнего окружения.*

Внутренняя часть, выделяемая границами, имеет в UML конкретное название — субъект.

***Субъект** (subject) — это классификатор, который реализует поведение, декларируемое вариантами использования.*

Если границы системы используются на диаграмме, то можно указать имя (и стереотип!), которые будут относиться к субъекту<sup>5</sup>. В примере на рис. 2.9 мы повторили рис. 2.4, но использовали другие возможности нотации UML.

---

<sup>5</sup> Это можно сделать и с помощью примечания, но использования границ системы более наглядно.



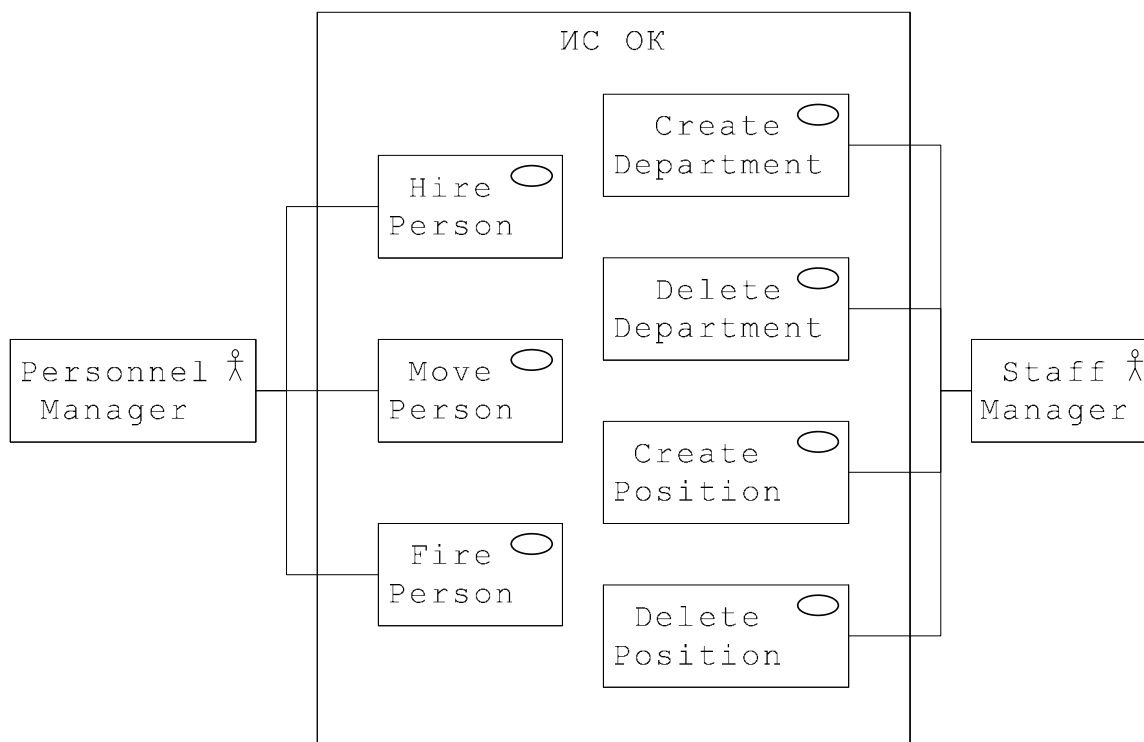


Рис. 2.9. Границы системы

## 2.3. РЕАЛИЗАЦИЯ ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ

После того, как построено представление использования (результат моделирования использования), то есть, выделены действующие лица, варианты использования и установлены отношения между ними, встает естественный вопрос: что дальше? То есть, как далее следует продолжать моделирование средствами UML?

Действующие лица находятся вне системы — с ними ничего делать не нужно. Можно сказать, что действующие лица уже выполнили свою задачу, просто появившись в модели системы. Таким образом, переход от моделирования использования к другим видам моделирования состоит в уточнении, детализации и конкретизации вариантов использования. В представлении использования мы показали, **что** делает система, теперь нужно определить, **как** это делается. Это обычно называется реализацией вариантов использования.

**Реализация варианта использования** (*use case realization*) — это описание всех или некоторых сценариев, составляющих вариант использования.

Повторим еще раз: вариант использования — это описание множества последовательностей событий или действий (сценариев), доставляющих значимый для действующего лица результат. Наиболее часто используемый метод описания множества последовательностей действий состоит в указании *алгоритма*, выполнение которого доставляет последовательность действий из требуемого множества<sup>6</sup>. Существует множество способов описания алгоритмов, более или менее формальных. Мы рассмотрим четыре, часто применяемых именно при реализации вариантов использования.

### 2.3.1. Текстовые описания

Исторически самый заслуженный и до сих пор один из самых популярных способов: составить текстовое описание типичного сценария варианта использования.

Рассмотрим следующий ниже текст в качестве примера одного из возможных сценариев.

**Сценарий варианта использования**      *Увольнение по собственному желанию*

1. *Сотрудник пишет заявление*
2. *Начальник подписывает заявление*
3. **Если** *есть неиспользованный отпуск,*  
    **то** *бухгалтерия рассчитывает компенсацию*
4. *Бухгалтерия рассчитывает выходное пособие*
5. *Системный администратор удаляет учетную запись*
6. *Менеджер персонала обновляет базу данных*

---

<sup>6</sup> Последовательность действий при конкретном выполнении алгоритма называется *протоколом* этого выполнения. Таким образом, сценарий можно рассматривать как протокол выполнения алгоритма варианта использования.

Казалось бы, что здесь неясного? А неясно, например, вот что: как должна вести себя система, если на шаге 2 начальник *не* подписывает заявление. Из текста сценария не только не ясен ответ, но, хуже того, при невнимательном чтении можно и не заметить, что есть вопрос.

Текстовые описания сценариев всем хороши: просты, всем понятны, легко и быстро составляются. Плохи они тем, что могут быть неполны и неточны, и эти недостатки незаметны.

### 2.3.2. Реализация программой на псевдокоде

Второй рассматриваемый нами способ реализации варианта использования — записать алгоритм на псевдокоде. Этот способ хорош тем, что понятен, привычен и доступен любому разработчику. Однако в настоящее время вряд ли можно рекомендовать такой способ реализации, как основной, по следующим причинам.

1. Реализация на псевдокоде плохо согласуется с современной парадигмой объектно-ориентированного программирования.

2. При использовании псевдокода теряются все преимущества использования UML: наглядная визуализация с помощью картинки, строгость и точность языка проектирования и реализации, поддержка распространенными инструментальными средствами.

3. Решения на псевдокоде практически невозможно использовать повторно.

Тем не менее, рассмотрим пример реализации вариантов использования на псевдокоде.

**Use case** Self Fire

Получить заявление

add\_payment:

Pay Compensation(Self Fire, add\_payment)

**Include** Delete Account

Обновить информацию в базе данных

**Use case** Adm Fire

Получить приказ

add\_payment:

Pay Compensation(Adm Fire, add\_payment)

**Include** Delete Account

Обновить информацию в базе данных

**Use case** Pay Compensation

if (add\_payment)

    if (from Self Fire)

        начислить за неиспользованный отпуск

    else if (from Adm Fire)

        начислить выходное пособие

Увольнение по собственному желанию запускается по инициативе сотрудника. Увольнение по инициативе администрации начинается с приказа об увольнении. В остальном последовательность действий в обоих случаях совпадает.

В этих текстах использовано ключевое слово Include, отражающее наличие зависимостей с таким стереотипом в модели. А именно, это означает, что в этом месте в текст псевдокода для данного варианта использования нужно включить текст псевдокода для варианта использования Delete Account, который мы здесь не приводим.

Вариант использования Pay Compensation запускается, если есть условия для выплаты компенсаций. При этом основные варианты использования не должны знать, каковы эти условия и как рассчитывается компенсация — за это отвечает вариант использования Pay Compensation. Зависимость со стереотипом «extend» означает, что псевдокод варианта использования Pay Compensation должен быть включен в текст основных вариантов использования. При этом вариант использования

Pay compensation должен знать, в какое место ему нужно включиться. Для этого в основных вариантах использования определена *точка расширения* (extension point) — по сути, просто метка в программе.<sup>7</sup> Этот пример в достаточной мере объясняет сходство и различие между зависимостями со стереотипом «include» и «extend».

### 2.3.3. Реализация диаграммами деятельности

Третий способ реализации варианта использования — описать алгоритм с помощью диаграммы деятельности. С одной стороны, диаграмма деятельности — это полноценная диаграмма UML, с другой стороны, диаграмма деятельности немногим отличается от блок-схемы (а тем самым и от псевдокода). Таким образом, реализация варианта использования диаграммой деятельности является компромиссным способом ведения разработки — в сущности, это проектирование сверху вниз в терминах и обозначениях UML.

Например, в информационной системе отдела кадров прием сотрудника может быть организован так, как показано на рис. 2.10.

Приблизило ли нас появление этой диаграммы в модели к завершению работы над системой? С одной стороны, вместо одной сущности, подлежащей реализации (вариант использования Hire Person) появилось пять новых: три сторожевых условия и две деятельности, которые в свою очередь теперь нуждаются в реализации. С другой стороны, каждая из этих новых сущностей кажется более простой и понятной, а значит быстрее и надежнее реализуемой. Кроме того, эту диаграмму можно показать заказчику, чтобы проверить, действительно ли проектируемая нами логика

---

<sup>7</sup> Для зависимости со стереотипом «include» никакой метки не нужно: место включения определяется тем, где в псевдокоде стоит ключевое слово **Include**.

работы системы соответствует тому бизнес-процессу, который существует в реальности.

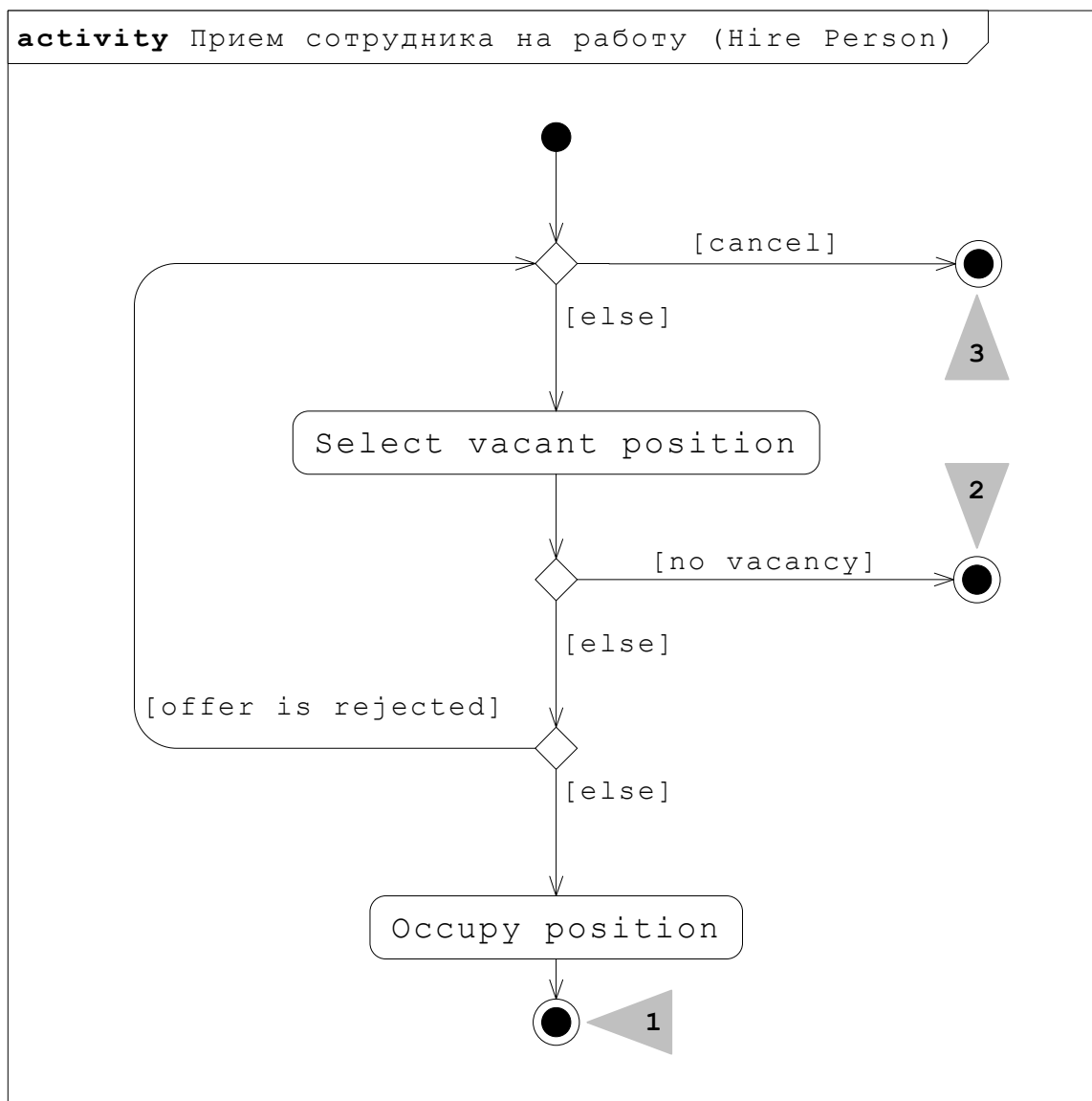


Рис. 2.10. Реализация варианта использования при помощи диаграммы деятельности

Применение диаграмм деятельности для реализации вариантов использования может привести к более глубокому пониманию существа задачи и даже открыть неожиданные возможности улучшения приложения, которые было трудно усмотреть в первоначальной постановке задачи.

Например, рассматривая (чисто формально) схему процесса на рис. 2.10, мы видим, что процесс может иметь три исхода.

1. Нормальное завершение (1 на рис. 2.1), которое предполагает обязательное выполнение деятельности Occupy position. Резонно предположить, что при выполнении этой деятельности в базу данных будет записана какая-то информация, что, несомненно, является значимым для пользователя результатом.

2. Исключительная ситуация (2 на рис. 2.10), возникающая в том случае, когда процесс не может быть нормально завершён, т. е. мы хотели принять человека на работу, и он был согласен, а текущее состояние штатного расписания не позволило этого сделать. Факт возникновения такой ситуации, хотя формально и не является значимым результатом для менеджера персонала, на самом деле может быть весьма важен для высшего руководства или менеджера штатного расписания.

3. Завершение процесса без достижения какого-либо видимого результата (3 на рис. 2.10). Например, менеджер персонала сделал кандидату какие-то предложения, но ни одно из них кандидата не устроило, после чего они расстались, как будто ничего и не было.

Этот простой анализ наталкивает на следующие соображения. Вариант использования **должен** доставлять значимый результат, значит, если результата нет, то что-то спроектировано не так, как нужно. Действительно, все практические информационные системы отдела кадров обязательно накапливают статистическую информацию обо всех **проведённых** кадровых операциях. Такая статистика необходима для анализа движения кадров. Однако далеко не все системы позволяют учитывать и **не проведённые** операции. Между тем, учёт причин, по которым кандидаты не принимают предложенной работы или, наоборот, причин, по которым организации отвергают кандидатов, может быть весьма полезен. Поэтому мы можем усовершенствовать нашу диаграмму, например, так, как показано на рис. 2.11.

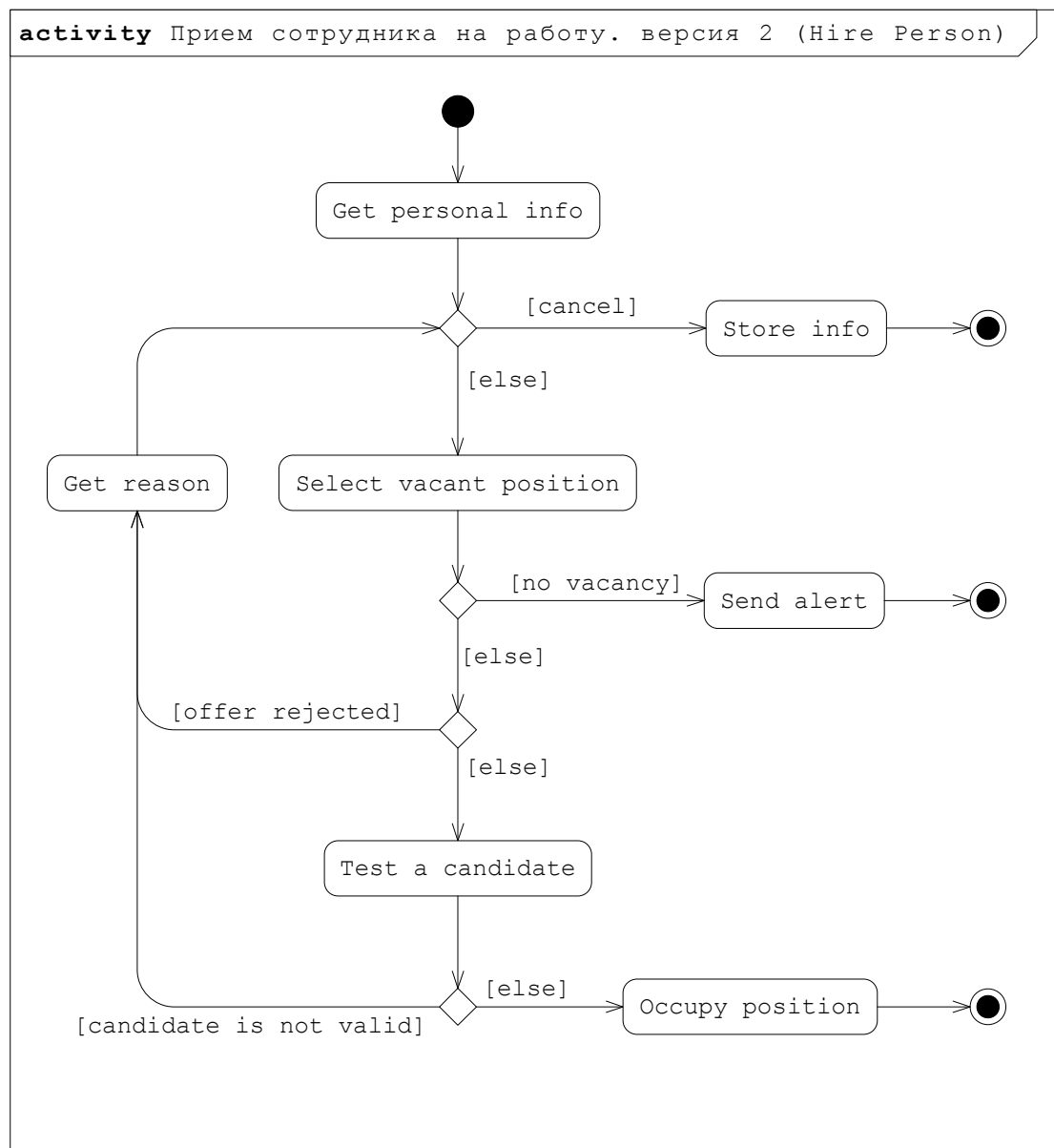


Рис. 2.11. Усовершенствованная реализация варианта использования

Вот теперь (формально) все хорошо: информация не теряется. Более того, имеет смысл вернуться к представлению использования и посмотреть, не нужно ли включить в модель новые варианты использования (может быть, как низкоприоритетные и подлежащие реализации в последующих версиях системы). Так реализация вариантов использования может приводить к изменению и усовершенствованию самих вариантов использования. **Моделирование имеет итеративный характер**, о чем мы уже говорили в главе 1.



### 2.3.4. Реализация диаграммами взаимодействия

Четвертый из основных способов реализации варианта использования — создать одну или несколько диаграмм взаимодействия в форме диаграмм коммуникации или диаграмм последовательности, которые описывают один или несколько сценариев данного варианта использования. Этот способ в наибольшей степени соответствует идеологии UML и рекомендуется авторами языка как основной и предпочтительный.

Рассмотрим пример реализации диаграммами взаимодействия варианта использования Hire Person (прием сотрудника на работу) информационной системы отдела кадров.

Сначала рассмотрим типовой сценарий, когда прием проходит безо всяких осложнений: есть вакантное рабочее место и кандидат готов его занять. Диаграмма последовательности для такого сценария приведена на рис. 2.12.

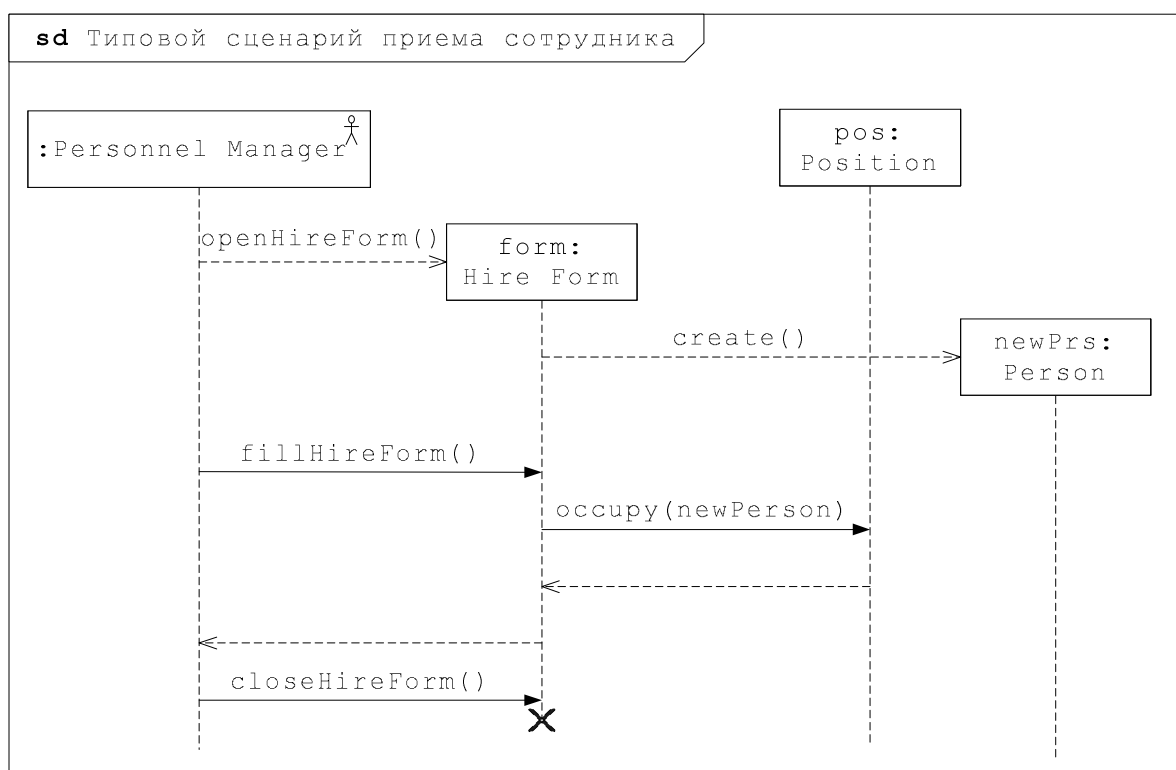


Рис. 2.12. Диаграмма последовательности для типового сценария

На приведенной диаграмме последовательность посылаемых сообщений примерно соответствует последовательности действий на диаграмме деятельности (см. рис. 2.11) в том случае, когда поток управления проходит по диаграмме сверху вниз один раз. Таким образом, диаграмма, представленная на рис. 2.12 до некоторой степени определяет типовой сценарий варианта использования Hire Person. Однако построив такую диаграмму, мы постулировали существование в системе некоторых классов, экземпляры которых должны взаимодействовать для обеспечения требуемого поведения моделируемого варианта использования. Здесь в нашей модели появились новые сущности:

- класс Hire Form, ответственный за интерфейс, необходимый для выполнения варианта использования прием сотрудника;
- класс Person, ответственный за хранение данных о конкретном сотруднике;
- класс Position, ответственный за хранение данных и выполнение операций с конкретной должностью.

Возвращаясь к нашему примеру, заметим, что диаграмма на рис. 2.12 семантически не полна: она не отражает все сценарии варианта использования, которые предусматриваются, например, на диаграмме на рис. 2.11. Как уже было сказано, в этом случае можно составить дополнительные диаграммы взаимодействия, реализующие альтернативные сценарии варианта использования. Например, на рис. 2.13 показан сценарий приема сотрудника, соответствующий исключительной ситуации, когда нет вакантных должностей. На этот раз мы описываем сценарий в форме диаграммы коммуникации.

Из материала этого раздела мы делаем следующий вывод: **реализация вариантов использования диаграммами взаимодействия является наиболее трудоемким и сложным методом, но этот метод лучше всего согласован с объектно-ориентированным подходом и в наибольшей мере приближает нас к конечной цели.**

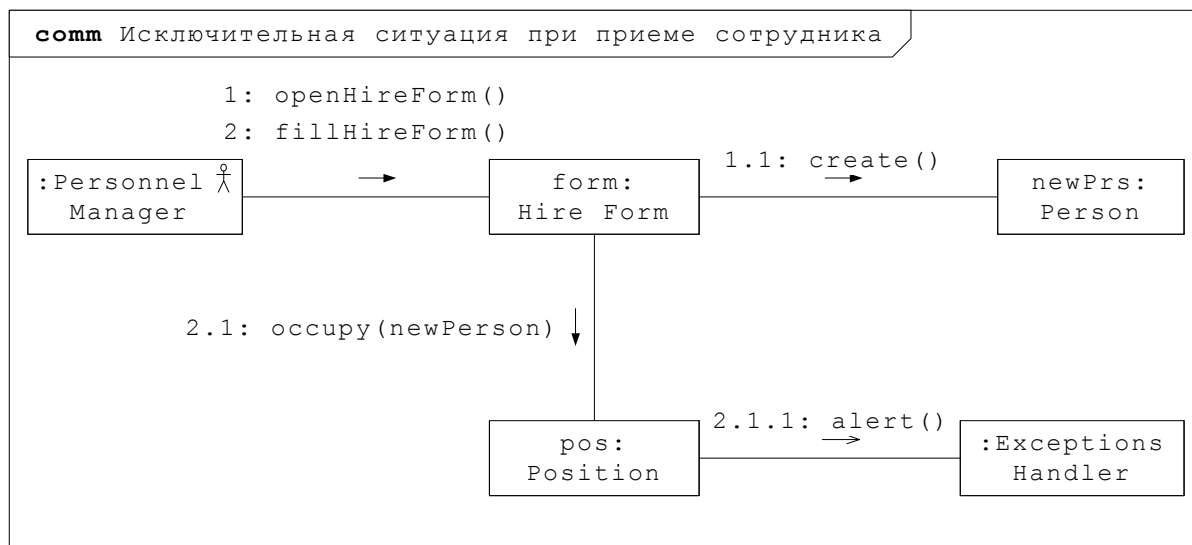


Рис. 2.13. Диаграмма кооперации для исключительной ситуации

## ВЫВОДЫ

1. Составление диаграмм использования — это первый шаг моделирования.
2. Основное назначение диаграммы использования — показать, что делает система во внешнем мире.
3. Диаграмма использования не зависит от программной реализации системы и поэтому не обязана соответствовать структуре классов, модулей и компонентов системы.
4. Идентификация действующих лиц и вариантов использования — ключ к дальнейшему проектированию.
5. В зависимости от выбранной парадигмы проектирования и программирования применяются различные способы реализации вариантов использования.

## **3. МОДЕЛИРОВАНИЕ СТРУКТУРЫ**

### **3.1. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ МОДЕЛИРОВАНИЕ СТРУКТУРЫ**

Моделируя структуру, мы описываем составные части системы и отношения между ними. UML в большинстве случаев применяется в качестве объектно-ориентированного языка моделирования, поэтому не удивительно, что основным видом составных частей, из которых состоит система при таком подходе, являются классы и отношения между ними.

#### **3.1.1. Назначение структурного моделирования**

Рассмотрим более детально, какие именно структуры нужно моделировать и зачем. Мы выделяем следующие структуры:

- структура связей между объектами во время выполнения программы;
- структура хранения данных;
- структура программного кода;
- структура компонентов в приложении;
- структура сложных объектов, состоящих из взаимодействующих частей;
- структура артефактов в проекте;
- структура используемых вычислительных ресурсов.

Наша классификация может быть не совсем полна и уж совсем не ортогональна (упомянутые структуры не являются независимыми, они связаны друг с другом), но в целом соответствует сложившейся практике разработки приложений, поскольку позволяет фиксировать основные решения, принимаемые в процессе проектирования и реализации. В этом разделе кратко мы обсудим назначение перечисленных структур и укажем средства UML, предназначенные для их моделирования.

**Структура связей между объектами во время выполнения программы.** В парадигме объектно-ориентированного программирования процесс выполнения программы состоит в том, что программные объекты взаимодействуют друг с другом, обмениваясь сообщениями. Наиболее распространенным типом сообщения является вызов метода объекта одного класса из метода объекта другого класса. Для моделирования структуры связей в UML используются отношения ассоциации на диаграмме классов.

**Структура хранения данных.** Программы обрабатывают данные, которые хранятся в памяти компьютера. В парадигме объектно-ориентированного программирования для хранения данных во время выполнения программы предназначены атрибуты классов. Однако большая часть приложений для автоматизации делопроизводства устроена так, что определенные данные (не все) должны храниться в памяти компьютера не только во время сеанса работы приложения, но постоянно, т. е. между сеансами.

Вопрос структуры хранения данных является первостепенным для приложений баз данных. К счастью, известны надежные методы решения этого вопроса, например, диаграммы "сущность—связь". Эти же методы (с точностью до обозначений) применяются и в UML в форме ассоциаций с указанием кратности полюсов.

**Структура программного кода.** Не секрет, что программы существенно отличаются по величине — бывают программы большие и маленькие. Удивительным является то, насколько велики эти различия: от сотен строк кода (и менее) до сотен миллионов строк (и более). Столь большие количественные различия не могут не проявляться и на качественном уровне. Действительно, для маленьких программ структура кода практически не имеет значения, для больших — наоборот, имеет едва ли не решающее значение. Поскольку UML не является языком программирования, модель не определяет структуру кода непосредственно, однако косвенным образом структура модели существенно влияет на структуру кода.

Большинство инструментов поддерживает полуавтоматическую генерацию кода для одного или нескольких, чаще объектно-ориентированных, языков программирования. В большинстве случаев классы модели транслируются в классы (или эквивалентные им конструкции) целевого языка. Кроме того, многие инструменты учитывают структуру пакетов в модели и транслируют ее в соответствующие "надклассовые" структуры целевой системы программирования. Таким образом, если задействовано средство автоматической генерации кода, то структура классов и пакетов в модели фактически полностью моделирует структуру кода приложения.

**Структура компонентов в приложении.** Приложение, состоящее из одной компоненты, имеет тривиальную структуру компонентов, моделировать которую нет нужды. Но большинство современных приложений на этапе проектирования представляют собой взаимосвязь многих компонентов, даже если и не являются распределенными. Компонентная структура предполагает описание двух аспектов: во-первых, как классы распределены по компонентам, во-вторых, как (через какие интерфейсы) компоненты взаимодействуют друг с другом. Оба эти аспекта моделируются диаграммами компонентов UML.

**Структура сложных объектов, состоящих из взаимодействующих частей.** Для моделирования этой структуры применяется новое средство UML 2 — диаграмма внутренней структуры классификатора. Данная диаграмма используется для описания внутренней структуры классов и компонентов. Существует еще одна сущность, которая также позволяет описать взаимодействие множества частей. Это сущность называется кооперацией и служит для описания взаимодействия в некотором контексте. С точки зрения внутренней структуры основное отличие кооперации от класса и компонента состоит в том, что кооперация не является владельцем своих частей, и соединители частей кооперации могут не иметь

явного выражения в виде ассоциации. Однако, как у классов и компонентов, у кооперации могут быть экземпляры, которые функционируют во время исполнения.

**Структура артефактов в проекте.** Только самые простые приложения состоят из одного артефакта — исполнимого кода программы. Большинство реальных приложений насчитывает в своем составе десятки, сотни и тысячи различных компонентов: исполнимых двоичных файлов, файлов ресурсов, файлов исходного кода, различных сопровождающих документов, справочных файлов, файлов с данными и т. д. Для большого приложения важно не только иметь точный и полный список всех артефактов, но и указать, какие именно из них входят в конкретный экземпляр системы. Дело в том, что для больших приложений в проекте сосуществуют разные версии одного и того же артефакта. Это исчерпывающим образом моделируется диаграммами компонентов и размещения UML, где предусмотрены стандартные стереотипы для описания артефактов разных типов.

**Структура используемых вычислительных ресурсов.** Приложение, состоящее из многих артефактов, как правило, бывает распределенным, т. е. различные артефакты размещаются на разных компьютерах. Диаграммы размещения позволяют включить в модель описание и этой структуры.

### 3.1.2. Классификаторы

Важнейшим типом дескрипторов являются классификаторы.

***Классификатор** (classifier) — это дескриптор множества однотипных объектов.*

Из этого определения непосредственно вытекает основное и характеристическое свойство классификатора: классификатор (прямо или косвенно) может иметь экземпляры.

В UML определено достаточно много классификаторов. Мы рассматриваем их частями. Во второй главе детально рассмотрены только два из них, а именно:

- действующее лицо (actor);
- вариант использования (use case).

Классификаторы, которые рассмотрены в этой главе, приведены ниже:

- артефакт (artifact);
- тип данных (data type);
- ассоциация (association);
- класс ассоциации (association class);
- интерфейс (interface);
- класс (class);
- кооперация (collaboration);
- компонент (component);
- узел (node).

Все классификаторы имеют некоторые общие свойства, которые используются в дальнейшем изложении. В этом параграфе мы опишем семь наиболее важных свойств классификаторов, которые нам понадобятся, прежде всего.

**Во-первых**, классификаторы (как и все элементы модели) имеют имена. Имя служит для идентификации элемента модели и потому должно быть уникально в данном пространстве имен.

**Во-вторых**, как уже было сказано, классификатор может иметь экземпляры. Экземпляры бывают прямые и косвенные.

*Если некоторый объект непосредственно порожден с помощью конструктора классификатора А, то этот объект называется **прямым экземпляром** (direct instance) классификатора А (1 рис. 3.1)<sup>8</sup>.*

*Если классификатор А является обобщением классификатора В или, что то же самое, классификатор В является специализацией классификатора А, то все экземпляры классификатора В являются **косвенными экземплярами** классификатора А (2 рис. 3.1).*

---

<sup>8</sup> Употребляют также термин "непосредственный экземпляр".



Данное свойство является транзитивным: если классификатор А является обобщением классификатора В, а классификатор В является обобщением классификатора С, то все экземпляры классификатора С также являются косвенными экземплярами А (3 рис. 3.1).

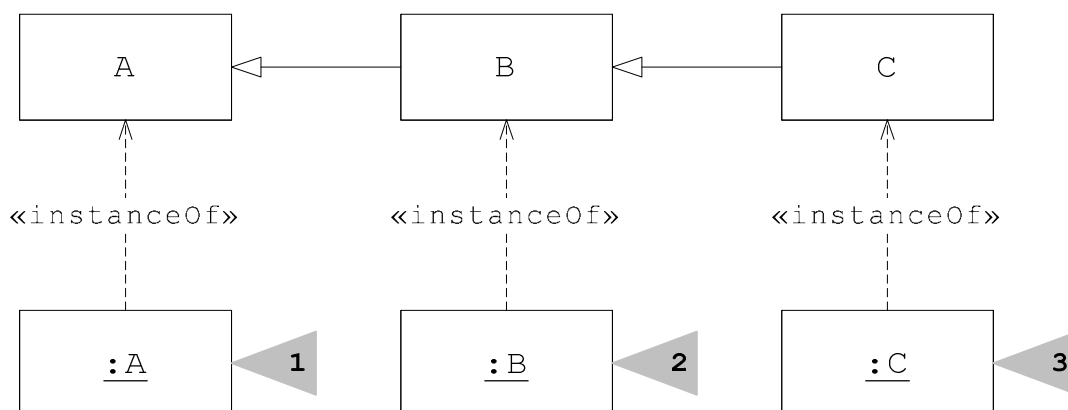


Рис. 3.1. Прямые и косвенные экземпляры классификатора А

**В-третьих**, классификатор может быть абстрактным или конкретным.

**Абстрактный** (*abstract*) классификатор не может иметь прямых экземпляров и в этом случае его имя выделяется курсивом<sup>9</sup>.

**Конкретный** (*concrete*) классификатор может иметь прямые экземпляры и в этом случае его имя записывается прямым шрифтом.

Абстрактный классификатор — это такой дескриптор множества объектов, в котором нет прямого описания элементов множества, но данный классификатор связан отношением обобщения с другими классификаторами и объединение множеств их экземпляров считается множеством экземпляров данного абстрактного классификатора. Другими словами, множество определяется не прямо, а через совокупность подмножеств. Например, интерфейс, будучи абстрактным классом, не может иметь

<sup>9</sup> Из данного правила есть исключение, а именно, если классификатор имеет стереотип «interface», то его имя не нужно выделять курсивом.

непосредственных экземпляров, но реализующий его класс может, стало быть, интерфейс является классификатором.

**В-четвертых**, классификатор (как и другие элементы модели) имеет видимость.

**Видимость** (visibility) определяет, может ли составляющая одного классификатора (в том числе имя) использоваться в другом классификаторе.

Другими словами, если в определенном контексте нечто доступно и может быть как-то использовано, то оно является видимым (в этом контексте). Если же оно не видимо, то и не может быть использовано. Видимость является свойством всех элементов модели (хотя не для всех элементов это свойство является существенным). Видимость может принимать одно из четырех значений:

- *открытый* (обозначается знаком + или ключевым словом public);

- *защищенный* (обозначается знаком # или ключевым словом protected);

- *закрытый* (обозначается знаком – или ключевым словом private).

- *пакетный* (обозначается знаком ~ или ключевым словом package).

Открытый элемент модели является видимым везде, где является видимым содержащий его элемент<sup>10</sup>. Например, открытый атрибут класса виден везде, где виден сам класс.

Защищенный элемент модели виден как в элементе его содержащем (контейнере), так и во всех элементах, для которых контейнер является обобщением. Например, защищенный атрибут класса виден в содержащем его классе и во всех подклассах.

---

<sup>10</sup> Элемент, который содержит другие элементы, часто называют *контейнером* (container).

Закрытый элемент модели виден только в элементе, которому он принадлежит. Например, закрытый атрибут класса виден только в этом классе.

Элемент модели со значением видимости пакетный, виден элементам только того пакета, в котором он сам определен.

**В-пятых**, все составляющие классификатора имеют область действия.

*Область действия (scope) определяет, как проявляет себя составляющая классификатора в экземплярах, т. е. имеют экземпляры свои значения составляющей или совместно используют одно значение.*

Область действия имеет два возможных значения:

- *экземпляр (instance)* — никак специально не обозначается, поскольку подразумевается по умолчанию;

- *классификатор (classifier)* — описание составляющей классификатора подчеркивается.

Если областью действия составляющей является экземпляр, то каждый экземпляр классификатора имеет свое значение составляющей. Например, областью действия атрибута по умолчанию является экземпляр. Это означает, что каждый объект — экземпляр класса — имеет свое собственное значение атрибута, которое может меняться независимо от значений данного атрибута других объектов, экземпляров этого же класса. Если областью действия составляющей является классификатор, то все экземпляры классификатора совместно используют одно значение составляющей. Например, конструктор обычно имеет областью действия классификатор (класс), поскольку является процедурой, общей для всех экземпляров данного класса.

**В-шестых**, классификатор имеет *кратность*, т. е. ограничение на количество экземпляров классификатора, как множества <sup>11</sup>. Не

---

<sup>11</sup> Кратность может быть не только у классификаторов, но также у атрибутов и полюсов ассоциаций (см. параграф 3.2.2).

следует путать кратность с количеством элементов (экземпляров). Множество, указанное в модели, во время выполнения может иметь различное количество элементов, и количество элементов может динамически меняться. Кратность определяет пределы этих изменений.

***Кратность** (multiplicity) множества — это множество чисел, которые задают все допустимые значения мощности для данного множества.*

Синтаксически кратность задается выражением, которое является непустой последовательностью элементов (разделенных запятыми), каждый из которых имеет следующий формат.

Нижняя граница .. ВЕРХНЯЯ ГРАНИЦА

В качестве верхний и нижней границы используются натуральные числа или ноль. Кроме того, в качестве верхней границы может использоваться символ \*. Если нижняя граница не задана, то она опускается вместе с символом .. (две точки). В табл. 3.1 приведены некоторые примеры выражений кратности.

Таблица 3.1

**Выражения кратности**

<b>Выражение кратности</b>	<b>Множество может иметь</b>
0..*	Произвольное число элементов
*	Произвольное число элементов (по определению эквивалентно предыдущему)
1..*	Один или более элементов
0..1	Не более одного элемента
1..10	От одного до десяти элементов
1..3, 5, 7..10	Один, два, три, пять, семь, восемь, девять или десять элементов
5..3	Некорректная кратность. Нижняя граница больше верхней
-1..3	Некорректная кратность. Отрицательные числа недопустимы

Обычно на практике используются следующие варианты кратности классификаторов.

И, наконец, **в-седьмых**, классификаторы (и только они!) могут участвовать в отношении *обобщения*.

## 3.2. СУЩНОСТИ НА ДИАГРАММЕ КЛАССОВ

На диаграммах классов в качестве сущностей применяются, прежде всего, классы, как в своей наиболее общей форме, так и в форме многочисленных стереотипов и частных случаев: интерфейсы, типы данных, активные классы и др. Кроме того, на диаграмме классов могут использоваться (как и везде) пакеты и примечания.

### 3.2.1. Классы

Класс — один из самых "богатых" элементов моделирования UML. Описание класса может включать множество различных элементов, и чтобы они не путались, в языке предусмотрено группирование элементов описания класса по *секциям* (compartment). Стандартных секций три:

- *секция имени* — наряду с обязательным именем может содержать также стереотип, кратность и список именованных значений;
- *секция атрибутов* — содержит список описаний атрибутов класса;
- *секция операций* — содержит список описаний операций класса.

Как и все основные сущности UML, класс обязательно имеет имя, а стало быть, секция имени не может быть опущена. Прочие секции могут быть пустыми или отсутствовать вовсе. Наряду со стандартными секциями, описание класса может содержать и произвольное количество дополнительных секций. Семантически дополнительные секции эквиваленты примечаниям.

Нотация классов очень проста — это всегда прямоугольник. Если секций более одной, то внутренность прямоугольника делится горизонтальными линиями на части, соответствующие секциям (рис. 3.2).

ClassName
+attribute -privateAttr -fio:String="Novikov" -array:char[10]
+operationName() <u>-staticOperation()</u> +function():int

Рис. 3.2. Типичная нотация класса

Содержимым секции в любом случае является текст. Текст внутри стандартных секций должен иметь определенный синтаксис.

Секция имени класса в общем случае имеет следующий синтаксис.

«стереотип» ИМЯ {свойства} кратность

Имени класса может предшествовать стереотип. В табл. 3.2 перечислены стандартные стереотипы классов.

Таблица 3.2

**Стандартные стереотипы классов**

Стереотип	Описание
«actor»	Действующее лицо
«enumeration»	Перечислимый тип данных
«exception»	Исключение (только в UML 1)
«interface»	Все составляющие абстрактные
«metaclass»	Экземпляры являются классами
«signal»	Класс, экземплярами которого являются сигналы
«stereotype»	Новый элемент на основе существующего
«dataType»	Тип данных
«utility»	Нет экземпляров, служба

Обязательное имя класса может быть выделено *курсивом* и в этом случае данный класс является абстрактным, т. е. не имеющим непосредственных экземпляров.

Класс, а также отдельные элементы его описания могут иметь произвольные заданные пользователем ограничения и именованные значения. Кратность класса задается по общим правилам.

### 3.2.2. Атрибуты

*Атрибут* — это именованное место (или, как говорят, *слот*), в котором может храниться значение.

Атрибуты класса перечисляются в секции атрибутов. В общем случае описание атрибута имеет следующий синтаксис.

```
видимость ИМЯ кратность : тип = начальное_значение  
{ свойства }
```

Видимость, как обычно, обозначается знаками +, −, #, ~. Еще раз подчеркнем, что если видимость не указана, то никакого значения видимости по умолчанию не подразумевается.

**Если имя атрибута подчеркнуто, то это означает, что областью действия данного атрибута является класс, а не экземпляр класса, как обычно.** Другими словами, все объекты — экземпляры этого класса совместно используют одно и то же значение данного атрибута, общее для всех экземпляров. В обычной ситуации (нет подчеркивания) каждый экземпляр класса хранит свое индивидуальное значение атрибута.

Кратность, если она присутствует, определяет данный атрибут как массив (определенной или неопределенной длины).

Тип атрибута — это либо примитивный (встроенный) тип, либо тип, определенный пользователем.

Начальное значение имеет очевидный смысл: при создании экземпляра данного класса атрибут получает указанное значение. Заметим, что если начальное значение не указано, то никакого значения по умолчанию не подразумевается. Если нужно, чтобы атрибут обязательно имел значение, то об этом должен позаботиться конструктор класса.

Как и любой другой элемент модели, атрибут может быть наделен дополнительными свойствами в форме ограничений и именованных значений.

Например, в информационной системе отдела кадров класс *Person*, скорее всего, должен иметь атрибут, хранящий имя

сотрудника. В табл. 3.3 приведен список примеров описаний такого атрибута. Все описания синтаксически допустимы и могут быть использованы в соответствии с текущим уровнем детализации модели.

Таблица 3.3

**Примеры описаний атрибутов**

<b>Пример</b>	<b>Пояснение</b>
name	Минимальное возможное описание — указано только имя атрибута
+name	Указаны имя и открытая видимость — предполагается, что манипуляции с именем будут производиться непосредственно
-name : String	Указаны имя, тип и закрытая видимость — манипуляции с именем будут производиться с помощью специальных операций
-name[1..3] String	: В дополнение к предыдущему указана кратность (для хранения трех составляющих; фамилии, имени и отчества)
-name String="Novikov"	: Дополнительно указано начальное значение
+name String{readOnly}	: Атрибут объявлен не меняющим своего значения после начального присваивания и открытым

### 3.2.3. Операции и методы

**Операция** — это спецификация действия с объектом: изменение значения его атрибутов, вычисление нового значения по информации, хранящейся в объекте и т. д.

Объявление конкретной операции в классе подразумевает наличие метода в этом же классе. Исключением является ситуация, когда операция объявлена абстрактной и ее реализация содержится в подклассах.



*Метод* — это реализация операции, т. е. выполняемый алгоритм.

Выполнение действий, определяемых операцией, инициируется вызовом метода.

При вызове метода могут, в свою очередь, быть вызваны методы этого же, а также других классов.

Описания операций класса перечисляются в секции операций и имеют следующий синтаксис.

видимость ИМЯ (параметры) : тип {свойства}

Здесь слово параметры обозначает последовательность описаний параметров операции, каждое из которых имеет следующий формат.

направление ПАРАМЕТР : тип = значение

Начнем по порядку. Видимость, как обычно, обозначается с помощью знаков +, −, #, ~ или с помощью ключевых слов `private`, `public`, `protected`, `package`. Подчеркивание имени означает, что область действия операции — класс, а не объект. Например, конструкторы имеют область действия класс. Курсивное написание имени означает, что операция абстрактная, т. е. в данном классе ее реализация не задана и должна быть задана в подклассах данного класса. После имени в скобках может быть указан список описаний параметров. Описания параметров в списке разделяются запятой. Для каждого параметра обязательно указывается имя, а также могут быть указаны направление передачи параметра, его тип и значение аргумента по умолчанию.

