Глава 4

Спецификация требований

Требования необходимо *специфицировать* (т.е. задать) графически или каким-либо иным формальным способом. Всесторонняя спецификация системы может потребовать использования многих типов моделей. Язык UML изобилует интегрированными методами моделирования, способными помочь бизнес-аналитику справиться с этой задачей. Спецификация — подобно процессу разработки ПО в целом — итеративный процесс с пошаговым наращиванием уровня детализации моделей. Немаловажную роль в успешном моделировании играет использование CASE-средств.

В результате спецификации требований вырабатываются три категории моделей: модели состояний, модели поведения и модели изменения состояния. Для каждой из категорий существует несколько методов работы с ними. В этой главе объясняются и иллюстрируются на примерах все основные методы моделирования языка UML.

Несмотря на то, что мы начнем изучение с моделей состояния, затем перейдем к моделям поведения, а затем — к моделям изменения состояний, это не отражает реальной последовательности, в которой проводится моделирование. Многие модели разрабатываются в параллель и служат источником взаимного развития. Это в особенности справедливо в отношении двух основополагающих типов моделей — моделей классов и моделей прецедентов.

4.1. Принципы спецификации требований

Спецификация требований связана с доскональным *моделированием* требований заказчиков, определенных в процессе *установления требований*. При этом рассматриваются только услуги, которые стремятся получить от системы заказчики (*формулировки сервисов*) (разд.3.1). На этапе спецификации требований *формулировки ограничений* не подлежат дальнейшей проработке, хотя и могут претерпеть изменения как результат обычного цикла *итерации*.

В качестве входной информации процесса спецификации требований выступают неформальные требования заказчиков, а результатом этого процесса являются модели спецификации проектных конструкций. Эти модели (разд. 2.2) дают более формальное определение различных сторон (представлений) системы. Обычно требования пользователей в процессе спецификации подразделяются на две основные категории: *функциональные требования* и *требования к данным*.

В качестве результата этапа спецификации выступает расширенный (“детально проработанный”) документ описания требований (разд. 3.6). Новый документ часто называют *документом спецификации требований* (или просто “спецификацией” на жаргоне разработчиков). Структура исходного документа не изменяется, однако содержание значительно расширяется за счет глав, которые определяют требования заказчиков. Постепенно для целей проектирования и реализации документ спецификации требований заменяет документ описания требований (на практике, расширенный документ может по-прежнему называться документом описания требований).

Модели спецификаций можно разделить на три группы.

1. Модели состояний.
2. Модели поведения.
3. Модели изменения состояний.

Модели *состояний* “детализируют” требования к данным. Модели *поведения* обеспечивают детализированные спецификации для функциональных требований. Модели *изменения состояний* охватывают два вида требований. Они призваны объяснить, каким образом действие функций приводит к изменению данных.

Модели представляются в виде диаграмм на *языке визуального моделирования* (*Visual Modeling Language*) — в нашем случае это язык UML. Обычно диаграмма служит целям моделирования одной из сторон системы — состояний, поведения или изменения состояний. Заметное исключение составляет диаграмма классов, которая определяет все три аспекта — состояние и поведение объектов, и, косвенно, изменения состояний объектов.

Каждая диаграмма дает *представление* об определенной стороне системы. Взятые вместе диаграммы дают возможность разработчикам и пользователям взглянуть на предлагаемое решение с разных точек зрения, выделяя одни его стороны и игнорируя другие. Ни одна из диаграмм в отдельности не дает полного определения системы. Систему можно понять только через взаимосвязанный набор диаграмм.

Аналогично случаю интерпретации завершенных моделей конструирование диаграмм — это не последовательный процесс построения одной диаграммы за другой. Диаграммы разрабатываются в параллель, и в результате каждой последующей итерации к ним добавляются новые детали. В то время, как разработчики должны следовать строго определенному процессу разработки, решение о том, какая из моделей должна играть роль “движущей силы” разработки, в значительной мере зависит от личных предпочтений аналитика. Обычно диаграммы прецедентов и модели классов — как наиболее важные типы моделей — конструируются параллельно, взаимно “обогащая” друг друга идеями.

С каждой новой итерацией разработки глубина и степень детализации спецификации возрастает. Многие более глубокие свойства объектов модели выражаются скорее в текстовом, нежели графическом виде. Некоторые свойства определяют замысел объекта модели, а не результат анализа. Некоторые другие свойства могут отражать особенности CASE-средств.

4.2. Спецификации состояний

Состояние объекта определяется значениями его атрибутов и ассоциаций. Например, объект `BankAccount` (Банковский счет) может находиться в состоянии “превышение кредитного лимита”, если значение атрибута `balance` (баланс) отрица-

тельно. Поскольку состояния объекта определяются структурам данных, модели структур данных называются *спецификациями состояний*.

Спецификация состояний дает *статический взгляд* на систему (поэтому моделирование состояний часто называют статическим моделированием). Здесь основной задачей является определение *классов* проблемной области, их *атрибутов* и *отношений* с другими классами. Вначале операции классов обычно не рассматриваются. Они выводятся из моделей *спецификации поведения*.

В типичной ситуации сначала определяются *классы-сущности*, т.е. классы, которые определяют проблемную область и характеризуются *постоянным* присутствием в базе данных системы. Подобные классы иногда называются "*бизнес-классами*". Классы, которые обслуживают системные события (*управляющие классы*) и классы, которые представляют GUI-интерфейс (*классы представления* или *пограничные классы*) не устанавливаются до тех пор, пока не станут известны поведенческие характеристики системы.

4.2.1. Моделирование классов

Модель классов — это краеугольный камень разработки объектно-ориентированной системы. Классы лежат в основании наблюдаемости свойств и поведения системы. К сожалению, классы всегда трудно поддаются определению, а свойства классов не всегда очевидны. Весьма маловероятно, чтобы два аналитика пришли к одному и тому же множеству классов и их свойств для одной и той же нетривиальной проблемной области. Хотя модели классов могут отличаться, конечный результат и степень удовлетворенности пользователя могут быть в равной мере достаточными (или в равной мере недостаточными).

Моделирование классов — это не детерминированный процесс. Не существует общего рецепта отыскания и определения наилучших классов. Этот процесс в значительной мере носит *итеративный* и *пошаговый наращиваемый* характер. К факторам, определяющим успешное проектирование классов, относится уровень квалификации и опыта аналитика, в частности, перечисленные ниже возможности.

1. Знания в области моделирования классов.
2. Понимание проблемной области.
3. Опыт в области аналогичных и успешных проектов.
4. Способность смотреть вперед и предвидеть последствия решений.
5. Готовность к пересмотру модели с целью устранения недостатков и т.д.

Последний момент связан с использованием CASE-средств. Широкомасштабное применение CASE-технологии может стать препятствием на пути разработки системы в технологически незрелых организациях (разд. 1.1.4.2). Однако использование CASE-средств для повышения личной продуктивности всегда оправдано.

4.2.1.1. Выявление классов

Два разных аналитика, как правило, не могут прийти к идентичным моделям классов для одной и той же проблемной области, и точно так же два разных аналитика не пользуются одним и тем же мыслительным процессом при выделении классов. Литература изобилует подходами, предлагаемыми для выявления классов. Аналитики могут поначалу даже следовать одному из этих подходов, однако последующие итерации,

как правило, обязательно приводят к использованию нешаблонных и в чем-то даже случайных механизмов.

Барами (Bahrami) [4] подробно изучил главные особенности четырех основных подходов к *выявлению классов* (*class discovery*). Ниже перечислены эти подходы.

1. Подход на основе использования именных групп.
2. Подход на основе использования общих шаблонов для классов.
3. Подход на основе использования прецедентов.
4. Подход CRC (class–responsibility–collaborators — класс–обязанности–“сотрудники”).

Барами [4] поставил в соответствие каждому из подходов опубликованные работы, однако — по нашему мнению— только последний подход обладает неоспоримым источником. Теперь мы обобщим эти подходы, а затем приведем примеры, в которых используется *комплексный подход* (*mixed approach*).

4.2.1.1.1. Подход на основе использования именных групп

Подход *на основе использования именных групп* (т.е. имен существительных в предложениях) предполагает, что аналитик читает формулировки документа описания требований в поисках именных групп. Каждое имя существительное рассматривается как *потенциальный класс* (*candidate class*). Затем список всех классов разделяется на следующие три группы.

1. Релевантные или подходящие классы.
2. Нечеткие или сомнительные классы.
3. Нерелевантные или неподходящие классы.

К *нерелевантным* (*irrelevant*) относятся классы, которые выходят за рамки проблемной области. Для них не удается дать формулировку их назначения. Опытные аналитики-практики вероятнее всего не включают неподходящие классы в первоначальный список потенциальных классов. Таким образом удается избежать формальных шагов по идентификации и исключению нерелевантных классов.