Направление передачи параметра в UML описывает семантическое назначение параметров, не конкретизируя конкретный механизм передачи. Как именно следует трактовать указанные в модели направления передачи параметров, зависит от используемой системы программирования. Возможные значения направления передачи приведены в табл. 3.4.

Таблица 3.4

**Ключевые слова для описания направления передачи параметров**

Ключевое слово	Назначение параметра
in	Входной параметр — аргумент должен быть значением, которое используется в операции, но не изменяется
out	Выходной параметр — аргумент должен быть хранилищем, в которое операция помещает значение
inout	Входной и выходной параметр — аргумент должен быть хранилищем, содержащим значение. Операция использует переданное значение аргумента и помещает в хранилище результат
return	Значение, возвращаемое операцией. Такое значение направления передачи устанавливается автоматически для возвращаемого значения

Типом параметра операции, равно как и тип возвращаемого операцией значения может быть любой встроенный тип или определенный в модели класс, интерфейс или тип данных.

*Все вместе (имя операции, параметры и тип результата) обычно называют **сигнатурой** (signature) операции.*

Рассмотрим примеры описания возможных операций класса Person информационной системы отдела кадров в табл. 3.5.

Таблица 3.5

**Примеры описания операций**

Пример	Пояснение
move	Минимальное возможное описание — указано только имя операции
+move(in from, in to)	Указаны видимость операции, направления передачи и имена параметров
+move(in from:Department, in to:Department)	Подробное описание сигнатуры: указаны видимость операции, направления передачи, имена и типы параметров
+getName():String{isQuery}	Функция, возвращающая значение атрибута и не имеющая побочных эффектов

В отличие от операции, которая может быть *абстрактно*, т. е. не иметь реализующего метода и *конкретной*, для которой метод определен, в UML не предусмотрена отдельная нотация для описания *самого* метода. Как и во многих других подобных случаях, не нашедших отражение в нотации, использование примечания может служить допустимой заменой.

### 3.2.4. Интерфейсы и типы данных

В UML имеется несколько частных случаев классификаторов, которые, подобно классам, предназначены для моделирования структуры, но обладают рядом специфических особенностей. Наиболее важными из них являются интерфейсы и типы данных.

***Интерфейс** — это именованный набор составляющих, описывающий контракт между поставщиками и потребителями услуг.*

Другими словами, интерфейс — это абстрактный класс, в котором все составляющие — атрибуты и операции — абстрактны.

Поскольку интерфейс — это абстрактный класс, он не может иметь непосредственных экземпляров.

Следующая тема для обсуждения — типы данных.

***Тип данных** — это совокупность двух вещей: множества значений (может быть очень большого или даже потенциально бесконечного) и конечного множества операций, применимых к данным значениям.*

В модели UML можно использовать три вида типов данных.

**Примитивные типы** (PrimitiveType), которые считаются предопределенными в UML. Таковыми являются следующие: целочисленный тип (Integer), булевский тип (Boolean), строковый тип (String). Существует еще один дополнительный тип, который описывает множество (может быть бесконечное) натуральных чисел (UnlimitedNatural). Используется этот тип в основном для указания кратности той или иной сущности. Инструменты вправе расширять этот набор и использовать другие подходящие названия.

**Типы данных, которые определены в языке программирования**, поддерживаемым инструментом. Это могут быть как названия встроенных типов, так и сколь угодно сложные выражения, доставляющие тип, если таковые допускаются языком.

**Типы данных, которые определены в модели пользователем.** Данные типы представляются в виде классификаторов со стереотипом «enumeration» или «dataType».

Особого внимания заслуживает *перечислимый тип данных* (enumeration). Например, тип Boolean определен в UML как перечислимый тип со значениями true и false. Если в проектируемом приложении нужно использовать не обычную двужначную логику, а трехзначную, то тогда соответствующий тип можно определить так, как показано на рис. 3.3.

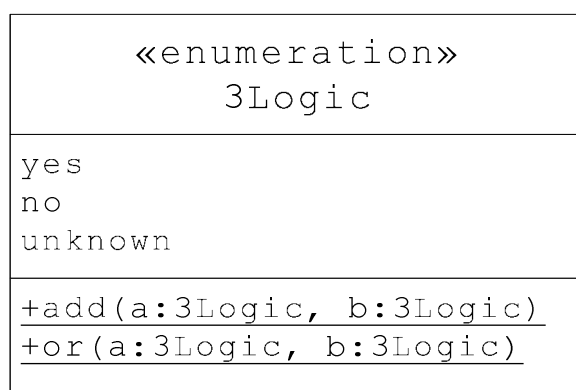


Рис. 3.3. Перечислимый тип данных «Трехзначная логика»

### 3.3. ОТНОШЕНИЯ НА ДИАГРАММЕ КЛАССОВ

Сущности на диаграммах классов связываются главным образом отношениями ассоциации (в том числе агрегирования и композиции) и обобщения. Отношения зависимости и реализации на диаграммах классов применяются реже, но, тем не менее, они также применяются, и мы начнем с них, как с более простых.

### 3.3.1. Отношения зависимости и реализации

Всего в UML определено довольно большое количество стандартных стереотипов отношения *зависимости*, которые можно разделить на несколько групп:

- между классами и объектами на диаграмме классов;
- между пакетами;
- между вариантами использования;
- другие.

Здесь рассматриваются зависимости первой группы, которые перечислены в табл. 3.6.

Таблица 3.6

**Стандартные стереотипы зависимостей на диаграмме классов**

Стереотип	Описание
«call»	Указывает зависимость между двумя операциями: операция зависимого класса вызывает операцию независимого класса.
«derive»	Буквально означает "может быть вычислен по". Зависимость с данным стереотипом применяется не только к классам, но и к другим элементам модели: атрибутам, ассоциациям и т. д. Суть состоит в том, зависимый элемент может быть восстановлен по информации, содержащейся в независимом элементе. Таким образом, данная зависимость показывает, что зависимый элемент, вообще говоря, излишен и введен в модель из соображений удобства, наглядности и т. д.
«instance Of»	Указывает, что зависимый объект (или класс) является экземпляром независимого класса (метакласса).
«instantiation»	Указывает, что операции независимого класса создают экземпляры зависимого класса.
«power type»	Показывает, что экземплярами зависимого класса являются подклассы независимого класса. Таким образом, в данном случае зависимый класс является метаклассом.
«refine»	Указывает, что зависимый класс уточняет (конкретизирует) независимый. Данная зависимость показывает, что связанные классы концептуально совпадают, но находятся на разных уровнях абстракции.

Рассмотрим отношение реализации. Между интерфейсами и другими классификаторами, в частности, классами, на диаграмме классов применяются два отношения:

- классификатор (в частности, класс) использует интерфейс — это показывается с помощью зависимости со стереотипом «call»;
- классификатор (в частности, класс) реализует интерфейс — это показывается с помощью отношения реализации.

Никаких ограничений на использование отношения реализации не накладывается: класс может реализовывать много интерфейсов, и наоборот, интерфейс может быть реализован многими классами. Нет ограничений и на использование зависимостей со стереотипом «call» — класс может вызывать любые операции любых видимых интерфейсов. Семантика зависимости со стереотипом «call» очень проста — эта зависимость указывает, что в операциях класса, находящегося на независимом полюсе, вызываются операции класса находящегося на зависимом полюсе.

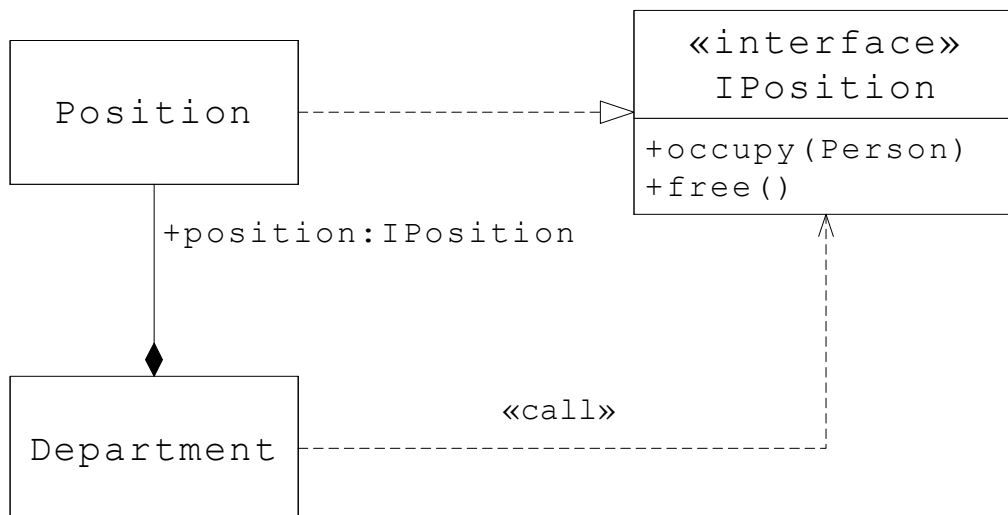


Рис. 3.4. Отношения реализации и использования интерфейсов

Рассмотрим пример из информационной системы отдела кадров. Допустим, что класс Department для реализации операций связанных с движением кадров, использует операции класса

Position, позволяющие занимать и освобождать должность — другие операции класса Position классу Department не нужны. Для этого, как показано на рис. 3.4 можно определить соответствующий интерфейс IPosition и связать его отношениями с данными классами.

### 3.3.2. Отношение обобщения

Отношение *обобщения* часто применяется на диаграмме классов. Действительно, трудно представить себе ситуацию, когда между классами в одной системе нет ничего общего. Как правило, общее есть, и это общее целесообразно выделить в отдельный класс. При этом общие составляющие, собранные в суперклассе, автоматически наследуются подклассами.

Таким образом, сокращается суммарное количество описаний, а значит, уменьшается вероятность допустить ошибку. Использование обобщений не ограничивает свободу проектировщика системы, поскольку унаследованные составляющие можно переопределить в подклассе. Для указания того, что та или иная составляющая переопределена в подклассе следует использовать появившееся в UML 2 дополнение *redefines* (см. рис. 3.5) .

По умолчанию обобщения являются подстановочными (*substitutable*), т. е. удовлетворяют принципу подстановки. Рассмотрим пример.

## ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

*Каждая структурная единица предприятия (подразделение, должность) должна иметь свое название.*

В информационной системе отдела кадров мы выделили классы Position, Department и Person. Для всех этих классов может быть указан атрибут, содержащий собственное имя объекта, выделяющее его в ряду однородных. Для простоты положим, что такой атрибут имеет тип String. В таком случае можно определить суперкласс,

ответственный за хранение данного атрибута и работу с ним, а прочие классы связать с суперклассом отношением обобщения. Однако более пристальный анализ предметной области наводит на мысль, что работа с собственным именем для выделенных классов производится не совсем одинаково. Действительно, назначение и изменение собственных имен подразделениям и должностям находится в пределах ответственности информационной системы отдела кадров, но назначение (тем паче изменение) собственного имени сотрудника явно выходит за эти пределы. Исходя из этих соображений, мы приходим к структуре обобщений, представленной на рис. 3.5. Операция `setName()`, объявленная в классе *Unit* переопределена для класса *Person*. На это указывает дополнение `redefines` (1). Переопределение состоит в том, что значение видимости для операции `setName()` изменено с “открытая” на “закрытая”. Обратите внимание, что суперкласс *Unit* определен как абстрактный, поскольку в системе не предполагается иметь объекты данного класса. Экземпляры существуют для конкретных подклассов *Department*, *Position* и *Person*.

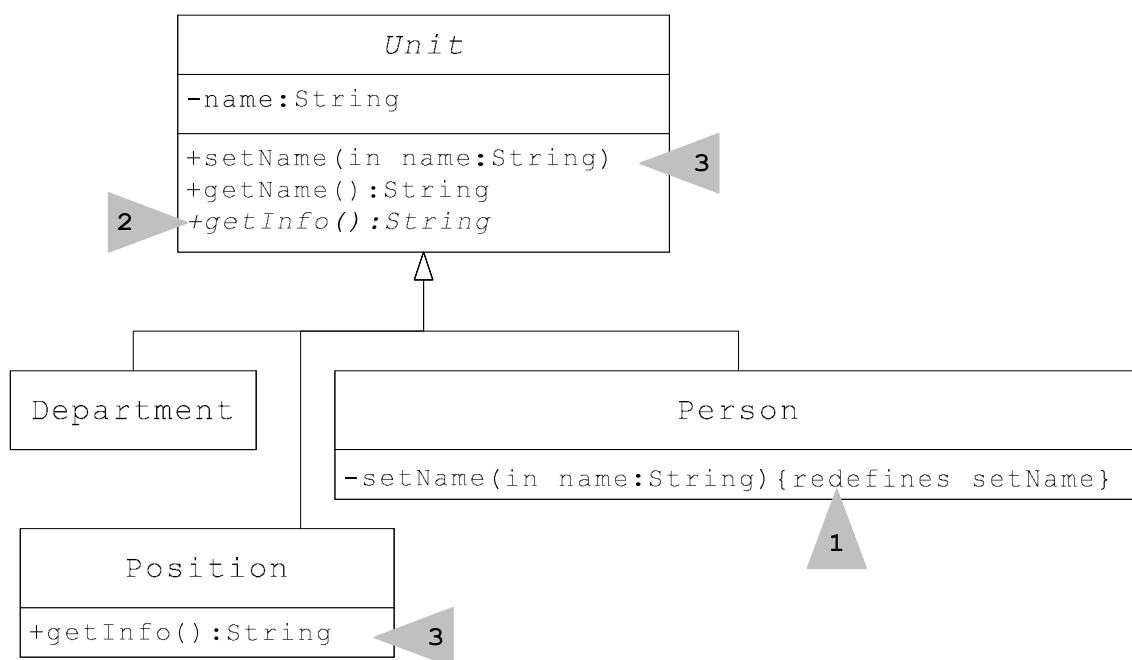


Рис. 3.5. Отношение обобщения



Отношения обобщения можно задавать в UML довольно свободно, но не совсем произвольно. Обобщения в модели должны образовывать строгий частичный порядок.

При *множественном наследовании* (multiple inheritance) возможны конфликты: суперклассы содержат составляющие, которые невозможно включить в один подкласс, например, атрибуты с одинаковыми именами, но разными типами. В UML конфликты при множественном наследовании считаются нарушением правил непротиворечивости модели. Если же конфликты отсутствуют, то множественное наследование в UML не только не запрещается, но даже поощряется! В частности, метамодели в стандарте изобилуют примерами множественного наследования.

### 3.3.3. Ассоциации и их дополнения

Отношение *ассоциации* является, видимо, самым важным на диаграмме классов. В общем случае ассоциация, нотация которой — сплошная линия, соединяющая классы, означает, что экземпляры одного класса связаны с экземплярами другого класса. Поскольку экземпляров может быть много, и каждый может быть связан с несколькими, ясно, что ассоциация является дескриптором, который описывает множество наборов связанных объектов. В UML ассоциация является классификатором, экземпляры которого называются связями.

*Связь (link)* — это экземпляр ассоциации (или соединителя), который представляет собой упорядоченный набор (*кортеж*, tuple) ссылок на экземпляры классификаторов на полюсах ассоциации.

Как уже было сказано, базовая нотация ассоциации (сплошная линия) позволяет указать, что объекты ассоциированных классов могут взаимодействовать во время выполнения. Но это только малая часть того, что можно моделировать с помощью отношения ассоциации. Для ассоциации в UML предусмотрено наибольшее количество различных дополнений, которые мы сначала перечислим, а потом рассмотрим по порядку. Дополнения, как обычно, не

являются обязательными: их используют при необходимости, в различных ситуациях по-разному. Если использовать все дополнения сразу, то диаграмма становится настолько перегруженной, что ее трудно читать. Итак, для ассоциации определены следующие дополнения:

- имя ассоциации (возможно, вместе с направлением чтения);
- кратность полюса<sup>12</sup> ассоциации;
- агрегации или композиция;
- возможность навигации для полюса ассоциации;
- роль полюса ассоциации;
- класс ассоциации;
- квалификатор полюса ассоциации.

Рассмотрим их по порядку.

#### **3.3.4. Имя ассоциации. Кратность полюса ассоциации**

*Имя ассоциации* указывается в виде строки текста над (или под, или рядом с) линией ассоциации. Имя не несет дополнительной семантической нагрузки, а просто позволяет различать ассоциации в модели. Обычно имя указывают в случаях многополюсных ассоциаций или, когда одна и та же группа классов связана несколькими различными ассоциациями. Однако строгого правила на этот счет нет.

Например, в информационной системе отдела кадров, если сотрудник занимает должность, то соответствующие экземпляры классов *Person* и *Position* должны быть связаны, т.е. между самими классами должно быть отношение ассоциации (1 рис. 3.15) и может быть имя (2), поясняющее ее назначение. Дополнительно можно указать направление чтения имени ассоциации (3). Фрагмент графической модели, приведенный на рис. 3.6, фактически можно прочесть вслух: *Person occupies Position*.

---

<sup>12</sup> Напомним, что полюсом называется конец линии ассоциации. Обычно используются двухполюсные (бинарные) ассоциации, но могут быть и многополюсные.

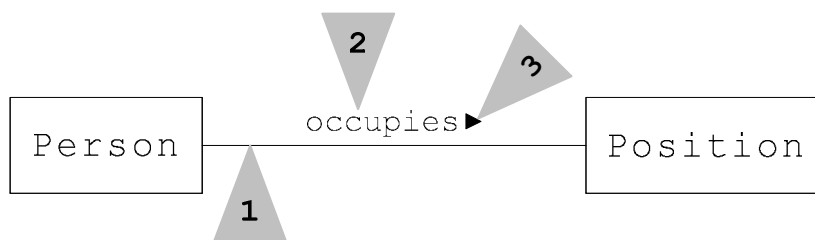


Рис. 3.6. Имя ассоциации и направление чтения

*Кратность полюса ассоциации* указывает, сколько объектов данного класса (со стороны данного полюса) участвуют в связи. Кратность может быть задана как конкретное число, и тогда в каждой связи со стороны данного полюса участвует ровно столько объектов, сколько указано. Более распространен случай, когда кратность указывается как диапазон возможных значений, и тогда число объектов, участвующих в связи должно находиться в пределах указанного диапазона. При указании кратности можно использовать символ \*, который обозначает неопределенное число (см. табл. 3.1). Например, если в информационной системе отдела кадров не предусматривается дробление ставок и совмещение должностей, то работающему сотруднику соответствует одна должность (1 рис. 3.16), а должности соответствует один сотрудник или ни одного (2) (должность вакантна). На рис. 3.7 приведен соответствующий фрагмент диаграммы UML.



Рис. 3.7. Кратность полюсов ассоциации

Более сложные случаи также легко моделируются с помощью кратности полюсов. Например, если мы хотим предусмотреть совмещение должностей и хранить информацию даже о

неработающих сотрудников, то диаграмма примет вид, приведенный на рис. 3.8 (запись \* эквивалентна записи 0..\*).



Рис. 3.8. Использование неопределенной кратности

### 3.3.5. Агрегация и композиция

В UML используются два частных, но очень важных случая отношения ассоциации, которые называются агрегацией и композицией. В обоих случаях речь идет о моделировании отношения типа "часть – целое". Ясно, что отношения такого типа следует отнести к отношениям ассоциации, поскольку части и целое обычно взаимодействуют.

**Агрегация** (*aggregation*) — это ассоциация между классом *A* (часть) и классом *B* (целое), которая означает, что экземпляры (один или несколько) класса *A* входят в состав экземпляра класса *B*.

Это отмечается с помощью специального графического дополнения: на полюсе ассоциации, присоединенному к «целому», изображается незакрашенный ромб (1). Например, на рис. 3.9 указано, что сотрудник является членом рабочей группы (Workgroup).

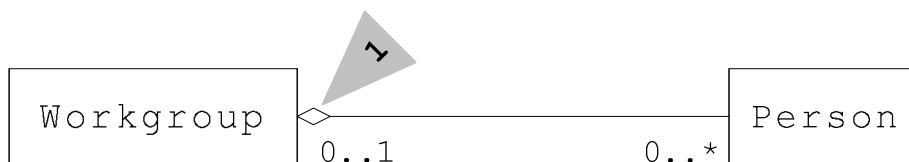


Рис. 3.9. Отношение агрегации

При этом никаких дополнительных ограничений не накладывается: экземпляр класса *Person* (часть) может быть связан с другими объектами (т. е. класс *Person* может участвовать в

нескольких агрегациях), создаваться и уничтожаться независимо от экземпляров класса Workgroup (целого).

**Композиция** (Composition) — это ассоциация между классом А (часть) и классом В (целое), которая дополнительно накладывает более сильные ограничения в сравнении с агрегацией: композиционно часть А может входить только в одно целое В, часть существует, только пока существует целое и прекращает свое существование вместе с целым.

Однако часть может быть отделена от целого до того, как оно будет удалено. В указанном случае композиция будет разрушена.

Графически отношение композиции отображается закрашенным ромбом (1 рис. 3.19). Для примера на рис. 3.10 приведен еще один взгляд на отношения между рабочими группами и сотрудниками в информационной системе отдела кадров. В этом случае, мы считаем, что в организации принята жесткая ("армейская") структура: каждый сотрудник входит ровно в одну рабочую группу и в каждой рабочей группе есть, по меньшей мере, один сотрудник. Для моделирования такой структуры используется композиция. Если же структура более аморфна: возможны "висящие в воздухе" сотрудники, бывают "пустые" рабочие группы и т. д., то для моделирования такой структуры более адекватным средством является агрегация (см. рис. 3.9).

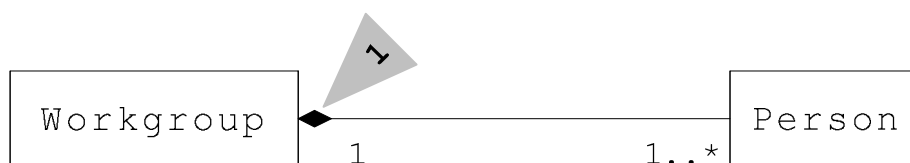


Рис. 3.10. Отношение композиции

В комбинации с указанием кратности, отношения ассоциации, агрегации и композиции позволяют лаконично и полно отобразить структуру классов: что из чего состоит и как связано. На рис. 3.11

приведен пример одного из вариантов такой структуры для информационной системы отдела кадров.

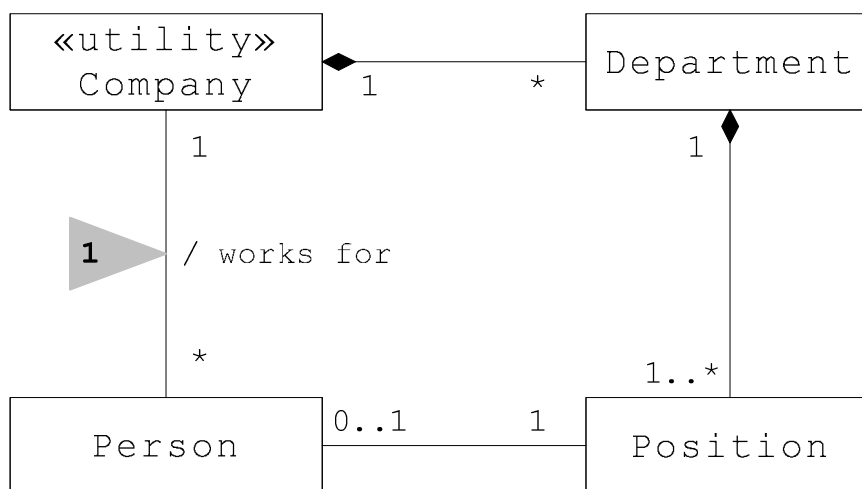


Рис. 3.11. Структура связей классов информационной системы отдела кадров

Продолжим рассмотрение информационной системы отдела кадров.

### ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

*Информационная система отдела кадров должна поддерживать иерархическую структуру подразделений на предприятии.*

Решение, которое позволяет легко учесть новое требование, приведено на рис. 3.12.

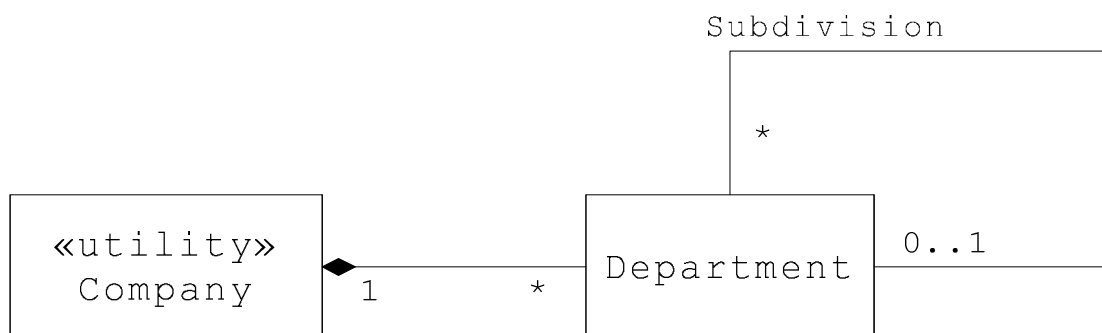


Рис. 3.12. Пример использования композиции

### 3.3.6. Роль полюса ассоциации

*Роль (role) — это интерфейс, который предоставляет классификатор в данной ассоциации.*

Напомним, что полюс ассоциации — это точка соприкосновения линии ассоциации с прямоугольником класса. Именно вблизи этой точки располагаются многочисленные дополнения полюсов ассоциации.

*Роль полюса ассоциации (association end role), называемая также спецификатором интерфейса — это способ указать, как именно участвует классификатор (присоединенный к данному полюсу ассоциации) в ассоциации.*

Нотация этого дополнения — текст, указанный на полюсе ассоциации. В общем случае роль полюса ассоциации имеет следующий синтаксис:

ВИДИМОСТЬ ИМЯ : тип

Имя является обязательным, оно называется *именем роли* и фактически является собственным именем полюса ассоциации, позволяющим различать полюса. Если рассматривается одна ассоциация, соединяющая два различных класса, то в именах ролей нет нужды: полюса ассоциации легко можно различить по именам классов, к которым они присоединены. Однако, если это не так, т. е. если два класса соединены несколькими ассоциациями, или же если ассоциация соединяет класс с самим собой, то указание ролей полюсов ассоциации является необходимым. Приведем еще один пример (см. рис. 3.13).

На рисунке изображена ассоциация класса Position с самим собой (1 рис. 3.13). На полюсах ассоциации указаны роли (2 рис. 3.13). Значок, показывающий направление чтения (3 рис. 3.13) (черный треугольник) позволяет прочесть данную ассоциацию как “Chief subordinates Subordinate”. Эта ассоциация призвана отразить наличие иерархии подчиненности должностей в организации. Однако из рис. 3.13 видно только, что объекты класса

Person образуют некоторую иерархию (каждый объект связан с некоторым количеством нижележащих в иерархии объектов и не более чем с одним вышележащим объектом), но не более того.

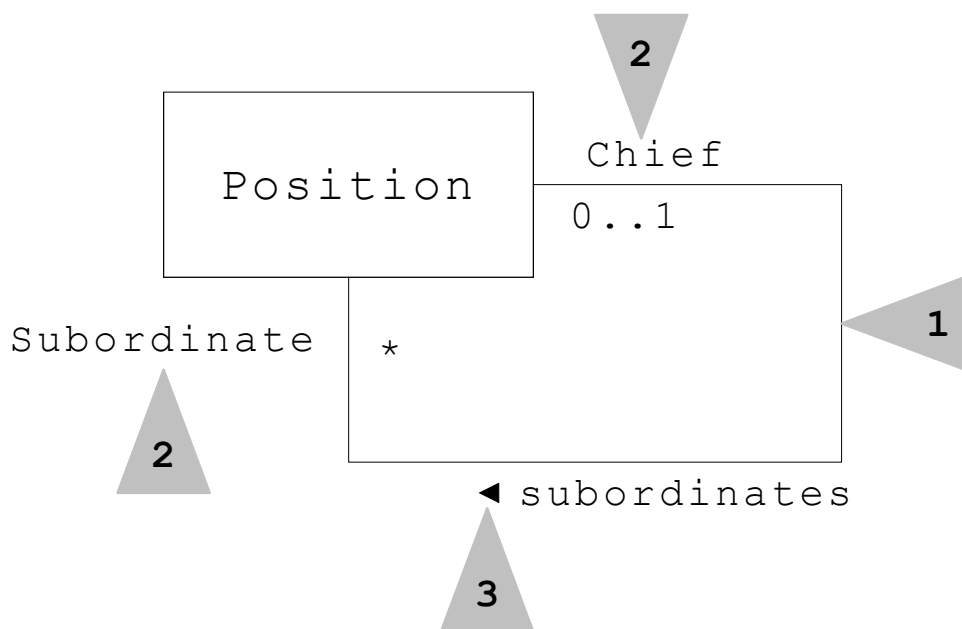


Рис. 3.13. Описание иерархии должностей

Используя роли и, заодно, отношения реализации, можно описать субординацию в информационной системе отдела кадров достаточно лаконично, но точно. Например, на рис. 3.14 указано, что в иерархии субординации каждая должность может играть две роли. С одной стороны, должность может рассматриваться как начальственная (1) (chief), и в этом случае она предоставляет интерфейс IChief (2)<sup>13</sup> имеющий операцию petition (начальнику можно подать служебную записку). С другой стороны, должность может рассматриваться как подчиненная (3) (subordinate), и в этом случае она предоставляет интерфейс ISubordinate (4), имеющий операцию report (от подчиненного можно потребовать отчет). У начальника

<sup>13</sup> Сложилась устойчивая традиция начинать имена интерфейсов с прописной буквы I.



может быть произвольное количество подчиненных (5), в том числе и 0, у подчиненного может быть не более одного начальника (6).

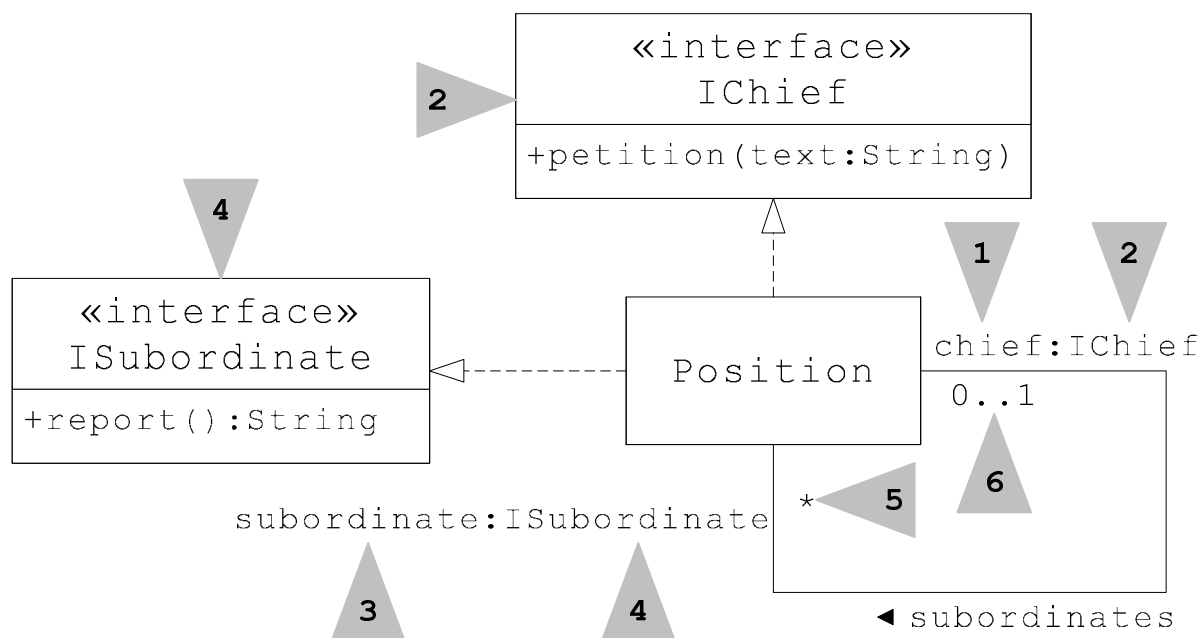


Рис. 3.14. Роли полюсов ассоциации

**Возможность навигации** (*navigability*) для полюса ассоциации — это свойство полюса, имеющее значение типа *Boolean*, и определяющее, можно ли эффективно получить с помощью данной ассоциации доступ к объектам класса, присоединенному к данному полюсу ассоциации.

Для отображения факта возможности или не возможности навигации для данного полюса ассоциации применяется следующая нотация: если навигация для некоторого полюса возможна, то этот полюс отмечают стрелкой на конце линии ассоциации (1), если же навигация не возможна, то на конце линии ассоциации рисуют косой крестик (2). В примере, приведенном на рис. 3.15, навигация возможна только в направлении от *Company* к *Person*, но не наоборот.

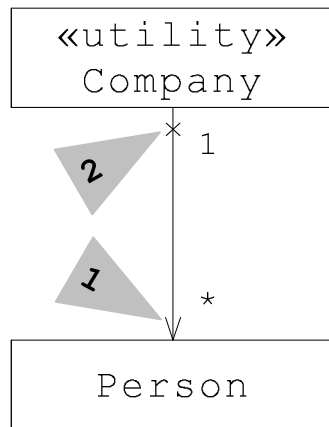


Рис. 3.15. Вариант использования направлений навигации

### 3.3.7. Класс ассоциации и квалификатор

В процессе проектирования возможны ситуации, когда ассоциация должна иметь собственные атрибуты (и даже операции), значения которых хранятся в экземплярах ассоциации — связях. В таком случае применяется класс ассоциации.

**Класс ассоциации** (*association class*) — это сущность, которая является ассоциацией, но также имеет в своем составе составляющие класса.

## ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

Допускается ситуация, когда сотрудник может работать на нескольких должностях в разных проектах, а также возможно, чтобы одну и ту же должность в одном проекте занимало несколько сотрудников (дробление ставки). Размер заработной платы зависит от того, сколько конкретно времени проработал данный сотрудник в данной должности в данном проекте.

Таким образом, допускается не только совмещение должностей (один сотрудник может работать на нескольких должностях в разных проектах), но и дробление ставок (одну должность могут занимать несколько сотрудников — полставки, четверть ставки и т. п.). Используя же разобранную нотацию ассоциации, мы можем

констатировать, что между классами *Person*, *Position* и *Project* имеет место ассоциация "многие ко многим". Однако этого недостаточно: необходимо указать, какую долю данной должности занимает данный сотрудник. Эту информацию нельзя отнести ни к должности, ни к сотруднику, ни к проекту — это атрибут ассоциации, которая всех их связывает. На рис. 3.16 показан способ использования класса ассоциации (1) для решения данной задачи.

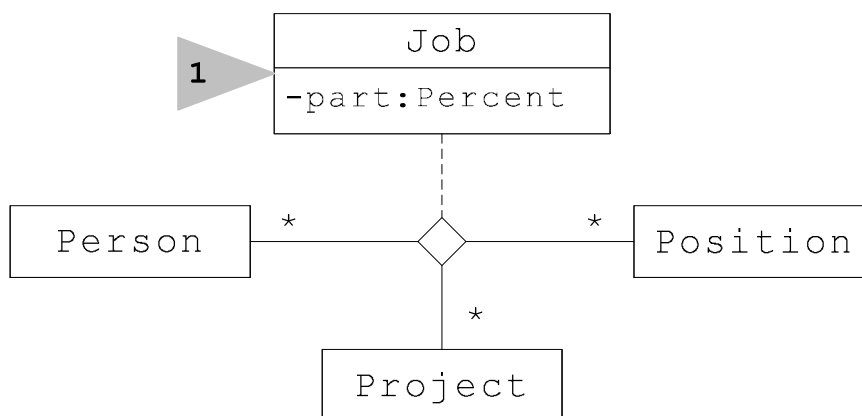


Рис. 3.16. Класс ассоциации

Продолжим разговор про специфику отношений между классами *Company* и *Person*. На рис. 3.15 они связаны отношением ассоциации с кратностями полюсов "один ко многим". Такая ассоциация доставляет для каждого экземпляра класса *Company*, находящегося на полюсе с кратностью "один" множество (иногда большое) объектов класса *Person*, находящегося на полюсе с кратностью "много". Однако часто возникают ситуации, когда нужно получить не все множество ассоциированных объектов *Person*, а некоторое небольшое подмножество, чаще всего один конкретный объект. Чтобы выделить из множества один конкретный объект, нужно располагать информацией, однозначно идентифицирующей этот объект. Такую информацию принято называть *ключом*. Например, индивидуальный номер налогоплательщика (ИНН) для сотрудника в информационной системе отдела кадров может считаться ключом.

Для того чтобы решить задачу поиска конкретного сотрудника (конкретный экземпляр класса `Person`) всегда можно применить следующее тривиальное решение: хранить ключ (ИНН) в самом объекте класса `Person` в качестве атрибута и, получив множество объектов, перебирать их все последовательно до тех пор, пока не найдется тот, который имеет искомое значение ключа. Такой прием называется *линейным поиском*. Для компаний, в которых не очень много сотрудников, данный способ может быть вполне приемлемым. Но в других случаях, когда к полюсу с кратностью "много" присоединено действительно **много** объектов, линейный поиск слишком неэффективен. Известно множество структур данных, позволяющих эффективно выделить (найти в множестве) объект по ключу: сортированные массивы, таблицы расстановки (хэш-таблицы), деревья сортировки, внешние индексы и др. Эти приемы обобщены в UML понятием квалификатора.

***Квалификатор полюса ассоциации** (*qualifier*) — это атрибут (или несколько атрибутов) полюса ассоциации, значение которого (которых) позволяет выделить один (или несколько) объектов класса, присоединенного к другому полюсу ассоциации.*

Квалификатор изображается в виде небольшого прямоугольника на полюсе ассоциации, примыкающего к прямоугольнику класса. Внутри этого прямоугольника (или рядом с ним) указываются имена и, возможно, типы атрибутов квалификатора. Описание квалифицирующего атрибута ассоциации имеет такой же синтаксис, что и описание обычного атрибута класса, только оно не может содержать начального значения.

Основное назначение квалификатора — снизить кратность противоположного полюса ассоциации, поэтому в основном он используется в ассоциациях с кратностями полюсов "один ко многим" или "многие ко многим" и стоит у полюса противоположному полюсу с кратностью "много".

При использовании квалификатора кратность противоположного полюса снижается, и это отображается на диаграмме. Сравните рис. 3.15 и рис. 3.17.

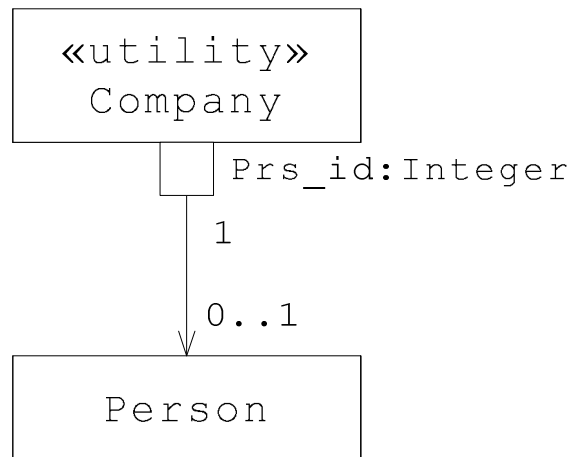


Рис. 3.17. Квалификатор

Кратность полюса у класса `Person` изменилась с  $*$  до  $0..1$ , так как экземпляр класса `Person` для данного ключа может быть найден, а может и отсутствовать (неправильное значение ключа).

Таким образом, если на полюсе ассоциации, противоположном полюсу квалификатора, задана кратность, то она указывает не допустимую мощность множества объектов, присоединенных к полюсу связи, а допустимую мощность того подмножества, которое определяется при задании значений атрибутов квалификатора.

### 3.4. ДИАГРАММЫ РЕАЛИЗАЦИИ

Данный раздел посвящен сразу двум диаграммам: компонентов и размещения, для которых можно использовать обобщающее название — *диаграммы реализации*. Связано это с тем, что данные диаграммы приобретают особую важность на позднейших фазах разработки — на фазах реализации и поставки (подробнее см. в главе 5). В то время как на ранних фазах разработки — анализа и

проектирования — эти диаграммы либо вообще не используются, либо имеют самый общий, не детализированный вид.

С точки зрения реализации проектируемая система состоит из компонентов (представленных на диаграммах компонентов), распределенных по вычислительным узлам (представленным на диаграммах размещения).

В UML 2 по сравнению с UML 1 произошло значительное изменение, а именно, понятие "компонент" было разделено на две составляющие: логическую и физическую. Логическая составляющая, продолжая носить имя *компонент* (component), является элементом логической модели системы, в то время как физическая составляющая, называемая *артефактом* (artifact), олицетворяет физический элемент проектируемой системы, размещающийся на *вычислительном узле* (node).

Диаграммы компонентов и размещения имеют много общего, объединяя воедино следующие, теснейшим образом связанные, вещи:

- структуру логических элементов (компонентов);
- отображения логических элементов (компонентов) на физические элементы (артефакты);
- структуру используемых ресурсов (узлов) с распределенными по ним физическими элементами (артефактами).

### 3.4.1. Интерфейс

Можно выделить две роли (см. рис. 3.18), которые играют интерфейсы (1) по отношению к классификаторам, например, компонентам (2). Интерфейс может быть *обеспеченным*<sup>14</sup> и *требуемым*<sup>15</sup>.

---

<sup>14</sup> Встречающиеся в литературе варианты перевода: "реализованный", "предоставляемый".

<sup>15</sup> Встречающийся в литературе вариант перевода — "запрашиваемый".

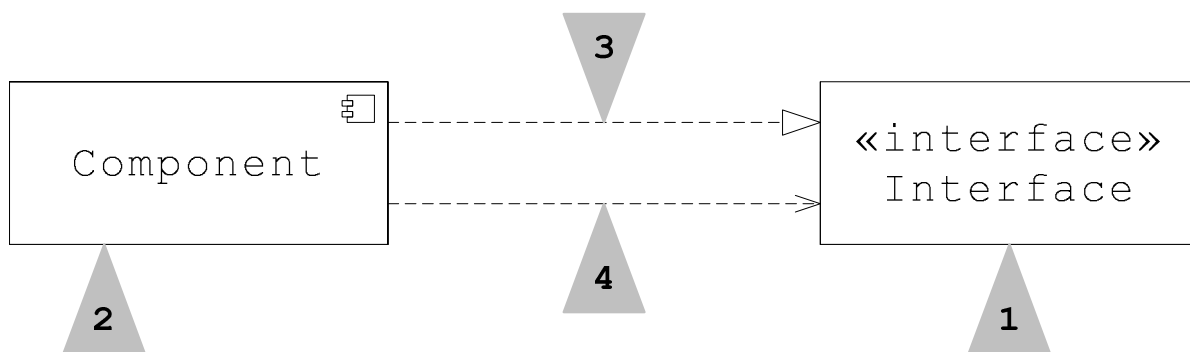


Рис. 3.18. Отношения между классификаторами и интерфейсами

Однако, нельзя забывать, что сам по себе интерфейс — это просто описание контракта, а обеспеченным или требуемым он становится в зависимости от того, как этот интерфейс используется:

- **если классификатор реализует интерфейс** — то для данного классификатора это *обеспеченный* интерфейс и данный факт показывается с помощью отношения реализации (3 на рис. 3.19);

- **если классификатор вызывает операции интерфейса** — то для данного классификатора это *требуемый* интерфейс и данный факт показывается с помощью отношения зависимости (4 на рис. 3.18).

Разобравшись с интерфейсами, давайте перейдем к компонентам.

### 3.4.2. Компоненты, артефакты и узлы

**Компонент** (*component*) — это модульный фрагмент логического представления системы, взаимодействие с которым описывается набором обеспеченных и требуемых интерфейсов.

Компонент UML является частью модели, и описывает логическую сущность, которая существует только во время проектирования (*design time*), хотя в дальнейшем ее можно связать с физической реализацией (артефактом) времени исполнения (*run time*). Стандартом UML для компонентов предусмотрены стереотипы, приведенные в табл. 3.7.

Таблица 3.7

## Стандартные стереотипы компонент

Стереотип	Описание
«buildComponent»	компонент, служащий для разработки приложения
«entity»	постоянно хранимый информационный компонент, представляющий некоторое понятие предметной области
«service»	функциональный компонент без состояния, возвращающий запрашиваемые значения без побочных эффектов
«subsystem»	единица иерархической декомпозиции большой системы

Аналогом компонента в смысле сборочного программирования является понятие артефакта в UML. Причем не любого артефакта, а только некоторых из его стереотипов.

***Артефакт** — это любой созданный искусственно элемент программной системы.*

К элементам программной системы, а, следовательно, и к артефактам, могут относиться исполняемые файлы, исходные тексты, веб-страницы, справочные файлы, сопроводительные документы, файлы с данными, модели и многое другое, являющееся физическим элементом информации. Другими словами, артефактами являются те информационные элементы, которые тем или иным способом используются при работе программной системы и входят в ее состав.

Для того чтобы как-то отражать такое разнообразие типов артефактов в UML 2 предусмотрены стандартные стереотипы, перечисленные в табл. 3.10.

Таблица 3.8

## Стандартные стереотипы артефактов

Стереотип	Описание
«file»	Файл любого типа, хранимый в файловой системе
«document»	артефакт, представляющий файл (документ), который не является ни файлом исходных текстов, ни исполняемым файлом



Стереотип	Описание
«executable»	выполнимая программа любого вида. Подразумевается по умолчанию, если никакой стереотип не указан
«library»	статическая или динамическая библиотека
«script»	файл, содержащий текст, допускающий интерпретацию соответствующими программными средствами
«source»	файл с исходным кодом программы

Однако реальные артефакты гораздо разнообразнее по своим типам, чем перечисленные выше. Чтобы как-то учесть это обстоятельство, многие инструменты, помимо стандартных стереотипов, поддерживают дополнительные стереотипы артефактов, часто со специальными значками и фигурами, обеспечивающими высокую наглядность диаграмм.

Самым важным аспектом использования понятия артефакта в UML является то, что артефакт может участвовать в отношении манифестации.

**Манифестация** — это отношение зависимости со стереотипом «manifest», связывающее элемент модели (например, класс или компонент) и его физическую реализацию в виде артефакта.

На рис. 3.19 представлен класс Company, который связан отношением манифестации (зависимость со стереотипом «manifest») с двумя артефактами со стереотипом «source», которые в свою очередь определяют артефакт времени выполнения — динамическую библиотеку (со стереотипом «library») Company.

Вообще говоря, отношение манифестации — это отношение типа "многие ко многим", один элемент модели может быть реализован многими артефактами, и один артефакт может участвовать в реализации многих элементов модели.

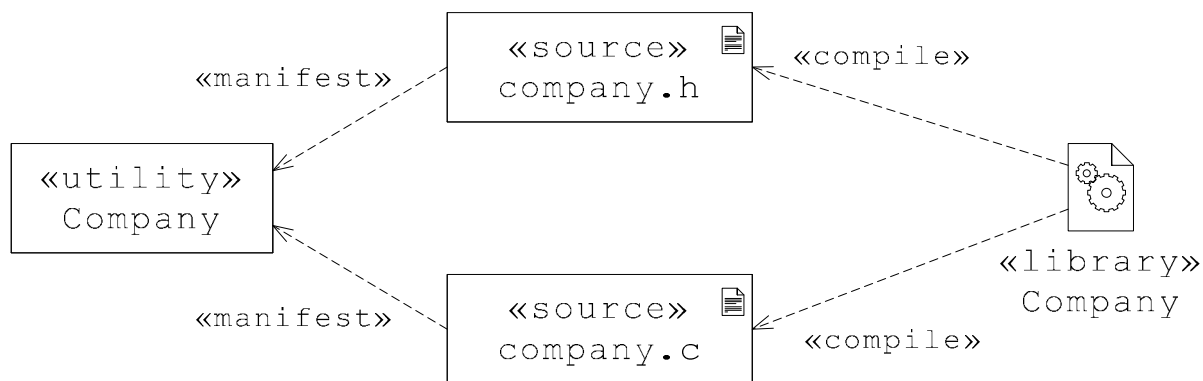


Рис. 3.19. Артефакты

Манифестацию графически изображают отношением зависимости со стереотипом «manifest» от артефакта к реализуемой сущности. Поскольку манифестация — это отношение типа "многие ко многим", для полного описания отношения манифестации могут потребоваться несколько отношений зависимости в модели.

Последняя сущность, рассматриваемая в этом параграфе — узел.

*Узел (node) — это физический вычислительный ресурс, участвующий в работе системы.*

В UML 2 предусмотрено два стереотипа для узлов «executionEnvironment» и «device».

Узел со стереотипом «executionEnvironment» позволяет моделировать аппаратно-программную платформу, на которой происходит выполнение приложения. Узел со стереотипом «device» также моделирует аппаратно-программную платформу, но допускает возможность вложение одного узла в другой, как это показано на рис. 3.20.

Артефакты системы во время ее работы размещаются на узлах, что графически выражается либо их перечислением внутри узла (1 на рис. 3.20), либо отношением зависимости со стереотипом «deploy» между артефактом и узлом (1 на рис. 3.21), либо изображением артефакта внутри изображения узла (2 на рис. 3.21). Все нотации равноправны.

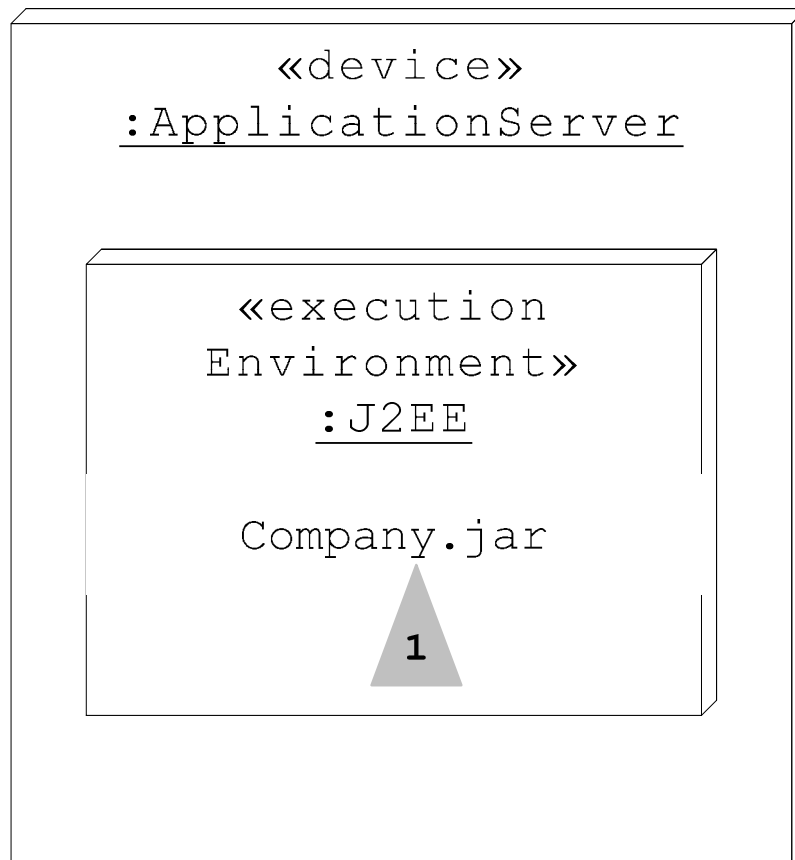


Рис. 3.20. Нотация узла

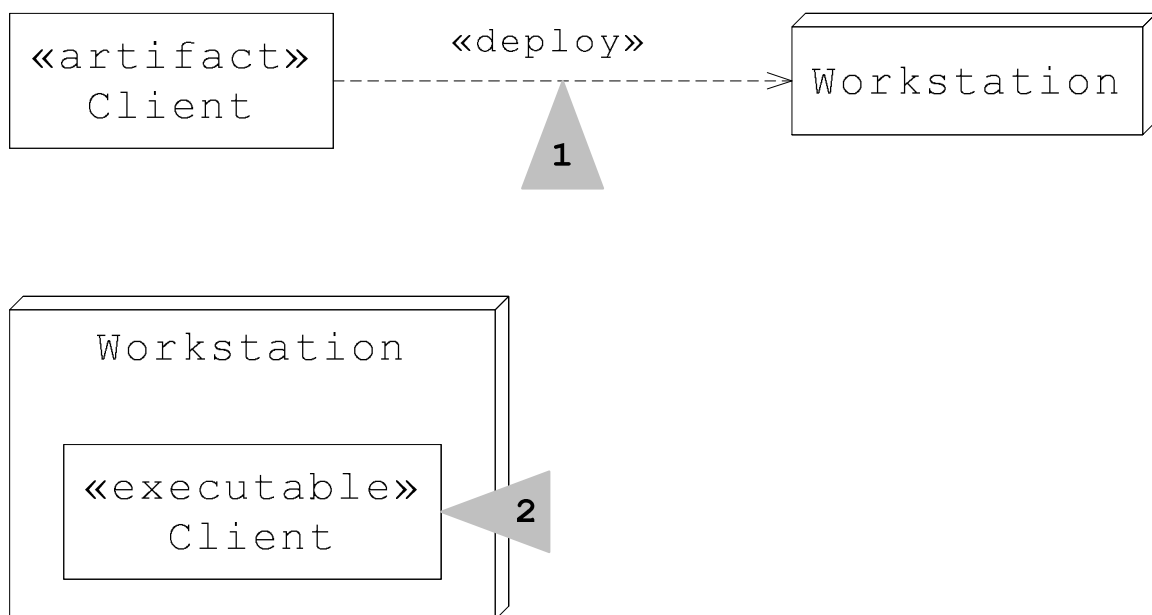


Рис. 3.21. Нотации размещения артефакта на узле

Последнее, что нам осталось рассмотреть в рамках данного параграфа – это отношение *ассоциации* между узлами.

Если узлы связаны между собой отношением ассоциации, то это означает то же, что и в других контекстах: возможность обмена сообщениями. Применительно к вычислительным сетям, состоящим из узлов, ассоциация означает наличие канала связи. Если нужно указать дополнительную информацию о свойствах канала, то это можно сделать, используя общие механизмы: стереотипы («tcp/ip» на рис. 3.22), ограничения и именованные значения.



Рис. 3.22. Ассоциация между узлами

### 3.5. МОДЕЛИРОВАНИЕ НА УРОВНЕ РОЛЕЙ И ЭКЗЕМПЛЯРОВ КЛАССИФИКАТОРОВ

На структурных диаграммах, рассматриваемых в первых разделах этой главы, сущности большей частью являются классификаторами. Экземпляры классификаторов, если и появляются, то играют вспомогательную роль. Однако бывают случаи, когда необходимо рассмотреть модель с более детальной, объектной точки зрения. В этом разделе рассматриваются средства UML, которые применяются в таких случаях.

#### 3.5.1. Диаграмма внутренней структуры

В процессе моделирования на UML может оказаться, что ряд классов или компонентов имеют ярко выраженную внутреннюю структуру. В предыдущих разделах мы встречались, например, с отношением композиции между классами, которая служит для

представления взаимосвязи между "целым" и его "частями". Это хоть и наиболее типичный пример наличия у класса (композиата) внутренней структуры, однако он не до конца отражает суть понятия "внутренняя структура", принятого в UML, и является только одной гранью этого понятия.

Дело в том, что под внутренней структурой классификатора понимается не просто возможность представить этот классификатор в виде некоторого контейнера, но также и способность обеспечить некоторый контекст, т. е. внутренняя структура — это логическое понятие, которое объединяет классификаторы по принципу согласованности для выполнения некоторой задачи, а, следовательно, подразумевает и поведение.

В UML различают две группы классификаторов, для которых может существовать внутренняя структура. В первую группу входят классы и компоненты, а ко второй группе принадлежит такая сущность, как кооперация.

Для каждой из групп в UML имеются свои средства представления внутренней структуры и именно обсуждению этих средств посвящены первые два параграфа данного раздела.

Диаграмма внутренней структуры — одно из самых важных нововведений UML 2. Оно распространяет механизм структурной декомпозиции на структурированные классификаторы (классы и компоненты). С точки зрения практического моделирования это очень важно. Невозможность показать на диаграммах прямо, как именно взаимодействуют составные части сложного классификатора, затрудняла составление сложных моделей в UML 1. В UML 2 это ограничение снято.

*Диаграмма внутренней структуры — это структурная диаграмма, которая раскрывает внутреннюю структуру классификатора и пути взаимодействия элементов (частей), составляющих эту структуру.*

На диаграммах внутренней структуры применяются следующие основные сущности.

- структурированный классификатор (structured classifier);
- часть (part);
- порт (port);
- соединитель (connector).

**Структурированный классификатор (structured classifier)** — это классификатор (класс или компонент), внутренняя структура которого описывается диаграммой внутренней структуры.

Это определение похоже на тавтологию, но формально оно правильно, так как само понятие структурированного классификатора достаточно искусственно. Структурированный классификатор изображается обычной для классификаторов фигурой — прямоугольником, внутри и на границах которого размещаются фигуры и значки других сущностей. Обычно на одной диаграмме внутренней структуры раскрывают структуру одного классификатора, показывая части только этого классификатора.

**Часть (part)** — это структурная составляющая, которая описывает роль, которую ее экземпляр играет внутри экземпляра структурированного классификатора.

**Порт (port)** — индивидуальная точка взаимодействия (*interaction point*) структурированного классификатора и его частей с внешними по отношению к ним сущностями.

Порт изображается как небольшой квадратик на границе структурированного классификатора или части (1). Для порта может быть указано имя (2), а также тип (3) и кратность (4). Если указан тип, то это должен быть интерфейс, по которому происходит взаимодействие через данный порт. Впрочем, тип можно и не указывать, а явно присоединить к порту один или несколько предоставляемых и/или требуемых интерфейсов (5), через которые осуществляется взаимодействие с окружающим миром. Общая нотация сущности порт приведена на рис. 3.23.

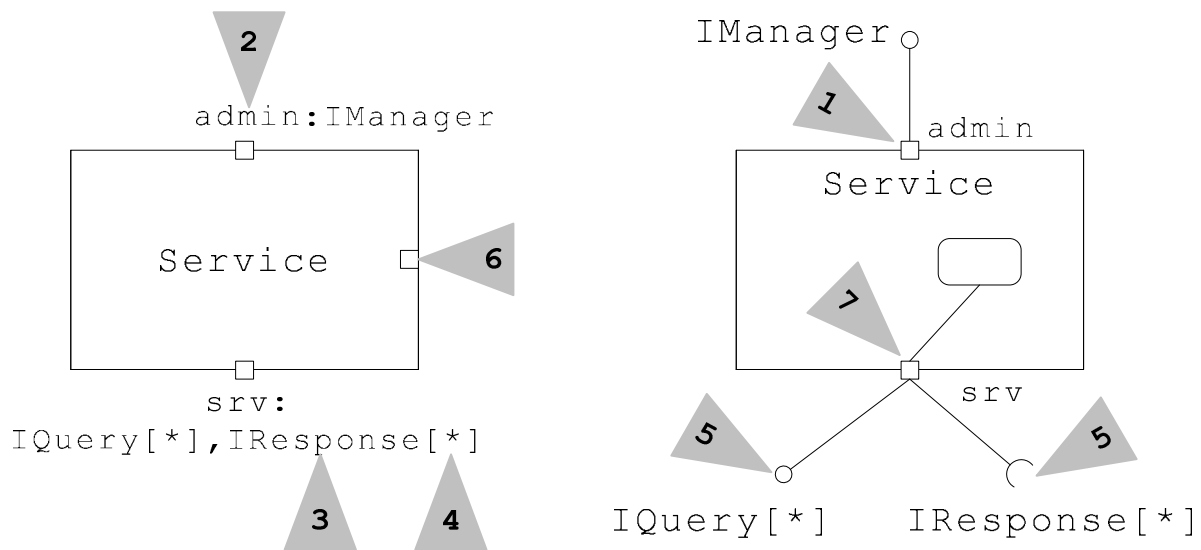


Рис. 3.23. Нотация порта

Как было уже сказано, части структурированного классификатора можно рассматривать как слоты для всевозможных экземпляров других классификаторов, лишь бы они соответствовали роли, которую играет часть. Эти экземпляры классификаторов могут быть связаны между собой явными (например, ассоциации) или неявными (например, зависимости) отношениями. В рамках структурированного классификатора эти отношения представляются в виде соединителей.

**Соединитель** (*connector*) служит для соединения частей структурированного классификатора между собой.

Соединитель может соединять порт структурированного классификатора с его частью или просто соединять части между собой. При этом порты на границах частей могут указываться, а могут и отсутствовать.

Отметим, что соединители используются не только в диаграммах внутренней структуры, но и в кооперациях.

**Соединители бывают двух видов: делегирующие и сборочные.**

Соединитель, который соединяет порт структурированного классификатора с его внутренней частью, называется **делегирующим соединителем** (*delegation connector*).

Соединитель, который соединяет две части структурированного классификатора, называется **сборочным соединителем** (*assembly connector*).

На рис. 3.24 показаны оба вида соединителей: делегирующие (1) и сборочные (2). Для делегирующих соединителей существует возможность использования альтернативной нотации с использованием стереотипа «delegate» (3).

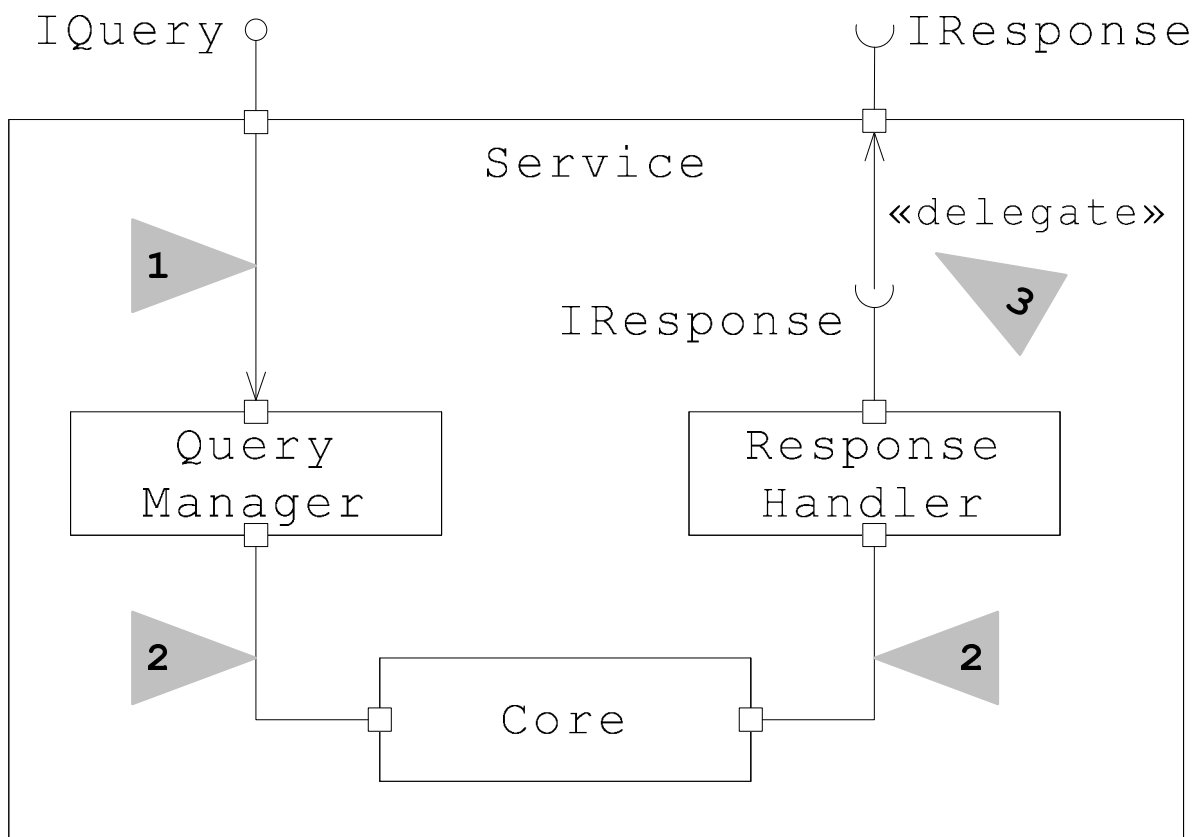


Рис. 3.24. Соединители на диаграмме внутренней структуры

Рассмотрим пример диаграммы внутренней структуры из модели информационной системы отдела кадров (см. рис. 3.25). На этой диаграмме, мы выделяем в подразделении (класс Department)



простую внутреннюю структуру, которая состоит из единственного начальника и некоторого множества подчиненных. Подчиненные (subordinates) и начальник (chief) взаимодействуют, причем предусмотрены различные интерфейсы взаимодействия в зависимости от направления передачи информации (IChief и ISubordinate). Кроме того, указанные части взаимодействуют с внешним миром через свои порты (Inbox и Resource).

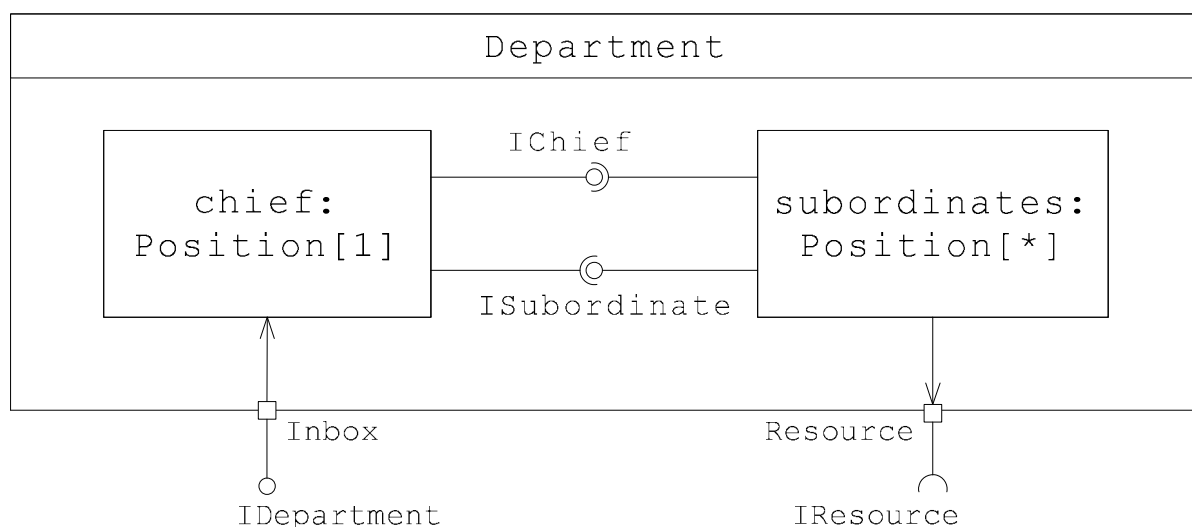


Рис. 3.25. Диаграмма внутренней структуры класса Department

### 3.5.2. Кооперация

Кооперация — это еще один классификатор, который имеет внутреннюю структуру. С некоторой натяжкой кооперацию можно назвать разновидностью структурированного классификатора, отличие которого состоит в том, что кооперация никогда не владеет своими частями. Части связаны между собой посредством участия в кооперации, а не физическим вхождением внутрь нее.

При определении внутренней структуры кооперации используются те же самые сущности, что и для описания структурированных классификаторов, поэтому мы не будем

пересказывать содержание предыдущего параграфа, а только внесем необходимые уточнения.

Кооперация определяет необходимый для решения поставленной задачи набор кооперирующихся участников в виде ролей. В каждом конкретном случае эти роли будут играть конкретные экземпляры классификаторов. Существенные для решаемой задачи взаимоотношения между ролями показываются на диаграмме соединителями, определяя, таким образом, необходимые связи.

В целях наглядности полезно описывать в кооперации только те аспекты классификаторов, которые существенны для решаемой задачи, и исключать остальные. В результате, один и тот же участник может одновременно играть разные роли в различных кооперациях, и каждая кооперация будет представлять только существенные для своей цели аспекты этого участника.

Отдельно от структуры кооперации описывается ее поведение, например, через диаграммы взаимодействия.

Обратимся к информационной системе отдела кадров. На рис. 3.25 приведена внутренняя структура подразделения (Department), описывающая взаимодействие начальника и подчиненных. Рассмотрим немного более сложный случай, когда у начальника есть один или несколько заместителей, через которых он взаимодействует с подчиненными. Если построить диаграмму внутренней структуры для такого случая, описание взаимодействия начальника с заместителями на ней будет дублировать описание взаимодействия заместителей с подчиненными (и начальника с подчиненными в исходной диаграмме). Избежать такой избыточности позволяет кооперация, с помощью которой можно описать схему взаимодействия один раз и применить к различным взаимодействующим частям.

Кооперация изображается в виде пунктирного эллипса, содержащего имя кооперации. Внутренняя структура кооперации в

форме ролей (1) и соединителей (2) может быть показана внутри эллипса на отдельной диаграмме. Например, на рис. 3.26 приведена кооперация, описывающая отношения начальника и подчиненных.

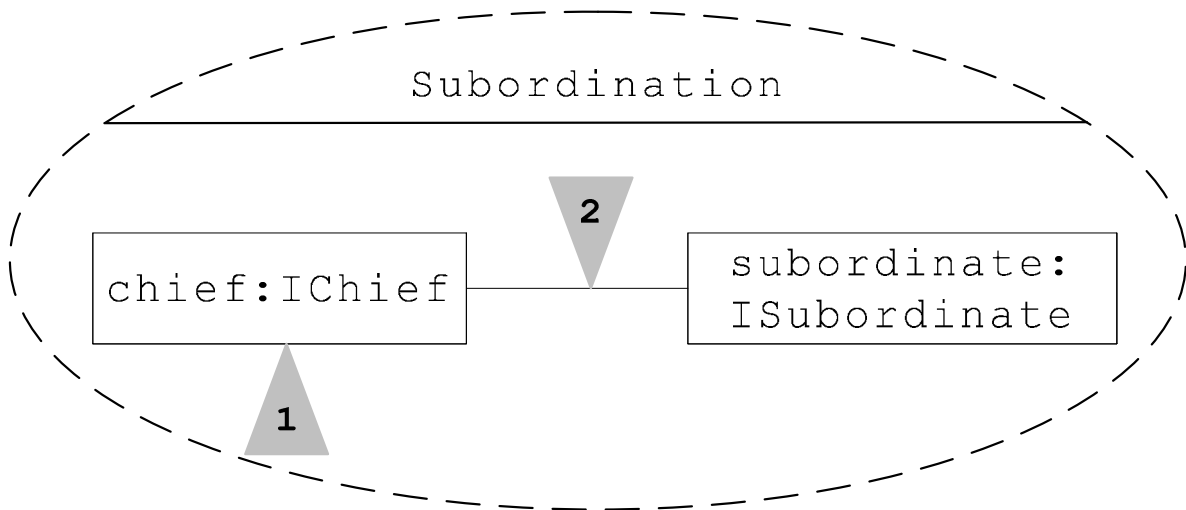



Рис. 3.26. Кооперация начальника и подчиненного

### 3.5.3. Образцы проектирования

Использование коопераций в UML тесно связано с таким понятием как **образцы проектирования**, которые в свою очередь **являются одним из видов паттернов**. Для изучения данного аспекта вернемся к моделированию информационной системы отдела кадров и еще раз посмотрим на реализацию варианта использования Hire Person, представленного на рис. 2.22. 

Хотя внешне все выглядит удовлетворительно, на самом деле в данной реализации полно недочетов. Все что нам остается сделать — это предложить новую диаграмму последовательности (см. рис. 3.27) для реализации типового сценария приема сотрудника, удовлетворяющую принципу Model-View Separation.

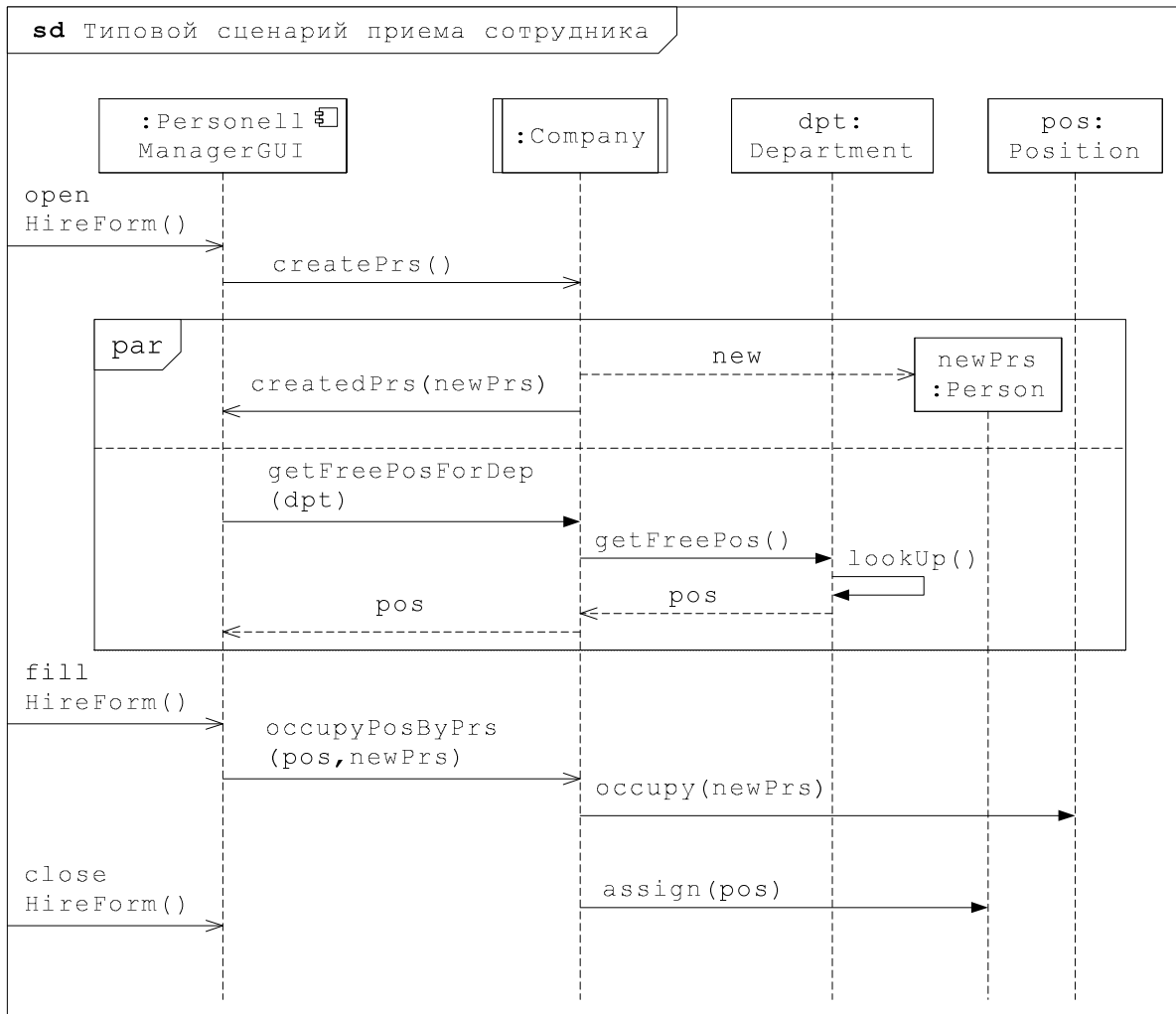


Рис. 3.27. Диаграмма последовательности для типового сценария приема сотрудника


### 3.5.4. Экземпляры классификаторов

В предыдущих темах мы уже рассмотрели достаточно много различных классификаторов. Однако при этом мы всегда делали упор на то, что классификатор является дескриптором, т. е. описателем однотипных объектов, но никогда не вели целенаправленно разговор об экземплярах классификаторов. Теперь у нас накопилось достаточно материала, чтобы восполнить этот пробел.

Для того чтобы указать, что элемент на диаграмме является именно экземпляром классификатора, а не самим классификатором, применяется следующая нотация — его имя подчеркивают.

Не могут иметь конкретных экземпляров абстрактные классификаторы, к которым относятся, например, интерфейс (стереотип «interface») и любые другие классификаторы с ограничением {abstract}.

Также особняком стоит экземпляр ассоциации, который носит специальное название — связь. Подробнее использование связей будет обсуждаться в следующей главе, посвященной моделированию взаимодействия, а сейчас мы лишь скажем, что связь не имеет имени и поэтому выделять подчеркиванием нечего. И хотя нотации ассоциации и связи одинаковы (сплошная линия), однако это не значит, что связь может быть спутана с ассоциацией — у них разные способы использования.

Экземпляры классификатора "тип данных" (стереотип «dataType») и его специализации "примитивный тип" (стереотип «primitive») на диаграммах в большинстве случаев представляются в виде начальных или константных значений для атрибутов других классификаторов. Для возможных экземпляров классификатора "перечисление" (стереотип «enumeration») предусмотрена специальная нотация (см. рис. 3.3). 

Пример использования экземпляров действующего лица для информационной системы отдела кадров приведен на рис. 3.28. Пользуясь случаем, мы продемонстрируем на этом примере альтернативную нотацию для представления действующего лица — класс со стереотипом «actor» (1).

Большую помощь при моделировании системы оказывают экземпляры узлов (стереотипы «device», «executionEnvironment») и размещаемые на них экземпляры артефактов (стереотип «artifact» и другие стереотипы, указанные в табл. 3.12), которые изображаются на диаграмме размещения.

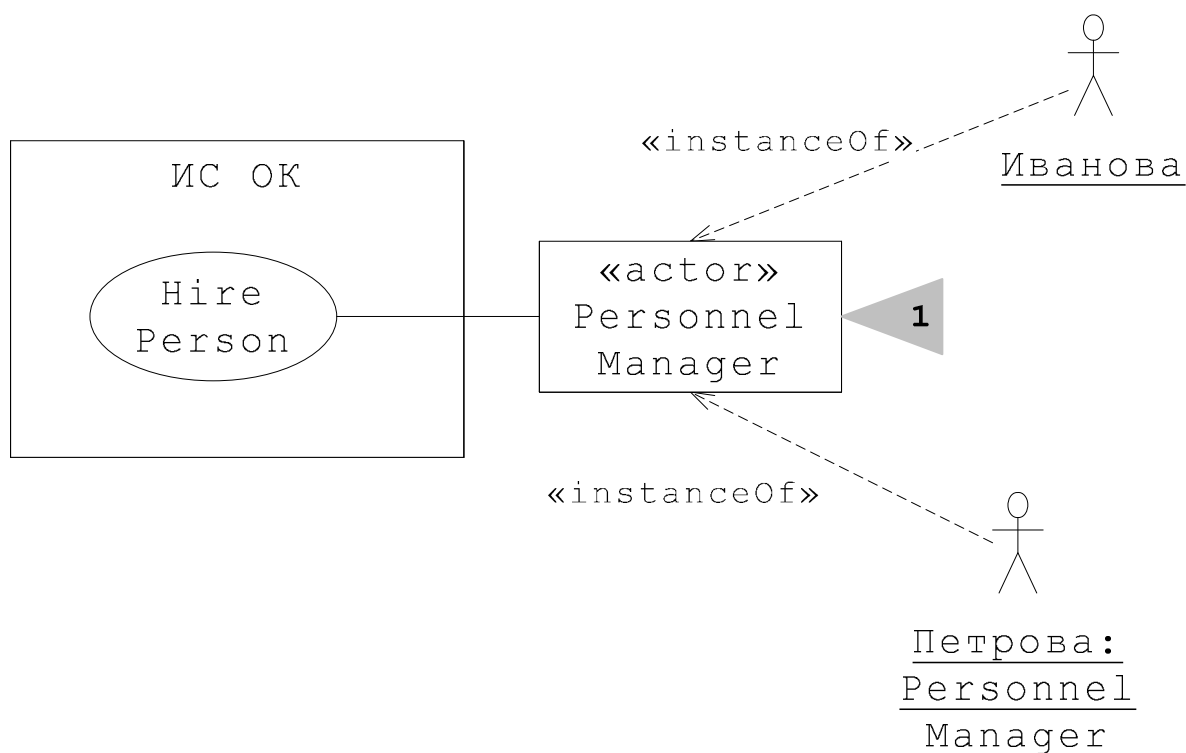


Рис. 3.28. Пример использования экземпляров действующих лиц

В случае информационной системы отдела кадров, например, вместо рассмотрения абстрактных вычислительных узлов, можно рассматривать конкретные компьютеры, которые имеются у организации в наличии, и размещать на этих конкретных компьютерах конкретные экземпляры программы.

Последний классификатор, рассматриваемый в рамках этого параграфа — компонент. **Компонент по определению не может иметь экземпляров, так как существует только на этапе моделирования системы, где никаких экземпляров быть не может.**

### 3.5.5. Объекты и диаграмма объектов

Для представления объектов в UML существует специальная диаграмма — **диаграмма объектов** (*object diagram*).

После столь подробного разбора диаграмм классов и ее составляющих в рамках данной главы, мало что осталось добавить относительно диаграмм объектов.

С одной стороны, диаграмма объектов — это не более чем частный случай диаграммы классов. Сущностями на диаграмме объектов являются объекты, т. е. экземпляры классов. Их имена подчеркиваются. **Отношениями на диаграмме объектов являются связи, т. е. экземпляры ассоциаций.** Отнюдь не все дополнения, предусмотренные для ассоциаций, имеют смысл и могут быть показаны для связей. В частности, кратность для полюсов связи не имеет смысла. Обычно связь отображается просто в виде линии, соединяющей объекты, без каких-либо дополнений.

С другой стороны, можно рассматривать диаграмму объектов как дамп памяти в некоторый момент выполнения системы (т. е. пример того, какие конкретные объекты сосуществуют в некоторый момент времени и какие между ними установлены связи).

Поскольку диаграмма объектов — это не более чем пример, ее описательная сила невелика. **Все, что можно показать на диаграмме объектов, можно показать и на диаграмме взаимодействия** в форме коммуникации, причем гораздо более информативно. Поэтому диаграммы объектов используются сравнительно редко.

## ВЫВОДЫ

1. Структура сложной системы описывается на уровне дескрипторов.

2. Диаграммы классов моделируют структуру классов и отношений между ними.

3. Классы выбираются на основе анализа предметной области, взаимного согласования элементов модели и общих теоретических соображений.

4. Взаимосвязь между классами описывается, прежде всего, с помощью отношений обобщения и ассоциации. Реже с помощью отношений зависимости и реализации.

5. Отношение ассоциации имеет большой набор различных дополнений, с помощью которых можно указать особенности отношений между классами.

6. Множества классов могут объединяться в логическую структуру – компонент.

7. Каждый компонент описывается набором требуемых и обеспеченных интерфейсов.

8. Компонент и классы как элементы модели связываются с физическими сущностями — артефактами — с помощью манифестации.

9. Диаграммы компонентов моделируют структуру компонентов (артефактов) и взаимосвязей между ними.

10. Диаграммы размещения моделируют структуру вычислительных ресурсов и размещенных на них артефактов.

11. Диаграммы внутренней структуры показывают контекст взаимодействия частей сложных классификаторов, причем части, в свою очередь, могут иметь внутреннюю структуру.

12. Кооперация — это способ показать контекста взаимодействия нескольких классификаторов

13. Диаграмма объектов — это пример связей программных объектов в отдельный момент выполнения системы.



## 4. МОДЕЛИРОВАНИЕ ПОВЕДЕНИЯ

### 4.1. МОДЕЛИ ПОВЕДЕНИЯ

При создании программной системы недостаточно ответить на вопросы "что делает система?" (глава 2) и "из чего она состоит?" (глава 3) — требуется ответить на вопрос "как работает система?". Ответ на этот вопрос дает **модель поведения**.

*Модель поведения (behavior model) — это описание алгоритма работы системы.*

Средства моделирования поведения в UML, ввиду разнообразия областей применения языка, должны удовлетворять набору различных и частично противоречивых требований. Перечислим некоторые из них.

**1. Модель должна быть достаточно детальной для того, чтобы послужить основой для составления компьютерной программы** — компьютер не сможет самостоятельно "додумать" опущенные детали.

**2. Модель должна быть компактной и обозримой, чтобы служить средством общения между людьми в процессе разработки системы и для обмена идеями.**

**3. Модель не должна зависеть от особенностей реализации конкретных компьютеров, средств программирования и технологий**, чтобы не сужать область применения языка UML.

**4. Средства моделирования поведения в UML должны быть знакомыми и привычными для большинства пользователей языка** и не должны противоречить требованиям наиболее ходовых 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.

Мы разделили все средства моделирования поведения в UML на четыре группы:

- **описание поведения с явным выделением состояний**, задается диаграммами автомата;

- **описание поведения с явным выделением потоков данных и управления**, задается диаграммами деятельности;

- **описание поведения как упорядоченной последовательности сообщений**; задается диаграммами взаимодействия в четырех формах;

- **описание параллельного поведения**, задается специальными средствами на каждой из диаграмм описывающих поведение.

Все эти средства описаны в четырех следующих разделах.

## 4.2. ДИАГРАММЫ АВТОМАТА

*Диаграммы автомата (state machine diagram) в UML являются реализацией основной идеи использования конечных автоматов как средства описания алгоритмов и, тем самым, моделирования поведения.*

Конечные автоматы в UML реализованы довольно своеобразно. С одной стороны, в основу положено классическое представление автомата в форме графа состояний-переходов. С другой стороны, к классической форме добавлено большое число различных расширений и вспомогательных обозначений, которые, строго говоря, не обязательны — без них в принципе можно было бы обойтись — но они весьма удобны и наглядны при составлении диаграмм. Большую часть этих расширений в свое время, независимо от UML, предложил Дэвид Харел. Он же ввел термин диаграмма состояний (statechart),

инкорпорированный в UML 1 и переименованный в диаграмму автомата в UML 2.

Итак, начиная обзор средств моделирования с самого верхнего уровня, можно констатировать, что на диаграммах автомата применяется всего один тип сущностей — состояния, и всего один тип отношений — переходы. Совокупность состояний и переходов между ними образует *конечный автомат*.

Таким образом, типов сущностей и отношений предельно мало, но подтипов, вариантов нотации и специальных случаев для них определено много (может быть, даже слишком много).

А именно, состояния бывают:

- простые (simple),
- составные (composite) двух видов: ортогональные (orthogonal) и нет,

- специальные (pseudo),
- ссылочные (submachine),

и каждый тип состояний имеет дополнительные подтипы и различные составляющие элементы.

Переходы бывают *простые* и *составные*, и каждый переход содержит от двух до пяти составляющих:

- исходное состояние (source),
- событие перехода (trigger event),
- сторожевое условие (guard),
- действие на переходе (effect),
- целевое состояние (target).

Рассмотрим все эти элементы по порядку.

#### **4.2.1. Простое состояние**

Простое состояние является в UML простым только по названию — оно имеет следующую структуру:

- имя (name);
- действие при входе (entry action);

- действие при выходе (exit action);
- описание множества внутренних переходов (internal transitions) и соответствующих действий;
- внутренняя деятельность (do activity);
- множество отложенных событий (defer events).

*Имя состояния* является обязательным. Все остальные составляющие простого состояния не являются обязательными.

*Действие при входе* (обозначается при помощи ключевого слова entry) — это указание атомарного действия, которое должно выполняться при переходе автомата в данное состояние. Действие при входе выполняется **после** всех других действий, предписанных переходом, переводящим автомат в данное состояние.

*Действие при выходе* (обозначается при помощи ключевого слова exit) — это указание атомарного действия, которое должно выполняться при переходе автомата из данного состояния. Действие при выходе выполняется **до** всех других действий, предписанных переходом, выводящим автомат из данного состояния.

Множество внутренних переходов — это множество простых переходов из данного состояния в это же самое. *Внутренний переход* (internal) отличается от простого *перехода в себя* (external) тем, что действия при выходе и входе **не** выполняются.

*Внутренняя деятельность* (обозначается при помощи ключевого слова do) — это указание деятельности, которая начинает выполняться при переходе в данное состояние после выполнения всех действий, предписанных переходом, включая действие на входе. Внутренняя деятельность либо продолжается до завершения, либо прерывается в случае выполнения перехода (в том числе и внутреннего перехода). В классической модели конечный автомат, находясь в некотором состоянии, ничего не делает: он находится в состоянии ожидания перехода. В модели UML считается, что автомат можно нагрузить какой-то полезной фоновой деятельностью, которая будет прерываться при выполнении любого перехода.

Если в то время, когда автомат находится в некотором состоянии, происходит событие, для которого в данном состоянии не определен переход, то согласно семантике UML ничего не происходит и событие безвозвратно теряется. В некоторых случаях этого требуется избежать. Для этого в UML предусмотрено понятие отложенного события.

*Отложенное событие* — это событие, для которого не определен переход в данном состоянии, но которое, тем не менее, не должно быть потеряно, если оно произойдет, пока автомат находится в данном состоянии (обозначается при помощи ключевого слова **defer**). Семантика отложенного события такова: если происходит отложенное событие, то оно помещается в конец некоторой системной очереди отложенных событий. После перехода автомата в новое состояние проверяется (начиная с начала) очередь отложенных событий. Если в очереди есть событие, для которого в новом состоянии определен переход, то событие извлекается из очереди и происходит переход.

#### 4.2.2. Простой переход

Простой переход всегда ведет из одного состояния в другое или в то же самое состояние. Переход не может приходить "ниоткуда" и уходить "в никуда". Существует несколько ограничений для специальных состояний, например, для начального состояния не может быть входящих переходов, а для заключительного — исходящих, а в остальном переходы между состояниями могут быть определены произвольным образом.

Прочие составляющие — событие перехода, сторожевое условие и действия на переходе не являются обязательными. Если они присутствуют, то изображаются в виде текста в определенном синтаксисе рядом со стрелкой, изображающей переход. Синтаксис описания перехода следующий:

Событие [ Сторожевое условие ] / Действие

Например, в информационной системе отдела кадров переход сотрудника из состояния Applicant (кандидат) в состояние Employed (нанятый на работу) может быть описан с помощью простого перехода (1), представленного на рис. 4.1.

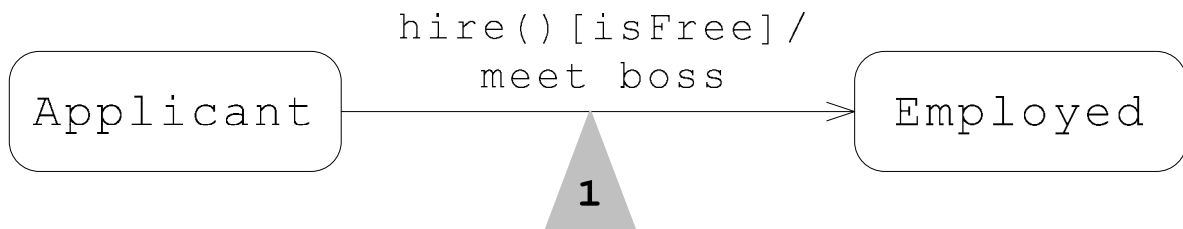


Рис. 4.1. Простой переход

Общая семантика перехода такова. Допустим, что автомат находится в состоянии Applicant, в котором определен исходящий переход с событием `hire()`, сторожевым условием `isFree` и действием `meet boss`, ведущий в состояние Employed.

Если возникает событие `hire()`, то переход *возбуждается* (в данном состоянии могут быть одновременно возбуждены несколько переходов — это не считается противоречием в модели). Далее проверяется сторожевое условие `isFree` (проверяются сторожевые условия всех возбужденных переходов в неопределенном порядке). Если сторожевое условие выполнено, то переход *срабатывает* — выполняется действие на выходе из состояния Applicant, выполняется действие на переходе `meet boss`, выполняется действие на входе в состояние Employed и автомат переходит в состояние Employed. Даже если у нескольких возбужденных переходов сторожевые условия оказываются истинными, то, тем не менее, срабатывает всегда только один переход из возбужденных. Какой именно переход срабатывает — не определено. Таким образом, поведение, описываемое подобным автоматом, является недетерминированным. Если же сторожевое условие `isFree` не выполнено, то переход **не** срабатывает. Если ни один из

возбужденных переходов не срабатывает, то событие `hire()` **теряется** и автомат остается в состоянии `Applicant`.

**Событие перехода** (*trigger event*)<sup>16</sup> — это тот входной символ (стимул), который вкупе с текущим состоянием автомата определяет следующее состояние.

UML допускает наличие переходов без событий — такой переход называется переходом по завершении.

**Переход по завершении** (*completion transition*) — это переход, который происходит по окончанию внутренней деятельности.

Рассмотрим пример.

## ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

Сразу после приема на работу кандидату может быть поручено выполнение специального задания. Если кандидат не справляется с ним в оговоренный период (испытательный срок), то это является достаточным основанием для увольнения. Если же кандидат успешно выполняет задание, то с ним заключается договор на постоянную работу.

На рис. 4.2 приведено одно из возможных решений, в котором используется как простой переход (1), так и переход по завершении (2).

**Сторожевое условие** (*guard*) — это логическое выражение, которое должно оказаться истинным для того, чтобы возбужденный переход сработал.

В сторожевом условии можно использовать значения атрибутов моделируемого элемента, с которым связана машина состояний, а также значения аргументов переключающего события. Таким образом, значение сторожевого условия вычислить заранее, на этапе моделирования, невозможно. Сторожевое условие должно проверяться динамически, во время выполнения.

---

<sup>16</sup> Используют также термин "переключающее событие".

Для каждого возбужденного перехода сторожевое условие проверяется ровно один раз, сразу после того, как переход возбужден и до того, как в системе произойдут какие-либо другие события. Если сторожевое условие ложно, то переход не срабатывает и событие теряется. Даже если впоследствии сторожевое условие станет истинным, переход сможет сработать, только если повторно возникнет событие перехода.