К *релевантным* (*relevant*) относятся классы, которые безусловно принадлежат проблемной области. Имена существительные, представляющие имена этих классов, часто встречаются в документе описания требований. Кроме того, значение и назначение этих классов можно обосновать на основе общих знаний о прикладной области, а также на основе изучения аналогичных систем, руководств, документов и патентованных программных пакетов.

К *нечетким* (*fuzzy*) относятся классы, которые нельзя уверенно и безоговорочно признать подходящими. Они составляют наибольшую проблему. Их необходимо проанализировать более глубоко, а затем либо включить в список релевантных классов, либо исключить из списка нерелевантных. Окончательное отнесение этих классов к той или другой группе, собственно, и проводит различие между хорошей и плохой моделью классов.

Подход на основе использования именных групп предполагает наличие полного и корректного документа описания требований. На практике это предположение редко соответствует действительности. Но даже если оно обоснованно, кропотливое изуче-

ние больших объемов текста не обязательно должно привести к получению исчерпывающего и точного результата.

4.2.1.1.2. Подход на основе использования общих шаблонов для классов

Подход *на основе использования общих шаблонов для классов* позволяет вывести потенциальные классы на основе теории родовой классификации объектов. *Теория классификации* касается разделения мира объектов на удобные группы, что позволяет более эффективно строить рассуждения о них.

Барами [4] приводит следующий перечень групп (шаблонов) для выявления потенциальных классов.

- *Понятийный* (или *концептуальный*) *класс* (*concept class*) представляет собой идею, которую разделяет или с которой согласна значительная общность людей. В отсутствие понятий люди не способны эффективно общаться или даже общаться на некоем удовлетворительном уровне. Например, `Reservation` (Резервирование) — это понятийный класс, относящийся к системе резервирования мест в авиакомпаниях.
- *Событийный класс* (*events class*). Событие — это нечто, что не требует времени применительно к нашей временной шкале. Например, `Arrival` (Прибытие) — это событийный класс, относящийся к системе резервирования мест в авиакомпаниях.
- *Организационный класс* (*organization class*). Организация — это любой вид целенаправленного объединения или совокупности сущностей. Например, `TravelAgency` (Бюро путешествий) — это класс, относящийся к системе резервирования мест в авиакомпаниях.
- *Класс "людей"* (*people class*). Под "людьми" здесь понимается скорее роль, которую человек играет в той или иной системе, а не физическое лицо. Например, `Passenger` (Пассажир) — это класс, относящийся к системе резервирования мест в авиакомпаниях.
- *Класс местоположений* (*places class*). Местоположение определяет физическое расположение объектов, связанных с информационной системой. Например, `TravelOffice` (Офис бюро путешествий) — подобный класс, относящийся к системе резервирования мест в авиакомпаниях.

Дж. Рамбау (J. Rumbaugh), А. Джекобсон (I. Jacobson) и Г. Буч (G. Booch) [76] предлагают другую схему классификации.

- *Физический класс* (*physical class*) (например, `Airplane` (Самолет)).
- *Бизнес-класс* (*business class*) (например, `Reservation`).
- *Логический класс* (*logical class*) (например, `FlightTimetable` (Расписание рейсов)).
- *Прикладной класс* (*application class*) (например, `ReservationTransaction` (Операция резервирования)).
- *Компьютерный класс* (*computer class*) (например, `Index` (Индекс)).
- *Поведенческий класс* (*behavioral class*) (например, `ReservationCancellation` (Отмена резервирования)).

Подход на основе использования общих шаблонов классов служит в качестве полезного руководства, но не определяет систематического процесса, посредством которого можно было бы выделить надежное и полное множество классов. Этот подход можно с успехом использовать для определения начального множества классов или для проверки того, должны ли некоторые классы (полученные другими способами) присутствовать в нашем множестве или, напротив, отсутствовать в нем. Однако, подход на основе использования общих шаблонов классов слишком слабо связан со специфическими требованиями пользователей, чтобы претендовать на исчерпывающее решение.

Особая опасность, связанная с подходом на основе использования общих шаблонов классов, заключается в неверном истолковании имен классов. Например, что означает `Arrival` (Прибытие)? Означает ли это прибытие на взлетно-посадочную полосу (время приземления), прибытие к терминалу (время высадки), прибытие в зал возврата багажа (время таможенного досмотра) и т.д.? Аналогично, слово `Reservation` (в данном случае резервация. *Прим. ред.*) в среде североамериканских индейцев имеет совершенно иное значение по сравнению с тем, что имелось в виду до сих пор.

4.2.1.1.3. Подход на основе использования прецедентов

Подходу *на основе использования прецедентов* придается особое значение в языке UML. Можно даже сказать, что этот подход рекомендуется использовать в рамках UML (если быть точным — то в рамках методологии RUP (Rational Unified Process)). Графическая модель прецедентов сопровождается неформальными описаниями, а также диаграммами последовательностей и кооперации для отдельных прецедентов (разд. 2.2 и далее в этой главе). Эти дополнительные описания и шаги определения диаграмм (и объектов) требуется выполнить для каждого прецедента. На основе этой информации можно прийти к обобщениям, необходимым для выявления потенциальных классов.

Подход, направляемый прецедентами, обладает особенностями, присущими подходу снизу-вверх. После того, как прецеденты становятся известны, а *представление* о системе *с точки зрения взаимодействия*, по меньшей мере, частично определено с помощью диаграмм последовательностей, объекты, используемые в этих диаграммах, приводят к выявлению классов.

В действительности этот подход в чем-то похож на подход, использующий именные группы. Их объединяет тот факт, что прецеденты специфицируют требования. Оба подхода направлены на изучение формулировок, изложенных в документе описания требований, чтобы выявить в итоге потенциальные классы. То, что эти формулировки излагаются в повествовательной форме или представлены графически, имеет второстепенное значение. В любом случае на этом этапе ЖЦ разработки ПО большую часть прецедентов можно описать только в текстовой форме без диаграмм взаимодействия.

Подход, основанный на прецедентах, страдает теми же недостатками, что и подход, использующий именные группы. Будучи по сути подходом снизу-вверх в смысле точности, он опирается на полноту и корректность моделей прецедентов. В результате, он даже может привести к нежелательному разбалансированию итеративного и наращиваемого процесса разработки ПО, при котором модели прецедентов должны быть завершены еще до построения моделей классов. В общем, каковы бы ни были цели и средства, это приводит к *функциональному подходу* (*function-driven approach*) (сторонники объектно-ориентированного подхода предпочитают называть его *проблемно-ориентированным* (*problem-driven*)).

4.2.1.1.4. Подход CRC

Подход *CRC* (**C**lass–**R**esponsibility–**C**ollaborators — класс–ответственность–"сотрудники") представляет собой нечто большее, чем метод выявления классов, — это привлекательный способ интерпретации и изучения объектов (а также и обучения объектному подходу). Наибольшую известность подход CRC получил благодаря работам Ребекки Вирфс-Брок (Rebecca Wirfs-Brock) и ее коллег Б. Вилкерсон (B. Wilkerson) и Л. Винер (L. Wiener) [94], [93].

Подход CRC включает в себя сеансы "мозгового штурма", проведение которых облегчается за счет использования специально подготовленных карточек. Карточки состоят из трех отделений: *имя класса* записывается в верхнем отделении, "*обязанности*" класса перечислены в левом отделении, а "*сотрудники*" перечислены в правом отделении. Обязанности — это услуги (операции), которые класс готов выполнить в интересах других классов. Для выполнения многих обязанностей необходимо участие (обслуживание) со стороны других классов. Такие классы перечисляются как "сотрудники".

Процесс CRC — это живой процесс, во время которого разработчики "играют в карты", они заполняют карточки именами классов и назначают им "обязанности" и "сотрудников" в ходе исполнения сценария обработки информации (например, сценария прецедента). В тех случаях, когда возникает потребность в некоей услуге, а существующие классы не покрывают ее, создается новый класс, которому назначаются соответствующие "обязанности" и "сотрудники". Если класс становится "слишком занятым", он разделяется на несколько меньших классов.

Подход CRC отличается от других подходов тем, что при его использовании выделение классов является результатом анализа сообщений, передаваемых между объектами для выполнения задач обработки информации. Акцент делается на унифицированном методе распределения "интеллекта" в системе, и некоторые классы могут быть скорее получены, исходя из подобной технической потребности, чем выявлены в качестве "бизнес-объектов" как таковых. В этом смысле метод CRC может быть более приемлемым для проверки правильности выбора классов уже выявленных с помощью других методов. Подход CRC также полезен при установлении свойств классов (которые логически следуют из "обязанностей" и типов "сотрудников" класса).

4.2.1.1.5. Комплексный подход

На практике процесс выявления классов в разное время, вероятнее всего, следует разным подходам. Зачастую, он включает элементы всех четырех подходов, рассмотренных выше. Немаловажными факторами при этом выступают общая эрудиция эксперта, его опыт и интуиция. Процесс в чистом виде не следует ни методу сверху-вниз, ни методу снизу-вверх — он все время идет "из средины". Подобный подход к выявлению классов мы называем *комплексным подходом* (*mixed approach*).

Вот один из возможных сценариев. Начальное множество классов можно сформировать на основе общих знаний и опыта эксперта. При этом дополнительно можно руководствоваться подходом на основе общих шаблонов классов. Остальные классы можно добавить, основываясь на анализе обобщенного описания проблемной области с использованием подхода на основе именных групп. Если в распоряжении аналитика имеются прецеденты, можно воспользоваться подходом, направляемым прецедентами, чтобы добавить новые и проверить состоятельность существующих классов. Наконец, подход CRC позволяет применить "мозговой штурм" для проверки правильности выбора выделенного к этому времени множества классов.

4.2.1.1.6. Некоторые правила выявления классов

Ниже приведен далеко не полный перечень *руководящих принципов* или *правил*, которым должен следовать аналитик при выборе потенциальных классов. Вновь напоминаем о том, что здесь мы имеем дело только с *классами-сущностями.*

1. Для каждого класса должно быть ясно *сформулировано* его *назначение* в системе.
2. Каждый класс — это шаблон описания *множества объектов. Единичные* классы, для которых можно представить существование только одного объекта, весьма маловероятны среди "бизнес-объектов". Подобные классы обычно составляют в приложении "общее знание" и как правило жестко запрограммированы в программах приложения. Например, если система спроектирована для единственной организации, существование класса `Organization` (Организация) может быть не оправданно.
3. Каждый класс (т.е. класс-сущность) должен содержать *набор атрибутов.* Хорошим приемом является установление идентифицирующих атрибутов (*ключей*), чтобы помочь нам судить о *мощности* (*cardinality*) класса (т.е. ожидаемом количестве объектов данного класса в базе данных). Следует, однако, помнить о том, что класс не обязательно должен обладать пользовательским ключом. Объекты классов идентифицируются с помощью *идентификаторов объектов* (*OID*) (разд. 2.1.1.3).
4. Каждый класс должен отличаться от *атрибута.* Представляется ли понятие классом или атрибутом зависит от области приложения. Цвет автомобиля обычно воспринимается как атрибут класса `Car` (Автомобиль). Однако на фабрике по производству красок `Color` (Цвет) — это определенно класс со своими собственными атрибутами (яркостью, насыщенностью, прозрачностью и т.д.).
5. Каждый класс содержит *набор операций.* Однако на данном этапе мы не касаемся вопросов идентификации операций. Операции, входящие в *интерфейс* класса (сервисы, предоставляемые классом системе), являются логическим следствием формулировки назначения класса (пункт 1).

4.2.1.1.7. Примеры выявления классов

Давайте проанализируем требования с целью выделения потенциальных классов. В первом утверждении подходящими классами являются классы `Degree` (Степень) и `Course` (Курс). Эти два класса удовлетворяют пяти правилам, перечисленным ранее. Мы пока не уверены, должен ли и каким образом класс `Course` быть сужен до классов `CompulsoryCourse` (Обязательный курс) и `ElectiveCourse` (Выборочный курс). Например, ясно, что курс является обязательным или выборочным в зависимости от степени. Возможно, что различие между обязательными и выборочными курсами может быть зафиксировано с помощью ассоциации или даже атрибута класса. Таким образом, `CompulsoryCourse` и `ElectiveCourse` рассматриваются как нечеткие классы.

Второе утверждение идентифицирует только атрибуты класса `Course`, а именно `course_level` (уровень курса) и `credit_point_value` (количество условных очков). Третье утверждение характеризует ассоциацию между классами `Course` и `Degree`. Четвертая формулировка вводит атрибут `min_total_credit_points` (минимальное общее количество условных очков) в качестве атрибута класса `Degree`.

Пример 4.1. Запись на университетские курсы

Рассмотрите следующие требования к системе *Запись на университетские курсы* и выделите потенциальные классы.

1. Для получения каждой университетской степени существует несколько обязательных и несколько выборочных курсов.
2. Каждому курсу соответствует заданный уровень и значение условных очков (Credit-point — условное очко, начисляемое за прослушивание какого-либо курса (за один курс может быть начислено несколько очков); студент обязан набрать на данном году обучения такое число курсов, чтобы число очков за них было не ниже определенного значения. *Прим. ред.*).
3. Курс может быть составной частью системы получения произвольного количества степеней.
4. Каждая степень определяет минимальное общее значение условных очков, требуемое для получения степени (например, для степени бакалавра (вычислительные и информационные системы) требуется 68 очков, включая обязательные курсы).
5. Студенты могут составлять из дисциплин программы обучения, приспособленные к их индивидуальным нуждам и обеспечивающие им получение степени, на которую они претендуют.

Последнее утверждение позволяет нам выделить три новых класса: `Student` (Студент), `CourseOffering` (Предлагаемый курс) и `StudyProgram` (Программа обучения). Первые два, безусловно, являются релевантными классами, а вот `StudyProgram` можно превратить в ассоциацию между классами `Student` и `CourseOffering`. Поэтому `StudyProgram` классифицируется как нечеткий класс. Наши рассуждения отражены в табл. 4.1.

Пример 4.2. Магазин видеопроката

Рассмотрите следующие требования к системе *Магазин видеопроката* и выделите потенциальные классы.

1. Магазин видеопроката держит в запасе огромную видеотеку современных популярных видеофильмов. Конкретный фильм может храниться на видеокассетах или дисках.
2. Видеокассеты могут быть записаны в формате “Beta” или “VHS”. Видеодиски записаны в формате DVD.
3. Для каждого фильма установлен конкретный период проката (исчисляемый в днях) с соответствующей платой за прокат за этот период.
4. Видеомагазин должен быть в состоянии немедленно дать ответ на любой запрос по наличию фильмов в запасе, а также количеству кассет или дисков (текущие условия по каждой ленте и диску должны быть известны и зафиксированы).

Таблица 4.1. Потенциальные классы (Запись на университетские курсы)

Релевантные классы	*Нечеткие классы*
`Course` (Курс)	`CompulsoryCourse` (Обязательный курс)
`Degree` (Степень)	и `ElectiveCourse` (Выборочный курс)
`Student` (Студент)	`StudyProgram` (Программа обучения)
`CourseOffering` (Предлагаемый курс)	

Первое утверждение содержит несколько имен существительных, но только некоторые из них можно превратить в потенциальные классы. “Видеомагазин” — это не класс, поскольку он составляет всю систему (в базе данных может быть только один объект этого класса — так называемый *единичный класс*). Аналогично, понятия “запаса” и “видеотеки” слишком общие, чтобы рассматривать их в качестве классов, по крайней мере на этом этапе. Релевантными классами представляются `MovieTitle` (Название фильма), `VideoTape` (Видеокассета) и `VideoDisk` (Видеодиск).

Второе утверждение вводит дополнительную *специализацию* видеопроката как проката видеокассет и дисков. Мы можем предложить три новых класса: `BetaTape`, `VHSTape` и `DVDDisk`. Однако поскольку мы имеем дело только с одним типом видеодисков, в конечной модели можно не оставлять оба класса: `VideoDisk` и `DVDDisk`. Какой из двух классов оставить, зависит от конечного уровня специализации применительно к видеокассетам и дискам. Заметим, что мы пока не уверены в том, будут ли классы `BetaTape` и `VHSTape` обладать какими-либо отличительными атрибутами (за исключением того факта, что один из них принадлежит типу Beta, а другой — VHS).

Третье утверждение говорит о том, что каждое наименование фильма отличается условиями проката, связанными с ним. Однако не ясно, что понимается под “фильмом” — наименование фильма или же носитель фильма (кассета или диск)? Нам необходимо прояснить это требование с заказчиками. Тем временем, мы можем предпочесть объявить класс `RentalConditions` (Условия проката) как нечеткий, вместо того, чтобы хранить информацию о периоде проката и плате за прокат в классе для наименования фильма или для носителя фильма.

Последняя формулировка убеждает нас в том, что классы `BetaTape`, `VHSTape` и `DVDDisk` (или `VideoDisk`) — релевантные. Нам требуется хранить информацию о текущих условиях для каждой кассеты и диска. Однако атрибуты наподобие `video_condition` (условия видеопроката) или `number_currently_available` (количество, имеющееся в наличии) можно в общем объявить в абстрактном классе верхнего уровня (назовем его `VideoMedium` (Видеоноситель)), после чего они могут быть унаследованы конкретным подклассом (таким как `VHSTape`). Итоги наших рассуждений отражены в табл. 4.2.