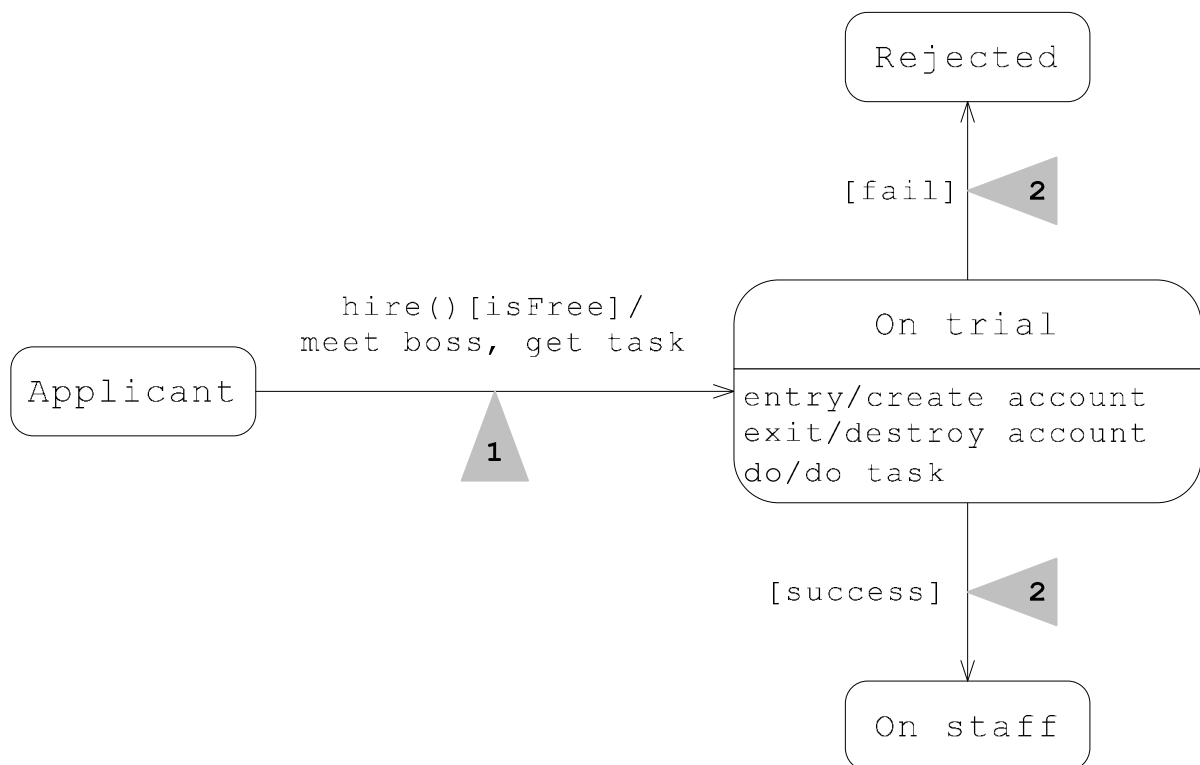


Рис. 4.2. Использование перехода по завершении

Последней составляющей простого перехода является действие.

**Действие** (*action*) — это непрерываемое извне атомарное вычисление, чье время выполнения пренебрежимо мало.

Авторы языка подразумевали, что инструменты моделирования будут связывать с понятием действия в модели UML понятие действия (или аналогичное) в целевом языке программирования. Например, для обычных языков программирования действиями являются вычисление значения выражения и присваивание его



переменной, запись блока данных в файл, посылка сигнала и т. д. В UML предусмотрено несколько типов действий, похожих по семантике на действия в наиболее распространенных языках программирования. Однако UML не является языком программирования и, тем самым, не претендует на то, чтобы быть универсальным языком описания действий. Поэтому понятие действия в UML сознательно недоопределено — оставлена свобода, необходимая инструментам для непротиворечивого расширения семантики действий UML до семантики действий конкретного языка программирования. Здесь, в контексте обсуждения машины состояний UML, стоит подчеркнуть два обстоятельства.

**Действие является атомарным и непрерываемым.** При выполнении действия на переходе или в состоянии не могут происходить события, прерывающие выполнение действия. Точнее говоря, событие может произойти, но система обязана задержать его обработку до окончания выполнения действия.

**Действие является безальтернативным и завершаемым.** Раз начавшись, действие выполняется до конца. Оно не может "раздумать" выполняться или выполняться неопределенно долго.

Действия являются важнейшей частью описания поведения с помощью конечных автоматов. В UML действия, составляющие процедуру реакции, фактически ничем не ограничены: в так называемых не интерпретируемых действиях могут быть скрыты, например, любые программистские трюки. Более того, последовательность действий на переходе также является действием. (Синтаксически, действия в последовательности разделяются запятыми). Поэтому формальные свойства машины состояний UML трудно проверить автоматически (в отличие от абстрактных конечных автоматов). С другой стороны, машины состояний UML выразительны и наглядны — многочисленные синтаксические добавления позволяют моделировать сложное поведение компактно и красиво.

### 4.2.3. Сегментированные переходы

В UML предусмотрены синтаксические средства, до некоторой степени, облегчающие семантически правильное построение сторожевых условий за счет более наглядного их изображения. Таковыми являются:

- сегментированные переходы, реализуемые с помощью переходных состояний и состояний выбора;
- предикат `else`.

**Сегменты перехода** (*transition segment*) — это части, на которые может быть разбита линия перехода.

Разбивающими элементами являются следующие фигуры:

- *переходное состояние* (*junction state*) (изображается в виде небольшого кружка);
  - *состояние выбора* (*choice*) (изображается в виде ромба);
- действия посылки и приема сигнала, изображаются в виде флажков.

Сегментирование перехода применяется в UML в нескольких ситуациях. Здесь мы рассматриваем их в связи со сторожевыми условиями, а прочие случаи опишем в соответствующем контексте.

Несколько переходов, исходящих из данного состояния и имеющих общее событие перехода, можно объединить в дерево сегментированных переходов следующим образом. Имеется один сегмент перехода, который начинается в исходном состоянии и заканчивается в переходном состоянии или состоянии выбора. Далее из этого переходного состояния или состояния выбора начинаются другие сегменты, которые заканчиваются в целевых состояниях или новых переходных состояниях или состояниях выбора. Сегмент перехода, начинающийся в исходном состоянии, называется *корневым*, сегменты, заканчивающиеся в целевых состояниях, называются *листовыми*. Событие перехода может быть указано только для корневого сегмента, действия на переходе могут быть

указаны только для листовых сегментов, а сторожевые условия могут быть указаны для любых сегментов. Такое дерево сегментированных переходов семантически эквивалентно множеству простых переходов, которое получится, если рассмотреть все пути из исходного состояния в целевые состояния, считая встречающиеся на пути сторожевые условия соединенными конъюнкцией (то есть, соединенными логической связкой "И").

Замысловатое определение предыдущего абзаца не таит в себе ничего необычного или нового. Покажем это на примере из информационной системы отдела кадров.

## ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

*При увольнении сотрудника с предприятия, в зависимости от причины, следует разделять три случая:*

- 1) увольнение сотрудника по собственному желанию,*
- 2) увольнение сотрудника по инициативе администрации в связи с нарушением сотрудником условий договора между ним и предприятием,*
- 3) вынужденное увольнение сотрудника в связи с проблемами, от сотрудника не зависящими. При этом подразумевается обратное зачисление в штат при первой возможности.*

Для реализации этих требований введем три различных состояния для уволенных с предприятия:

1. Unemployed — работник, уволившийся по собственному желанию, повторный прием которого должен проходить на общих основаниях.

2. Non grata — скандалист, бездельник и нарушитель трудовой дисциплины, уволенный по инициативе администрации, которого ни при каких обстоятельствах нельзя нанимать на работу;

3. Welcome back — хороший работник, с которым администрации пришлось расстаться ввиду временных трудностей,

переживаемых предприятием, и которого при первой возможности следует пригласить обратно.

На рис. 4.3 представлен соответствующий фрагмент диаграммы состояний, в котором дерево сегментированных переходов построено с использованием переходных состояний (1).

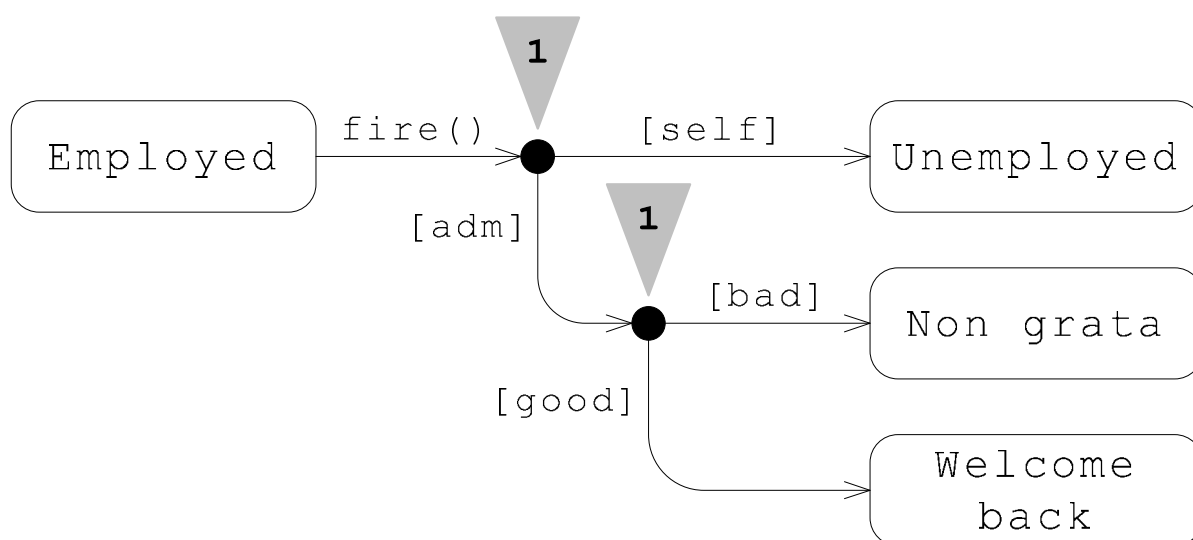


Рис. 4.3. Дерево сегментированных переходов

Продолжим рассмотрение данного примера. Известно, что условия увольнения *adm* и *self* взаимно исключают друг друга, и одно из них при увольнении обязательно имеет место. Для этого используется ключевое слово *else*, которое обозначает условие, считающееся истинным во всех случаях, когда ложны все другие условия, приписанные к альтернативным сегментам. Чтобы подчеркнуть альтернативность условий, мы использовали *состояния выбора* (1 на рис. 4.4). В результате описание сложной системы условий становится нагляднее и надежнее. Таким образом, фрагмент диаграммы состояний на рис. 4.3 семантически эквивалентен фрагменту на рис. 4.4. При этом совершенно ясно (и это нетрудно проверить автоматически), что система условий на рис. 4.4 полна и дизъюнктивна.

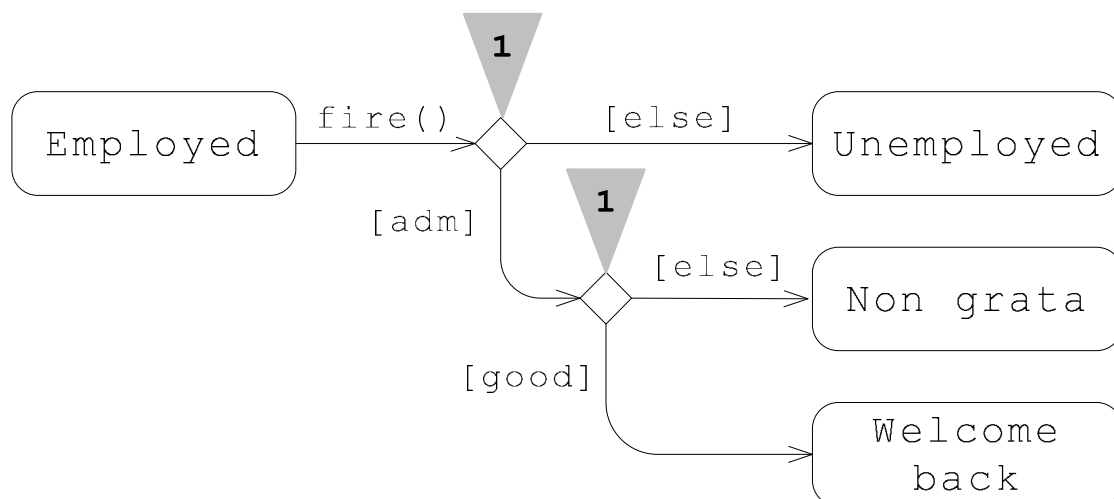


Рис. 4.4. Использование предиката else

Подводя итог обсуждению сторожевых условий, еще раз подчеркнем, что сегментированные переходы, реализованные через переходные состояния или состояния выбора, ничего не добавляют (и не убавляют) в семантике модели: это просто синтаксические обозначения, введенные для удобства и наглядности. Ту же самую семантику, которую имеют фрагменты диаграмм состояний на рис. 4.3 и рис. 4.4, можно передать с помощью фрагмента, приведенного на рис. 4.5.

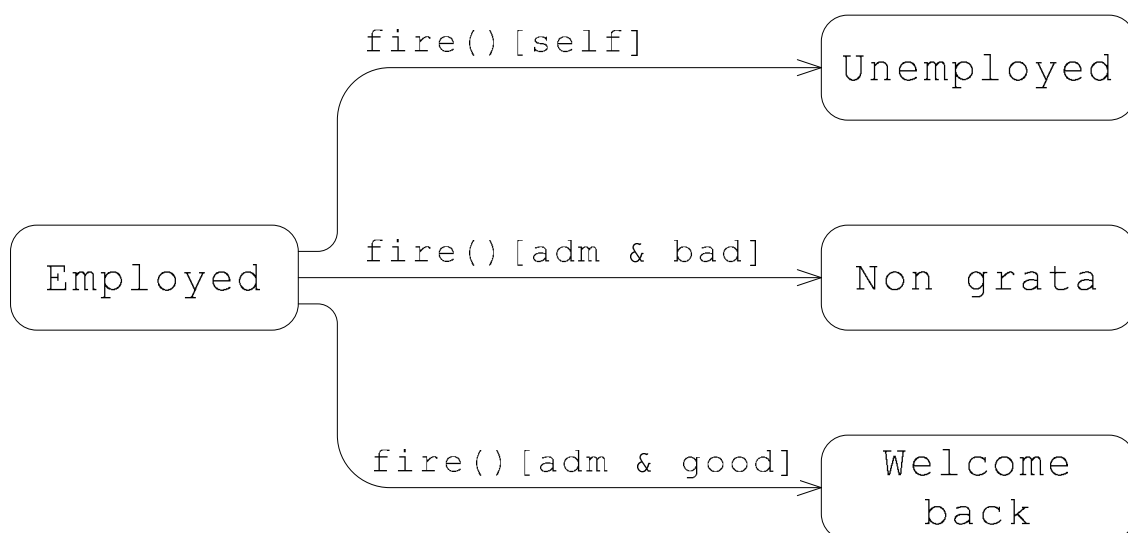


Рис. 4.5. Множество простых переходов с одним событием перехода и различными сторожевыми условиями

#### 4.2.4. Составные состояния

Выше мы рассмотрели простые состояния, соответствующие состояниям обычной модели конечного автомата. Пришла пора рассмотреть другие понятия, близкие к понятию состояния, но специфические для UML. Их можно разделить на две группы:

- составные состояния (composite state),
- специальные состояния (pseudo state).

Составное состояние может быть

- последовательным (sequential / non-orthogonal state),
- параллельным (ортогональным) (concurrent / orthogonal state).

Специальные состояния в UML 1 бывают следующих типов:

- начальное (initial);
- слияние (join);
- развилка (fork);
- переходное (junction);
- выбор (choice);
- поверхностное историческое (shallow history);
- глубинное историческое (deep history);
- синхронизирующее (synch);
- прекращение выполнения (terminate),
- заключительное (final);
- ссылочное состояние (submachine state),
- состояние "заглушка" (stub state).

В UML 2 из этого списка удалены синхронизирующее состояние и состояние "заглушка", но вместо последнего введены два новых специальных состояния: "точка входа" (entry point) и "точка выхода" (exit point).

***Составное состояние** — это состояние, в которое вложена машина состояний. Если вложена только одна машина, то состояние называется последовательным, если несколько — параллельным.*

Глубина вложенности в UML неограниченна, т. е. состояния вложенной машины состояний также могут быть составными. В UML 2 параллельные состояния переименованы в ортогональные. Мы используем оба термина как синонимы.

Мы начнем с простого примера, чтобы сразу пояснить прагматику составного состояния, т. е. зачем это понятие введено в UML, а затем опишем тонкости семантики и связь с другими понятиями машины состояний UML.

Рассмотрим всем известный прибор: светофор. Он может находиться в двух основных состояниях:

- Off — вообще не работает — выключен или сломался, как слишком часто бывает;

- On — работает.

Но работать светофор может по-разному:

- Blinking — мигающий желтый, дорожное движение не регулируется;

- Working — работает по-настоящему и регулирует движение.

В последнем случае у светофора есть 4 видимых состояния, являющихся предписывающими сигналами для участников дорожного движения:

- Green — зеленый свет, движение разрешено;

- YellowGreen — состояние перехода из режима разрешения в режим запрещения движения (это настоящее состояние, светофор находится в нем заметное время);

- Red — красный свет, движение запрещено;

- RedYellow — состояние перехода из режима запрещения в режим разрешения движения (это состояние отличное от YellowGreen, светофор подает несколько иные световые сигналы, и участники движения обязаны по-другому на них реагировать).

На рис. 4.6 приведена соответствующая диаграмма автомата светофора с использованием составных состояний (1) (несколько

забегая вперед, мы использовали здесь событие таймера, выделяемое ключевым словом `after`).

Общая идея составного состояния ясна. Теперь нужно внимательно разобраться с деталями составных состояний и связанных с ними переходов. Мы сделаем это постепенно, несколько раз возвратившись к описанию семантики составных состояний и переходов в подходящем контексте.

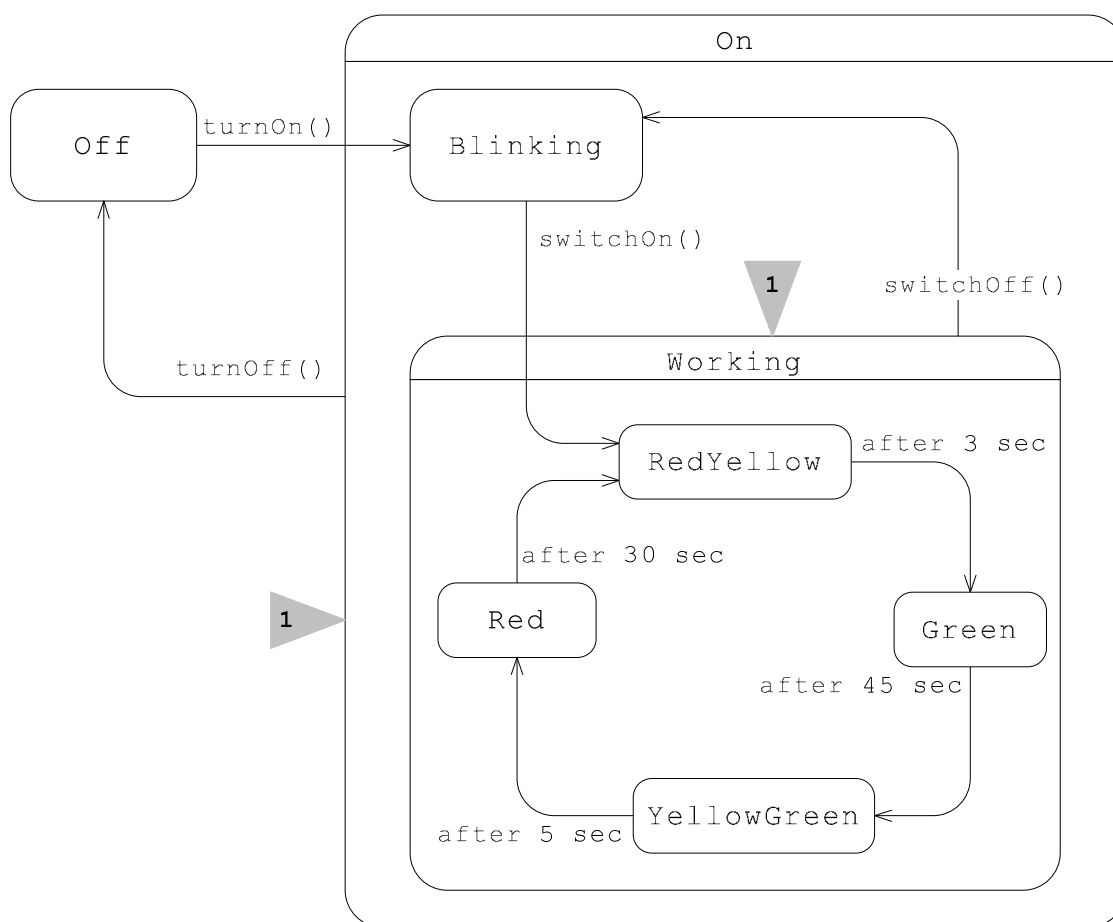


Рис. 4.6. Составные состояния



Первое правило можно сформулировать прямо здесь: переход из составного состояния наследуется всеми вложенными состояниями. Мы не случайно употребили характерный термин объектно-ориентированного программирования "наследование" в данном контексте. По нашему мнению, назначение составных состояний аналогично назначению суперклассов:<sup>17</sup> выявить общее в нескольких элементах и описать это общее только один раз. Тем самым сокращается описание модели, и она становится более удобной для восприятия человеком (сравните рис. 4.6 и рис. 4.7 еще раз).

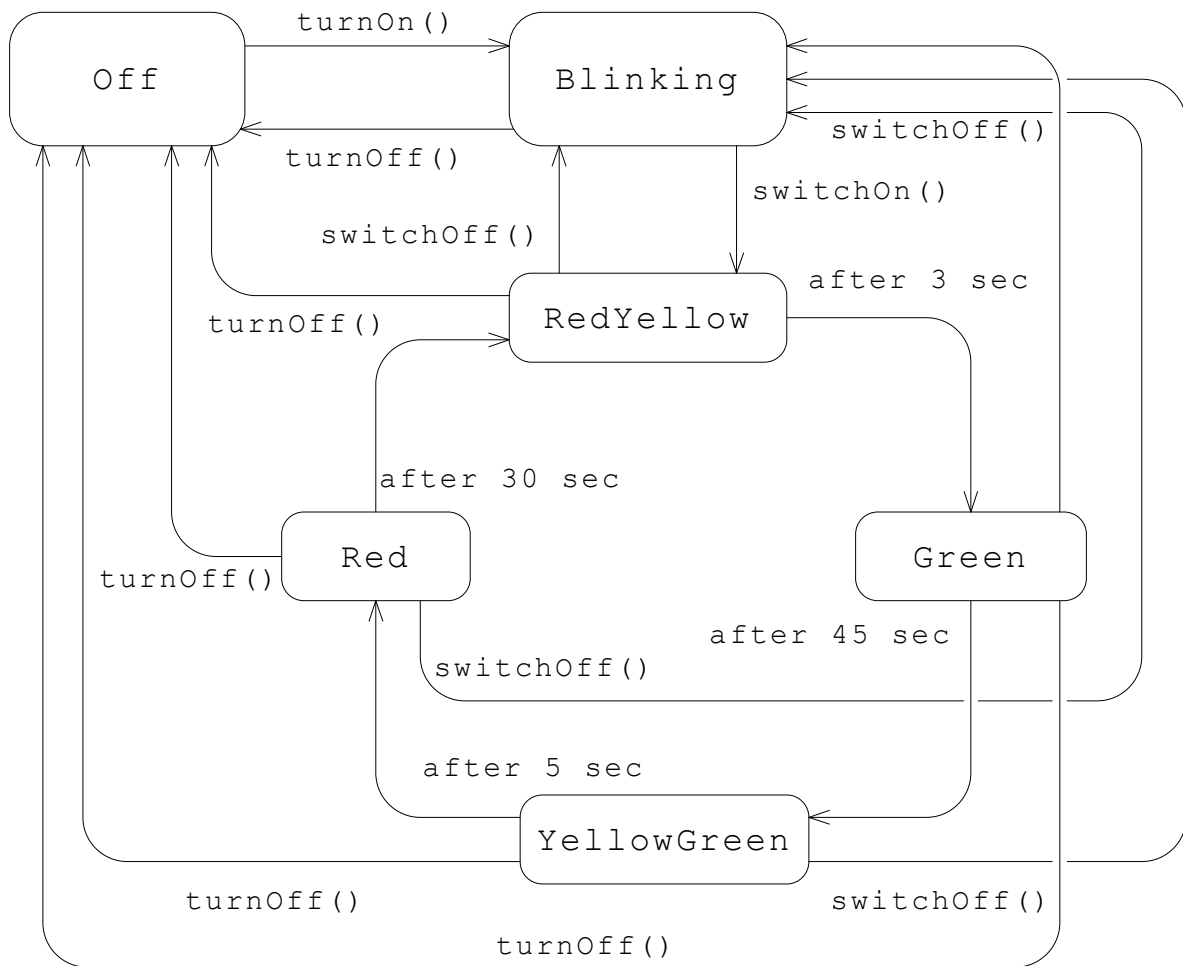


Рис. 4.7. Эквивалентная диаграмма, не содержащая составных состояний

<sup>17</sup> Но состояния не являются классификаторами, поэтому прямо использовать отношение обобщения было бы синтаксически неправильным.

#### 4.2.5. Специальные состояния

Перейдем к рассмотрению специальных состояний. Прежде всего, специальное состояние состоянием не является, в том смысле, что автомат не может в нем "пребывать" и оно не может быть текущим активным состоянием.

***Начальное состояние** (*initial state*) — это специальное состояние, соответствующее ситуации, когда машина состояний еще не работает.*

На диаграмме начальное состояние изображается в виде закрашенного кружка. Начальное состояние не имеет таких составляющих, как действия на входе, выходе и внутренняя активность, но оно обязано иметь исходящий переход,<sup>18</sup> ведущий в то состояние, которое будет являться по настоящему первым состоянием при работе машины состояний. Исходящий переход из начального состояния не может иметь события перехода, но может иметь сторожевое условие. В последнем случае должны быть определены несколько переходов из начального состояния, причем один из них обязательно должен срабатывать. В программистских терминах начальное состояние — это метка точки входа в программу. Управление не может задержаться на этой метке. Даже графически типичный случай начального состояния с одним непомеченным переходом очень похож на бытовую пиктограмму "начинать здесь". Начальное состояние может иметь действие на переходе — это действие выполняется до начала работы машины состояний.

Насколько обязательным является использование начального состояния на диаграмме автомата? Этот вопрос не имеет однозначного ответа — все зависит от ситуации. Например, в диаграммах на рис. 4.6 и рис. 4.7 мы обошлись без начального

---

<sup>18</sup> Разумеется, начальное состояние не может иметь входящих переходов — машина состояний не может вернуться в ситуацию до начала своей работы.

состояния, и поведение светофора не стало от этого менее понятным. Однако в других случаях наличие начального состояния может быть желательно или даже необходимо. Прежде всего, если имеется переход на границу составного состояния, то внутри этого составного состояния обязано присутствовать начальное состояние — в противном случае неясно, куда же ведет данный переход. Далее, если машина состояний описывает поведение программного объекта, создаваемого и уничтожаемого в программе, то присутствие начального состояния на диаграмме является весьма желательным: начальное состояние показывает, в каком состоянии находится объект при создании его конструктором (а в действия на переходе из начального состояния удобно поместить инициализацию атрибутов). С другой стороны, если начало и окончание жизненного цикла объекта, поведение которого моделируется, выходят за пределы моделируемого периода (например, нас не интересует ни процесс изготовления новых светофоров, ни процесс утилизации отслуживших свое), то начальное состояние на диаграмме состояний является излишним и может даже мешать восприятию.

*Заключительное состояние (final state) — это специальное состояние, соответствующее ситуации, когда машина состояний уже не работает.*

На диаграмме заключительное состояние изображается в виде закрашенного кружка, который обведен дополнительной окружностью.<sup>19</sup> Подобно начальному состоянию, заключительное состояние не имеет таких составляющих, как действия на входе, выходе и внутренняя активность, но имеет входящий переход,<sup>20</sup> ведущий из того состояния, которое является последним состоянием в данном сеансе работы конечного автомата.

---

<sup>19</sup> Жаргонное название этого символа — "бычий глаз".

<sup>20</sup> Разумеется, заключительное состояние не может иметь исходящих переходов — чтобы машина состояний заново заработала, ее нужно снова запустить.

Вообще говоря, работа конечного автомата может завершаться несколькими различными способами. Это соответствует общепринятой программистской практике: программа может иметь вариант нормального завершения и несколько вариантов завершения при возникновении исключительной ситуации или при ошибке. Отражая данную особенность поведения на диаграмме состояний, можно указать несколько переходов в одно и то же заключительное состояние. Синтаксически это допустимо. Однако мы настойчиво рекомендуем так не делать и помещать на диаграмму столько заключительных состояний, сколько в действительности существует семантически различных вариантов завершения работы данной машины состояний.

Прежде чем переходить к описанию других специальных состояний, еще раз уточним связь между составными состояниями, переходами между ними, начальным и заключительным состоянием. Напомним, что:

- если имеется входящий переход в составное состояние, то машина состояний, вложенная в данное составное состояние, **обязана** иметь начальное состояние;

- если машина состояний, вложенная в составное состояние, имеет заключительное состояние, то данное составное состояние **может** иметь исходящий переход по завершении;

- машина состояний верхнего уровня **считается** вложенной в составное состояние, которое не имеет ни исходящих, ни входящих переходов.

Следующее специальное состояние, которое мы рассмотрим — историческое состояние. *Историческое состояние* может использоваться во вложенной машине состояний внутри составного состояния. При первом запуске машины состояний историческое состояние означает в точности тоже, что и начальное: оно указывает на состояние, в котором находится машина в начале работы. Если в данной машине состояний используется историческое состояние, то

при выходе из объемлющего составного состояния запоминается то состояние, в котором находилась вложенная машина перед выходом. При повторном входе в данное составное состояние в качестве текущего состояния восстанавливается то запомненное состояние, в котором машина находилась при выходе. Проще говоря, историческое состояние заставляет автомат помнить, в каком состоянии его прервали в прошлый раз и "продолжать начатое".

Историческое состояние имеет две разновидности.

***Поверхностное историческое состояние** (shallow history state) запоминает, какое состояние было активным на том же уровне вложенности, на каком находится само историческое состояние.*

***Глубинное историческое состояние** (deep history state) помнит не только активное состояние на данном уровне, но и на всех вложенных уровнях.*

Рассмотрим пример из информационной системы отдела кадров.

## ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

*Информационная система должна отслеживать состояние, в котором находятся сотрудники, а именно: в офисе, в отпуске или на больничном. В случае болезни действует следующее правило – если сотрудник заболел в отпуске, то отпуск прерывается, а по выздоровлении возобновляется.*

Согласно техническому заданию, если сотрудник заболел, находясь в отпуске, то отпуск прерывается, а по выздоровлении возобновляется. Для того чтобы построить модель такого поведения, нужно воспользоваться историческим состоянием. В данном случае достаточно поверхностного исторического состояния (1) (shallow history state), поскольку на данном уровне вложенности все состояния уже простые. На рис. 4.8 приведен соответствующий фрагмент машины состояний.

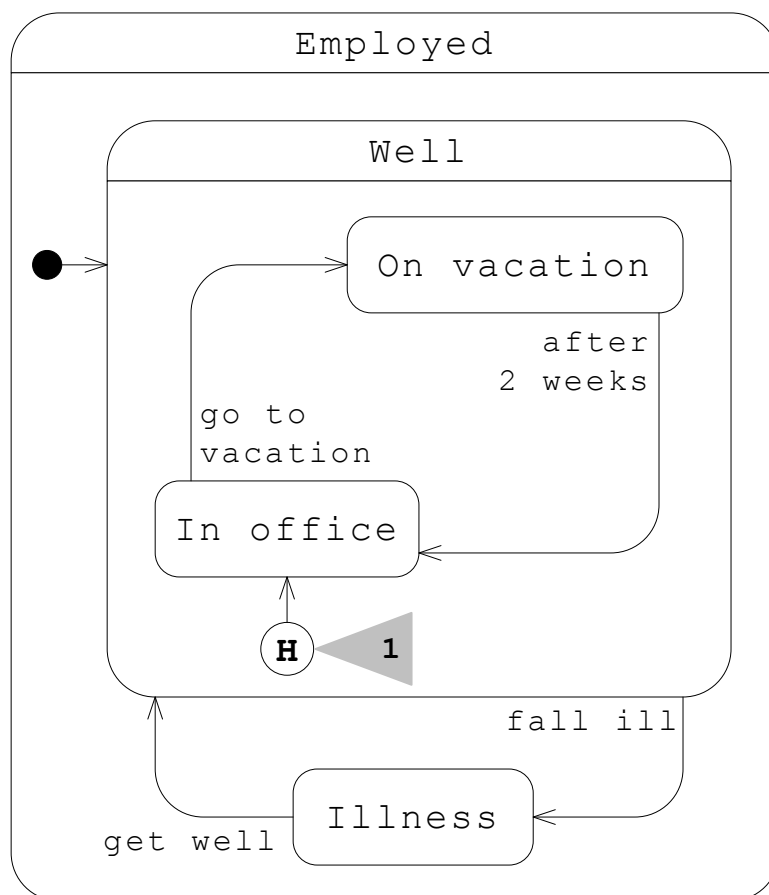


Рис. 4.8. Историческое состояние

#### 4.2.6. Вложенные машины состояний

Если мы хотим описать на UML действительно сложное поведение, то мы должны иметь возможность сделать это по частям, используя принцип "разделяй и властвуй". И такая возможность в UML предусмотрена — это ссылочное состояние и состояния заглушки в UML 1, которым на смену в UML 2 пришел вложенный автомат с точками входа и выхода.

Эти механизмы похожи, но не тождественны. Мы изложим их в историческом порядке: сначала UML 1, потом UML 2.

**Ссылочное состояние** (*submachine state*) — состояние, которое обозначает вложенный в него автомат.

На диаграмме ссылочное состояние изображается в виде фигуры простого состояния с именем, которому предшествует ключевое

слово `include` (1 на рис. 4.9). Семантика ссылочного состояния заключается в следующем. Если на диаграмме присутствует ссылочное состояние, то это означает, что в модели вместо ссылочного состояния присутствует составное состояние (и, соответственно, вложенный автомат), на которое делается ссылка. Таким образом, на одной диаграмме мы можем представить поведение, нарисовав его "крупными мазками", т. е. в терминах составных состояний верхнего уровня, а на других диаграммах раскрыть содержание составных состояний, нарисовав соответствующие автоматы и т. д. Критерий Дейкстры хорошего стиля программирования — "правильно написанный модуль должен помещаться на одной странице" — соблюдается.

Прежде чем привести пример, поговорим о переходах между составными состояниями. Их можно подразделить на два типа: переходы "в" и "из" составных состояний (т. е. переходы, пересекающие границы составных состояний), и переходы, начинающиеся и заканчивающиеся на границах состояний. Вообще говоря, без переходов, пересекающих границы состояний, можно обойтись (равно как и без многого другого), но жаль упускать возможность. Если переход начинается и заканчивается на границе состояний, то вся работа вложенного автомата инкапсулирована в составном состоянии и никаких проблем нет — ссылочного состояния достаточно для включения одного автомата внутрь другого. Однако, если такие переходы есть (а мы договорились, что не хотим от них отказываться), то возникает проблема: грубо говоря, нам нужно провести стрелку с одной диаграммы на другую. Решением этой проблемы в UML 1 является состояние заглушки.

***Состояние заглушка** (*stub state*) — это специальное состояние, которое обозначает в ссылочном состоянии некоторое вложенное состояние того составного состояния, на которое делается ссылка.*

Звучит замысловато, но все очень просто: мы разрешаем себе показать в ссылочном состоянии необходимый нам минимум деталей

того автомата, который скрыт в ссылочном состоянии. А нужны нам только имена вложенных состояний, в которые или из которых делается переход. На диаграмме состояние заглушка изображается в виде короткой вертикальной (или горизонтальной) черты внутри фигуры ссылочного состояния и с именем соответствующего вложенного состояния (2 и 3 на рис. 4.9). У этой черты начинается или заканчивается стрелка перехода, пересекающего границу ссылочного состояния.<sup>21</sup>

Приведем пример из информационной системы отдела кадров, чтобы проиллюстрировать приведенные сухие определения.

## ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

*Вновь принимаемый сотрудник всегда принимается с испытательным сроком. Ранее работавший сотрудник всегда принимается без испытательного срока.*

Давайте рассмотрим жизненный цикл сотрудника на предприятии, и раскроем детали поведения объекта `Person`, находящегося в самом важном для предприятия состоянии — `Employed`, исходя из требований технического задания. В этом нам поможет ссылочное состояние. На рис. 4.9 приведена диаграмма верхнего уровня, описывающая поведение в целом, без деталей. Состояние `Employed` — ссылочное (1), что подразумевает наличие другой диаграммы, на которой раскрыта внутренняя структура состояния `Employed`. Состояние заглушка `On trial` (2) указывает, что внутри составного состояния `Employed` есть вложенное состояние `On trial`. Состояние заглушка `On staff` (3) указывает, что внутри составного состояния `Employed` есть еще одно вложенное состояние `On staff`. Данные состояния выявлены на диаграмме верхнего уровня, чтобы подчеркнуть, что переходы по событию `hire()` ведут именно в эти вложенные состояния, а не в другие

---

<sup>21</sup> Если бы такого перехода не было, то и состояние "заглушка" было бы ненужным.



(новый сотрудник сначала обязательно должен попасть в состояние On trial, а ранее работавший разу оказывается в состоянии On staff).

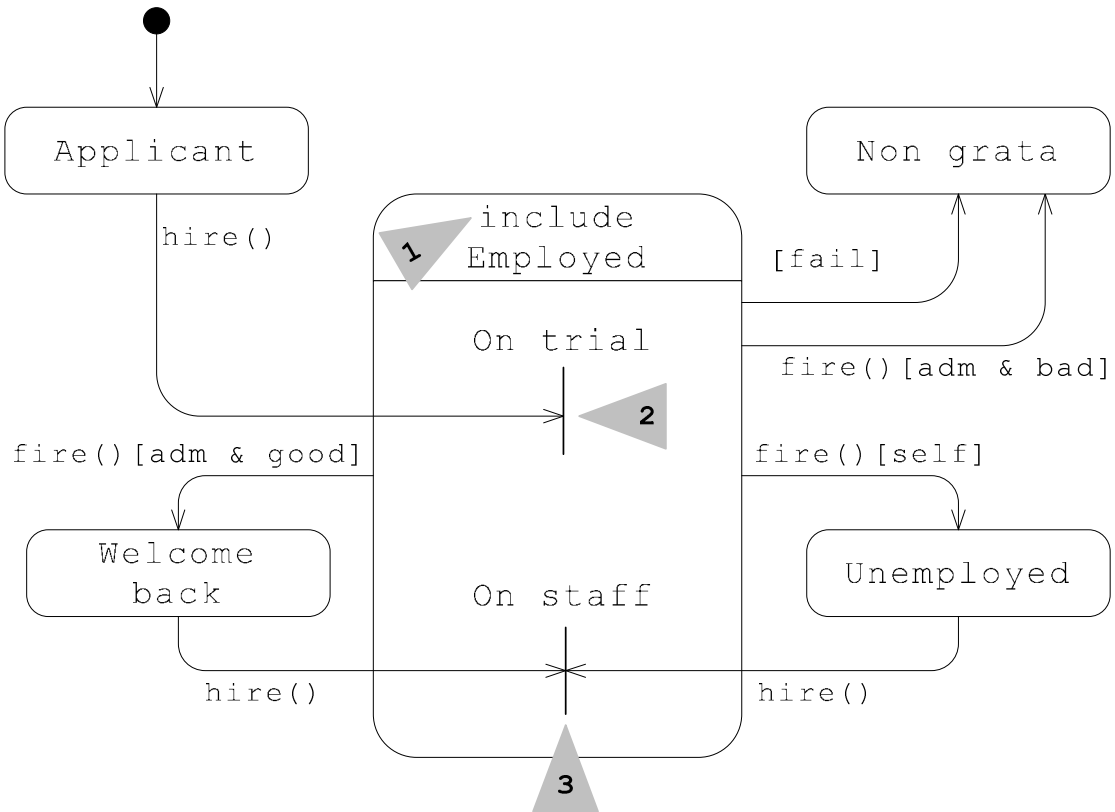


Рис. 4.9. Ссылочное состояние и состояние заглушка

На другой диаграмме (рис. 4.10) раскрывается внутренняя структура состояния Employed.

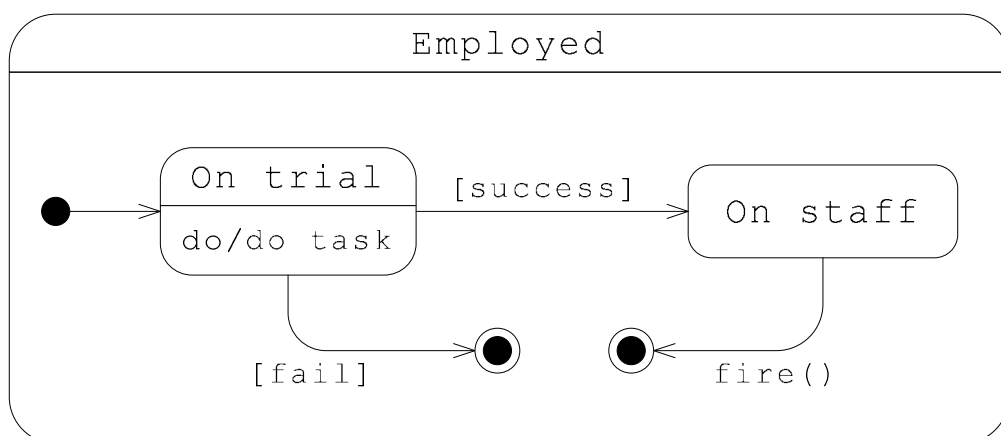


Рис. 4.10. Составное состояние, раскрывающее ссылочное состояние

Здесь мы сразу показали все вложенные состояния Employed. В данном случае диаграмма получилась простой, но если бы все нужные детали перестали помещаться на диаграмму, то нужно было бы ввести еще несколько ссылочных состояний и раскрыть их детали на отдельных диаграммах.

Для возможности практического использования метода пошагового уточнения при описании сложного поведения с помощью автоматов в UML 2 явным образом введено понятие вложенного автомата вместо понятия ссылочного состояния и заглушки.

**На диаграмме верхнего уровня вложенный автомат указывается как простое ссылочное состояние, а на диаграмме нижнего уровня вложенный автомат раскрывается как составное состояние.** При этом строка имени ссылочного состояния имеет следующий синтаксис.

ИМЯ СОСТОЯНИЯ : ИМЯ ВЛОЖЕННОГО АВТОМАТА

Подчеркнем, что ссылочное состояние заимствует структуру и поведение вложенного автомата, но не владеет им композиционно. Тот же самый вложенный автомат может быть использован в другом месте и в другой машине состояний. Таким образом, в UML 2 вложенные автоматы по назначению и способу использования подобны подпрограммам в обычных языках программирования.

Вместо понятия заглушки указываются точки входа и выхода на границе вложенного автомата. Если продолжать аналогию с подпрограммами, то это аналог входных и выходных параметров, через которые во вложенный автомат передаются соответствующие события.

Переходы на границу и с границы ссылочного состояния имеют тот же самый смысл, что и переходы на границу и с границы составного состояния, как если бы копия составного состояния вложенного автомата была подставлена вместо ссылочного состояния.

Уточненное в UML 2 понятие вложенного автомата представляется нам настолько простым и естественным, что вместо дальнейших пояснений мы просто сошлемся на рис. 4.11 и рис. 4.12, где еще раз представлен пример, аналогичный примеру на рис. 4.9 и рис. 4.10.

Мы надеемся, что читатель оценит красоту и точность нотации UML 2.

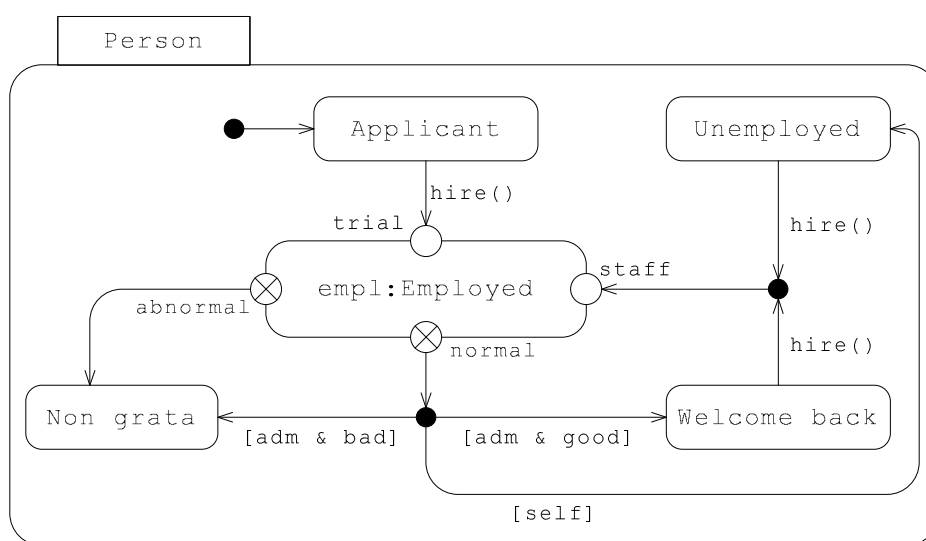


Рис. 4.11. Составное состояние, использующее вложенный конечный автомат

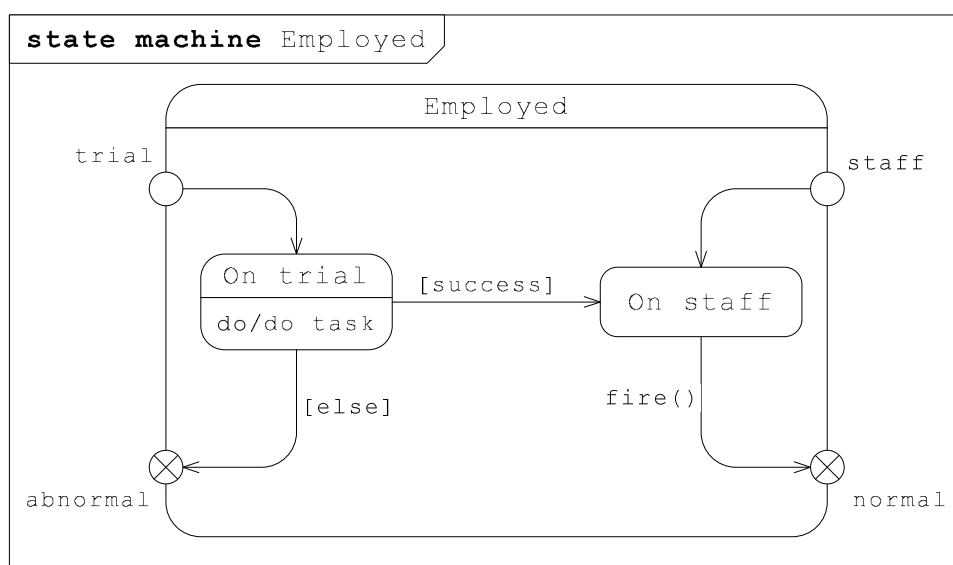


Рис. 4.12. Составное состояние, раскрывающее вложенную машину состояний

#### 4.2.7. События

Различные типы состояний, рассмотренные в предыдущем разделе, позволяют задавать структуру автомата. Но основная семантическая нагрузка при описании поведения падает на переходы различных типов. Помимо структурных составляющих — исходного и целевого состояний — переход может быть нагружен событием перехода, сторожевым условием и действиями на переходе.