Таблица 4.2. Потенциальные классы (Магазин видеопроката)

Релевантные классы	*Нечеткие классы*
`MovieTitle` (Название фильма)	`RentalConditions` (Условия проката)
`VideoMedium` (Видеоноситель)	
`VideoTape` (Видеокассета)	
`VideoDisk` (или `DVDDisk`)	
`BetaTape`	
`VHSTape`	

Первое утверждение содержит понятия “клиент”, “контракт” и “товар”. Наша общая эрудиция и опыт подсказывают нам, что это типичные классы. Однако понятия контракта и товара не входят в рамки системы *Управление контактами с клиентами* и должны быть отброшены.

Пример 4.3. Управление контактами с клиентами

Рассмотрите следующие требования к системе *Управление контактами с клиентами* и выделите потенциальные классы.

1. Система поддерживает функции “постоянного контакта” с нашей наличной и потенциальной клиентской базой, так чтобы откликаться на ее нужды и заполучать новые контракты на приобретение наших товаров.
2. Система хранит имена, номера телефонов, обычные почтовые и курьерские адреса и т.д. организаций и наших контактных лиц в этих организациях.
3. Система позволяет нашим сотрудникам планировать задания и мероприятия, которые необходимо провести в отношении наших контактных лиц. Сотрудники планируют задания и мероприятия для других сотрудников или для себя.
4. Задание — это группа мероприятий, которые осуществляются для достижения определенного результата. Результатом может быть превращение потенциального клиента в клиента, организация доставки товара или решение проблемы клиента. К обычным типам мероприятий относятся телефонный звонок, визит, отправка факса, устройство обучения и т.д.

`Customer` (Клиент) — это релевантный класс, однако мы можем предпочесть назвать его `Contact` (Контакт), имея в виду, что не все контакты осуществляются с нашими настоящими клиентами. Различие между наличным и потенциальным клиентом может оправдать или не оправдать введение таких классов как `CurrentCustomer` (Наличный клиент) и `ProspectiveCustomer` (Потенциальный клиент). Поскольку уверенности у нас нет, мы объявляем эти классы нечеткими.

Второе утверждение проливает новый свет на приведенные выше рассуждения. Нам необходимо провести различие между контактной организацией и контактным лицом. Имя `Customer` не кажется слишком удобным для названия класса. Помимо прочего, понятие `Customer` подразумевает только наличного клиента и, кроме того, это может заключать в себе как понятие организации, так и контактного лица. Наше новое предложение состоит в том, чтобы назвать классы следующим образом: `Organization` (Организация), `Contact` (Контакт) (подразумевается контактное лицо), `CurrentOrg` (Наличная организация) (т.е. организация, являющаяся нашим наличным клиентом) и `ProspectiveOrg` (Потенциальная организация) (т.е. организация, являющаяся нашим потенциальным клиентом).

Во втором утверждении упоминается несколько атрибутов классов. Однако обычный и курьерский почтовые адреса представляют собой составные атрибуты и применимы к обоим классам: `Organization` и `Contact`. Поэтому `PostalAddress` и `CourierAddress` — законные нечеткие классы.

Таблица 4.3. Потенциальные классы (Управление контактами с клиентами)

Релевантные классы	*Нечеткие классы*
`Organization` (Организация)	`CurrentOrg`
`Contact` (Контакт)	`ProspectiveOrg`
`Employee` (Сотрудник)	`PostalAddress`
`Task` (Задание)	`CourierAddress`
`Event` (Мероприятие)	

В третьей формулировке вводится три релевантных класса `Employee` (Сотрудник), `Task` (Задание) и `Event` (Мероприятие). Это предложение объясняет суть плановой деятельности.

Последнее утверждение посвящено дальнейшему прояснению смысла и взаимоотношений между классами, однако здесь не вводится ни одного нового класса.

4.2.1.2. Спецификация классов

После того, как перечень потенциальных классов сформирован, необходима их дальнейшая спецификация: классы требуется включить в диаграмму классов и определить их свойства. Некоторые свойства можно ввести и отобразить внутри графических пиктограмм, представляющих классы на диаграмме классов. Многие другие свойства, включенные в спецификацию класса, имеют только текстовое представление. CASE-средства, как правило, обладают возможностями редактирования, позволяющими легко вводить или модифицировать подобную информацию посредством диалоговых окон, снабженных вкладками, или с помощью аналогичных способов.

Как объяснялось в начале этой главы, классы задаются на определенном уровне абстракции. Более развитые возможности моделирования языка UML здесь не используются — они рассматриваются в главе 5 и последующих главах.

4.2.1.2.1. Именование классов

Каждому классу необходимо присвоить *имя*. При работе с некоторыми CASE-средствами помимо имени классу можно также присвоить *код*, возможно, отличный от имени. Код может удовлетворять соглашениям по именованию, требуемым целевым языком программирования или СУБД. Для генерации программного кода из проектной модели используется именно код класса, а не имя.

Мимоходом, мы приняли определенное соглашение в отношении имен классов. Это соглашение заключается в том, что имя класса *начинается с заглавной буквы.* Для составных имен в *качестве первой буквы* каждого слова также *используется заглавная буква* (вместо того, чтобы отделять слова знаком подчеркивания или дефисом). Это всего лишь рекомендуемое соглашение, однако среди разработчиков оно нашло довольно много приверженцев. (В разд. 6.1.3.2 к имени каждого класса рекомендуется добавить однобуквенный префикс для обозначения программного слоя, к которому принадлежит класс).

Имя класса должно быть именем существительным в единственном числе (например, `Course`) либо, при возможности, сочетанием прилагательного и существительного в единственном числе (например, `CompulsoryCourse`). Ясно, что класс представляет собой шаблон для множества объектов и использование в качестве имен существительных во множественном числе не несет никакой дополнительной информации. Иногда существительное в единственном числе не отражает истинного предназначения класса. В подобных ситуациях допустимо использование существительных во множественном числе (например, `RentalConditions` (Условия проката) в примере 4.2).

Имя класса должно быть *осмысленным.* Оно должно отражать истинную природу класса. Оно должно заимствоваться из *словаря пользователей* (а не жаргона разработчиков).

Лучше использовать более длинные имена, чем скрывать их смысл за шифром. Можно с уверенностью сказать, что имена длиной более *тридцати символов* слишком громоздки (а некоторые программные среды их просто не воспринимают, если CASE-

средства работают с именами классов, а не кодами классов). Возможно также использование более длинных описательных имен в дополнение к именам и кодам классов.

4.2.1.2.2. Выявление и спецификация атрибутов классов

Графическая пиктограмма, представляющая класс, состоит из трех отделений (имя класса, атрибуты, операции) (разд. 2.1.2). Спецификация атрибутов классов принадлежит к *спецификации состояний* и рассматривается в этом разделе. Спецификация операций рассматривается позже в этой главе в разделе, посвященном *спецификации поведения* (разд. 4.3).

Выделение атрибутов осуществляется параллельно с выделением классов. Идентификация атрибутов своего рода "побочный эффект" установления классов. Это не означает, что выявление атрибутов — простая задача. Напротив, это процесс, требующий значительных усилий и многократных итераций.

Исходные модели спецификации определяют только атрибуты, являющиеся существенными для понимания *состояний*, в которых могут находиться объекты класса. Остальные атрибуты можно до поры до времени игнорировать (однако аналитик должен быть уверен в том, что установленная, но проигнорированная на определенном этапе информация не будет по ошибке утеряна и будет зафиксирована впоследствии). Маловероятно, чтобы все атрибуты класса были приведены в документе описания требований, однако важно не включать в спецификацию те атрибуты, которые не вытекают из требований. В последующих итерациях можно добавить больше атрибутов.

Для имен атрибутов мы рекомендуем придерживаться простого соглашения: в именах атрибутов использовать только *строчные буквы*, а слова в составных именах отделять *подчеркиванием*.

4.2.1.2.3. Примеры спецификации классов

Пример 4.4. Запись на университетские курсы

Обратитесь к примеру 4.1 и рассмотрите следующие дополнительные требования, изложенные в документе описания требований.

1. Выбор студентами учебных курсов может быть ограничен из-за конфликтов расписания, а также за счет ограничения на количество студентов, которое может быть набрано на текущий предлгаемый курс.
2. Предлагаемая студентом программа обучения вводится в интерактивную систему записи на курсы. Система проверяет программу на непротиворечивость и сообщает о любых проблемах. Проблемы требуется решать при помощи научного руководителя. Окончательная программа является предметом научного согласования со стороны представителя заведующего кафедрой, а затем направляется секретарю учебного заведения.

В первом утверждении упоминаются конфликты расписания, однако мы не знаем достоверно, как следует моделировать эту проблему. Возможно, что речь здесь идет о прецеденте, который процедурно определяет конфликты расписания. Вторую часть этой же формулировки можно смоделировать за счет ведения атрибута `enrolment_quota` (квота набора) в класс `CourseOffering`. Теперь также ясно, что класс должен обладать атрибутами `year` (год) и `semester` (семестр).

Вторая формулировка укрепляет нас во мнении о необходимости введения класса `StudyProgram`. Можно видеть, что класс `StudyProgram` сочетает в себе ряд дисцип-

лин, предлагаемых к изучению в текущий момент. Поэтому класс `StudyProgram` также должен обладать атрибутами `year` и `semester`.

Ближайшее рассмотрение нечетких классов `CompulsoryCourse` и `ElectiveCourse` приводит нас к выводу, что учебный курс является обязательным или выборочным *относительно* определенной ученой степени. Один и тот же курс может быть обязательным по отношению к одной степени, выборочным, что касается другой, и вообще не допустимым применительно к некоторым другим степеням. Раз так, то `CompulsoryCourse` и `ElectiveCourse` не являются классами в полном смысле слова. (Заметим, что здесь мы не затрагиваем область моделирования классов с использованием обобщения (разд. 2.1.5) — моделирование обобщения рассматривается в разделе 4.2.4.)

На рис. 4.1 представлена модель классов, соответствующая проведенным нами рассуждениям. Кроме того, на рисунке используются символы (*стереотипы* (*stereotype*)) `<<PK>>` и `<<CK>>` для обозначения первичных ключей и потенциальных ключей, соответственно (разд. 8.4.1.2). Это уникальные идентификаторы объектов для рассмотренных классов. Здесь же заданы типы данных для атрибутов.

Классы `StudyProgram` и `CourseOffering` не имеют пока идентифицирующих атрибутов. Они будут введены в эти классы после установления ассоциативных связей между классами (разд. 2.1.3 и 4.2.2).

Пример 4.5. Магазин видеопроката

Обратитесь к примеру 4.2. Предположим, что мы прояснили требование 3 об условиях проката. Ниже приведен перечень дополнительных требований.

1. Плата за прокат отличается в зависимости от видеоносителя: кассета или диск (однако для двух типов кассет — Beta и VHS — она одинакова).
2. Хотя магазин держит в запасе видеодиски только одного формата — DVD, пользователи желали бы расширить в будущем систему проката и на другие форматы дисков.
3. Работники видеомагазина стремятся запомнить коды наиболее популярных лент. Зачастую при идентификации фильма они используют именно код фильма, а не его название. Это полезная практика, поскольку фильм с одним названием мог выпускаться разными режиссерами.

Первое утверждение разъясняет, что условия проката отличаются для кассеты (`VideoTape`) и диска (`VideoDisk`), содержащих один и тот же фильм. Поэтому имеет смысл ввести отдельный класс `RentalConditions` (который должен ассоциироваться с классами `VideoTape` и `VideoDisk`).

Из формулировки второго требования вытекает необходимость сосуществования обоих классов: `VideoDisk` и `DVDDisk`. Иерархия обобщения с корнем `VideoMedium` теперь становится довольно очевидной, однако мы откладываем обсуждение отношения обобщения до раздела 4.2.4.

На основании анализа третьего предложения мы вводим в класс `MovieTitle` атрибуты кода фильма `movie_code` (в качестве ключевого атрибута) и режиссера — `director`. Остальные атрибуты были рассмотрены в примере 4.2.

На рис. 4.2 показана модель классов для приложения *Магазин видеопроката*, построенная в результате обсуждения требований примеров 4.2 и 4.3. Атрибут `VideoMedium.number_currently_available` является *атрибутом с областью действия-класс* (*статическим атрибутом*) (разд. 2.1.6). Атрибут `MovieTitle.is_in_stock`, отражающий наличие в запасе фильма с данным названием, является *производным*

(*derived*) атрибутом, т.е. он может быть вычислен на основе имеющегося в наличии текущего количества фильмов — VideoMedium.number_currently_available.

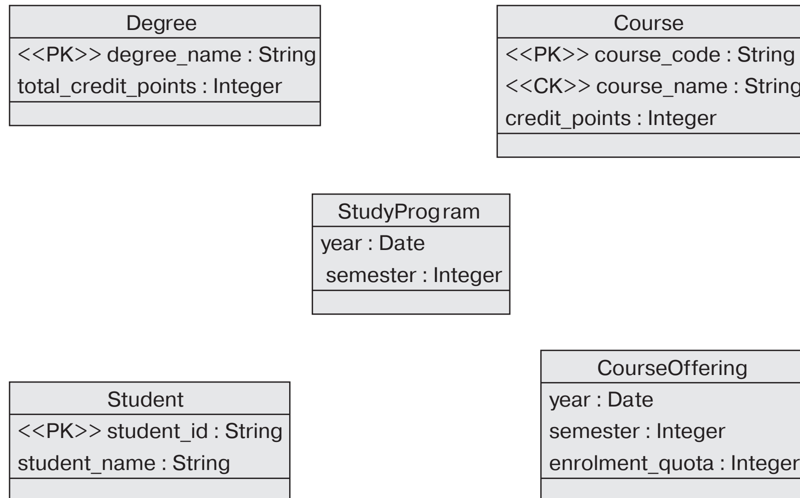

Рис. 4.1. Спецификация классов (Запись на университетские курсы)

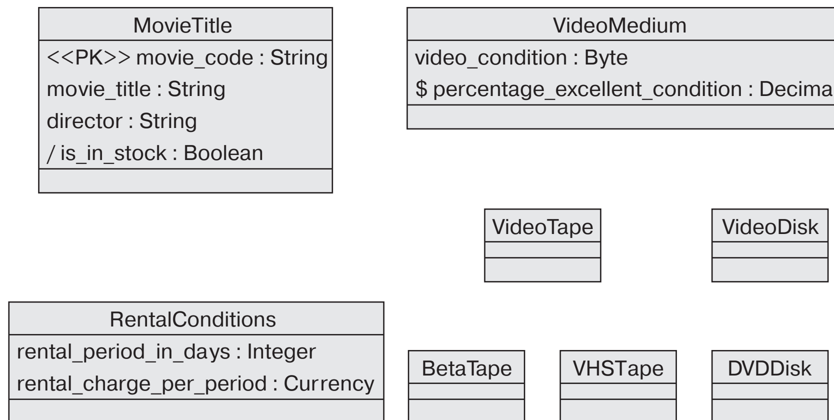

Рис. 4.2. Спецификация классов (Магазин видеопроката)

Анализ первого утверждения говорит нам о том, что понятие наличного клиента выводится на основе ассоциации между классами Organization и Contract. Эта ассоциативная связь может быть довольно динамичной. Следовательно, нужда в классах CurrentOrg и ProspectiveOrg отпадает. Более того, наша система не касается управления контрактами и не отвечает за поддержку класса Contract. Лучший вариант, который просматривается в данном случае — смоделировать решение с помощью производного атрибута Organization.is_current, который обозначает наличную организацию и при необходимости может изменяться подсистемой управления контрактами.

Второе утверждение наводит на мысль о необходимости введения двух классов для адресов: PostalAddress и CourierAddress.

Оставшиеся формулировки требований дают дополнительную информацию о содержании атрибутов классов. Кроме того, здесь можно высказать несколько соображений, связанных с ассоциативными отношениями и ограничениями целостности (они будут рассмотрены позже).

Пример 4.6. Управление контактами с клиентами

Обратитесь к примеру 4.3 и рассмотрите следующую дополнительную информацию.

1. Клиент рассматривается как наличный, если существует контракт с этим клиентом на поставку наших товаров или услуг. Однако, функции управления контрактами выходят за рамки нашей системы.
2. Система позволяет вырабатывать различные отчеты по нашим контактам на основе почтового и курьерского адресов (например, находить всех клиентов по почтовому коду).
3. Дата и время создания задания фиксируются. Можно также сохранить значение “дохода”, ожидаемого от осуществления задания.
4. Мероприятия для сотрудника отображаются на экране его компьютера в виде страницы календаря (один день на страницу). Приоритет каждого мероприятия (низкий, средний, высокий) визуально выделяется на экране.
5. Не со всеми мероприятиями связано понятие “срок исполнения” — некоторые из них являются “бессрочными” (они могут выполняться в любое время в течение дня, на который они запланированы).
6. Время создания мероприятия не может изменяться, а срок исполнения — может.
7. По завершении мероприятия дата и время его завершения фиксируются.
8. Система также хранит отличительные признаки для *сотрудников*, которые *создают* задания и мероприятия, которым *запланировано* осуществление мероприятия (“поручено сотруднику”) и которые *завершили* мероприятие.