В UML используются четыре типа событий:

- событие вызова,
- событие сигнала,
- событие таймера,
- событие изменения.

**Событие вызова** (*call event*) — это событие, возникающее при вызове метода класса.

Если событие вызова используется как событие перехода в машине состояний, описывающей поведение класса, то класс должен иметь соответствующую операцию. Событие вызова — наиболее часто используемый тип событий перехода. В большей части рассмотренных примеров в качестве событий перехода использовались именно события вызова. Поскольку событие вызова — это вызов метода, то оно может иметь аргументы, как всякий вызов метода. Значения аргументов могут использоваться в действиях перехода. Если метод возвращает значение, то этот факт отмечается с помощью действия возврата в последовательности действий данного перехода.

Проиллюстрируем вышесказанное на примере информационной системы отдела кадров. Определим в классах `Person` и `Position` операции, приведенные на рис. 4.13.

Операция `hasPosition()` класса `Person` проверяет, занимает ли сотрудник какую-нибудь должность, а операция `assign()` переводит сотрудника на новую должность `newPos` и возвращает значение `true`, если перевод успешно произведен.

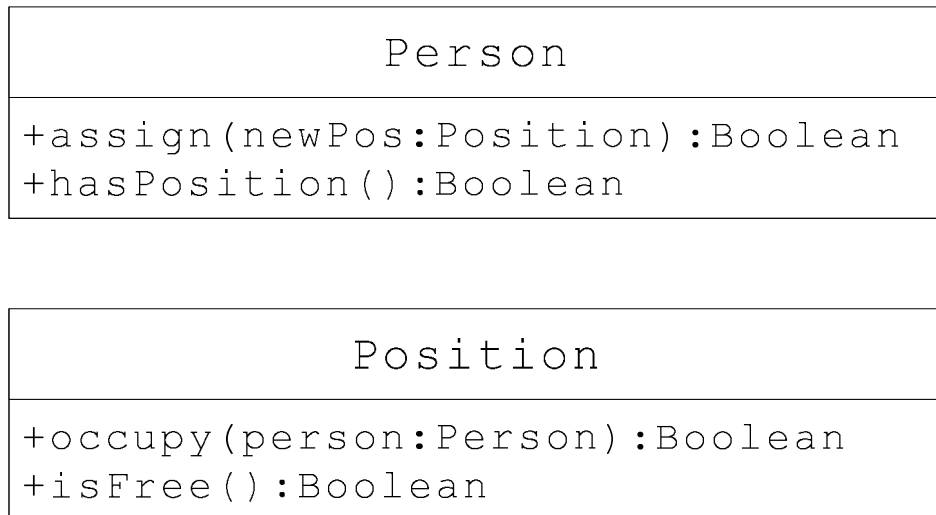


Рис. 4.13. Операции, определенные в классах Person и Position

Операция `isFree()` класса `Position` проверяет, свободна ли должность, и операция `occupy()` позволяет назначить сотрудника на должность и в случае успешного завершения возвращает значение `true`.

Тогда типичное поведение операции `assign()` можно описать диаграммой состояний, приведенной на рис. 4.14.

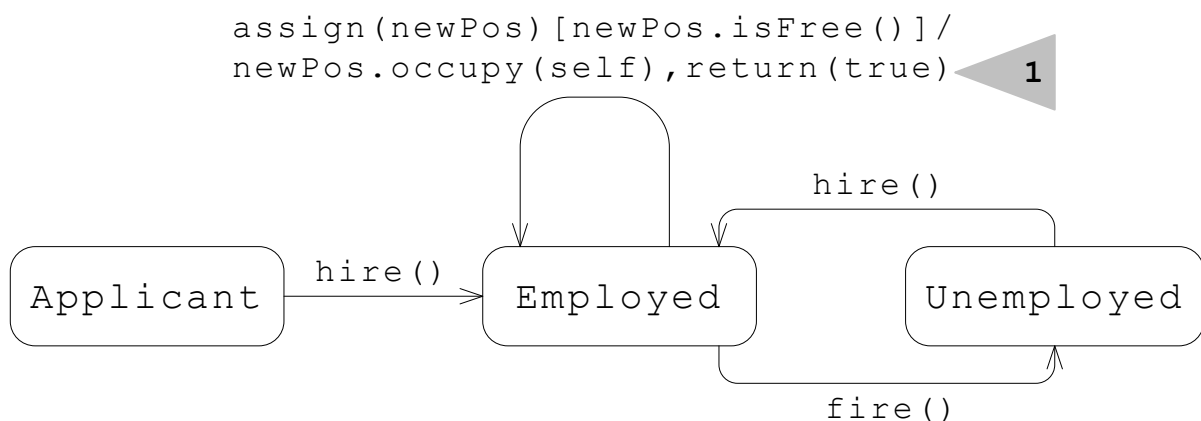


Рис. 4.14. Событие вызова

На этом рисунке наибольшее внимание заслуживает переход из состояния `Employed` в себя. Давайте разбираться. По событию

`assign()` данный переход возбуждается, но для того, чтобы этот переход завершился должно быть выполнено условие `newPos.isFree()`. Если должность свободна (`newPos.isFree()` вернул `true`), то тогда осуществляется вызов метода `newPos.occupy(self)`, а вызов метода `assign(newPos)` возвращает `true` (1 на рис. 4.14).

***Событие сигнала** (*signal event*) — это событие, возникающее при посылке сигнала.*

Чтобы разъяснить, что такое событие сигнала в UML, нужно рассмотреть, прежде всего, саму концепцию сигнала. Здесь опять имеет место пересечение моделирования структуры и поведения: определяются сигналы в структурной части модели, а используются в поведенческой.

*Синтаксически **сигнал** (в UML 1) — это экземпляр класса со стереотипом «*signal*». В UML 2 сигнал — это самостоятельный классификатор.*

*Семантически **сигнал** — это именованный объект, который создается другим объектом (отправителем) и обрабатывается третьим объектом (получателем).*

Сигнал может иметь атрибуты (параметры). Сигнал может иметь операции. Одна из них считается определенной по умолчанию. Она имеет стандартное имя `send()` и параметр, являющийся множеством объектов, которым отправляется сигнал. Это операция — конструктор, который создает экземпляр классификатора, то есть сигнал. Объявлять эту операцию не нужно. Можно объявлять другие операции, которые служат для доступа к значениям атрибутов сигнала, но и это не обязательно.

Концепция сигнала в UML принадлежит к числу наиболее фундаментальных и имеет ясную и строго определенную семантику. Объект, являющийся отправителем, обращается к классификатору сигнала (вызывает операцию `send()`), указывая аргументы сигнала (значения атрибутов) и целевое множество объектов, которым должен

быть отправлен сигнал. После этого объект-отправитель продолжает свою работу — дальнейшее его не касается.

На диаграмме классов классификатор сигнал связывается с классом отправителем зависимостью со стереотипом «send». Объекты, которым отправляется сигнал, обязаны иметь операцию для получения и обработки сигнала. Такая операция имеет стереотип «signal», ее имя совпадает с именем классификатора сигнала, а имена и типы параметров совпадают с именами и типами атрибутов сигнала. Операция получения сигнала не может возвращать результат. После того, как объект-отправитель передал классификатору сигнала все необходимые данные, тот создает новый объект — экземпляр сигнала и отправляет его копии всем объектам целевого множества, т. е. вызывает в объектах целевого множества операции приема сигнала с указанными значениями аргументов.

Сигналы подходят для описания асинхронных видов взаимодействия. Все получатели сигнала должны быть экземплярами активных классов, т. е. иметь свой поток управления.

В UML существует еще одно важное понятие, связанное с асинхронным взаимодействием — *исключение* (exception). В разных версиях UML исключения определены и используются по-разному.

*Исключения* в UML 1 во всем подобны сигналам (на уровне метамодели исключение определено как подкласс сигнала), в то время как в UML 2 от исключения осталось только действие, которое генерирует исключение (RaiseExceptionAction) и элемент (ExceptionHandler), который описывает действия для обработки исключения. Как видно, в UML 2 более сильно выражен тот факт, что для поддержки исключений используется встроенный механизм обработки, который имеется в большинстве современных объектно-ориентированных систем программирования и состоит в том, что прерывается нормальный поток управления без помех для основной логики программы.

*Событие таймера (time event) — это событие, которое возникает, когда истек заданный интервал времени с момента попадания автомата в данное состояние.*

Синтаксически событие таймера записывается с помощью ключевого слова *after*, за которым указывается выражение, определяющее длину интервала времени. Семантически событие таймера означает следующее. Подразумевается, что у состояния имеется таймер, который сбрасывается в 0 (начинает отсчет), когда автомат переходит в данное состояние (напомним, что автомат считается перешедшим в состояние, когда закончено выполнение всех действий, предписанных переходом). Таймер ведет отсчет времени. Если до истечения указанного интервала времени сработает другой переход, то событие таймера не возникает. Когда указанный интервал времени истекает, наступает событие таймера и возбуждается соответствующий переход.

Если переход срабатывает, то автомат переходит в новое состояние.

Если переход по событию таймера не срабатывает (из-за ложности сторожевого условия), то событие таймера теряется, и таймер продолжает отсчет времени, так что позже может сработать другой переход по событию таймера с большим интервалом времени. Событие таймера не может быть отложено.

Приведем пример использования события таймера на переходах в конечных автоматах (рис. 4.15).

## ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

*Информационная система должна хранить информацию не только о работающих, но и об уволенных сотрудниках. По прошествии 10 лет после увольнения, если эта информация не нужна более, она уничтожается.*



Таким образом, по истечении десяти лет после увольнения объект, представляющий уволенного сотрудника (запись в базе данных) уничтожается, если только этот объект не помечен специальным образом (сторожевое условие `keepForever`). Если объект оставлен в базе, то через десять лет проверка повторяется.

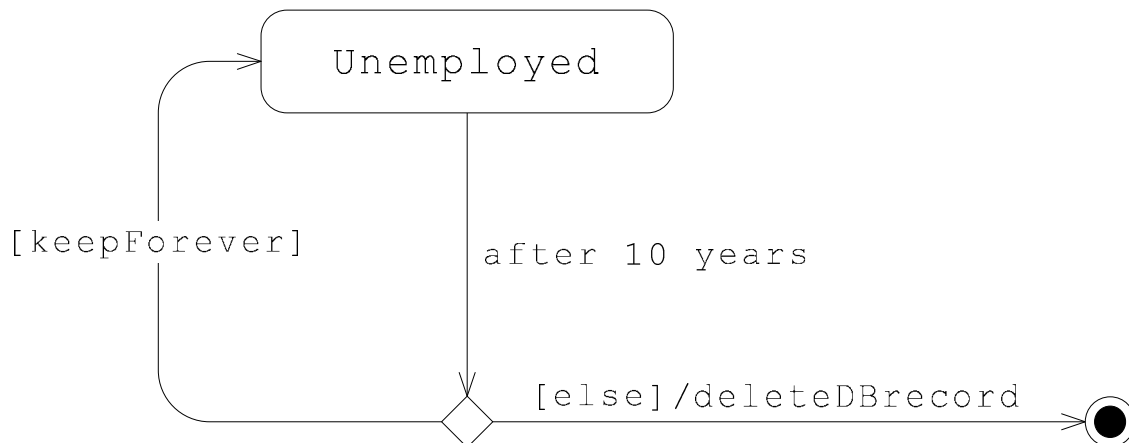


Рис. 4.15. Переход по событию таймера со сторожевым условием

В UML 2 появился очень полезный вариант события таймера, которое связано не с интервалом времени пребывания в локальном состоянии, а с глобальным временем — системными часами. Данное событие записывается с помощью ключевого слова `at`, за которым нужно указать некоторый момент абсолютного времени (год, месяц, день, час, минуту, секунду...). Когда заданный момент наступает, происходит событие таймера. Если указать момент в прошлом, событие `at` никогда не наступит.

**Событие изменения** (*change event*) — это событие, которое возникает, когда некоторое логическое условие становится истинным, будучи до этого ложным.

Синтаксически событие изменения записывается с помощью ключевого слова `when`, за которым указывается логическое выражение (условие). Семантически событие изменения означает следующее. Подразумевается, что в системе имеется механизм, работающий как демон (например, приложение, запущенное в

фоновом режиме), который генерирует событие, если в результате изменения состояния системы изменяется значение логического выражения. Если выражение, являющееся аргументом события изменения, принимает значение `true` (имея до этого значение `false`), то переход возбуждается. Если выражение имеет значение `true` в тот момент, когда автомат переходит в данное состояние, то переход сразу возбуждается. Если переход срабатывает, то автомат, как обычно, переходит в новое состояние. Если переход не срабатывает, то событие изменения теряется. При этом если условие продолжает оставаться истинным, то нового события изменения не возникает. Для того чтобы снова возникло событие изменения, нужно, чтобы условие стало сначала ложным, а потом истинным.

Рассмотрим элементарный пример из информационной системы отдела кадров.

## ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

*По достижении определенного возраста (55 лет для женщин и 60 лет для мужчин) сотрудник увольняется на пенсию.*

Реализация данного требования приведена на рис. 4.16.

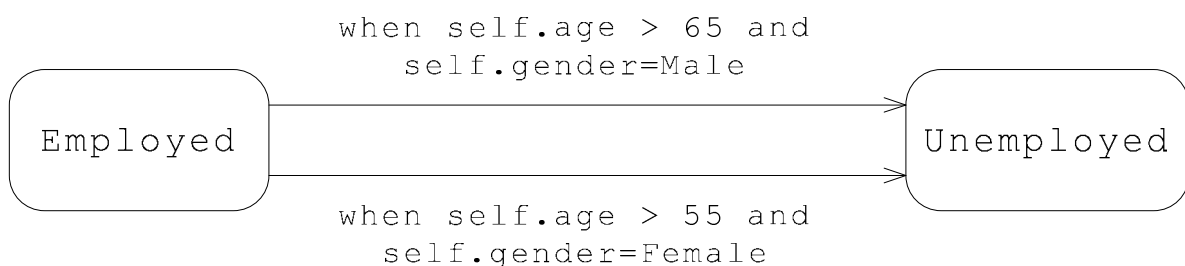


Рис. 4.16. Переход по событию изменения

Заметим, что условие достижения пенсионного возраста сотрудников записано с помощью специального языка (OCL) и подразумевает, что класс `Person` (рис. 4.16 — это часть диаграммы автомата именно этого класса) имеет атрибут с именем `age`.

### 4.3. ДИАГРАММЫ ДЕЯТЕЛЬНОСТИ

*Диаграммы деятельности* являются средством описания поведения в UML, причем их место в языке допускает некоторые разночтения.

Основных сущностей и отношений, применяемых на диаграмме деятельности, в некотором смысле еще меньше, чем на диаграмме автомата (хотя, казалось бы, меньше уже некуда — на диаграмме состояний только состояния и переходы). Дело в том, что основная сущность на диаграмме деятельности является частным случаем простого состояния (состояние деятельности), а основное отношение — частным случаем простого перехода (переход по завершении). В тоже время всевозможных украшений и вариантов нотации на диаграмме деятельности, особенно в UML 2, намного больше, чем на диаграмме автомата. Поэтому, чтобы не затеряться в деталях, в следующем параграфе мы обсудим содержание базовых понятий, затем определим основные сущности и отношения, применяемые на диаграммах деятельности, а уже потом перейдем к примерам и картинкам.

#### 4.3.1. Действие и деятельность

Мы уже использовали понятие действия, постулировав, что **действие является атомарным, непрерываемым извне, безусловным и завершаемым**. Действия используются на переходах и в состояниях машины состояний UML и играют там ключевую роль. Каждое действие имеет присущие ему наборы входных и выходных параметров (они называются *контактами* (pin)). Как правило, эти наборы фиксированы по числу и типам параметров, но бывают и действия с изменяемым числом параметров. Среди элементарных действий нет привычных действий по управлению ходом выполнения программы (ветвления, циклы, переходы и т. д.) — управление не считается примитивом и вынесено на следующий уровень, уровень деятельности.

Вторым важнейшим понятием, применяемым при описании поведения, является деятельность.

*Деятельность (activity) в UML — это описание поведения в форме графа деятельности.*

Деятельность в UML моделирует то же, что и действие, т. е. какую-то содержательную активность во время работы системы; в этом смысле деятельность подобна действию, но деятельность противопоставляется действию по всем характеристическим признакам. В табл. 4.1 проведено сопоставление понятий "действие" и "деятельность" в UML.

Таблица 4.1

**Сопоставление действия и деятельности**

Характеристика	Действие	Деятельность
Внешнее событие	Не прерывает выполнения	Может прерваться и завершить выполнение
Завершаемость	Всегда завершается самостоятельно	Может продолжаться неограниченно долго
Внутренняя структура	Не моделируется в UML	Может быть раскрыта на отдельной диаграмме
Время выполнения	Пренебрежимо мало	Продолжительное

Если нам неважно различие между действием и деятельностью и нужно употребить более общее понятие, то применяется термин *активность*.

#### 4.3.2. Граф деятельности

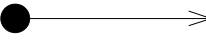
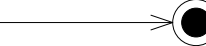
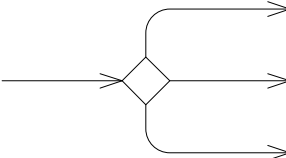
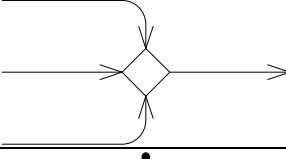
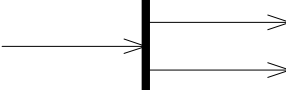
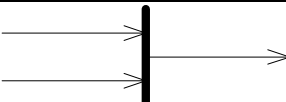


*Семантически граф деятельности (activity graph) — это множество сущностей, которыми являются действия или деятельности, и отношения между этими сущностями, которые задают порядок их выполнения.*

*Синтаксически граф деятельности — это нагруженный ориентированный (псевдо) гиперграф, в котором используются узлы четырех типов: узлы действий, узлы деятельности, узлы управления и узлы объектов, а дуги являются потоками управления или потоками данных.*

На диаграмме деятельности применяется ряд значков, которые на самом деле не являются сущностями, хотя и являются узлами графа деятельности. Это так называемые узлы управления. Для UML 1 узлы управления перечислены в табл. 4.2.

Таблица 4.2

Узлы управления UML 1

Название	Изображение	Что обозначает
Начальное состояние (initial node)		Начало деятельности
Заключительное состояние (final node)		Завершение деятельности
Разветвление управления (decision node)		Начало альтернативных ветвей деятельности
Объединение управления (merge node)		Конец альтернативных ветвей деятельности
Развилка управления (fork node)		Начало параллельных ветвей деятельности
Слияние управления (join node)		Конец параллельных ветвей деятельности
Посылка сигнала (send)		Действие посылки сигнала
Прием сигнала (accept)		Ожидание события прихода сигнала

На диаграмме деятельности UML 1 применяется один основной тип отношений — простые переходы по завершении (а также поток объектов). Переход по завершении не имеет переключающего события — событием является завершение внутренней активности (деятельности) в состоянии. Как правило, исходящий переход по завершении один; если их несколько, они должны быть снабжены сторожевыми условиями, образующими полную дизъюнкцию

систему предикатов. Кроме того, в UML 1 можно<sup>22</sup> использовать и переходы, возбуждаемые событиями. Срабатывание такого перехода означает прерывание выполнения деятельности в состоянии и переход в другое состояние. Однако повторим сказанное при описании диаграмм автомата: использование переходов, управляемых событиями, настолько же неуместно на диаграмме деятельности, насколько неуместно использование на диаграмме автомата переходов по завершении. Мы не рекомендуем такой стиль моделирования.

Таким образом, в рамках семантики машины состояний авторы UML 1 сумели описать семантику обычных блок-схем (в которых нет никаких событий, а есть последовательная передача управления следующей деятельностью по завершении предыдущей деятельности). Хорошо это или плохо — трудно сказать. Во всяком случае, в UML 2 от этого способа определения семантики отказались.

Приведем пример диаграммы состояний в стиле UML 1. Для этого рассмотрим увольнение сотрудника как бизнес-процесс, реализующий соответствующий вариант использования. Приведенная на рис. 4.17 блок-схема буквально воспроизводит текстовое описание сценария. Никаких пояснений, как именно выполняется, например, деятельность Написать заявление здесь нет, но бизнес-процесс описан совершенно ясно.

Операционная семантика графов деятельности в UML 2 определена иным способом, не через машины состояний, а через сети Петри. Это определение очень наглядно и изящно, хотя может быть не совсем привычно.

В основе определения лежит понятие *маркера* (token). Маркер может не содержать никакой дополнительной информации (пустой маркер) и тогда он называется *маркером управления* (control flow token) или же может содержать ссылку на объект или структуру данных, и тогда маркер называется *маркером данных* (data flow token).

---

<sup>22</sup> По крайней мере этому нет опровержения

Следует сразу подчеркнуть, что маркеры **не** являются элементами графа деятельности. **Маркеры — это абстрактные конструкции, которые вводятся для удобства описания динамического процесса выполнения статически определенного графа деятельности.** Никаких маркеров на диаграмме деятельности нет — на диаграмме деятельности присутствуют узлы. Но во время выполнения деятельности маркеры появляются, перемещаются между узлами и исчезают. Правила появления, перемещения и исчезновения маркеров описаны ниже, после описания типов узлов графа деятельности.

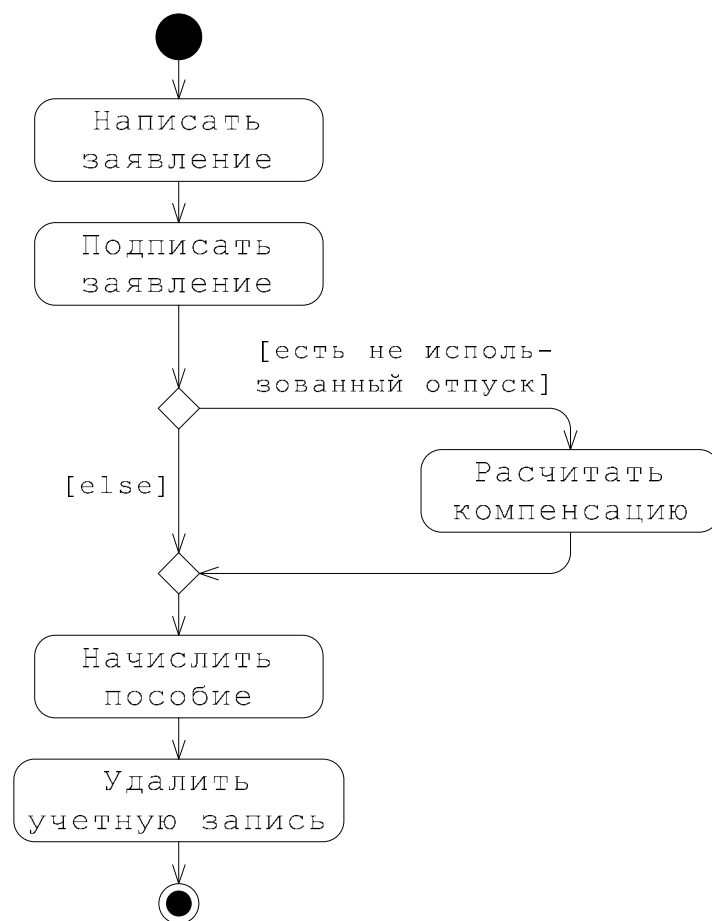


Рис. 4.17. Диаграмма деятельности UML 1

Узлы графов деятельности в UML 2 бывают трех видов: *узлы действий*, *узлы объектов* и *узлы управления*. Кроме того, на диаграммах деятельности UML 2 бывают составные узлы — они называются *областями*. Следует заметить, что каждый из видов узлов

делится на подвиды, так что всего вариантов сущностей довольно много. Мы охватим примерами и объяснениями все, но не все в этом параграфе.

Самыми важными являются *узлы действий* (action node). Действительно, трудно представить себе разумную деятельность, которая не включает каких либо действий. Номенклатура действий в UML 2 описана в предыдущем параграфе.

Мы включаем вызов деятельности в число действий и никак синтаксически не выделяем на диаграммах. В UML 2 между действием и деятельностью существует еще одно существенное различие. Все действия заранее предопределены и описаны в спецификации. Если разработчику нужно описать какой-либо особое поведение, он должен использовать для этого деятельность. При этом поведение можно описать отдельно, а в конструируемую в данный момент деятельность включить в действие по вызову деятельности.<sup>23</sup>

Узлы действий в UML 2 похожи на состояния действия в UML 1, но имеют важнейшее отличие. Для действий в UML 2 можно указать, что является входной и выходной информацией действия, задав входные и выходные контакты.

**Контакт** (*pin*) — это указание аргумента или результата действия.

Контакт может иметь имя и тип.

Именно через контакты путешествуют маркеры данных по графу деятельности в процессе выполнения. Через входные контакты действие принимает свои аргументы, а через выходные контакты выдает свои результаты.

---

<sup>23</sup> Некоторые авторы выделяют вызовы деятельности в отдельный вид узлов на графе деятельности и используют специальный значок в форме трезубца направленного вниз для графического выделения таких узлов. Трезубец показывает, что где-то существует другая диаграмма деятельности, на которой описана последовательность действий, составляющих данную деятельность.



Графически контакты изображаются в виде маленьких квадратиков, прикрепленных к сторонам фигуры, изображающей деятельность (см. третий вариант на рис. 4.18).

Следующим по порядку являются *узлы объектов* (object node). Они включают в себя следующие четыре разновидности:

- *контакт* — мы уже дали определение выше и несколько раз вернемся к этому важнейшему нововведению ниже. Фактически это то, во что превратился "объект в состоянии" из UML 1;

- *параметр деятельности* (activity parameter) — это тот же контакт, только он изображается на границе рамки деятельности. Контакты действия по вызову некоторой деятельности должны соответствовать по количеству, именам и типам параметрам этой деятельности;

- *центральный буфер* (central buffer) — искусственная конструкция, используемая в тех случаях, когда необходимо показать неоднозначность пути маркеров данных в графе деятельности. Центральный буфер имеет несколько входящих и несколько исходящих дуг. Он может принять маркер данных по любой из входящих дуг и немедленно отправить его далее по любой из исходящих дуг;

- *хранилище данных* (data store) — аналог понятия переменной, в которую можно несколько раз (с заменой данных) записать значение (поместить маркер данных), и потом неограниченное число раз его читать (брать копию маркера данных). В некотором смысле хранилище данных — это частный случай центрального буфера, применимый для хранения маркера данных.<sup>24</sup>

Предыдущий список должен породить у внимательного читателя недоумение: как же так, мы определяем контакт и как

---

<sup>24</sup> Хранилища данных применяются очень часто, это фактически локальные переменные, они нужны практически на любой диаграмме деятельности. Центральный буфер применяется очень редко, это особый случай параллелизма.

составляющую действия, и как отдельную сущность. Здесь нет противоречия. На рис. 4.18 представлены различные варианты нотации для контактов. Сверху изображен "объект в состоянии" (1) в нотации UML 1, а затем допустимая промежуточная форма (2). Характерная для UML 2 нотация (3) как видно гораздо лаконичнее принятой в UML 1. К основной нотации контакта в UML 2 можно добавить примечание со стереотипом «transformation» (4), чтобы специфицировать требуемое приведение типов. И, наконец, поток передаваемых данных может быть просто не специфицированным (5). Семантически все формы нотации эквивалентны.

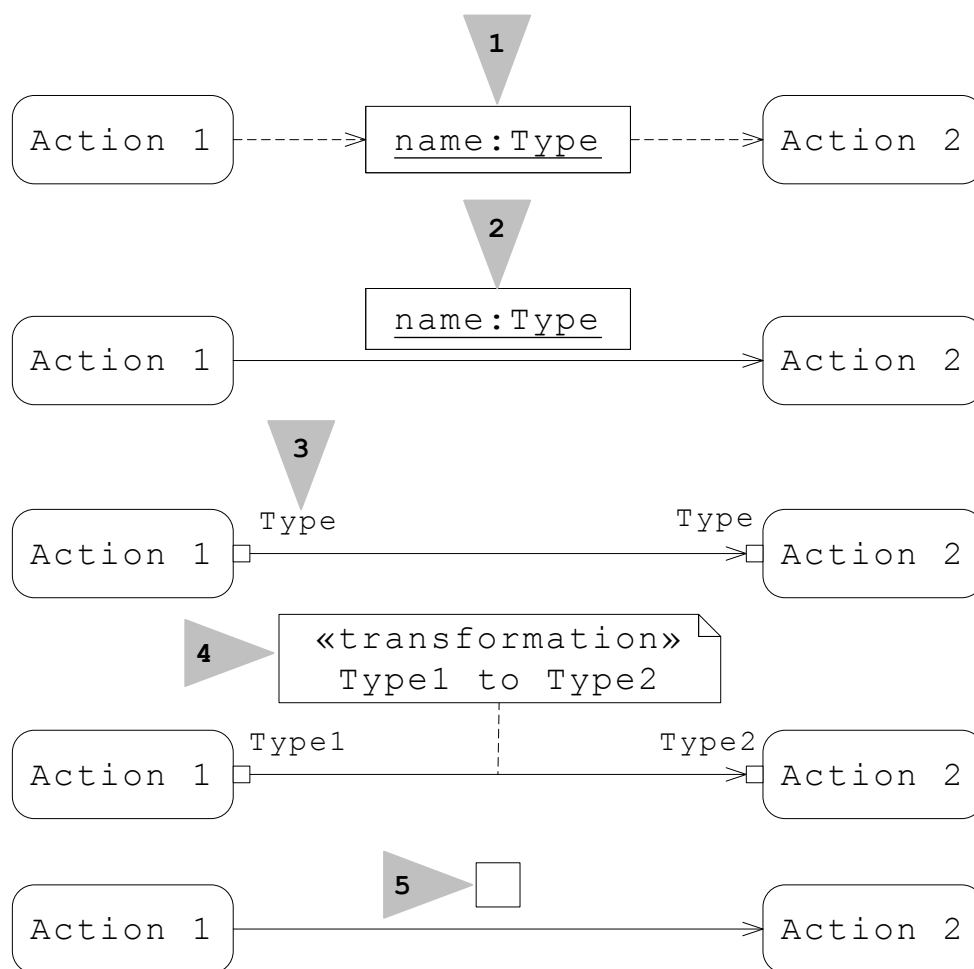

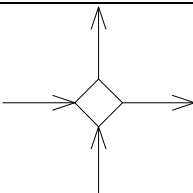
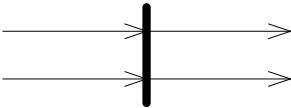



Рис. 4.18. Варианты нотации контактов

Третий тип узлов в графе деятельности — *узлы управления* (control node). В UML 2 используются те же узлы управления, что перечислены в табл. 4.2, с тремя дополнениями, которые приведены в табл. 4.3.

Таблица 4.3

**Дополнительные узлы управления UML 2**

Название	Изображение	Что обозначает
Заключительное состояние потока (final flow node)		Завершение одного потока в деятельности. Если в деятельности есть другие параллельные потоки, они продолжаются.
Комбинированное соединение/разветвление управления		Последовательность из узла соединения и узла разветвления.
Комбинированное слияние/развилка управления		Последовательность из узла слияния и узла развилки.
Прием сигнала от таймера		Узел, являющийся источником маркера управления по истечении заданного интервала времени

Теперь все готово, чтобы описать операционную семантику графов деятельности в UML 2. Семантика описывается в терминах правил определения того, в каких случаях какие дуги готовы передавать какие маркеры.

### **Правила для узлов управления**

1. Начальное состояние (№ 1 в табл. 4.2) создает один маркер управления и все исходящие дуги готовы передать этот маркер.

2. Если единственная входящая дуга развилки (№ 5 в табл. 4.2) готова передать маркер (управления или данных), то все исходящие дуги готовы одновременно (параллельно) передать копии этого маркера. Развилка создает параллельные потоки.

3. Если все входящие дуги слияния (№ 6 в табл. 4.2) готовы передать маркеры (управления или данных), то исходящая дуга готова передать маркер управления. Слияние обеспечивает синхронизацию потоков.

4. Если единственная входящая дуга разветвления (№ 3 в табл. 4.2) готова передать маркер (управления или данных), то те исходящие дуги разветвления, на которых сторожевые условия выполняются, готовы передать этот маркер.<sup>25</sup>

5. Если любая входящая дуга соединения (№ 4 в табл. 4.2) готова передать маркер (управления или данных), то единственная исходящая дуга соединения готова передать этот маркер.

6. Если хотя бы одна входящая дуга заключительного состояния потока (№ 1 в табл. 4.8) готова передать маркер, то заключительное состояние потока поглощает этот маркер.

7. Если хотя бы одна входящая дуга заключительного состояния деятельности (№ 2 в табл. 4.7) готова передать маркер, то заключительное состояние деятельности поглощает все маркеры управления и все маркеры данных, завершая, тем самым, выполнение деятельности.

### **Правила для узлов объектов**

1. Параметр деятельности (1 на рис. 4.19) создает один маркер данных и все исходящие дуги (2 на рис. 4.19) готовы передать этот маркер.

2. Хранилище данных (3 на рис. 4.19) поглощает один маркер данных и создает неограниченное количество его копий. Все выходные дуги хранилища (4 на рис. 4.19) всегда готовы передать копию хранимого маркера данных.

3. Центральный буфер перенаправляет маркеры данных, не создавая и не поглощая их. Как только входная любая входная дуга

---

<sup>25</sup> Но маркер один — конкуренция, кто первый готов, тот и получит маркер.

готова передать маркер, все выходные дуги готовы передать этот маркер.

4. Входной контакт действия (5 на рис. 4.19) поглощает маркер данных.

5. Выходной контакт действия (6 на рис. 4.19) создает маркер данных.

### **Правила для контактов и действий**

1. Если все дуги данных, входящие во все входные контакты действия, готовы передать маркеры данных, и если есть входящие дуги управления и хотя бы одна готова передать маркер управления, то действие выполняется.

2. Если действие выполнено, то все дуги данных, выходящие из выходных контактов, готовы передать маркеры данных, и если есть исходящие ребра управления, то те исходящие дуги управления, на которых сторожевые условия выполняются, готовы передать маркер управления.

Продолжим пример об увольнении сотрудника. Теперь мы рассмотрим этот пример не как бизнес-процесс высокого уровня, а как операцию информационной системы. Рассмотрим рис. 4.19.

Выполнение этого графа деятельности происходит следующим образом.

Начальное состояние (7 на рис. 4.19) готово передать маркер управления, а параметр деятельности (1 на рис. 4.19) готов передать маркер данных. Таким образом, все входящие дуги деятельности Get Person Info готовы передать маркеры и деятельность выполняется. В результате выполнения появляются три маркера данных в контактах pos, fnd, dpt, соответственно. Маркер данных fnd имеет булевский тип и его значение определяет, присутствует ли в базе данных объект p. Если это так, то pos и dpt содержат должность и подразделение увольняемого, в противном случае они пусты.

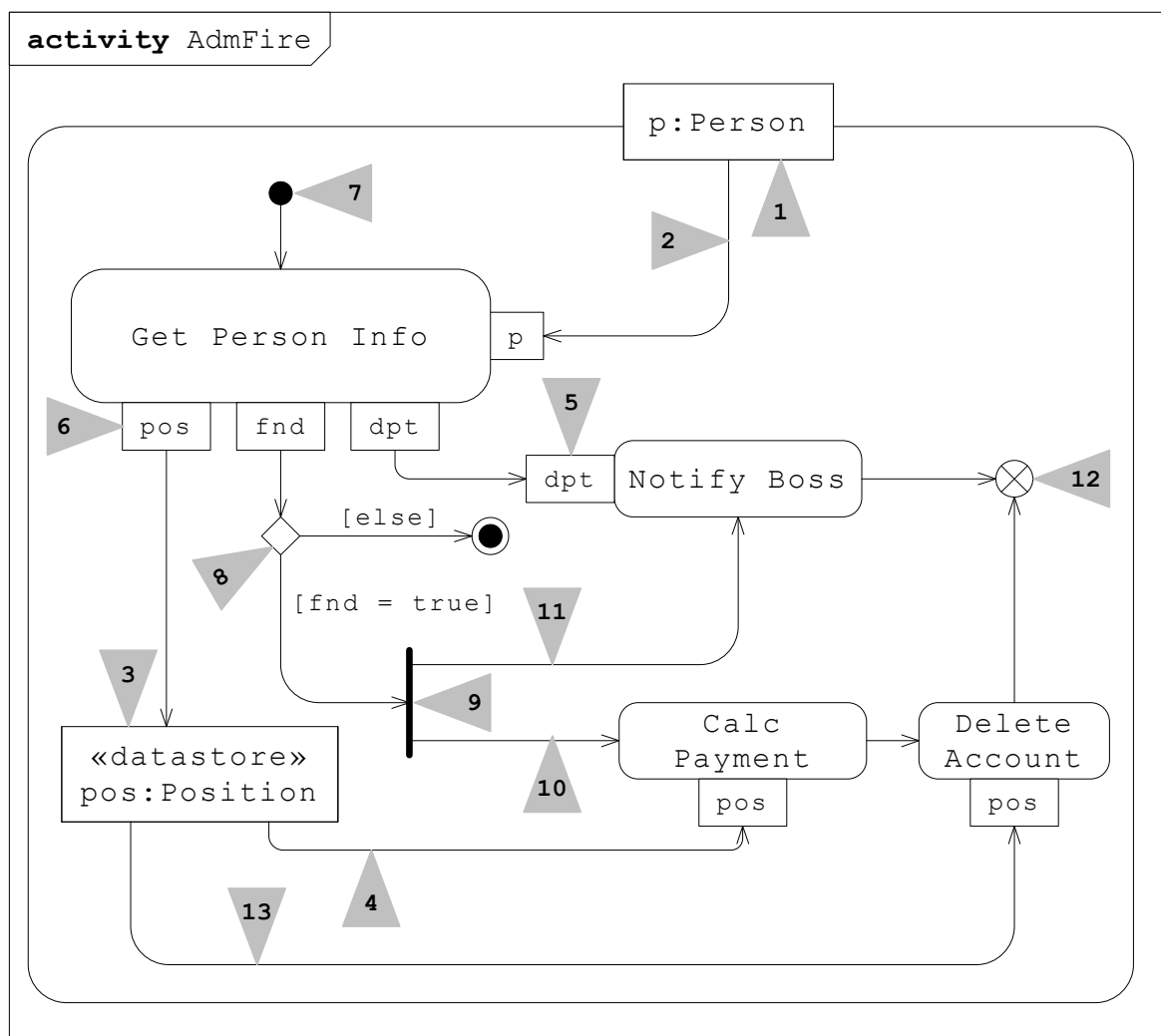


Рис. 4.19. Диаграмма деятельности по увольнению сотрудника (AdmFire)

Маркер данных `pos` немедленно отправляется в хранилище данных (3 на рис. 4.19), поскольку оно имеет единственную входящую дугу, и сохраняется там для дальнейшего использования.

Маркер данных `fnd` отправляется в узел управления "разветвление" (8 на рис. 4.19), где проверяется значение этого маркера данных. Если выполняется условие `fnd = true`, то маркер данных отправляется в узел управления "развилка" (9 на рис. 4.19), и выполняется следующий шаг. В противном случае выполнение графа деятельности заканчивается.

Узел управления "развилка" (9 на рис. 4.19) размножает полученный маркер на два и отправляет их дальше, запуская два

параллельных потока управления (10 и 11 на рис. 4.19). Заметим, что в данном случае содержимое маркеров теряется, поскольку принимаются они не через контакты, а непосредственно, как маркеры управления.

Деятельность `Notify Boss` готова принять маркер управления по переходу (11 на рис. 4.19) и маркер данных `dpt` (5 на рис. 4.19) от деятельности `Get Person Info`. Эта деятельность запускается, а после завершения отправляет маркер управления в заключительное состояние потока (12 на рис. 4.19), где он поглощается, и выполнение этого потока управления завершается.

Деятельность `Calc Payment` готова принять маркер управления по переходу (10 на рис. 4.19) и маркер данных `pos` из хранилища. Эта деятельность запускается, а после завершения отправляет маркер управления деятельности `Delete Account`.

Деятельность `Delete Account` готова принять маркер управления от предыдущей деятельности и маркер данных `pos` из хранилища по переходу (13 на рис. 4.19). Эта деятельность запускается, а после завершения отправляет маркер управления в заключительное состояние потока (12 на рис. 4.19), где он поглощается, и выполнение этого потока управления завершается.

### 4.3.3. Дорожки и разбиения

В UML 1 имеется своеобразное графическое средство, которое называется дорожкой.

*Дорожка (swim lane) — это графический комментарий, позволяющее классифицировать сущности по некоторому признаку. Обычно используется на диаграмме классов или на диаграмме деятельности.*

Дорожка — это именно графический комментарий, подобный границам системы (субъекта) на диаграмме использования. Поэтому никаких формальных правил применения дорожек в UML 1 нет.

В UML 2 ситуация несколько изменилась: дорожки, переименованные в UML 2 в *разбиения*, переведены из разряда графических комментариев в разряд сущностей метамодели языка, и инструменты могут (но не обязаны!) использовать это обстоятельство.

**Разбиение** (*partition*) — это разбиение в математическом смысле (то есть дизъюнктное покрытие) множества сущностей на диаграмме.

На рис. 4.20 представлена диаграмма деятельности, отражающая простейший бизнес-процесс найма на работу. Мы считаем, что наш процесс включает четыре деятельности:

- Interview — сбор информации;
- Analysis — анализ собранной информации и принятие решения;
- Fill Forms — заполнение документов;
- Refuse — отказ в найме.

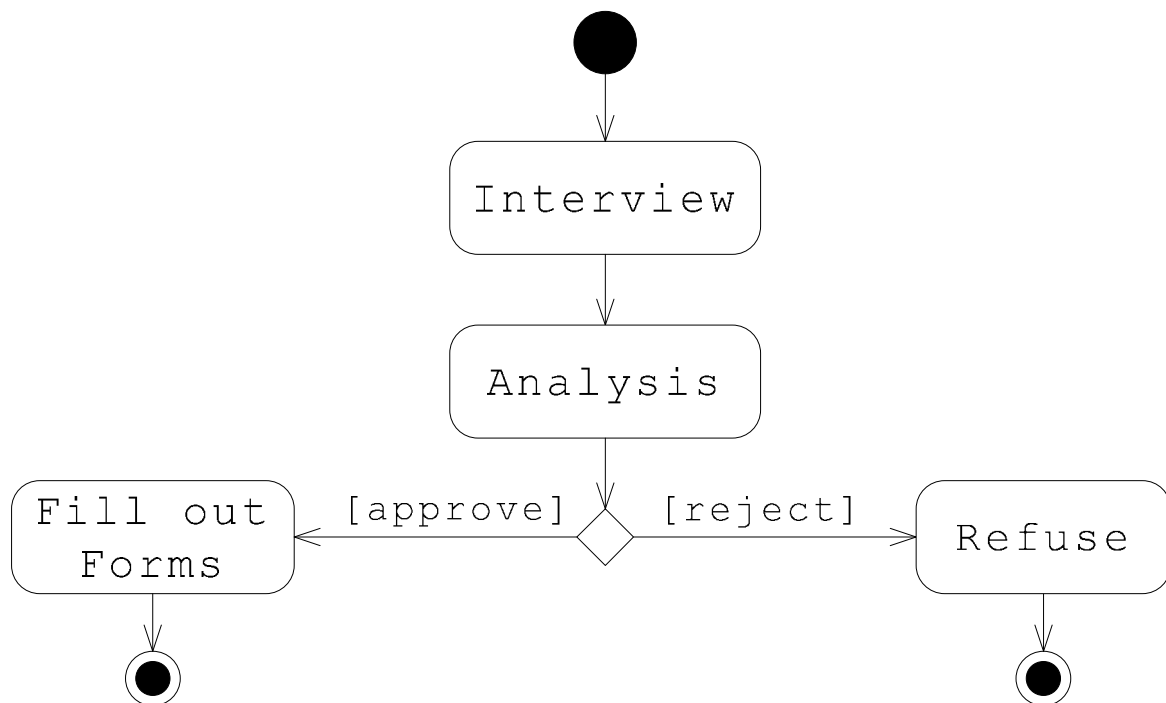


Рис. 4.20. Диаграмма деятельности процесса найма на работу



На диаграмме рис. 4.20 нет никаких дорожек — все деятельности равноправны и однородны. Допустим теперь, что деятельности, в которые нанимаемый вовлечен непосредственно (Interview, Fill out Forms, Refuse) происходит в одном месте и, так сказать, у него на глазах, а важная деятельность (о которой нанимаемый может не догадываться) по анализу информации и принятию решения (Analysis) осуществляется в другом месте и, может быть, другими действующими лицами (техническими специалистами, руководителями подразделений и т. д.). Эта важная информация не является частью модели, так как не имеет отношения к поведению системы, но мы можем отразить ее на диаграмме с помощью дорожек (рис. 4.21). В данном случае мы подразумеваем, что дорожка с названием HR Department содержит деятельности, выполняемые в приемной отдела кадров, а дорожка с названием Target Department содержит деятельности, выполняемые в том подразделении, куда предполагается принять кандидата.

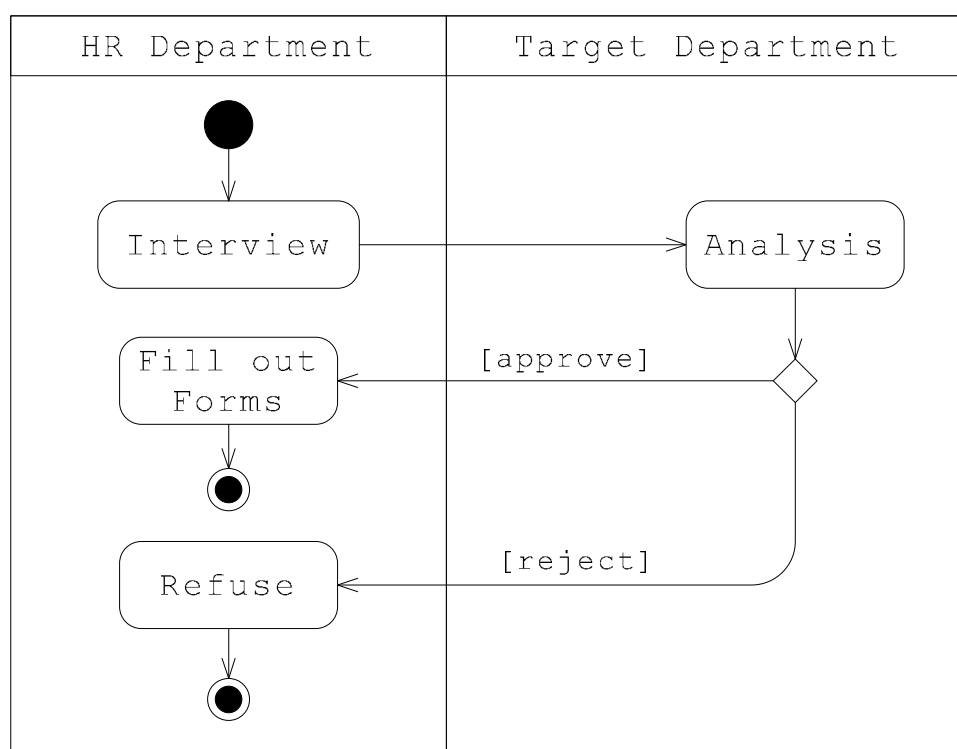


Рис. 4.21. Дорожки

Графически дорожки изображаются в виде прямоугольников с названиями. Как правило, их изображают со смыкающимися боковыми сторонами, хотя это и не обязательно. Авторы UML утверждают, что изображение, подобное приведенному на рис. 4.21, напоминает плавательные дорожки в бассейне, откуда и произошло название данной графической конструкции. Возложим ответственность за правомерность такой ассоциации на авторов UML и завершим этот несложный раздел еще одним примером применения наследников дорожек — разбиений в UML 2.