Спецификация классов для приложения *Управление контактами с клиентами* представлена на рис. 4.3. Как видно из рисунка, отношения между классами в модели отсутствуют. Поэтому, к примеру, восьмое утверждение, которое связывает сотрудников с заданиями и мероприятиями, не нашло отражения в модели.

Первая формулировка позволяет нам установить несколько атрибутов класса Campaign. Класс Campaign содержит следующие атрибуты campaign_code (код кампании) (первичный ключ), campaign_title (название кампании), date_start (дата начала) и date_closed (дата закрытия).

Последнее предложение утверждения 1 касается призов кампании. При его ближайшем рассмотрении можно прийти к выводу о том, что Prize (Приз) — самостоятельный класс: приз разыгрывается, ему присуще такое свойство как победитель, и он должен обладать другими явно не выраженными свойствами, такими как описание, ценность и место в ряду других призов кампании.

Мы вводим класс Prize в модель классов вместе с атрибутами, о которых говорилось выше: prize_descr, prize_value и prize_ranking. Мы замечаем, что дата розыгрыша призов совпадает для всех призов кампании. Мы вводим атрибут даты розыгрыша date_drawn в класс Campaign. Приз выигрывает благотворитель. Мы можем зафиксировать этот факт позже в связи с введением ассоциации классов Prize и Supporter.

Формулировка 2 гласит, что у каждого билета есть номер, однако этот номер не уникален среди всех билетов (он уникален только в рамках кампании). Мы вводим ат-

рибут ticket_number, однако, не придаем ему статус первичного ключа для класса CampaignTicket. Два других атрибута CampaignTicket представляют цену билета (ticket_value) и статус билета (ticket_status). Общее количество билетов и количество проданных билетов — атрибуты класса Campaign (num_tickets и num_tickets_sold).

Пример 4.7. Телемаркетинг

Обратитесь к разделу 2.3.4 (Постановка задачи 4) и к разделу 3.5, в котором представлены бизнес-модель требований для приложения *Телемаркетинг*. Рассмотрите, в частности, диаграмму бизнес-классов на рис. 3.6 (разд. 3.5.3.1). Примите к сведению следующую дополнительную информацию.

1. Каждая кампания имеет свое название, которое, как правило, используется при ее упоминании. Кроме того, кампания обладает уникальным внутренним кодом для внутренних ссылок. Каждая кампания проводится в течение заданного периода времени. Вскоре после закрытия кампании проводится розыгрыш призов и объявляются владельцы выигрышных билетов.
2. Все билеты пронумерованы. В рамках кампании все билеты обладают уникальными номерами. Общее количество билетов, выпущенных для кампании, количество билетов, проданных до настоящего времени, и текущее состояние каждого билета известны (например, “наличный”, “заказан”, “оплачен”, “выигрышный”).
3. Для установления производительности телемаркетеров благотворительного общества фиксируется продолжительность звонков и успешные результаты звонков (т.е. звонков, завершившихся заказом билетов).
4. В системе ведется обширная информационная база, касающаяся благотворителей. Помимо обычных деталей, относящихся к контактам (адрес, телефонный номер и т.д.) эта информация включает исторические подробности, такие как дата первого и последнего участия благотворителя в кампаниях вместе с общим количеством кампаний, в которых они участвовали. В системе также хранятся данные обо всех известных предпочтениях и ограничениях, связанных с благотворителем (например, таких как нежелательное время для звонков и обычно используемая им при покупке билетов кредитная карточка).
5. Телемаркетинговые звонки должны обрабатываться в соответствии с определенными приоритетами. Звонки, на которые не получен ответ или зафиксирован ответ автоответчика, требуется перепланировать, чтобы попробовать дозвониться позже. Важно изменять время повторных попыток дозвониться.
6. Можно пытаться дозвониться снова и снова до тех пор, пока не будет исчерпан лимит попыток. Этот лимит может отличаться для различных типов звонков. Например, лимит для обычного “звонка-приглашения” может отличаться от лимита для “звонка-напоминания” благотворителю о невыполненном платеже.
7. Возможные результаты звонков классифицируются, чтобы облегчить ввод данных в систему телемаркетерами. Типичными видами результата являются: “успех” (т.е. билеты заказаны), “неудача”, “перезвонить позже”, “нет ответа”, “занято”, “автоответчик”, “факс”, “неверный номер”, “разъединение“.

Утверждение 3 обнаруживает несколько открытых вопросов. Какие структуры данных требуются нам для расчета продуктивности телемаркетеров? Чтобы ответить на этот вопрос нам требуется определить некоторые показатели, с помощью которых можно выразить продуктивность. Один из возможных вариантов состоит в том, чтобы подсчитывать среднее количество звонков в час и среднее количество успешных звонков в час. Затем можно вычислить показатель продуктивности, разделив количество ус-

пешных звонков на общее количество звонков. Теперь введем в класс `Telemarketer` соответствующие атрибуты: `average_per_hour` и `success_per_hour`.

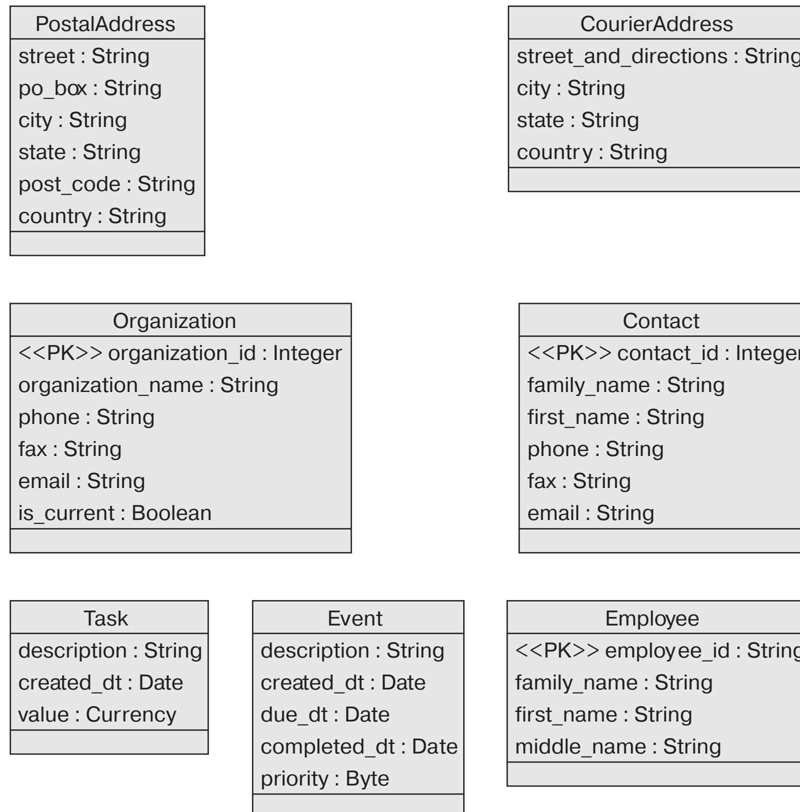

Рис. 4.3. Спецификация классов (Управление контактами с клиентами)

Для вычисления показателей продуктивности телемаркетера необходимо запоминать продолжительность каждого звонка. Хранилищем для этой информации служит класс `CallOutcome` (Результат звонка). В этот класс мы вводим атрибуты для начала и закрытия кампании: `start_time` и `end_time`. Мы предполагаем, что результат каждого звонка связан телемаркетером посредством ассоциации.

Результатом анализа утверждения 4 является включение ряда атрибутов в класс `Supporter`. Вот эти атрибуты: `supporter_id` (первичный ключ),. `supporter_name`, `phone_number`, `mailing_address`, `date_first`, `date_last`, `campaign_count`, `preferred_hours` и `credit_card_attributes`. Некоторые из этих атрибутов (`mailing_address`, `preferred_hours`) достаточно сложны для того, чтобы превратить их в дополнительные классы позже в процессе разработки. Пока же мы храним их как атрибуты.

Утверждение 5 касается класса `CallScheduled` (Запланированный звонок). Мы вводим в него атрибуты `phone_number`, `priority` и `attempt_number`. У нас нет полного понимания того, как поддерживать требование о том, что последующие звонки должны производится в разное время дня. Очевидно, что это должно быть возложено

на алгоритм планирования, однако нам потребуется поддержка структур данных. К счастью, некоторый свет на эту проблему проливает следующая формулировка.

Результатом анализа утверждения 6 является введение класса CallType (Тип звонка). Этот класс содержит следующие атрибуты: type_descr, call_attempt_limit, а также alternate_hours. Последний атрибут — это сложная структура данных, аналогичная атрибуту preferred_hours класса Supporter, которая в конце концов станет отдельным классом.