В этом примере мы наложим на действия по приему сотрудника две ортогональных классификации. Первая совпадает с уже разобранный классификацией по месту действия, а вторая указывает, как выполняется действие — устно (деятельности Interview или Analysis) или иначе (рис. 4.22).

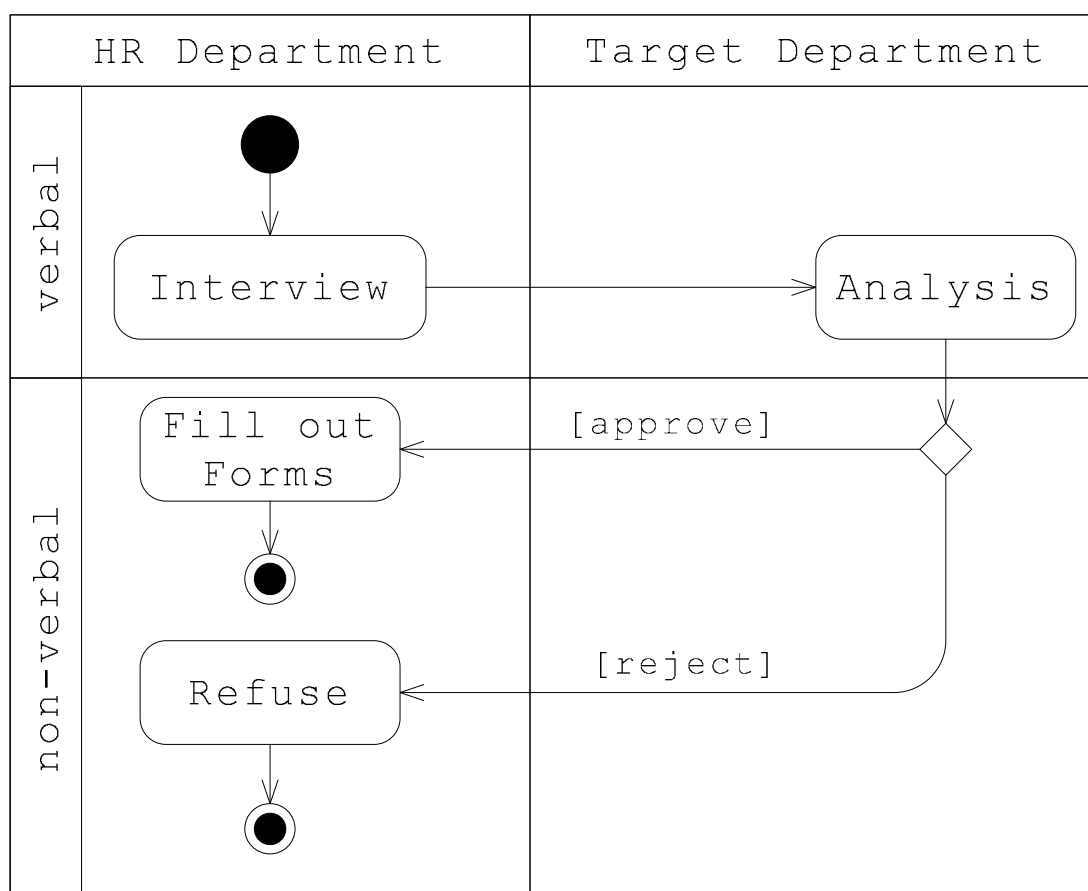


Рис. 4.22. Ортогональные дорожки

#### 4.3.4. Траектория объекта и поток данных

Диаграммы деятельности UML позволяют моделировать поведение, определяя не только поток управления, как в приведенных выше примерах, но и поток данных. Это утверждение вполне справедливо для UML 2 и справедливо с некоторыми оговорками для UML 1.

При объектно-ориентированном подходе к моделированию поток данных — это изменение состояний объектов во времени, и описание такого изменения существенным образом характеризует поведение.

Для описания данной характеристики поведения в UML 1 используются понятия "траектория объекта" и "объект в состоянии".

**Объект в состоянии** (*object in state*) — это экземпляр некоторого класса, про который известно, что он находится в определенном состоянии в данной точке вычислительного процесса.

Синтаксически объект в состоянии изображается, как обычно, в виде прямоугольника и его имя подчеркивается, но дополнительно после имени объекта в квадратных скобках пишется имя состояния, в котором в данной точке вычислительного процесса находится объект. В некоторых случаях состояние объекта не важно, например, если достаточно указать, что в данной точке вычислительного процесса создается новый объект данного класса, и в этом случае применяется обычная нотация для изображения объектов. Важно подчеркнуть, что объект в состоянии на диаграммах деятельности "по определению" считается состоянием, т. е. вершиной графа модели, которая может быть инцидентна траектории объекта.

**Траектория объекта**<sup>26</sup> — это переход особого рода, исходным или целевым состоянием которого является объект в состоянии.

---

<sup>26</sup> Некоторые авторы предпочитают использовать слово "поток", поскольку словосочетания "поток данных" и "поток управления" являются давно и хорошо устоявшимися терминами. Мы все-таки остановились на термине "траектория объекта", поскольку "поток объекта" по-русски звучит уж очень нескладно, хотя и соответствует устоявшейся традиции.

Траектория объекта изображается в виде пунктирной стрелки (в отличие от сплошной стрелки обычного перехода). Семантически траектория объекта, проведенная от состояния деятельности к объекту в состоянии, означает, что результатом деятельности является переход указанного объекта в данное состояние (или, может быть, создание нового объекта в указанном состоянии, что является частным случаем изменения состояния). Траектория объекта, проведенная от объекта в состоянии к состоянию деятельности, означает, что объект в данном состоянии является необходимым входным параметром указанной деятельности.

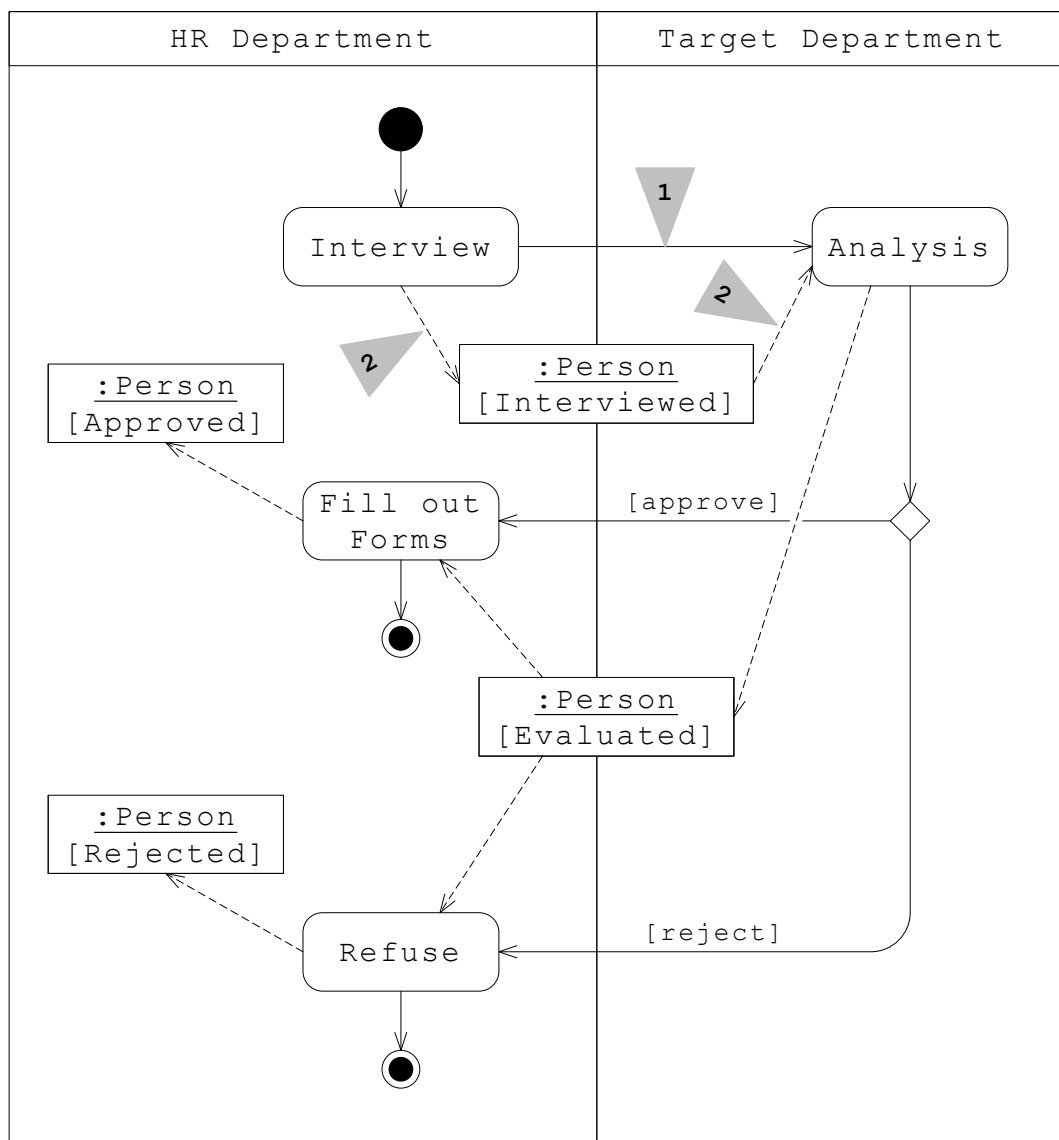


Рис. 4.23 Траектория объекта

Рассмотрим это на примере диаграммы деятельности, описывающей процесс найма сотрудника в информационной системе отдела кадров. Рис. 4.23 в основном повторяет рис. 4.21, но здесь представлена траектория объекта класса `Person`, хранящего данные о принимаемом сотруднике. На диаграмме хорошо видно, что именно является входными и выходными данными каждого из состояний деятельности: в результате деятельности `Interview` создается новый объект, который далее обрабатывается, меняя свое состояние.

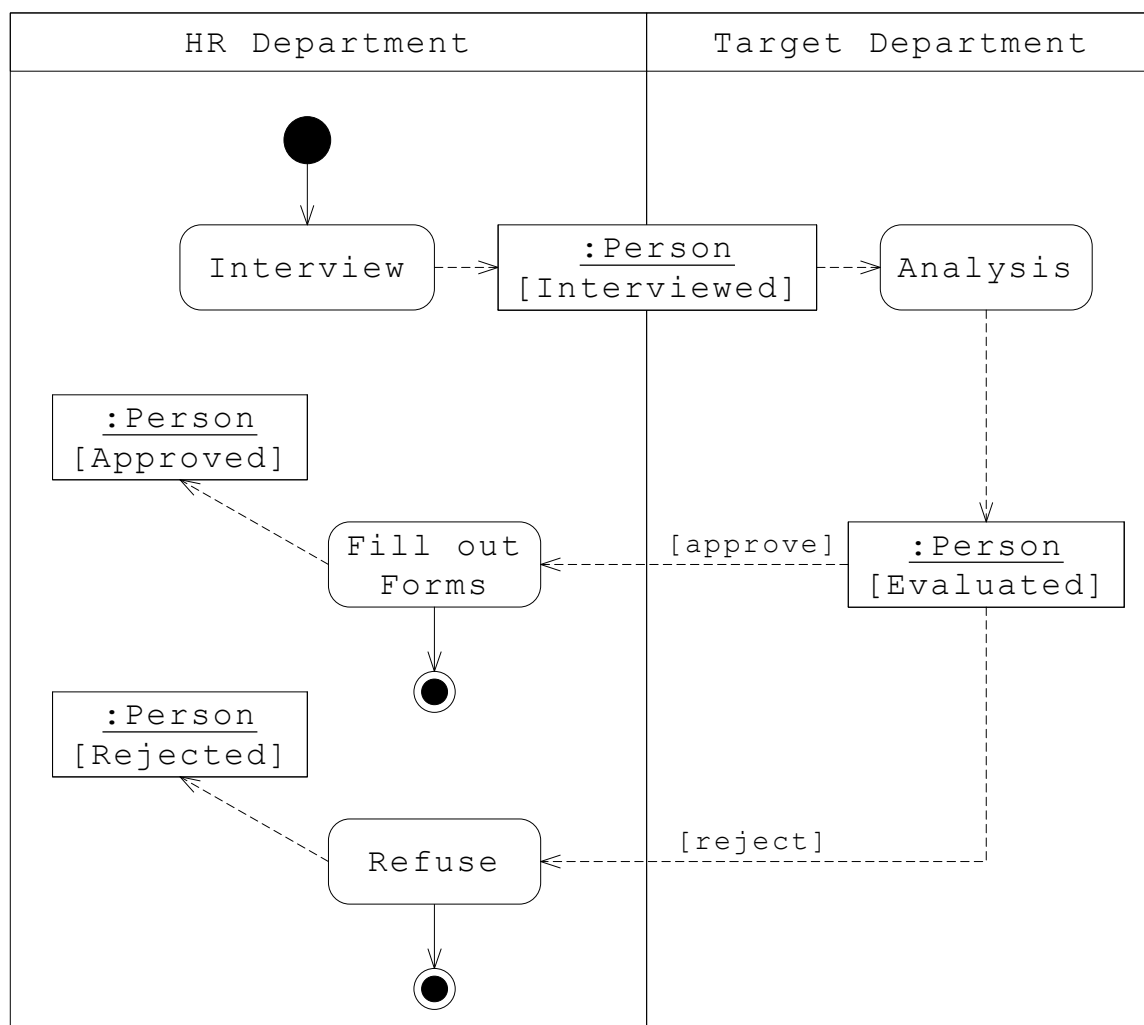


Рис. 4.24. Использование траекторий объектов вместо переходов по завершении

Нетрудно заметить, что в данном случае при моделировании поведения мы фактически повторяемся, описывая поведение системы. Из диаграммы на рис. 4.23 следует, что деятельность `Analysis`

выполняется после деятельности Interview, причем это указано дважды: один раз с помощью перехода по завершении (1 на рис. 4.23) из Interview в Analysis и второй раз с помощью траектории объекта (2 на рис. 4.23), показывающей, что для выполнения деятельности Analysis необходим объект, создаваемый деятельностью Interview. Разумеется, даже UML 1 позволяет не говорить лишнего: диаграмма на рис. 4.24 описывает то же самое поведение, что и диаграмма на рис. 4.23.

Нотация UML 2 позволяет использовать для описания поведения поток данных (траекторию объекта) еще шире, поскольку в определении семантики графа деятельности маркеры данных и маркеры управления практически равноправны. На рис. 4.25 приведена еще одна диаграмма деятельности, описывающая процесс приема на работу, но в нотации UML 2 с применением контактов и параметров деятельности.

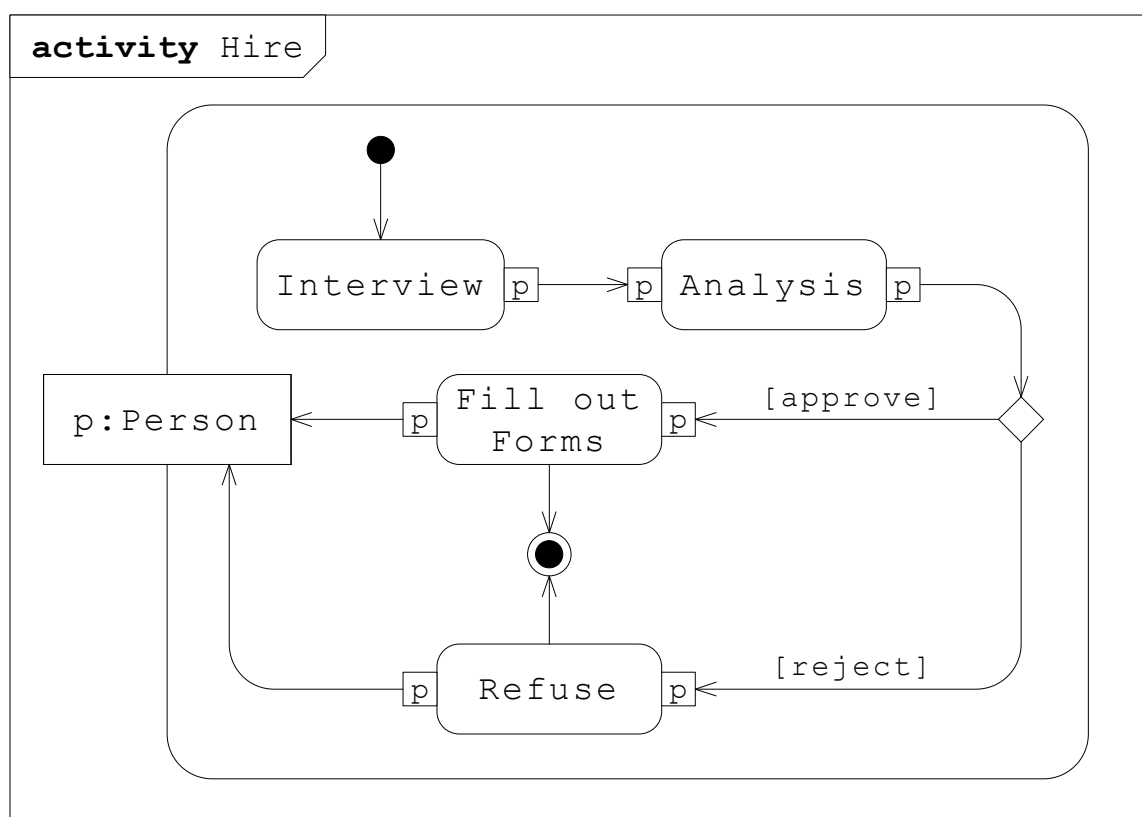


Рис. 4.25. Применение контактов и параметров деятельности

#### 4.3.5. Отправка и прием сигналов

Вернемся еще раз к примеру с наймом на работу и допустим, что мы хотим отразить в модели несколько иной вариант поведения. В диаграммах на рис. 4.20 – 4.24 процесс, происходящий в приемной отдела кадров, приостанавливается на то время, пока не будет завершена деятельность по оценке кандидата и принятию решения, которая фактически происходит в другом месте. Такое вынужденное ожидание может быть психологически неприятно кандидату (равно как и менеджеру по персоналу). Допустим, что в проектируемой информационной системе отдела кадров требуется обеспечить асинхронное проведение процесса приема: после сбора сведений о кандидате менеджер по персоналу отправляет сигнал в соответствующие инстанции и в ожидании ответного сигнала с решением переводит себя и кандидата в состояние ожидания с внутренней активностью — угощает чаем, рассказывает о задачах организации и т. п. В рамках уже рассмотренных обозначений такая ситуация может быть описана диаграммой деятельности, как показано на рис. 4.26. Здесь мы предполагаем, что в не отображаемых на диаграмме "инстанциях" принимается сигнал `Request` с аргументом `person` (1), проводится деятельность по анализу кандидата и в ответ отправляется сигнал `Response` с аргументом `decision` (2).

Приведенная на рис. 4.26 диаграмма точно описывает желаемое поведение, но может показаться не слишком наглядной: читатель должен знать синтаксические детали обозначений UML, чтобы понять описание процесса с первого взгляда. Между тем имеется хорошо знакомая очень многим наглядная система обозначений для передачи и приема сигналов (см. рис. 4.27). Эта система обозначений также включена в UML. Суть состоит в том, что действие по отправке (1) и приему (2) сигнала изображаются в виде фигур, сегментирующих соответствующие переходы. Применение данных обозначений приводит нас к диаграмме на рис. 4.27.

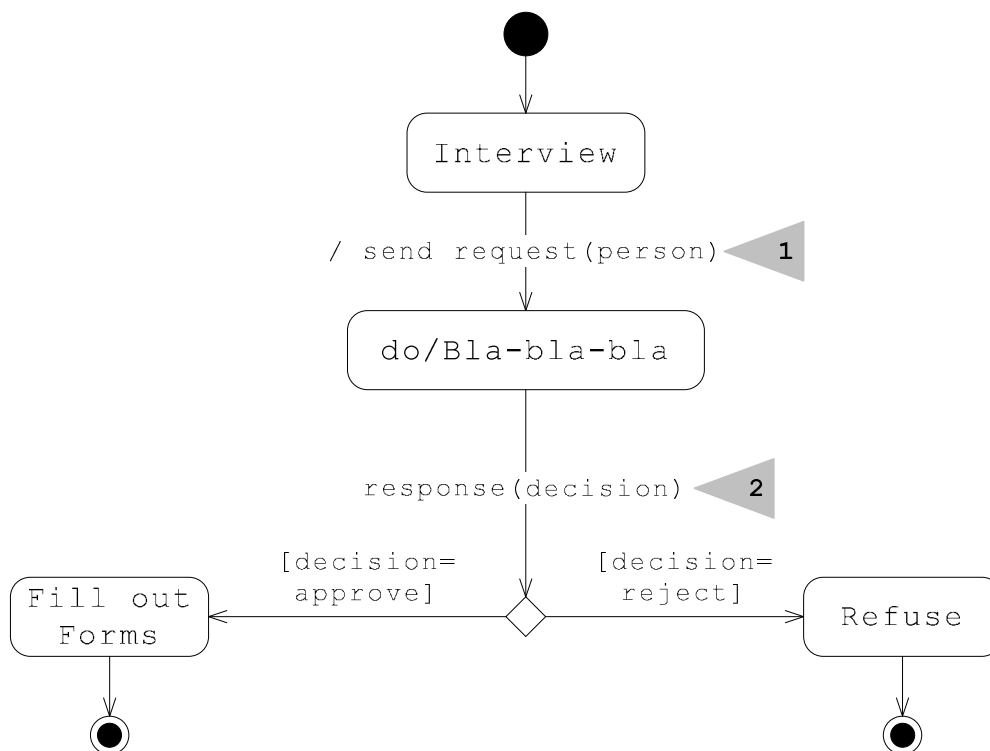


Рис. 4.26. Асинхронный процесс принятия решения при найме

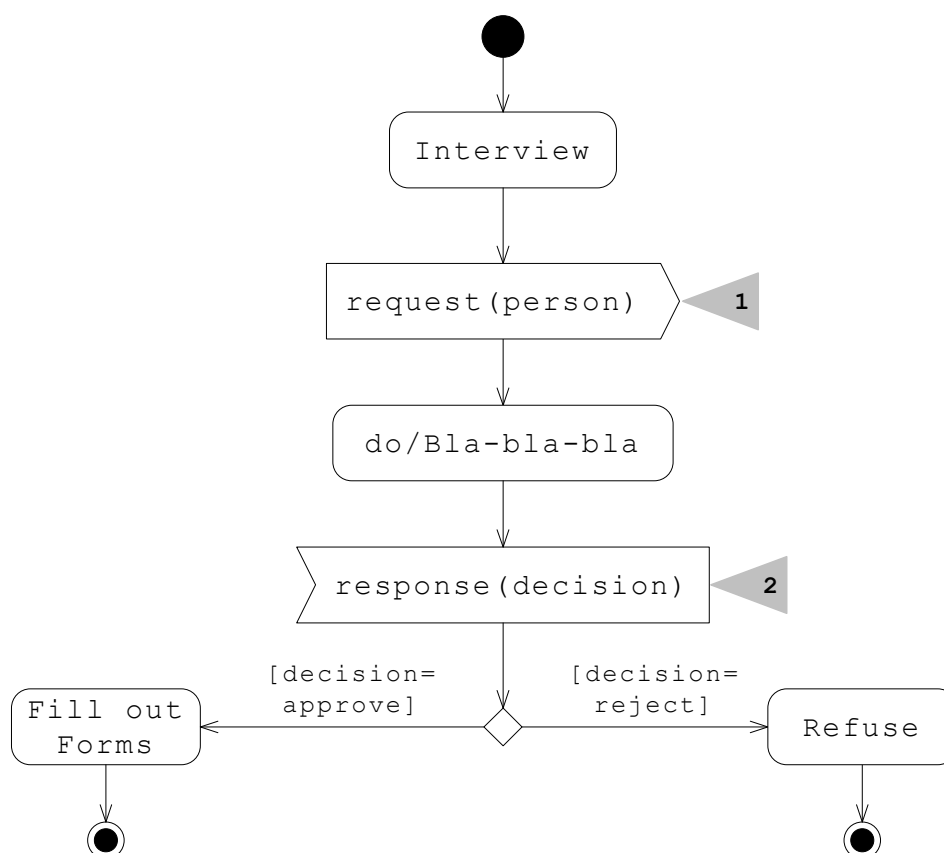


Рис. 4.27. Специальные обозначения для отправки и приема сигналов



#### 4.3.6. Прерывания и исключения

Прерывания и исключения — это примеры давно и хорошо известных, можно сказать базовых, механизмов, применяемых в программировании.

Начнем с прерываний. В UML 2 введена явным образом конструкция, которая называется область прерывания.

***Область прерывания** (interruptible activity region) — это структурированный узел (рамка) на диаграмме деятельности, внутри которого возможно прерывание обычного порядка выполнения действий при возникновении определенного события.*

Синтаксически область прерывания изображается в виде пунктирной рамки, ограничивающей некоторый фрагмент графа деятельности. Прерывающие события изображаются в виде "флажка" приема сигнала, от которого проводится стрелка-"молния" (interruptible activity edge) к некоторой внешней деятельности — *обработчику прерывания* (interrupt handler). Флажок должен находиться обязательно **внутри** области прерывания, чтобы прерывание не спутать с исключением, которое имеет похожую нотацию.<sup>27</sup>

Семантически область прерывания означает следующее. Если при выполнении фрагмента графа деятельности в области прерывания произойдет указанное прерывающее событие, то управление передается обработчику прерывания. Возобновить прерванный фрагмент невозможно.<sup>28</sup>

На рис. 4.28 мы привели фактически тот же пример, что и на рис. 2.10 и рис. 2.11. Прерывающим событием является событие

---

<sup>27</sup> Флажок исключения располагается за границей области, которая генерирует исключение

<sup>28</sup> Стандарт хранит молчание по поводу того, позволяет ли обработчик прерывания сохранить контекст и освободить ресурсы прерываемому процессу. Реальные системы обработки прерываний в операционных системах, как правило, это делают.

Cancel (1), которое может произойти в любой момент, а обработчиком прерываний — соответствующее действие Cancel hiring (2).

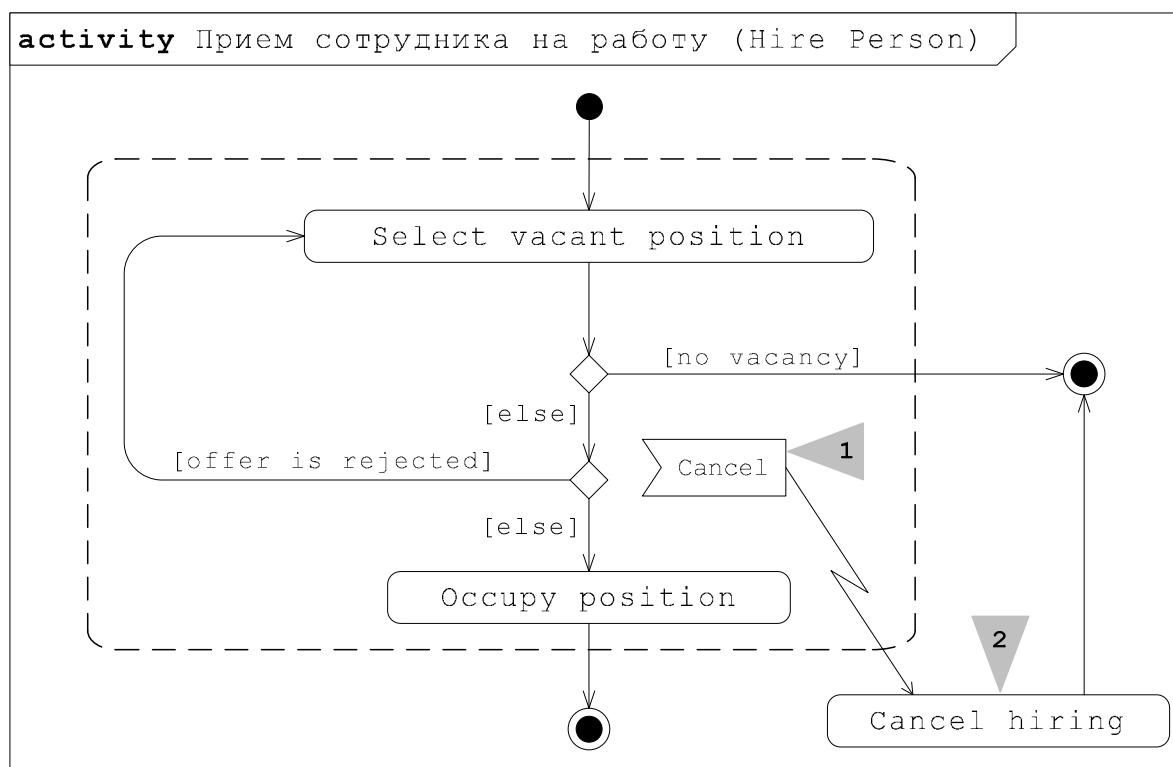


Рис. 4.28. Область прерывания

Обратимся теперь к исключениям. В UML 1 исключения трактуются как частный случай сигналов, что хотя и допустимо, но не очень удобно. В UML 2 для моделирования исключений введена специальная конструкция.

*Распространение исключения.* Для действия или иного узла графа деятельности может быть указано, какие исключения этот узел может генерировать. Кроме того, для узла указываются обработчики исключений, которые с ним связаны. Работает это следующим образом. Допустим, произошло исключение. Если с узлом, в котором произошло исключение, ассоциирован обработчик этого исключения, то обработчик выполняется и на этом обработка заканчивается. Если же нет ассоциированного обработчика, то исключение передается в

следующий объемлющий по иерархии вложенности узел и обработчик ищется там, и так далее. Только если исключение передано на самый верхний уровень деятельности, так и не будучи обработанным, то выполнение деятельности прекращается.

*Иерархия исключений.* Исключения не являются классификаторами и не могут образовывать иерархии обобщения. Однако моделировать исключения можно с помощью объектов определенного типа и именно эти типы могут образовывать иерархии. Обработчик исключения связан с определенным типом исключения и может обрабатывать также все его специализации. Таким образом, типы исключений можно дифференцировать очень детально, и это не потребует очень большого числа обработчиков.

*Параметры исключения.* В описание типов исключений могут быть введены атрибуты, которые при генерировании исключения заполняются требуемыми значениями и позволяют передавать в обработчик столько информации, сколько нужно для детальной и корректной обработки.

*Сохранение выходных контактов.* Когда возникает исключение, выполнение текущего действия прерывается без генерации выходных значений. Выходные контакты обработчика исключения должны в точности совпадать по числу и типам с выходными контактами защищаемого узла, и когда обработка исключения заканчивается, то выходные значения обработчика исключения подставляются вместо выходных значений деятельности, сгенерировавшей исключения и выполнение программы может продолжиться обычным образом, "как будто ничего и не было".

Нотация для исключений следующая.

Чтобы показать, что некоторый узел может генерировать исключения, у этого узла определяется выходной контакт с типом исключения, возле которого рисуется маленький треугольник.

Чтобы показать, что некоторый узел имеет обработчик исключения, от защищаемого узла рисуется стрелка-"молния" к

входному контакту узла-обработчика, причем типом этого контакта должен быть тип исключения.

Обратимся еще раз к примеру про удаление подразделения, и в качестве примера (рис. 4.29) рассмотрим диаграмму для деятельности `deletePos()`.

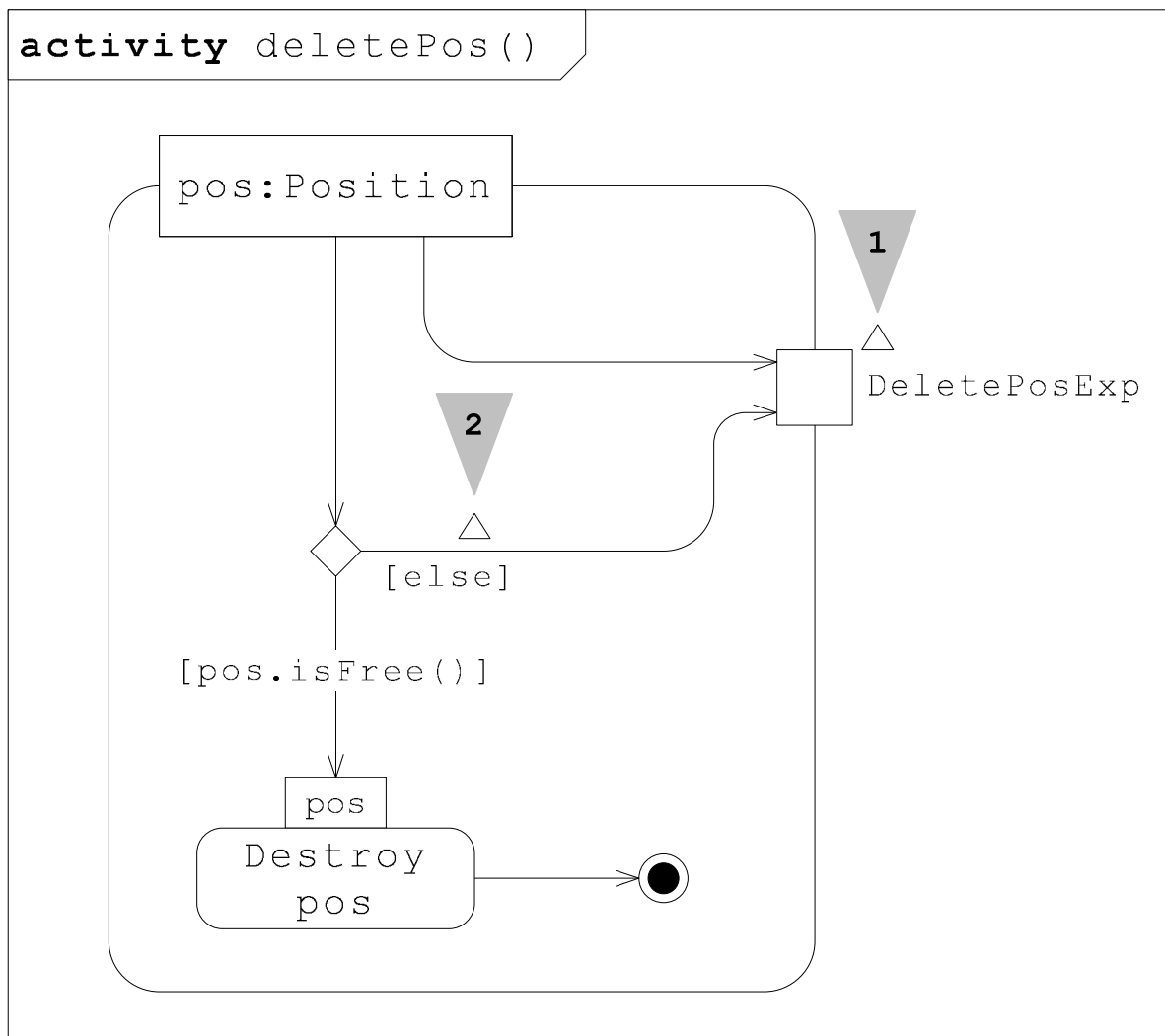


Рис. 4.29. Диаграмма деятельности deletePos

Все что требуется теперь сделать — связать исключение с его обработчиком, что мы и сделали на рис. 4.30 с использованием двух разных нотаций.

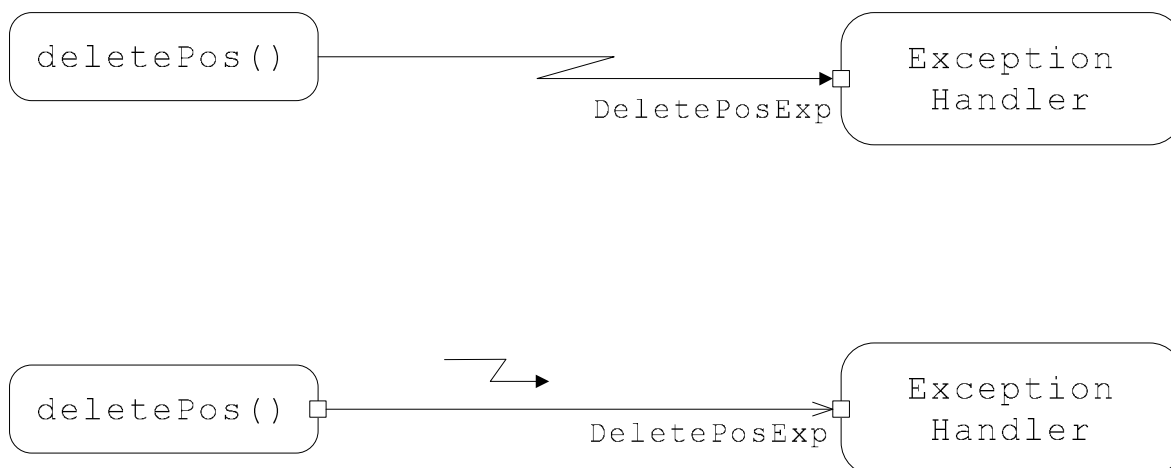


Рис. 4.30. Варианты нотации стрелки-"молнии"

## 4.4. ДИАГРАММЫ ВЗАИМОДЕЙСТВИЯ

Диаграммы взаимодействия предназначены для моделирования поведения путем описания взаимодействия объектов для выполнения некоторой задачи или достижения определенной цели. Взаимодействие происходит путем обмена сообщениями.

Диаграммы взаимодействия изображаются в нескольких различных графических формах, из которых самыми важными являются диаграммы последовательности и диаграммы коммуникации. Наряду с этими основными формами описания взаимодействия в UML 2 применяются диаграммы синхронизации и обзорные диаграммы взаимодействия.

Мы уже отмечали, что **диаграммы последовательности и диаграммы коммуникации семантически эквиваленты, хотя графически выглядят совсем по-разному**. Семантически эти диаграммы эквиваленты потому, что описывают одно и то же: последовательность передачи сообщений между объектами в процессе их взаимодействия. А выглядят по-разному они потому, что в диаграмме последовательности графически подчеркивается упорядоченность во времени передаваемых сообщений, в то время

как в диаграмме коммуникации на передний план выдвигается структура связей между объектами, по которым передаются сообщения.

Сразу подчеркнем главное: оба типа диаграмм моделируют поведение "по индукции", от частного к общему, путем описания конкретного протокола передачи сообщений. В этом и сила и слабость данного способа описания поведения. Сильная сторона состоит в том, что в объектно-ориентированной парадигме обмен сообщениями — это и есть само выполнение программы, поэтому **протокол передачи сообщений является наиболее точной моделью поведения**. Диаграммы взаимодействия находятся "ближе" к реальному выполнению программы, чем другие средства описания поведения. Слабость диаграмм взаимодействия состоит в том, что эти диаграммы описывают поведение на уровне экземпляров, а не классификаторов; на уровне протоколов выполнения алгоритма, а не самого алгоритма. Диаграммы взаимодействия менее "алгоритмичны", чем диаграммы автомата и диаграммы деятельности.

Наряду с основными сущностями и отношениями на диаграммах последовательности и коммуникации применяется множество дополнительных элементов семантики и нотации. В следующем параграфе мы рассмотрим основной элемент этих диаграмм — сообщение.

#### 4.4.1. Сообщения

*Сообщение (message) — это передача управления и данных от одного объекта (отправителя) к другому (получателю).*

Заметим, что отправка сообщения является действием, а получение сообщения — событием. В UML 1 следующие действия связаны с передачей информации и отправкой сообщений:

- вызов метода (call);
- создание объекта (create);
- уничтожение объекта (destroy);
- возврат значения (return);

- посылка сигнала (send).

Действие записывается в виде текста над (или рядом со) стрелкой, символизирующей сообщение. Если действие имеет параметры (вызов метода, создание объекта, посылка сигнала), то аргументы, соответствующие параметрам по числу и типам, записываются справа от имени действия в круглых скобках.

Синтаксис вызова метода имеют различия в UML 1 и в UML 2. Если действием является вызов метода, возвращающего значения, то в UML 1 слева от имени метода записывается список переменных для возвращаемых значений (их может быть несколько) и знак присваивания =.<sup>29</sup> Таким образом, та часть нотации сообщений, которая относится к выполняемому действию по вызову метода, в UML 1 имеет следующий синтаксис.

переменные := ИМЯ ( аргументы )

В UML 2 используется несколько иной синтаксис:

атрибуты = ИМЯ ( аргументы ) : переменные

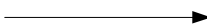



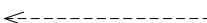
Поскольку получение сообщения является событием, то получатель сообщения вместе с информацией получает и управление (для того, чтобы иметь возможность выполнить действия, инициируемые полученным сообщением). В UML различается несколько типов передачи управления с помощью сообщения. Чтобы отличить тип передачи сообщения, в UML применяется специальная графическая нотация, а именно, различаются виды стрелок, которыми обозначаются сообщения. Хотя на диаграммах коммуникации и последовательности сообщения обозначаются различным образом, принципы изображения одинаковы и перечислены в табл. 4.4.

---

<sup>29</sup> Напомним, что инструменты вправе использовать любой синтаксис для текстовых фрагментов при условии, что имеется взаимно-однозначное соответствие между используемым синтаксисом и стандартным синтаксисом. Так, в частности, в UML 1 чаще используется знак присваивания := (как в Паскале), а в UML 2 — знак = (как в Си).

Таблица 4.4

### Типы передачи сообщений

Вид стрелки	Тип передачи сообщения
	<i>Вложенный поток управления.</i> Данный тип передачи сообщения подразумевает, что отправитель может отправить следующее сообщение только после того, как завершится выполнение всех действий, инициированных данным сообщением. Обычно применяется при вызове методов.
 только UML 1	<i>Простой поток управления.</i> Данный тип передачи подразумевает, что управление передается от отправителя сообщения получателю (возможно, безвозвратно). Обычно применяется при моделировании поведения на уровне действующих лиц и вариантов использования.
 в UML 1  в UML 2	<i>Асинхронный поток управления.</i> Данный тип передачи подразумевает, что сообщение асинхронно передается от отправителя получателю, при этом у отправителя сохраняется свой поток управления, независимый от потока управления получателя. Обычно применяется при отправке сигналов.
	<i>Возврат управления.</i> Данный тип передачи подразумевает возврат управления после выполнения всех действий, инициированных передачей сообщения с вложенным потоком управления. При этом могут быть указаны возвращаемые значения. Данный тип передачи сообщения можно не отображать на диаграмме, поскольку он подразумевается по умолчанию при вызове методов.
не определяется	Допускается использование при моделировании других, не определяемых в UML, типов передачи управления, например, передача управления по истечении времени.

Для того чтобы сообщение могло быть передано от отправителя к получателю, отправитель должен "знать" получателя, т. е., например, должна существовать ассоциация между классификаторами отправителя и получателя, экземпляр которой (связь) и служит тем путем, по которому передается сообщение. На диаграмме коммуникации эта связь всегда изображается в явном виде, как линия, а на диаграмме последовательности она подразумевается как часть самой стрелки сообщения.



Однако поведение определяется не только и не столько тем, какие объекты посылают какие сообщения, но прежде всего тем, в каком порядке это происходит. UML позволяет определить относительный порядок сообщений во взаимодействии, причем это делается несколькими различными способами.

На диаграмме последовательности порядок сообщений определяется временем их отправки, а время отсчитывается на диаграмме сверху вниз. Таким образом, сообщения, изображенные выше, предшествуют сообщениям, изображенным ниже.

Порядок можно задать с помощью последовательного *номера сообщения*. Данные номера уникальны и обладают тем свойством, что сообщения с меньшими номерами предшествуют сообщениям с большими.

Таким образом, сообщение может быть довольно сложной синтаксической конструкцией. Сразу отметим, что абсолютно все возможные части описания сообщения, как правило, нет нужды использовать — обязательным является только имя. Общий синтаксис текста описания сообщения следующий.

предшественники / повторность номер : атрибуты =  
ИМЯ ( аргументы ) : переменные

#### 4.4.2. Диаграммы последовательности

*Диаграмма последовательности предназначена для моделирования поведения в форме описания протокола сеанса обмена сообщениями между взаимодействующими экземплярами классификаторов во время выполнения одного из возможных сценариев.*

Из этого определения вытекают несколько следствий, определяющих состав сущностей и набор отношений, используемых на диаграмме последовательности (равно как и на диаграмме коммуникации).

На диаграмме присутствуют только те экземпляры классификаторов, которые задействованы в данном взаимодействии.

Прочие экземпляры не показываются, хотя возможно и присутствуют в системе.

Отображаются только те связи, которые нужны для передачи данной последовательности сообщений, прочие связи не показываются.

Состав сообщений (а тем самым операций и сигналов) определяется назначением данного взаимодействия; в других взаимодействиях эти же экземпляры классификаторов могут обмениваться другими сообщениями.

Вышесказанное нуждается в пояснении. Утверждение о том, что на диаграмме последовательности присутствуют экземпляры классификаторов не совсем верное, точнее, не совсем полное. Чтобы понять это, давайте обратимся к понятию связи и посмотрим, откуда оно берется.

***Связь (link)** — это экземпляр отношения (например, ассоциации или зависимости), представляющий собой набор ссылок на экземпляры классификаторов, связанных между собой посредством этого отношения.*

Первое, на что требуется обратить внимание, это то, что связи могут быть как временными, так и постоянными. Постоянные связи суть экземпляры ассоциаций. Временные связи могут и не быть экземплярами ассоциаций. Для того чтобы один экземпляр классификатора смог вызвать метод другого, первый должен знать про второй. Знание это может быть получено не только вследствие наличия ассоциации между соответствующими классификаторами, но и, например, по причине передачи второго экземпляра классификатора в качестве параметра операции для первого. Такая связь на диаграмме классов могла бы быть выражена отношением зависимости.

Перейдем к описанию особенностей нотации диаграммы последовательности. Напомним, что в UML семантика первична, а нотация вторична: возможны различные вариации в способах

рисования диаграмм, особенно это характерно для диаграмм последовательности. В наших примерах мы придерживаемся самого "скромного" стиля отображения диаграмм, с минимумом украшений.

На диаграмме последовательности считается выделенным одно направление, соответствующее течению времени. По умолчанию считается, что время течет сверху вниз, но это не обязательно, например, можно считать, что время течет слева направо, оговорив это специальным примечанием. В наших примерах используется исключительно нотация по умолчанию: время всегда течет сверху вниз. Саму ось времени не отображают.

Среди сообщений есть первое, которое кладет начало данному взаимодействию. Стрелка этого сообщения расположена выше всех других стрелок сообщений. Все экземпляры классификаторов, которые находятся выше первого сообщения, существуют **до** начала данного взаимодействия; остальные, которые расположены ниже или на уровне первой стрелки, возникают **в процессе** данного взаимодействия. Обычно экземпляр классификатора возникает в результате выполнения конструктора классификатора. Стрелку сообщения, которая рисуется пунктирной линией, соответствующую вызову операции конструктора, принято направлять к фигуре (прямоугольнику), обозначающей созданный экземпляр.

Параллельно оси времени от всех участвующих во взаимодействии объектов отходит прямая пунктирная линия, которая называется *линией жизни* (lifeline). Линия жизни представляет объект во взаимодействии: если стрелка отходит от линии жизни объекта, то это означает, что данный объект отправляет сообщение, а если стрелка сообщения входит в линию жизни, то это означает, что данный объект получает сообщение. Если же стрелка **пересекает** линию жизни объекта, то это ничего не значит — сообщение пролетело мимо. Если в процессе взаимодействия объект заканчивает свое существование, то линия жизни обрывается и в этом месте ставится косой крест.

Над стрелкой сообщения указывается текстовая часть описания сообщения. Заметим, что номер сообщения, равно как и номера предшествующих сообщений, на диаграммах последовательности обычно не указывают, поскольку в этом нет нужды: относительный порядок сообщений и так хорошо определяется направлением оси времени.

Мы еще не закончили описание особенностей нотации диаграммы последовательности, но чувствуем, что пора привести пример. Рассмотрим взаимодействие, возникающее при одном из простых сценариев в нашей информационной системе отдела кадров, а именно, создание подразделения. Данное взаимодействие инициируется внешним действующим лицом — менеджером штатного расписания, который открывает соответствующую форму и запускает выполнение операции создания подразделения, после чего закрывает более не нужную ему форму. Общая схема взаимодействия для данного сценария представлена на рис. 4.31. Обратите внимание, что помимо сообщений, которые соответствуют вызовам методов (например, 1), на диаграмме присутствует сообщение, соответствующее вызову конструктора (2).

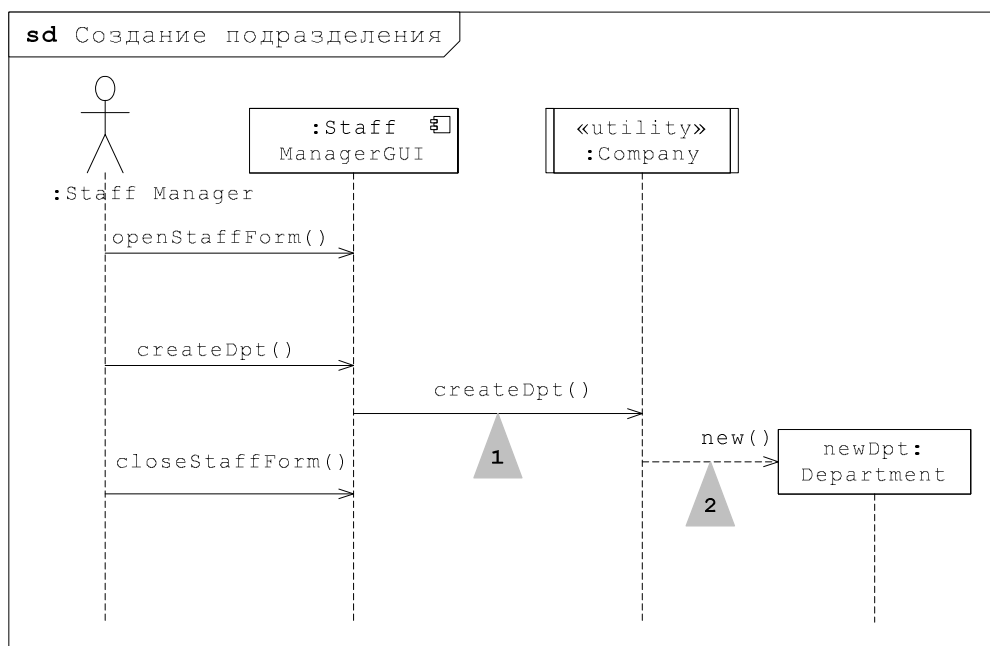


Рис. 4.31. Диаграмма последовательности

Продолжим описание нотации диаграмм последовательности.

Если же нужно в явном виде указать ограничения по времени, например, указать, что время задержки доставки сообщения должно быть ограничено сверху, то на диаграмму в нужном месте (имеет значение только положение по вертикали) рядом с началом или концом стрелки сообщения помещают произвольные идентификаторы, которые называются *метками времени* (time observation), и добавляют временные ограничения, задающие требуемые условия на значения меток времени.

**Метка времени** — это именованная точка на линии жизни. Перед меткой времени ставится символ "@".

Допустим, что наша информационная система отдела кадров предназначена для организации, имеющей удаленный филиал. В этом филиале имеется и работает свой экземпляр информационной системы, который, очевидно, должен быть проинформирован, что в головной организации создано новое подразделение. Возможно, эта информация об изменениях в головной организации дойдет в филиал с некоторой задержкой, поскольку для связи используется медленный канал передачи данных. Такую ситуацию можно промоделировать диаграммой, приведенной на рис. 4.32, где метка времени (1) в совокупности с временным ограничением (2), описывают задержанную доставку сообщения notify().

Обратите внимание на линии (3 и 4 на рис. 4.32). Это графические комментарии, которые нужны для привязки метки времени и временного ограничения к требуемым точкам линии жизни.

Мы уже отмечали, что сообщение передает не только данные, но и поток управления. Чтобы показать, что некоторый объект в определенный период взаимодействия имеет *фокус управления*, или, как еще говорят, *активизирован*, на диаграмме последовательности используют специальный графический элемент — *спецификацию выполнения* (execution specification), который изображают в виде

узкой полоски на соответствующей части линии жизни. В UML 1 данный графический элемент назывался *активацией* (activation). Начало спецификации выполнения соответствует получению сообщения о вызове метода, а конец — завершению выполнения метода и возврату управления. При этом если во время выполнения данного метода будет вызван еще раз метод этого же объекта (тот же самый метод, или другой), то это отмечается с помощью еще одной полоски активации, которая накладывается сбоку на первую. Глубина стека вызовов и, соответственно, количество наложенных полосок для одного объекта в UML, естественно, не ограничиваются.

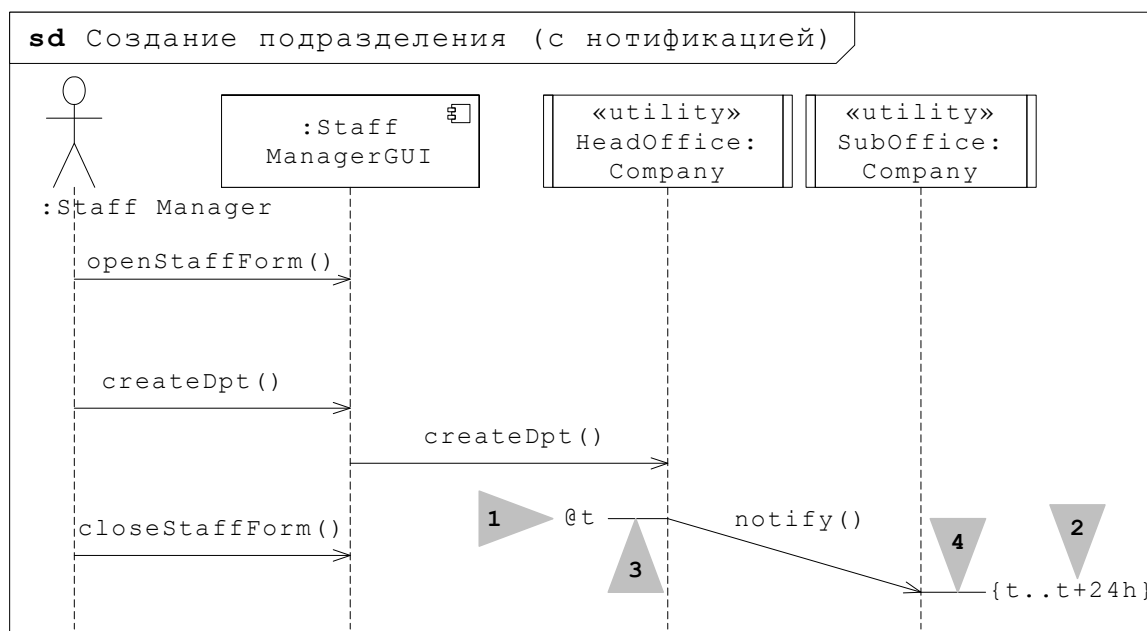


Рис. 4.32. Метки времени и задержанная доставка сообщения

Для наглядности на диаграмме последовательности можно показать в явном виде возврат управления (и, может быть, возвращаемое значение), хотя это не обязательно: возврат управления подразумевается при использовании сообщения типа вызов метода. Более того, если использовать полоски для явного указания активации объектов, стрелки возврата не нужны: их легко можно мысленно восстановить.

Рассмотрим применение этой группы обозначений на следующем примере из информационной системы отдела кадров. Допустим, при создании нового подразделения немедленно выполняется метод `createBossPos()` по созданию новой должности (начальника) в этом подразделении и эта вакансия заполняется (свято место пусто не бывает), а после успешного создания подразделения форма демонстрирует менеджеру штатного расписания измененную организационную диаграмму компании (рис. 4.33).

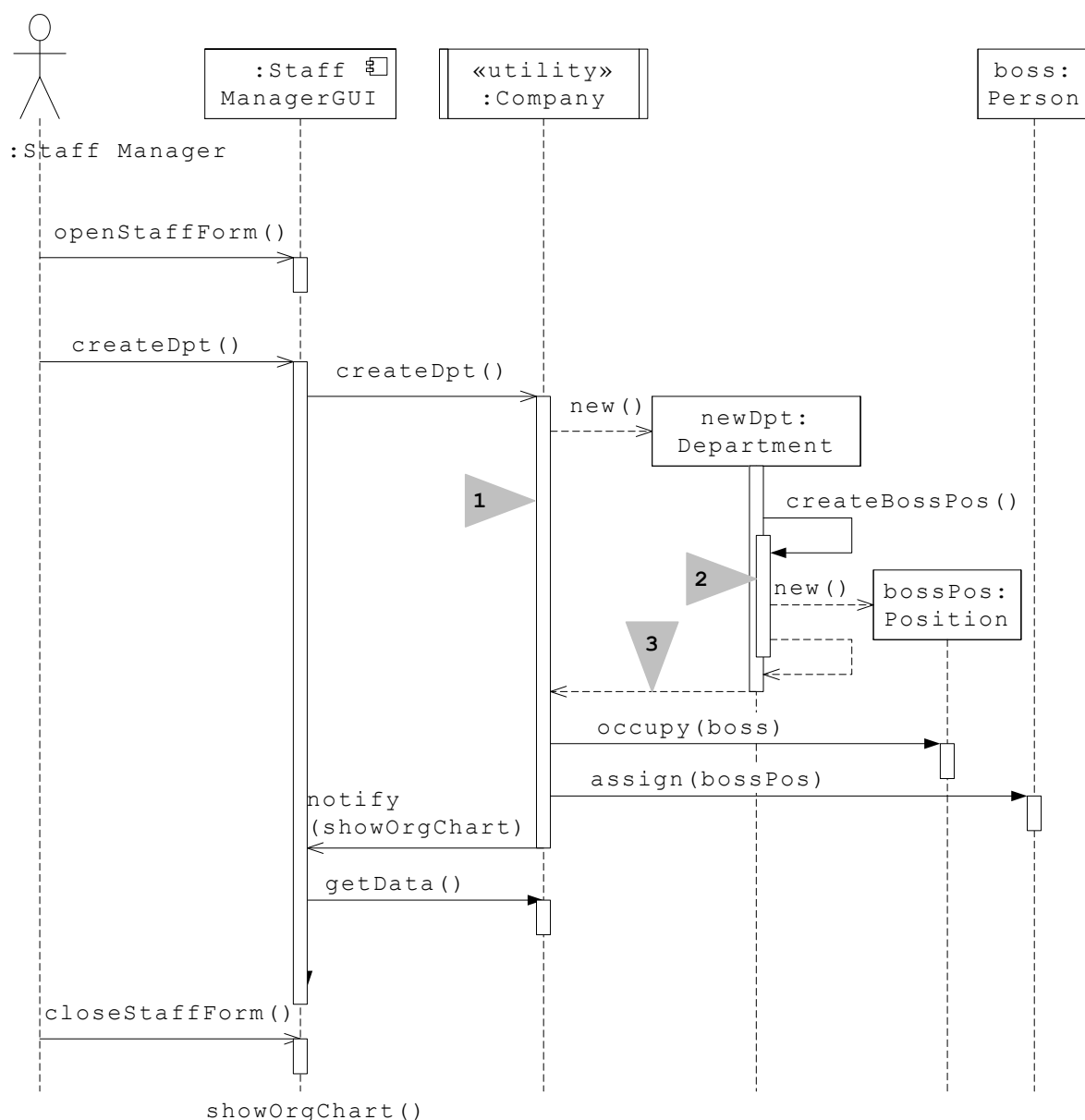


Рис. 4.33. Активации и возвраты

Здесь мы используем как активации (1), в том числе вложенные (2), так и возвраты (3), чтобы показать применение всех средств, хотя, может быть, это немного перегружает диаграмму.

#### 4.4.3. Составные шаги взаимодействия

Надо сказать, что диаграммы последовательности UML по существу заимствованы из другого графического языка описания поведения — MSC (Message Sequence Chart), который был разработан и успешно применяется производителями встроенных систем. Часть конструкций MSC была заимствована еще в UML 1, а оставшиеся пришли с UML 2. В том числе в UML 2 были заимствованы *составные шаги взаимодействия* (combined fragment). Составные шаги позволяют на диаграмме последовательности, которая фактически является диаграммой протокола взаимодействия, отражать и алгоритмические аспекты, а не только последовательность передачи сообщений. Составные шаги позволяют графически изображать на диаграмме последовательности ветвления, циклы и другие полезные конструкции управления. Делается это очень просто: на диаграмме рисуется рамка, в углу которой указывается тип составного шага с помощью ключевого слова, а внутри шага указываются частичные последовательности сообщений в соответствии с правилами, присущими шагам данного типа. Например, для ветвлений используется ключевое слово `alt`, а альтернативные последовательности сообщений рисуются внутри рамки просто последовательно, друг под другом. Операнды составного шага взаимодействия отделяются пунктирной линией и для них указываются соответствующие сторожевые условия.

На рис. 4.34 приведена диаграмма последовательности с использованием составного шага взаимодействия.



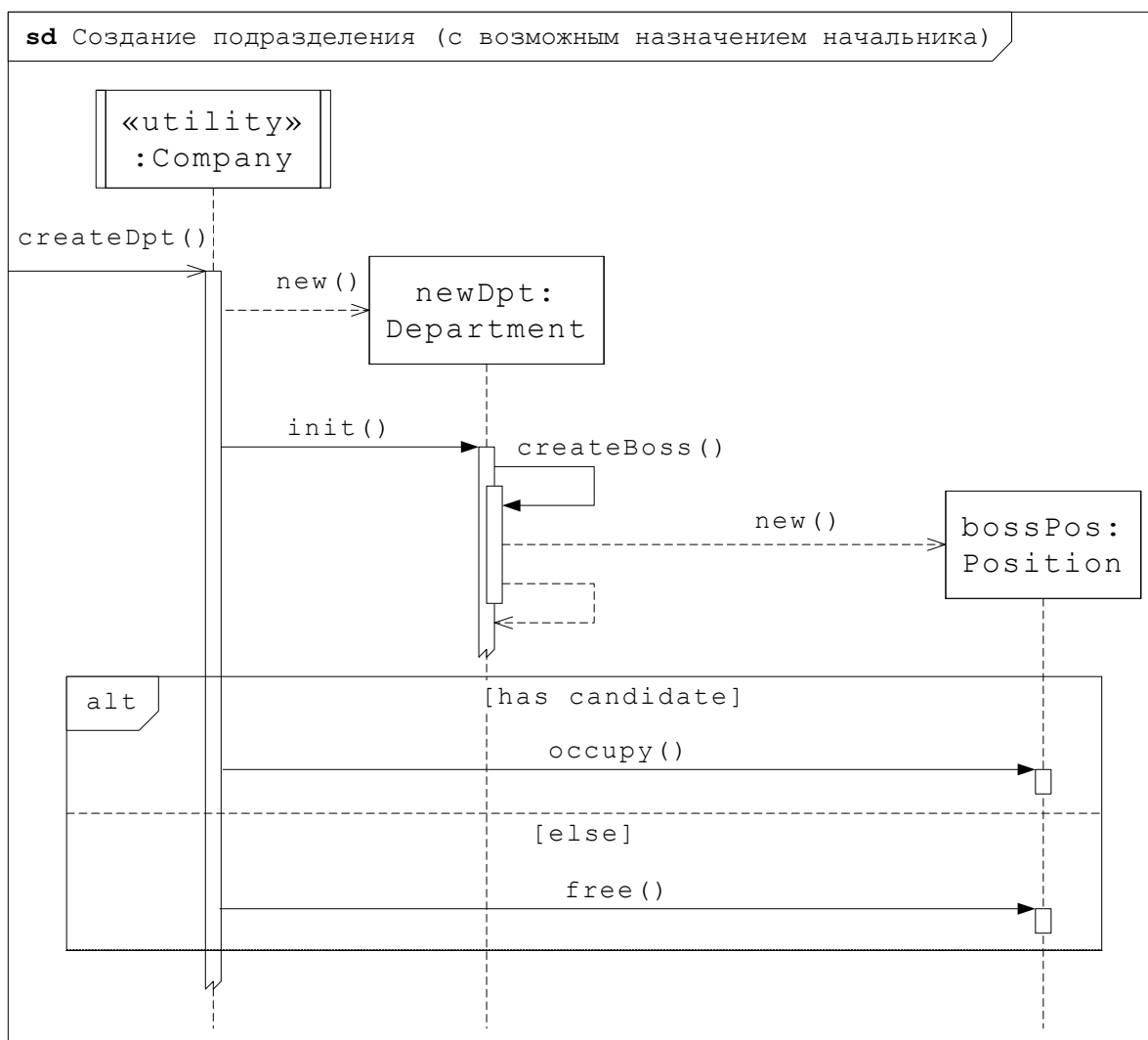


Рис. 4.34. Составной шаг взаимодействия

Рассмотрим еще пример из информационной системы отдела кадров. В этом примере рассматривается одна процедура низкого уровня, а именно, открытие сессии информационной системы. При запуске клиента нужно выполнить две задачи: во-первых, пользователь должен ввести допустимое имя (name) и пароль (password), а во-вторых, нужно проверить наличие лицензии на работу для данного клиента. Обе эти проверки — составные части одной большой задачи — открытие сессии (start session), протокол выполнения которой приведен на рис. 4.35.

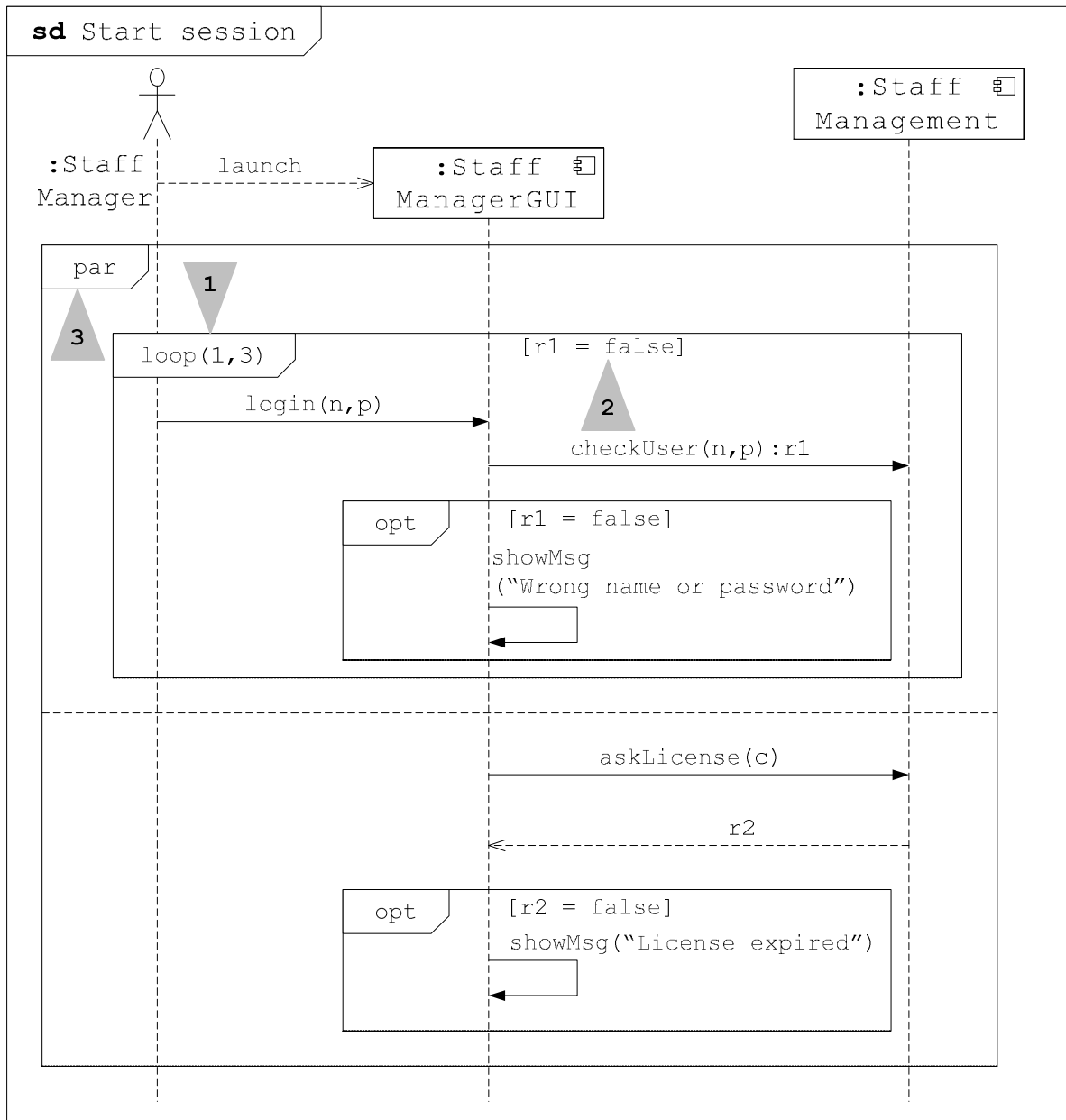


Рис. 4.35. Использование вложенных составных шагов взаимодействия

На это пользователю дается три попытки, что передано составным шагом `loop(1,3)` (1 на рис.4.35) и сторожевым условием продолжения попыток `[r1=false]` (2 на рис.4.35). Результат идентификации пользователя сохраняется в переменной `r1`. Результат проверки лицензии сохраняется в переменной `r2`. Эти две

проверки можно выполнять независимо друг от друга, то есть параллельно (3 на рис. 4.35).

Разумеется, также как и в случае со структурами управления в обычных языках программирования, любой шаг, входящий в составной шаг взаимодействия, в свою очередь может быть составным шагом любого типа и так далее на любую глубину вложенности. Однако в текстовой записи программы на обычном языке легко обеспечить наглядность для вложенных конструкций — достаточно применить обычную технику отступов. На диаграммах это не так просто: вложенный шаг придется нарисовать во вложенной рамке и так далее. Одна-две вложенных рамки — вполне терпимо (см. рис. 4.35). Но три, четыре, пять уровней вложенности — придется использовать слишком мелкий масштаб, диаграмму будет трудно читать.

Для решения этой проблемы применяется *использование взаимодействия* (interaction use), которое означает ссылку на взаимодействие (обычно представленное диаграммой последовательности), определенное в другом месте. Синтаксически использование взаимодействия задается такой же рамкой, как и составной шаг, но с ключевым словом `ref`. Внутри рамки пишется имя взаимодействия, на которое делается ссылка, возможно с аргументами и возвращаемыми значениями. Использование данной конструкции позволяет нарисовать сложное взаимодействие на нескольких диаграммах, оставляя каждую из них обзримой. Например, задав взаимодействие по открытию сессии на рис. 4.35, на рис. 4.36 мы можем использовать это взаимодействие.

Особенно часто конструкция `ref` применяется на обзорных диаграммах взаимодействия, которые рассматриваются в следующем параграфе и где приведена диаграмма для этого же примера.

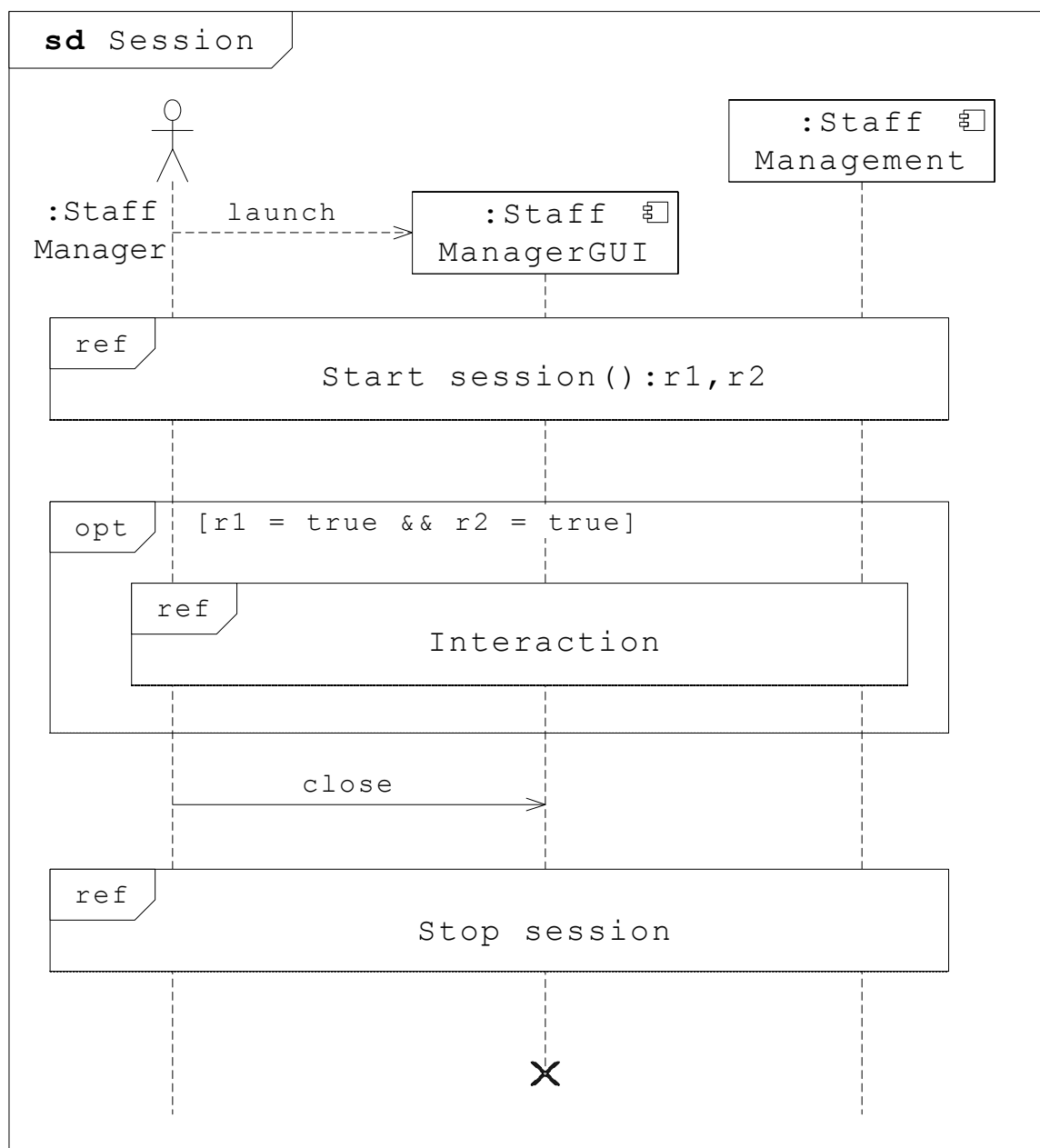


Рис. 4.36. Использование взаимодействия

#### 4.4.4. Обзорные диаграммы взаимодействия

В UML 2 появился еще один тип диаграмм, который по названию принято относить к диаграммам взаимодействия — обзорные диаграммы взаимодействия.

**Обзорная диаграмма взаимодействия** (interaction overview diagram) — это диаграмма деятельности, в которой используются

узлы управления, не используются узлы данных, а вместо узлов действий и деятельности используются фрагменты диаграмм взаимодействия в форме диаграмм последовательности или в форме ссылок на взаимодействие.

Мы считаем, что их включение в язык оправдано, потому что иногда эти диаграммы бывают очень удобны и позволяют лаконично и наглядно описать то, для чего они предназначены — общую схему взаимодействия, оставляя раскрытие деталей другим диаграммам.

Для примера рассмотрим еще раз общее описание работы информационной системы отдела кадров (см. рис. 4.36), но теперь уже представленное на обзорной диаграмме взаимодействия (рис. 4.37).

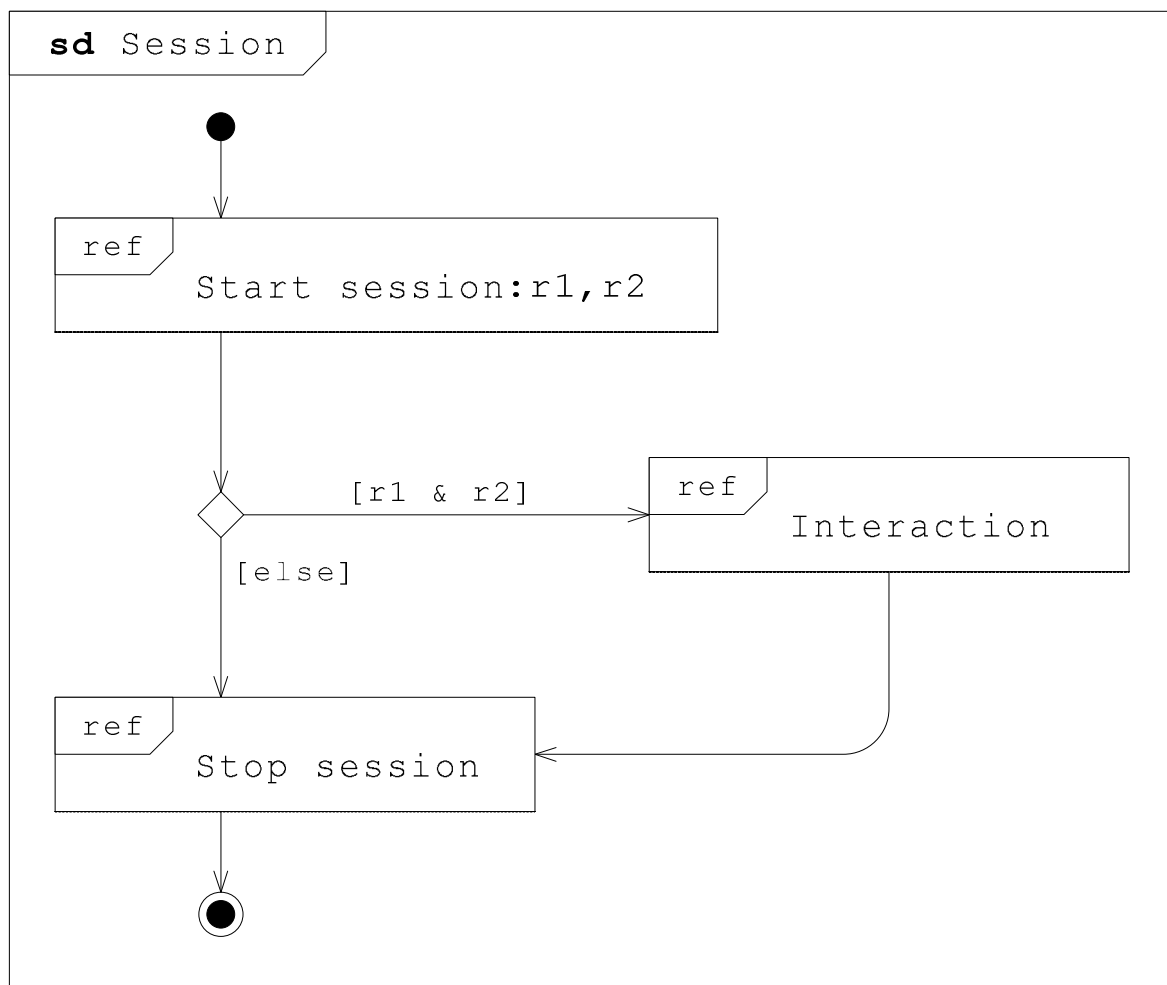


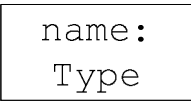


Рис. 4.37. Обзорная диаграмма взаимодействия

#### 4.4.5. Диаграммы коммуникации

Графических ухищрений на диаграмме коммуникации<sup>30</sup> значительно меньше — данный тип диаграмм графически очень лаконичен и, тем не менее, чрезвычайно выразителен. Поэтому при рассмотрении элементов диаграмм коммуникации мы больше внимания уделяем семантике, оставляя самоочевидную нотацию для примеров. В частности, описания некоторых семантических тонкостей, относящиеся также и к диаграммам последовательности и опущенные в предыдущих параграфах, здесь восполнены. Для начала мы остановимся на трех семантических понятиях, присущих диаграммам коммуникации, которые не имеют броского графического выражения, но являются важным аспектом прагматики диаграмм взаимодействия и могут быть просто не замечены и пропущены читателем, если он знакомится с языком, глядя только на картинку. Вдобавок эти понятия были переименованы и переопределены при переходе от UML 1 к UML 2, а потому возникает терминологическая путаница, особенно в русских переводах. Читателю не следует пугаться необычных сочетаний слов в табл. 4.5.

Таблица 4.5

**Некоторые понятия, связанные с кооперацией**

UML 1	UML 2	Обозначение	Определение
Роль классификатора	Роль (часть)		Обозначение слота объекта, участвующего во взаимодействии
Роль ассоциации	Соединитель		Обозначение слота связи, вдоль которой осуществляется взаимодействие
Контекст взаимодействия	Кооперация		Статическая структура (граф), показывающая роли и связи

---

<sup>30</sup> Напомним, что изначально эти диаграммы назывались диаграммами кооперации и были переименованы в диаграммы коммуникации в UML 2.0.

Как уже много раз было сказано, сущностями на диаграммах взаимодействия (в частности, на диаграмме коммуникации) являются объекты. Это действительно так, но с одной оговоркой: объект на диаграмме взаимодействия может выступать в двух ипостасях. Это может быть конкретный индивидуальный объект и тогда диаграмма описывает конкретное взаимодействие с участием данного объекта (условно можно сказать, что диаграмма является экземпляром взаимодействия). Но также же это может быть слот во фрейме взаимодействия, подлежащий заполнению некоторым подходящим объектом, и тогда диаграмма описывает множество взаимодействий, задавая их общую схему (то есть условно можно сказать, что диаграмма коммуникации является дескриптором класса взаимодействий).

Второй из рассмотренных случаев, т. е. когда на диаграмме подразумевается слот, подлежащий заполнению объектом, называется в UML 1 *ролью классификатора* (classifier role), а в UML 2 просто *ролью* или *частью* (part) в зависимости от контекста.<sup>31</sup> Синтаксически роль классификатора и конкретный объект почти неразличимы: в обоих случаях изображается стандартная фигура классификатора (прямоугольник), в которой вписано имя и классификатор, разделенные двоеточием. Различие заключается в том, что в случае использования роли классификатора имя предлагается не подчеркивать.

Совершенно аналогично понятию роли классификатора вводится понятие *роли ассоциации* (или *соединителя* в терминологии UML 2). Отношения между объектами (ролями классификаторов) — это связи, которые чаще всего бывают экземплярами ассоциаций. Но если связь связывает роли классификаторов, т. е. слоты объектов, то

---

<sup>31</sup> Название, прямо скажем, не очень удачное, а потому трудно переводимое. Все-таки речь идет об экземпляре определенного классификатора.

она сама является слотом — слотом связи, который называется ролью ассоциации.

Диаграмма коммуникации (равно как и диаграмма последовательности) описывает поведение как *взаимодействие*, т. е. как протокол обмена сообщений между объектами. Один и тот же объект может участвовать в различных взаимодействиях, играя в них различные роли. Таким образом, взаимодействие всегда происходит в определенном контексте, который определяется множеством участвующих во взаимодействии объектов и связей. Несколько утрируя, можно сказать, что диаграмма коммуникации, в которой не указаны сообщения (и которая, тем самым, синтаксически неотличима от диаграммы объектов) является *контекстом взаимодействия*.

*Номер сообщения* определяется в соответствии с положением сообщения в последовательности сообщений данного взаимодействия. Если во взаимодействии используются только простые или асинхронные сообщения (см. табл. 4.4), то сообщения просто нумеруются, обычно подряд: 1, 2, 3 и т. д. Сообщения с меньшими номерами предшествуют сообщениям с большими номерами. Если же используются вложенные потоки управления, т. е. сообщения типа вызова операции (см. табл. 4.4), то сообщения нумеруются более сложным образом. Допустим, сообщение вызова некоторой операции имеет номер  $x$ . Тогда сообщения, отправляемые при выполнении этой операции, будут иметь номера  $x.1$ ,  $x.2$   $x.3$  и т. д. Первое сообщение, отправляемое при выполнении вызова  $x.1$  будет иметь номер  $x.1.1$  и т. д. Количество точек в номере соответствует уровню вложенности потока управления, т. е. глубине стека вызовов. Таким образом, например, сообщение с номером 1.2 предшествует сообщению с номером 1.2.3, а сообщение 2.1, напротив, следует за сообщением 1.2.3. Если первым во взаимодействии является сообщение вызова метода, то его номер часто не указывают (такое сообщение как бы имеет неявный номер 0), чтобы не загромождать



диаграмму повторяющимся всюду номером первого сообщения и ненужной точкой.

Рассмотрим пример из информационной системы отдела кадров. Рис. 4.38 семантически эквивалентен рис. 4.31 — эти диаграммы описывают одно и то же взаимодействие. Мы рекомендуем читателю просто сравнить эти две диаграммы.

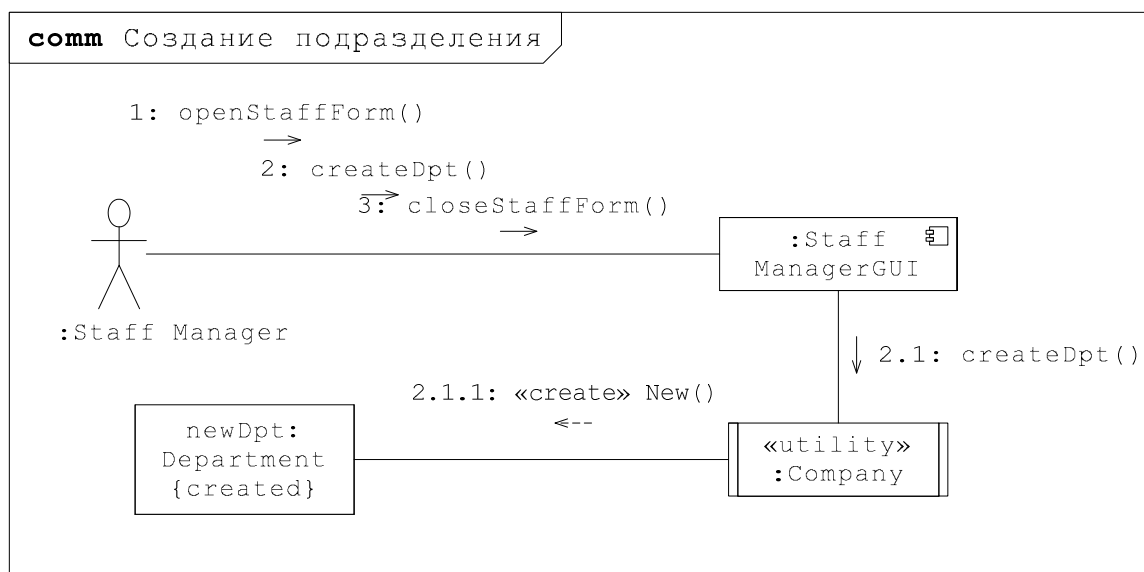


Рис. 4.38. Диаграмма коммуникации

#### 4.4.6. Диаграммы синхронизации

В UML 2 введен новый<sup>32</sup> тип диаграмм взаимодействия, созданный для описания изменения состояния объектов с течением времени в результате взаимодействия. Эти диаграммы в UML получили название *диаграмм синхронизации* (timing diagram).

Рассмотрение диаграммы начнем сразу с примера из информационной системы отдела кадров. На рис. 4.39 изображена диаграмма синхронизации для объекта newDpt (например, см. рис. 4.38) экземпляра класса Department.

<sup>32</sup> Надо отметить, что инженеры-электронщики подобные диаграммы использовали издавна.

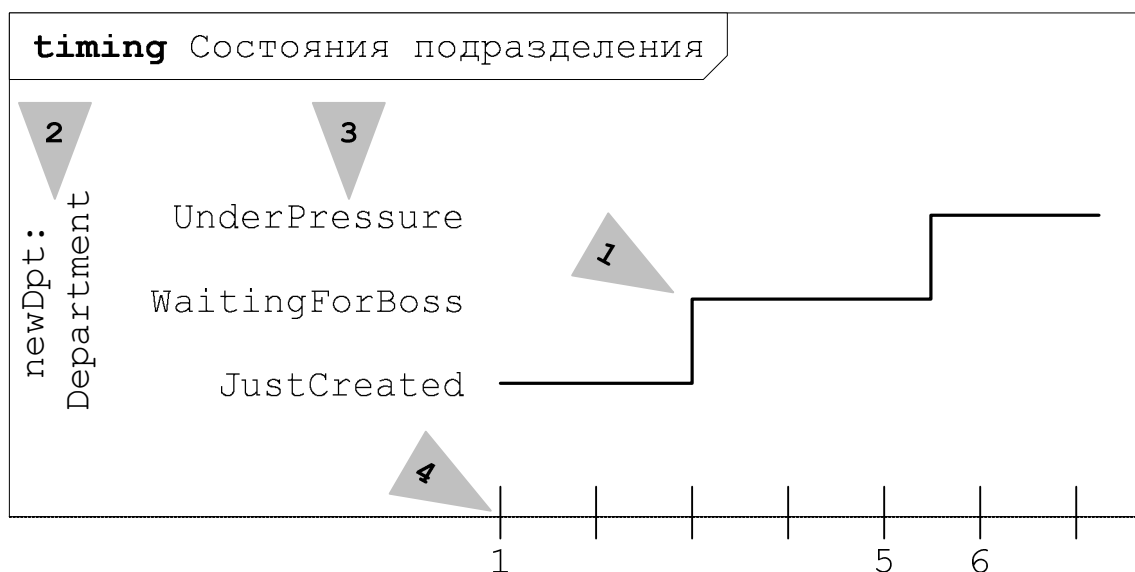


Рис. 4.39. Диаграмма синхронизации для одного объекта

В данном примере рассматривается смена состояний подразделения в процессе его создания. Линия жизни (1 на рис. 4.39) описывает изменение состояния объекта с течением времени. В каждый момент времени объект `newDpt` (2 на рис. 4.39) находится в одном из нескольких заданных состояний (3 на рис. 4.39), отображаемых в диаграмме на оси ординат. По оси абсцисс отображается время (4 на рис. 4.39) в некоторых единицах измерения. В нашем примере подразделение последовательно меняет три состояния:

- `JustCreated` — подразделение только что создано и еще не имеет структуры;
- `WaitingForBoss` — имеет незаполненную вакансию начальника;
- `UnderPressure` — подразделение имеет начальника и может, наконец, спокойно работать.

Как видно, диаграмма синхронизации становится довольно громоздкой при большом количестве возможных состояний. В связи с этим существует упрощенный вариант нотации диаграммы синхронизации, представленный на рис. 4.40.

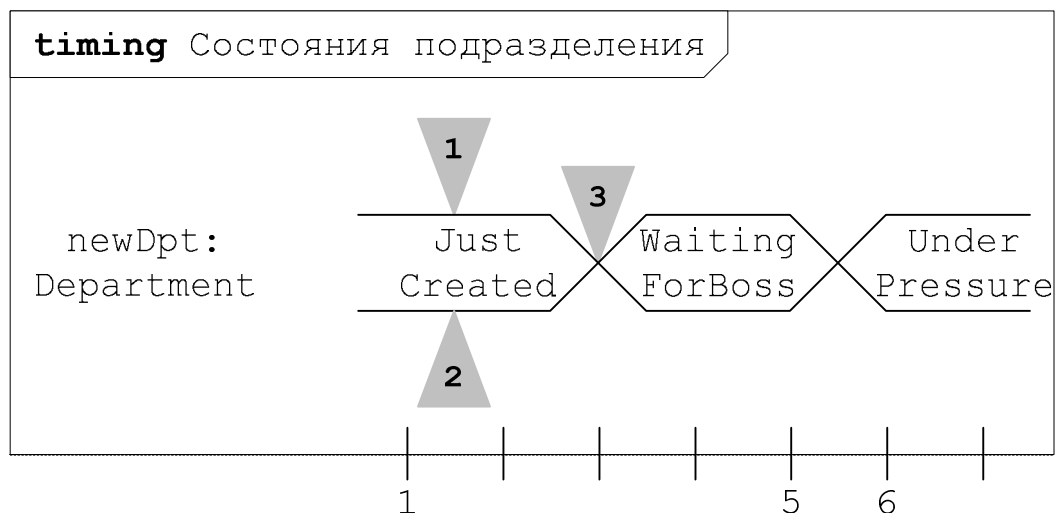


Рис. 4.40. Упрощенное представление диаграммы синхронизации

В упрощенном виде диаграммы синхронизации состояния объекта записываются между двух параллельных горизонтальных линий (1 и 2 на рис. 4.40) – упрощенное представление линии жизни, которые пересекаются в точках смены состояния (3 на рис. 4.40). Такой вид диаграмм более компактен и предпочтительней при большом количестве возможных состояний.

Рассмотрим теперь, как можно показать взаимодействие объектов на диаграммах синхронизации. За основу возьмем взаимодействие в случае, когда при создании подразделения существует кандидат на позицию начальника подразделения. Сравните это взаимодействие с диаграммой на рис. 4.41.

Если для показа сообщения требуется нарисовать линию, пересекающую большое количество линий жизни, то можно использовать метки продолжения (2 на рис. 4.41). Само сообщение при этом показывается у начального или у конечного сегмента линии.

На диаграммах синхронизации можно показать не только относительный порядок событий, но и их привязку к реальному ходу времени. Разумеется, можно использовать любые единицы измерения, шкала времени не обязана быть равномерной, и не обязана быть непрерывной. Так же можно объединять над общей шкалой времени

диаграммы различных типов (полного и упрощенного) для представления взаимодействия большого количества объектов.