Последнее утверждение классифицирует результаты звонков. Это прямое указание на необходимость введения соответствующего класса — OutcomeType. Возможные типы результата могут храниться в атрибуте outcome_type_descr. Не ясно, какие еще атрибуты можно включить в OutcomeType, однако мы убеждены, что по мере изучения детализированных требований, эти атрибуты будут установлены. Одним из них может быть атрибут follow_up_action, назначение которого — хранить информацию о типичных последующих шагах применительно к каждому типу результата.

На рис. 4.4 представлена модель классов, завершающая приведенные выше рассуждения. Ассоциативные отношения, уже установленные в модели бизнес-классов (рис. 3.6), сохранены. Новые ассоциации не введены.

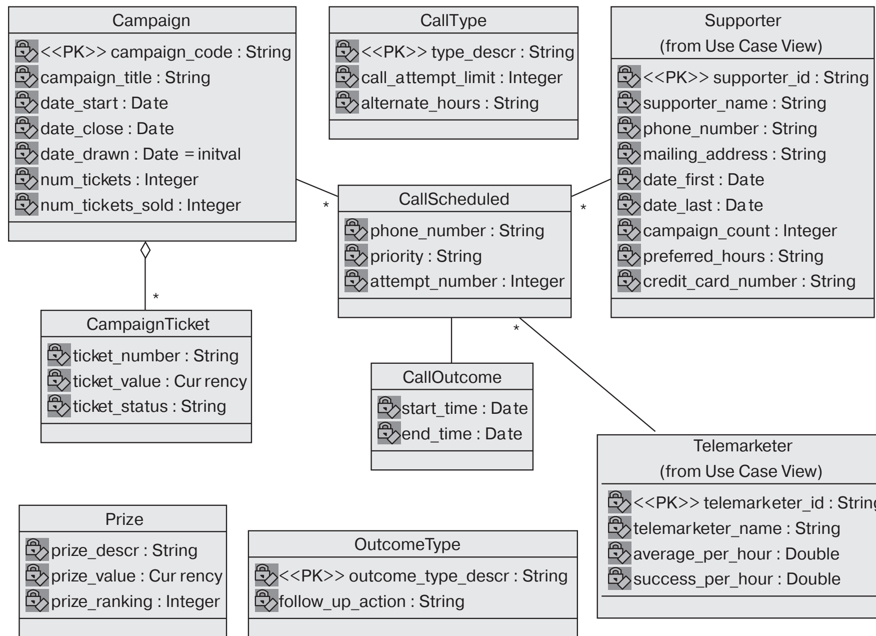

Рис. 4.4. Спецификация классов (Телемаркетинг)

4.2.2. Моделирование ассоциаций

Ассоциации служат объединению объектов в системе. Они способствуют взаимодействию между объектами. Без ассоциаций объекты могут устанавливать связь только во время прогона программы, если они совместно используют одни и те же атрибу-

ты или они имеют доступ (с помощью других средств, таких как глобальные переменные) к идентифицирующим объекты значениям других объектов.

Ассоциации представляют собой наиболее существенный вид отношений моделей, в частности, моделей постоянных "бизнес-объектов". Ассоциации поддерживают выполнение прецедентов и, таким образом, обеспечивают совмещение спецификации состояний и поведения.

4.2.2.1. Выявление ассоциаций

Нахождение основных ассоциаций представляет собой побочный эффект процесса выявления классов. При определении классов аналитик принимает решение об атрибутах классов, и некоторые из этих атрибутов являются ассоциациями с другими классами. Атрибуты могут относиться к элементарным типам данных либо могут вводиться в качестве других классов, устанавливая таким образом отношения с другими классами. По существу, любой атрибут, относящийся к *неэлементарным типам данных*, должен моделироваться как ассоциация (или агрегация) по классу, представляющему этот тип данных.

Выполнение пробного прогона прецедентов позволяет выявить остающиеся ассоциации. Устанавливаются пути взаимодействия между классами, необходимые для прогона прецедентов. Обычно ассоциации должны поддерживать эти пути взаимодействия.

Каждая *тернарная ассоциация* должна быть заменена циклом или бинарной ассоциацией. Тернарные ассоциации привносят риск неверного семантического истолкования.

Иногда для того, чтобы полностью выразить базовую семантику, *циклы, образуемые ассоциациями*, не должны коммутировать (быть замкнутыми) [51]. Это значит, что по меньшей мере одна из ассоциаций в цикле может быть *производной* (*derived*). Подобная ассоциация является избыточной в семантическом смысле и должна быть исключена (хорошая семантическая модель должна быть лишена избыточности). Вполне допустимо, что многие производные ассоциации, тем не менее, все же войдут в проектную модель (например, из соображений эффективности).

4.2.2.2. Спецификация ассоциаций

Спецификация ассоциаций подразумевает выполнение следующих действий.

1. Присваивание имен ассоциациям.
2. Присваивание имен ассоциативным ролям.
3. Установление кратности ассоциации (разд. 2.1.3.2).

Правила именования ассоциаций должны соответствовать соглашениям по именованию атрибутов — имена ассоциаций состоят из строчных букв, отдельные слова в имени ассоциации разделяются подчеркиванием (разд. 4.2.1.2.2).

Если два класса связаны только одним ассоциативным отношением, задавать имя ассоциации и ассоциативные *ролевые имена* между этими классами необязательно (разд. 2.1.2.1.1). CASE-средства могут внутренне различать каждую ассоциацию через системные идентификационные имена.

Ролевые имена можно использовать для раскрытия более сложных ассоциаций, в частности *самоассоциативных отношений* (*self associations*) (*рекурсивных* ассоциаций, которые связывают объекты одного и того же класса). При задании ролевых имен их следует выбирать с учетом того, что в проектной модели они станут атрибутами классов, расположенных на противоположных концах ассоциативной связи.

Кратность должна быть задана для обоих концов (ролей) ассоциации. Если вопрос кратности на этом этапе не ясен, нижняя и верхняя границы кратности можно опустить.

4.2.2.3. Пример спецификации ассоциации

Модель ассоциаций для приложения *Управление контактами с клиентами* показана на рис. 4.5. Чтобы продемонстрировать гибкость ассоциативного моделирования, имена ассоциаций и ролевые имена используются бессистемно.

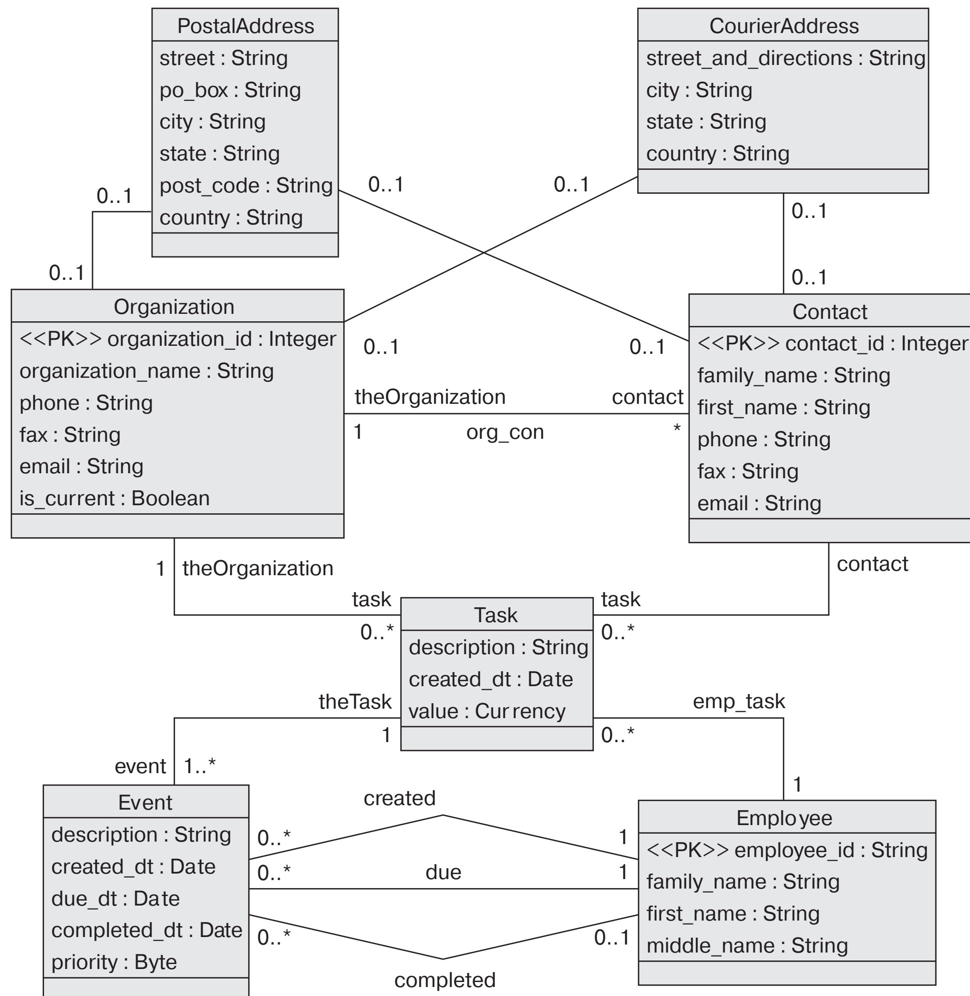

Рис. 4.5. Спецификация ассоциаций (Управление контактами с клиентами)

Пример 4.8. Управление контактами с клиентами

Обратитесь к примерам 4.3 и 4.6. Требования, приведенные в этих примерах, позволяют нам выявить и специфицировать ассоциации на классах для системы *Управление контактами с клиентами*.