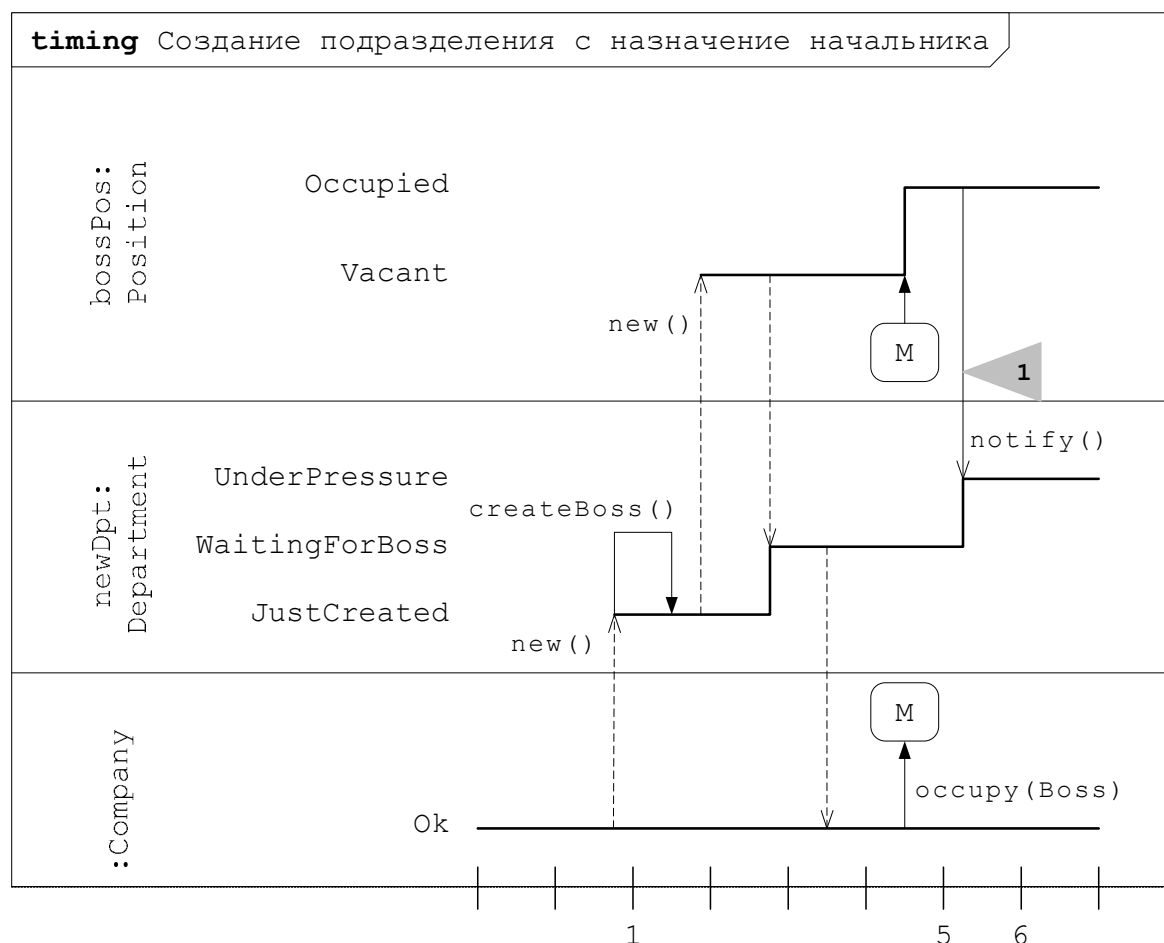


Рис. 4.41. Диаграмма синхронизации для нескольких объектов

## 4.5. МОДЕЛИРОВАНИЕ ПАРАЛЛЕЛИЗМА

Термин *параллельность* в программировании, вообще говоря, означает "одновременное" выполнение нескольких действий. Слово "одновременное" в данном контексте означает, что невозможно (или не нужно) указывать, какая из действий происходит раньше другой во времени. Другими словами, параллельные процессы не упорядочены во времени. Тем самым термин "параллельный" противопоставляется термину "последовательный":

последовательные деятельности упорядочены во времени, причем линейно.

Средства описания параллелизма в UML отнюдь не противопоставлены средствам описания последовательного поведения, напротив, они образуют единое целое, поскольку параллельное программирование скорее общее правило, нежели экзотическое исключение. Мы отделили обсуждение средств описания параллельного поведения от средств описания последовательного поведения только с целью упростить изложение основных идей каждого из типов канонических диаграмм, используемых для описания поведения. В последующих разделах поочередно рассматриваются отдельные опущенные выше детали и конструкции диаграмм описания поведения, относящиеся к моделированию параллелизма. Мы начинаем с простых и часто используемых средств и постепенно переходим к более запутанным, а потому реже используемым.

#### **4.5.1. Взаимодействие последовательных процессов**

Взаимодействие в UML моделируется с помощью сообщений: синхронных (вызовы операций) и асинхронных (посылка сигналов). Типичный вариант: взаимодействие машин состояний разных классов с помощью действий, выполняемых на переходах.

Рассмотрим два класса из информационной системы отдела кадров: `Person` и `Position` (рис. 4.42).

Рассмотрим теперь, как должна выполняться операция назначения сотрудника на новую должность, т. е. операция перевода. Мы оставим в стороне вопрос о том, в каком классе разумно определить данную операцию (на самом деле это совершенно не важно), и положим, что операция назначения сотрудника на должность имеет два параметра — сотрудник и должность:

```
assignP2P(person:Person, position:Position)
```

Допустим, что требуется обеспечить элементарный порядок в учете кадров (на программистском языке — целостность данных):

если сотрудник А назначен на должность Б, то и в должности Б должно быть записано, что ее занимает сотрудник А и наоборот.

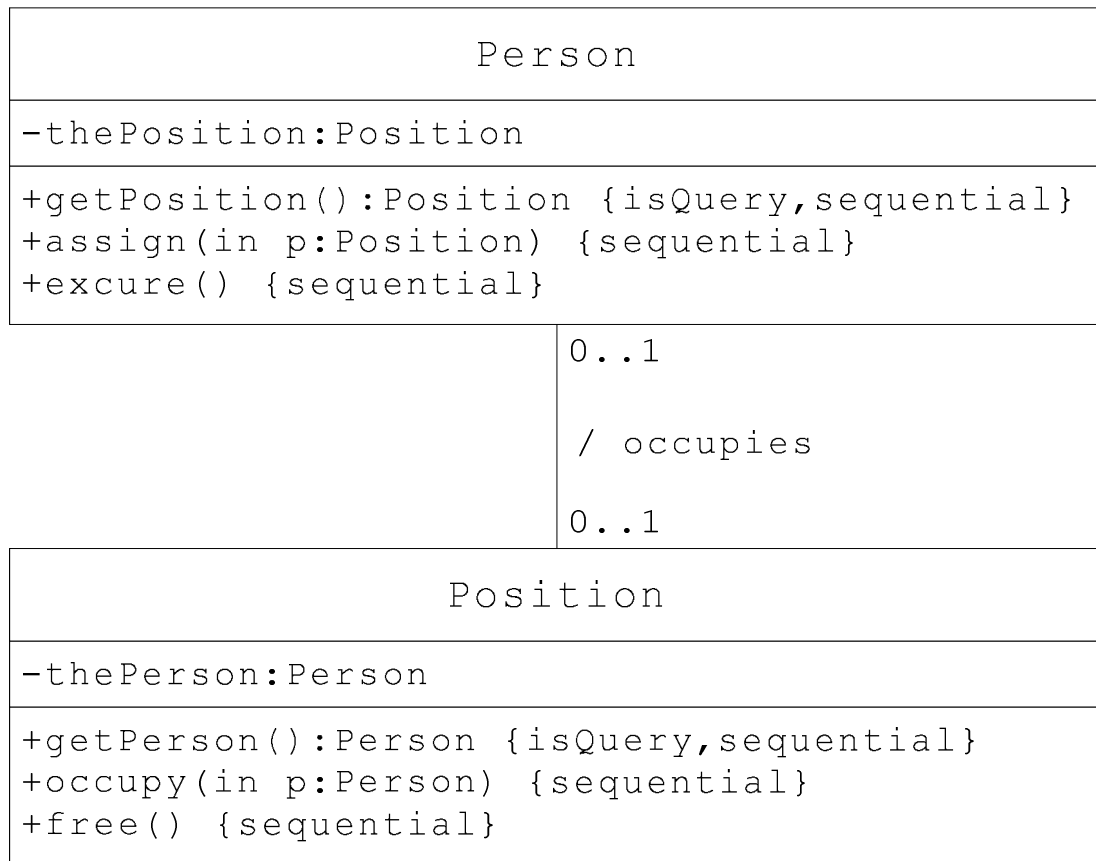


Рис. 4.42. Классы Position и Person

Другими словами, занятые должности и сотрудники должны взаимно однозначно соответствовать друг другу, а свободные должности и сотрудники должны быть действительно свободны и не должны содержать неадекватных ссылок друг на друга.

Требуемое поведение операции assignP2P() можно описать с помощью диаграммы объектов (фактически, это контекст взаимодействия), на которой показано, как должны измениться связи между объектами в результате выполнения операции (см. рис. 4.43). В данном описании контекста рассматривается типичный сценарий, в котором до выполнения операции сотрудник занимает некоторую должность, а целевая должность вакантна.

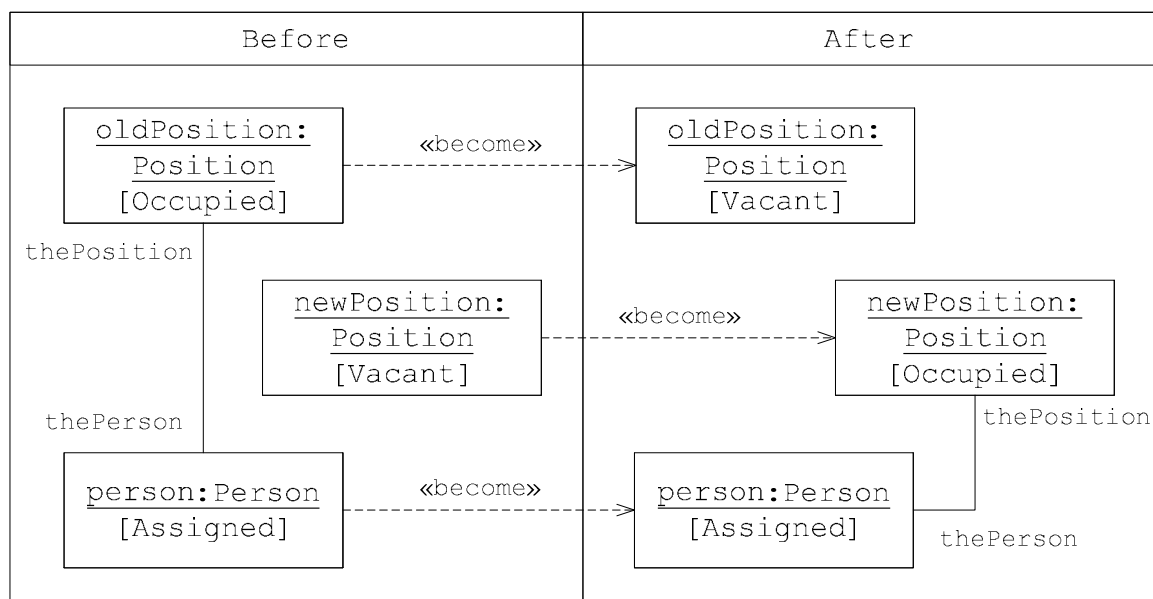


Рис. 4.43. Изменение связей и состояний объектов при выполнении операции перевода сотрудника

Мы видим, что при назначении сотрудника на должность задействованы три объекта: `oldPosition`, `newPosition` и `person`. Требуемое поведение может быть обеспечено за счет взаимодействия автоматов, реализующих поведение каждого из этих объектов. На рис. 4.44 и рис. 4.45 приведены диаграммы машин состояний для классов `Position` и `Person`, соответственно.

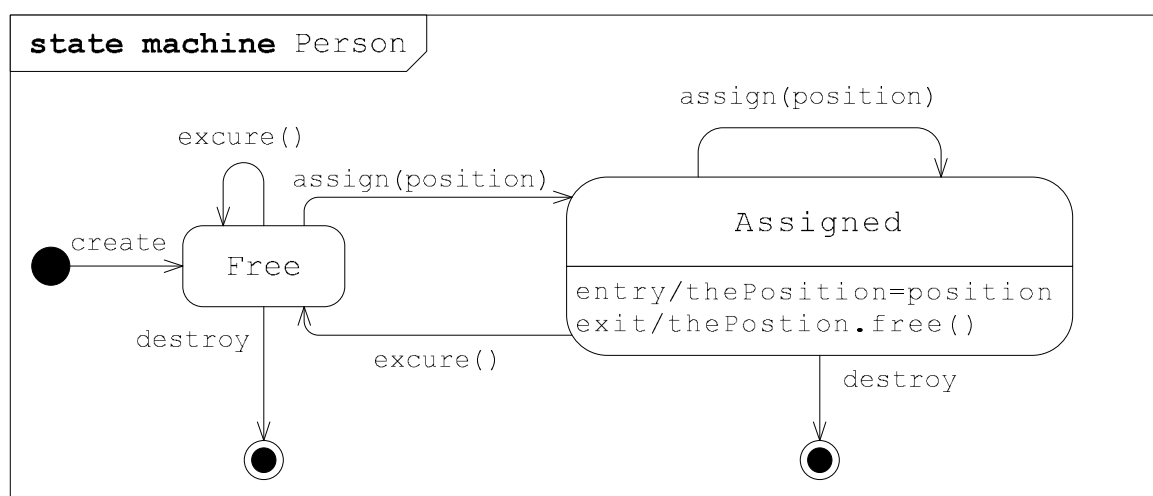


Рис. 4.44. Машина состояний класса `Position`

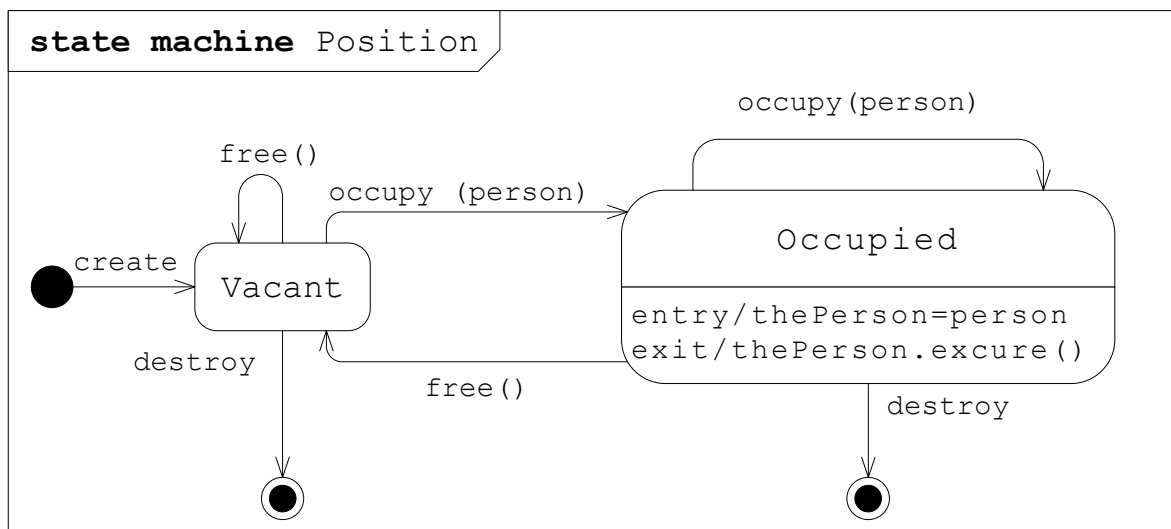


Рис. 4.45. Машина состояний класса `Person`

В таком случае, операция назначения сотрудника на должность `assignP2P()` может быть реализована двумя вызовами операций над объектами, являющимися ее аргументами:

```

position.occupy(person)
person.assign(position)
  
```

Мы подошли к кульминации данного примера: указанные две операции можно вызвать в любом порядке или **параллельно**, более того, их можно вызывать с ожиданием возврата управления или без ожидания, т.е. синхронно или асинхронно — в любом случае взаимодействие автоматов, приведенных на рис. 4.44 и рис. 4.45, обеспечит требуемое поведение (если только в процесс обмена сообщениями не вмешается "посторонний" вызов операции одного из этих объектов).

**Параллельное программирование — мощный способ моделирования поведения, и взаимодействующие конечные автоматы — великолепное средство параллельного программирования.**



#### **4.5.2. Ортогональные состояния и составные переходы**

В этом параграфе рассматриваются средства описания параллельности, применяемые на диаграммах автомата.

На диаграммах автомата параллельное выполнение моделируется с помощью ортогональных (параллельных) составных состояний. По существу, идея параллельных состояний очень проста и легко воспринимается: внутрь составного состояния вложено несколько машин состояний, которые работают параллельно.

Переход в ортогональное составное состояние означает одновременный переход в начальные состояния всех областей данного ортогонального составного состояния (т. е., фактически, параллельный запуск всех вложенных машин состояний); в случае наличия перехода в ортогональное составное состояние каждая область должна иметь единственное начальное (или историческое) состояние, в противном случае модель считается противоречивой.

Переход по событию и/или со сторожевым условием (т. е. любой переход не по завершении) из ортогонального составного состояния означает распространение данного перехода на все вложенные состояния всех областей; если унаследованный переход конфликтует с локально определенным переходом, то последний имеет приоритет (модель не считается противоречивой).

Переход по завершении из ортогонального составного состояния срабатывает в том и только в том случае, когда машина состояний в каждой области перешла в заключительное состояние (т. е., фактически, это означает синхронное завершение всех вложенных машин состояний); в случае наличия перехода по завершении из ортогонального составного состояния каждая область должна иметь заключительные состояния и должен быть определен единственный переход по завершении, в противном случае модель считается противоречивой.

Таким образом, семантика переходов, не пересекающих границу состояния, для ортогональных и последовательных составных

состояний аналогична. Для описания переходов, пересекающих границу ортогонального составного состояния, вводится специальное понятие — составной переход.

***Составной переход** (compound transition) — это переход, который начинается и/или заканчивается в нескольких состояниях.*

Если переход имеет **одно исходное и несколько целевых состояний**, то это соответствует разветвлению потока управления на несколько параллельных потоков; при этом целевые состояния должны быть вложенными состояниями областей ортогонального составного состояния — по одному на каждую область.

Если переход имеет **несколько исходных и одно целевое состояние**, то это соответствует слиянию нескольких потоков управления в один; при этом исходные состояния должны быть вложенными состояниями областей ортогонального составного состояния — по одному на каждую область.

Если переход имеет **несколько исходных и несколько целевых состояний**, то это соответствует синхронизации нескольких параллельных потоков управления; при этом исходные состояния должны быть вложенными состояниями областей одного ортогонального составного состояния — по одному на каждую область, и целевые состояния должны быть вложенными состояниями областей ортогонального составного состояния (возможно, другого) — также по одному на каждую область.

В любом случае составной переход должен переводить машину состояний из одной допустимой конфигурации активных состояний в другую допустимую конфигурацию активных состояний, в противном случае модель синтаксически неправильна. Неформально говоря, это значит, что ортогональное составное состояние нельзя покинуть и нельзя войти в него "частично" — в составном переходе должны участвовать по одному "представителю" (по одному вложенному состоянию) от каждой области ортогонального составного состояния, участвующего в переходе.

Составной переход может иметь событие и сторожевое условие. Как всегда, переход срабатывает, если произошло событие и выполнено условие. При этом, для того чтобы сработал переход, имеющий несколько исходных состояний, необходимо, чтобы все они были активны при наступления события.

Приведем пример из информационной системы отдела кадров. Рассмотрим тот же пример, что был использован при описании последовательных простых и составных состояний, а именно жизненный цикл сотрудника на предприятии. Диаграмма автомата сотрудника в информационной системе отдела кадров приведена на рис. 4.46.

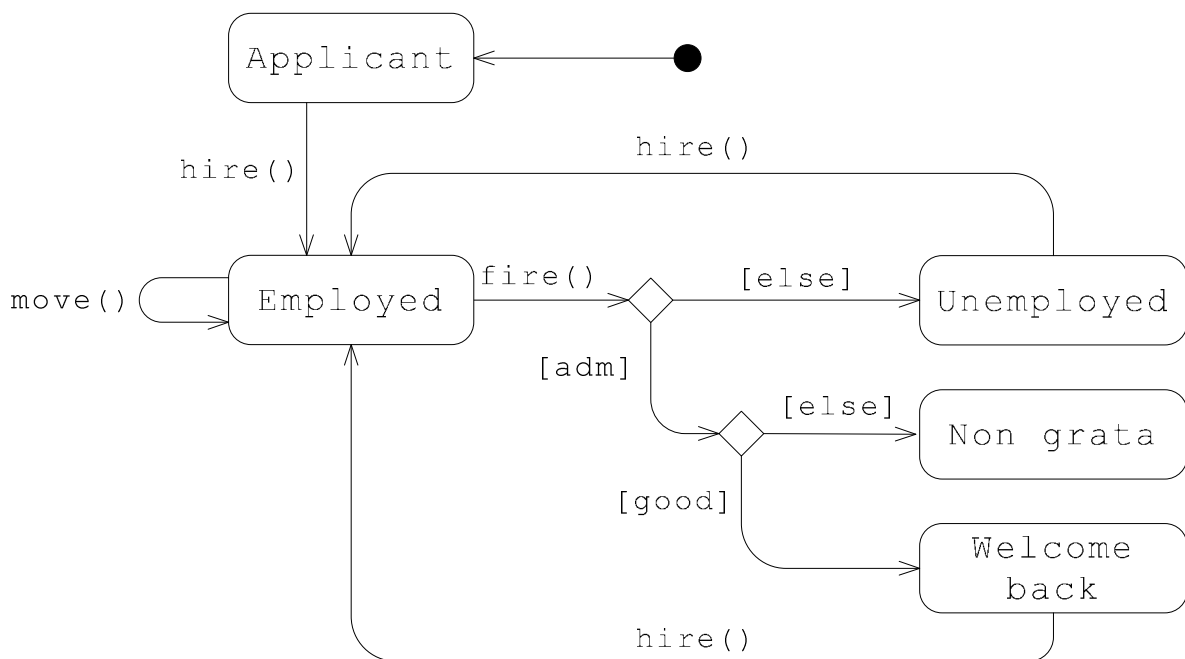


Рис. 4.46. Диаграмма автомата сотрудника ИС ОК

Рассмотрим состояние "работающий сотрудник" (на диаграммах обозначается Employed). Очевидно, что данное составное состояние — самое важное для системы и должно быть рассмотрено с наибольшей степенью подробности. Мы уже выделили некоторые вложенные состояния (см. рис. 4.8): сотрудник может находиться в офисе (In office), болеть (Illness) или быть в отпуске

(On vacation). Но параллельно с этим набором состояний, описывающим статус сотрудника в смысле присутствия на рабочем месте, сотрудник переживает и другие смены состояний, связанные с его статусом на предприятии (см. рис. 4.10).

## ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

*После успешного выполнения задания на испытательном сроке сотрудник либо зачисляется в штат, либо становится внештатным сотрудником.*

К состояниям On trial (сотрудник проходит испытательный срок) и On staff (сотрудник состоит в штате), определенным на рис. 4.10, следует добавить еще одно — Contractor, означающее, что сотрудник работает по контракту (по трудовому соглашению на определенный срок). Ясно, что смена этих состояний не зависит (формально) от смены состояний из первого списка и, таким образом, образует ортогональную машину состояний. На рис. 4.47 представлено составное состояние Employed (пока без внешних переходов), а внутренние переходы в каждой машине мы определили с помощью нескольких операций: go() и goBack() для первого набора состояний и promote() для второго. Для этих операции в качестве параметра указывается состояние, в которое нужно перейти.

Рассмотрим теперь переходы. При первом приеме на работу сотрудник должен пройти испытательный срок и в первый день работы должен быть в офисе. Поэтому переходы из соответствующих начальных состояний (1 и 2 на рис. 4.47) областей машины состояний ведут в состояния In office и On trial. Таким образом, внешний переход из состояния Applicant можно провести в состояние Employed как **простой** переход (1 на рис. 4.48) по событию hire(). С другой стороны, переход из состояния Unemployed, не должен приводить к новому испытательному сроку.

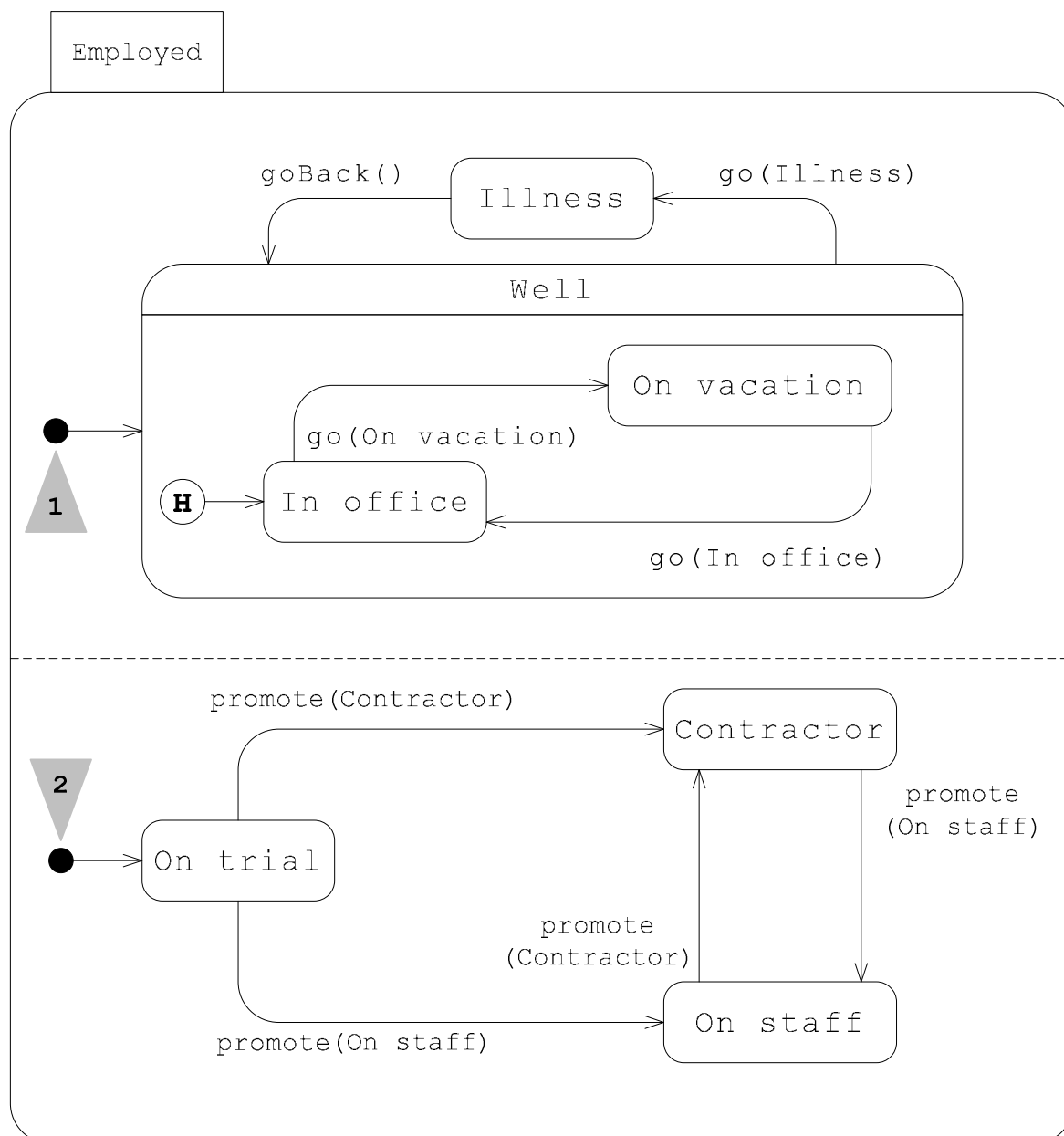


Рис. 4.47. Ортогональное составное состояние Employed

В таком случае переход по событию `hire()` из состояния `Unemployed` разумно определить как **составной** переход в состояния `In office` и `Contractor`. Аналогично, переход в случае увольнения должен происходить, во-первых, когда сотрудник находится в офисе (увольнять заочно не принято), и, во-вторых, когда сотрудник занимает должность постоянно. Прекращение трудовых

отношений с контрактником или проходящим испытательный срок обычно даже не называется увольнением. Это можно отразить с помощью соответствующего **составного** перехода из состояний `In office` и `On staff`. Далее, переход в себя по событию `move()` (4 на рис. 4.48), видимо, не должен менять конфигурацию активных состояний, поэтому в параллельных вложенных машинах целесообразно заменить начальные состояния историческими (5 и 6 на рис. 4.48). Обратите внимание на необходимость использование **глубинного** исторического состояния (5 на рис. 4.48).

На рис. 4.48 приведен соответствующий фрагмент диаграммы автомата. События на переходах во вложенных параллельных областях мы не стали указывать повторно, чтобы не загромождать диаграмму. Составные переходы изображаются с помощью специального значка, который выглядит как узкая закрашенная полоска (может быть расположен вертикально или горизонтально) и называется *линейкой синхронизации*. Все сегменты составного перехода начинаются или заканчиваются на линейке синхронизации. В зависимости от того, сколько сегментов переходов начинается и заканчивается на линейке синхронизации, они получают специальное название (обозначение не меняется). Если на линейке начинается один сегмент и заканчивается несколько, то линейка синхронизации называется *соединением* (`join`). Если на линейке заканчивается один сегмент и начинается несколько, то линейка синхронизации называется *развилкой* (`fork`). В данном случае на рис. 4.48 использованы одно соединение (3 на рис. 4.48) и одна развилка (2 на рис. 4.48).

Поведение машин состояний в параллельных областях составного состояния `Employed` на диаграммах рис. 4.47 и рис. 4.48 независимо — каждая из групп параллельных состояний "живет своей жизнью".

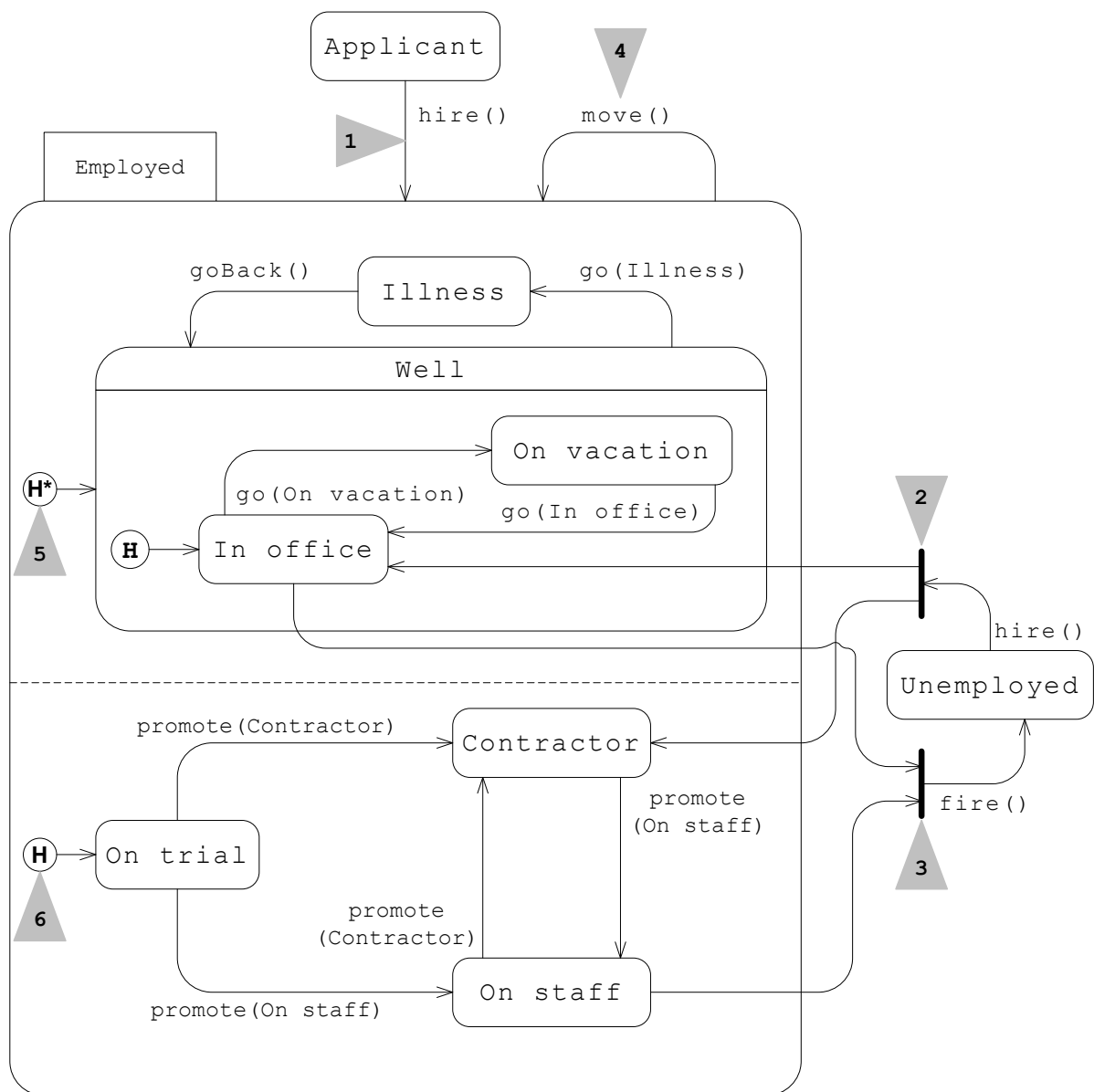


Рис. 4.48. Составные переходы

Поведение машин состояний в параллельных областях составного состояния Employed на диаграммах рис. 4.47 и рис. 4.48 независимо — каждая из групп параллельных состояний "живет своей жизнью".

### 4.5.3. Развилки и слияния

В этом параграфе рассматриваются средства описания параллельности, применяемые на диаграммах деятельности.

Диаграмма деятельности похожа на привычную блок-схему, но допускает использование различных дополнительных обозначений. Это правило распространяется, прежде всего, на моделирование параллелизма: основные средства моделирования параллельного поведения на диаграммах деятельности имеют много общего с рассмотренными в предыдущем параграфе, плюс некоторые дополнительные особенности. Основными средствами являются уже рассмотренные развилки и слияния, а дополнительным — обусловленный поток управления. Рассмотрим их все по порядку.

**Развилки и слияния — это средства визуализации составных переходов.** Диаграмму деятельности можно трактовать как диаграмму состояний, в которой используются (в основном) переходы по завершении. Что же такое составной переход по завершении? В сущности, это очень простая и естественная конструкция, может быть даже более естественная, чем общий случай составного перехода.

*Составной переход по завершении* на диаграмме деятельности имеет следующие особенности.

Во-первых, никто не использует данный термин: все применяют названия частных случаев нотации: *развилка*, *слияние*, *линейка синхронизации*. Все эти случаи являются составными переходами по завершении.

Во-вторых, семантика данных конструкций определяется в UML 1 в терминах потоков управления. В общем случае имеется линейка синхронизации, которой инцидентны входящие (один или более) и исходящие (один или более) сегменты переходов. Семантика состоит в следующем. Сначала все деятельности, инцидентные



входящим сегментам, должны завершиться. После этого параллельно запускаются все деятельности, инцидентные исходящим сегментам.

В-третьих, семантика данных конструкций определяется в UML 2 альтернативным, но эквивалентным способом в терминах маркеров. Развилка создает столько копий пришедшего маркера (управления или данных), сколько имеется исходящих дуг и запускает созданные маркеры во все исходящие дуги. Слияние, напротив, дожидается получения маркеров по всем входящим дугам, после чего отправляется маркер по единственной исходящей дуге. Общий случай — **линейка синхронизации (с несколькими входящими и несколькими исходящими дугами)** рассматривается как **композиция слияния, сразу за которым следует развилка.**

В-четвертых, развилки и слияния должны быть *сбалансированы*. Правило сбалансированности аналогично правилу вложенности на диаграммах состояний. Мы определим его следующим образом. Назовем диаграмму деятельности *строго сбалансированной*, если ее можно получить путем декомпозиции одной деятельности, причем на каждом шаге декомпозиции одна из деятельностей декомпозируется либо на некоторое количество последовательных деятельностей, либо на некоторое число параллельных деятельностей. Диаграмма деятельности является сбалансированной, если ее можно сделать строго сбалансированной путем введения промежуточных деятельностей и синхронизирующих состояний на имеющихся переходах. Неформально говоря, сбалансированность означает возможность выполнить блок-схему от начала к концу: потоки управления не должны возникать "ниоткуда" и исчезать "в никуда". На рис. 4.49 приведен пример сбалансированной (слева) и не сбалансированной (справа) диаграмм деятельности.

Подводя итоги параграфа, отметим, что, по нашему мнению, **средства моделирования параллелизма на диаграммах деятельности принадлежат к числу лучших в UML: они интуитивно понятны, наглядны и удобны в использовании.**

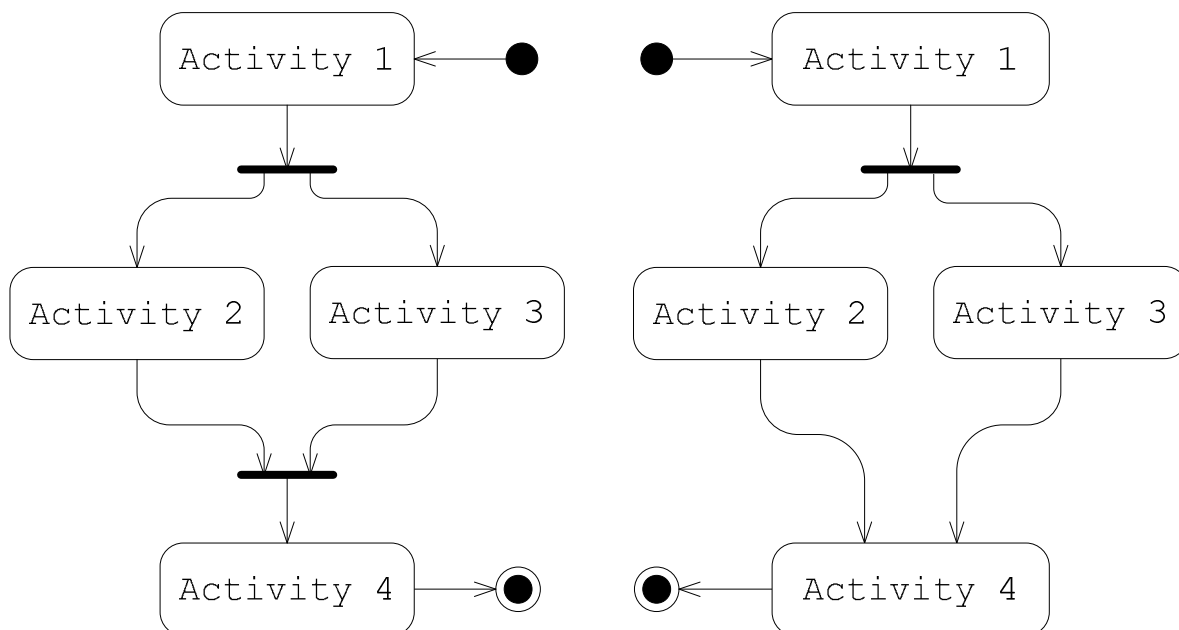


Рис. 4.49. Баланс развилок и слияний

#### 4.5.4. Параллелизм на диаграммах взаимодействия

На диаграммах коммуникации графических средств моделирования параллелизма, фактически, не предусмотрено. Параллелизм указывается текстуально, с помощью номеров сообщений. В номера сообщений можно включать не только цифры, но и буквы. **Сообщения, номера которых различаются в последней букве, считаются параллельными на данном уровне вызовов.** Их относительный порядок не определен. Они могут быть отправлены в любом порядке или даже физически одновременно, если это допускает реализация. В то же время, например, сообщения с номерами 1.1a и 1.1b **оба** предшествуют сообщению с номером 1.2 и **оба** следуют за сообщением с номером 1.

В UML 2 предусмотрены сразу **три** типа составных шагов взаимодействия, предназначенных для моделирования параллелизма. Покажем некоторые возможности их использования на примере

реализации операции `move()`. Описание параллельного взаимодействия, реализующего операцию, заключается в следующем.

Во-первых, нужно проверить, свободна ли целевая позиция. Если она занята, то с помощью составного шага взаимодействия `break` (1) нужно возбудить исключение (стрелка, идущая на границу диаграммы (2)) и прекратить взаимодействие.

Во-вторых, нужно выполнить три действия: освободить старую должность, занять новую и сделать отметку в самом объекте класса `Person`. Последовательность выполнения этих действий имеет значение. Сначала обязательно нужно обязательно освободить старую должность (3). Для этого можно использовать составной шаг взаимодействия `seq` (4), чтобы гарантировать нужный нам порядок выполнения действий. В то же время методы `occupy()` и `assign()` могут быть выполнены параллельно (5). В результате мы гарантированно обеспечиваем выполнение следующего бизнес-правила: сотрудник может в процессе выполнения операции временно "висеть в воздухе" (т. е. на него нет ссылок из должностей), но никогда, даже случайно и временно, не допускается, чтобы он "сидел на двух стульях" (т. е. на него есть две ссылки из разных должностей). Посмотрите на рис. 4.50 и сравните его с другими диаграммами, посвященными реализации операции перевода сотрудника.

На данной диаграмме в дополнение к уже известным нам нотациям сообщений (см. табл. 4.4) добавились еще два. Это так называемые найденные сообщения (`found message`) и потерянные сообщения (`lost message`). Найденные сообщения (6 на рис. 4.50) используются в случае, когда важно отразить факт получения сообщения, а факт его отправки не представляет интереса в рассматриваемом контексте. С потерянными сообщениями (7 на рис. 4.50) все в точности наоборот: главное то, что сообщение было отправлено, а получено или нет, не важно.

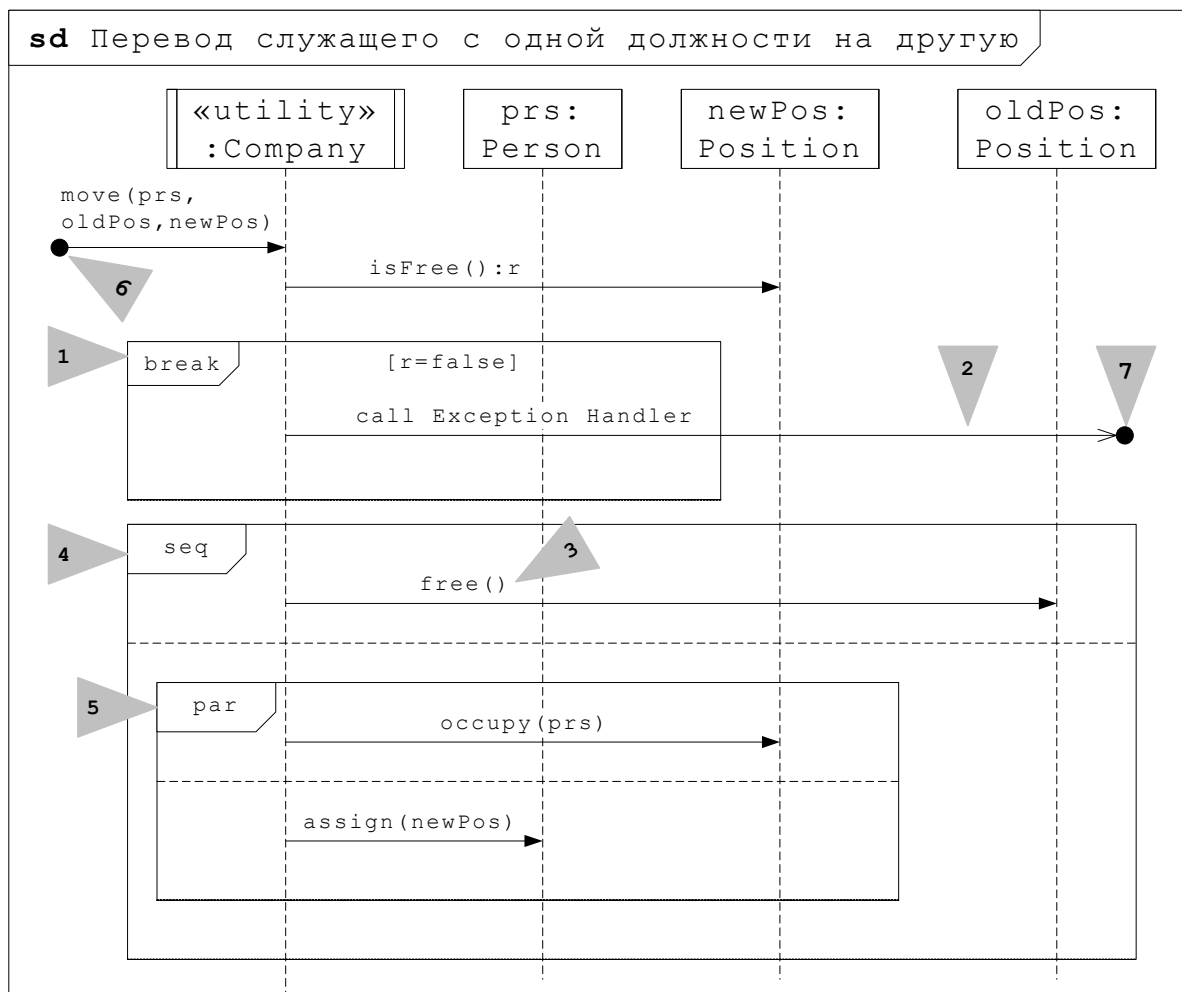


Рис. 4.50. Параллельное взаимодействие

## БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Новиков Ф.А, Иванов Д.Ю. Моделирование на UML. Теория, практика, видеокурс. — СПб, Профессиональная литература, Наука и Техника, 2010, 640 с.
2. Буч Г., Рамбо Д., Якобсон А. Язык UML. Руководство пользователя. Второе издание. — ДМК, 2006, 496 с.
3. Фаулер М. UML. Основы. 3-е издание. — Символ-Плюс, 2005, 192 с.
4. Буч Г., Якобсон А., Рамбо Д. UML. 2-е издание Классика CS. — СПб., Питер, 2005, 736 с.
5. Буч Г., Якобсон А., Рамбо Д. Унифицированный процесс разработки программного обеспечения. Питер, 2002, 496 с.
6. Крэг Л. Применение UML 2.0 и шаблонов проектирования, 3-е издание. Вильямс, 2007, 736 с.
7. Рамбо Д., Блаха М. UML 2.0. Объектно-ориентированное моделирование и разработка. Питер, 2007, 540 с.

Иванов Денис Юрьевич  
Новиков Федор Александрович

# УНИФИЦИРОВАННЫЙ ЯЗЫК МОДЕЛИРОВАНИЯ UML

Учебное пособие

Лицензия ЛР № 020593 от 07.08.97

Налоговая льгота – Общероссийский классификатор продукции  
ОК 005-93, т. 2; 95 3005 – учебная литература

---

Подписано в печать 20.05.2008. Формат 60×84/16 Печать цифровая

Усл. печ. л.    . Уч.-изд. л.    . Тираж    . Заказ

---

Отпечатано с готового оригинал-макета, предоставленного автором  
в цифровом типографском центре Издательства Политехнического  
университета:

195251, Санкт-Петербург, Политехническая ул., 29.

Тел. (812) 540-40-14

Тел./факс: (812) 927-57-76