Кратность всех ассоциаций между классами PostalAddress и CourierAddress с одной стороны и классами Organization и Contact с другой равна “ноль или один”. Дадим пояснения к использованию ассоциации между классами Organization и PostalAddress).

На одном конце ассоциации объект Organization связан максимум с одним объектом PostalAddress, но только в том случае, если почтовый адрес организации известен. На противоположном конце ассоциации конкретный объект PostalAddress связан с объектом Organization или объектом Contact. Следовательно, чтобы удовлетворять этим ограничениям, кратность должна быть равна “ноль или один”. (Даже при этих условиях, само ограничение необходимо отдельно зафиксировать документально с помощью средств моделирования ограничений языка UML (разд. 5.1.2)).

Ассоциация между классами Organization и Contact демонстрирует использование как имен ассоциаций, так и ролевых имен. Ролевые имена преобразуются в имена атрибутов модели реализации приложения *Управление контактами с клиентами*. Модель реализации содержит следующие атрибуты: Organization.contact и Contact.theOrganization. Префикс “the”означает, что роль theOrganization имеет кратность точно равную “один”. (Артикль “the” употребляется в английском языке для указания на определенный, конкретный объект. *Прим. ред.*). Кратность роли contact равна “много” (нижняя и верхняя границы не установлены).

Кратность роли contact между классами Task и Contact не задана. Требования не объясняют, должно ли задание быть непосредственно связано с контактом. Поэтому у нас нет уверенности в том, может ли оно быть связано более, чем с одним контактом.

Наконец, существует три ассоциации между классами Event и Employee. Эти ассоциации устанавливают, кто из сотрудников создает мероприятие, кто отвечает за его выполнение и кто, в конце концов, завершает его. Во время создания мероприятия сотрудник, который работает над его завершением, неизвестен (поэтому кратность на конце ассоциации completed со стороны работника равна “ноль или один”).

4.2.3. Моделирование отношений агрегации и композиции

Агрегация и ее более строгая форма *композиция* служит проводником семантики “часть-целое” между составным (супермножество) классом и компонентным (подмножество) классом (разд. 2.1.4). В языке UML агрегация трактуется как ограниченная форма ассоциации. Это колоссальное умаление роли агрегации в моделировании. Достаточно сказать, что агрегация, наряду с обобщением, является наиболее важным методом многократного использования функциональных возможностей в объектно-ориентированных системах.

Моделирующая способность языка UML значительно усилилась, если бы язык поддерживал четыре возможных семантики для агрегации [55].

1. Агрегация типа “Безраздельно обладает”.
2. Агрегация типа “Обладает”.
3. Агрегация типа “Включает”.
4. Агрегация типа “Участник”.

Агрегация типа *Безраздельно обладает* устанавливает следующее:

- между компонентными классами и их составными классами установлено отношение *зависимости по существованию* (следовательно, удаление составного

объекта распространяется вниз по иерархии отношения, так что связанные компонентные объекты также удаляются);

- агрегация *транзитивна* (если объект *C1* является частью объекта *B1*, а *B1* является частью *A1*, тогда *C1* является частью *A1*);
- агрегация *асимметрична* (*нерефлексивна*) (если объект является частью объекта *A1*, то объект *A1* не является частью *B1*);
- агрегация *стационарна* (если объект *B1* является частью объекта *A1*, то он не может быть частью объекта *Ai* (i≠1)).

Агрегация типа *Обладает* поддерживает первые три свойства агрегации типа "Безраздельно обладает", к которым относятся:

- зависимость существований;
- транзитивность;
- асимметричность.

Агрегация типа *Включает* слабее, чем агрегация типа *Обладает*. Агрегация типа *Включает* поддерживает следующие свойства:

- транзитивность;
- асимметричность.

Агрегация типа *Участник* обладает свойством целенаправленного группирования независимых объектов — группирования, при котором не делается предположений относительно свойства зависимости по существованию, транзитивности, асимметричности или стационарности. Это абстракция, посредством которой совокупность членов-компонентов рассматривается как составной объект более высокого уровня. Компонентный объект в агрегации типа *Участник* может одновременно принадлежать более, чем одному составному объекту (поэтому, кратность *агрегации* типа *Участник* может иметь значение "*многие ко многим*").

Хотя *агрегация* получила признание как фундаментальная концепция моделирования, по меньшей мере, одновременно с обобщением [80], в объектно-ориентированном анализе и проектировании ей уделяется лишь незначительное внимание (за исключением областей приложений "совершенной парочки", наподобие систем мультимедиа). К счастью, эта тенденция в будущем может перемениться, благодаря вкладу и проницательности методологов, работающих над *шаблонами проектирования*. Это ясно видно, например, из трактовки агрегации (композиции) в плодотворной книге Гамма (Gamma) и др. [28].

4.2.3.1. Выявление агрегаций и композиций

Поиск агрегаций ведется параллельно с поиском ассоциаций. Если ассоциация проявляет одно или более из четырех семантических свойств, рассмотренных выше, то ее можно моделировать как агрегацию.

При объяснении отношения агрегации лакмусовой бумажкой выступают фразы "*включает*" ("*has*") и "*является частью*" ("*is-part-of*"). При истолковании отношения сверху-вниз по иерархии классов используется фраза "включает" (например, `Книга` "включает" `Главу`). При интерпретации снизу-вверх используется фраза "является частью" (например, `Глава` "является частью" `Книги`). Если предложение, описывающее отношение, прочитывается вслух с использованием этих фраз и оно лишено смысла на естественном языке, то это отношение *не* является агрегацией.

Со структурной точки зрения агрегация часто связывает воедино большое количество классов, тогда как ассоциация степени выше двух бессмысленна (см. разд. 2.1.3.1). Когда требуется связать более двух классов воедино, отличным вариантом моделирования может быть агрегация типа *Участник*.

4.2.3.2. Спецификация агрегаций и композиций

Язык UML обеспечивает только ограниченную поддержку агрегации. Сильная форма агрегации называется в UML *композицией*. В композиции составной объект может физически содержать компонентные объекты (семантически это отношение берется "по значению"). Компонентный объект может принадлежать только одному составному объекту. Отношение композиции языка UML в большей или меньшей степени соответствует нашим агрегациям типа *Безраздельно обладает* и *Обладает*.

Слабая форма агрегации в UML называется просто *агрегацией*. Это отношение семантически берется "по ссылке" — составной объект физически не содержит компонентный объект. Один компонентный объект может обладать несколькими ассоциативными или агрегативными связями в модели. Попросту говоря, *агрегация* в языке UML соответствует нашим агрегациям типа *Включает* и *Участник*.

Сплошной ромб в языке UML представляет композицию. *Пустой ромб* используется для определения агрегации. В остальном спецификация агрегации совпадает с обозначениями для ассоциации.

4.2.3.3. Пример спецификации агрегации и композиции

Пример 4.9. Запись на университетские курсы

Обратитесь к примерам 4.1 и 4.5. Рассмотрите следующие дополнительные требования.

1. Академическая характеристика студента должна быть доступна по требованию. Характеристика должна включать информацию об оценках, полученных студентом по каждому из курсов, на которые записался студент (и которые он не покинул без взыскания, т.е. в первые три недели с начала семестра).
2. За каждый курс отвечает преподаватель, однако этот курс могут вести и другие преподаватели. В течение разных семестров за один курс могут отвечать разные преподаватели; кроме того, в течение разных семестров курс могут вести разные преподаватели.

На рис. 4.6 показана модель классов, в которой выделено отношение агрегации. Понятие "студент" (объект `Student`) "включает" академическую характеристику (объект `AcademicRecord`)" — это описание композиции в языке UML (с семантикой "по значению"). Каждый объект `AcademicRecord` физически помещен в один из объектов `Student`. Несмотря на существование ассоциации `takes` (брать [курс обучения]) объект `AcademicRecord` включает атрибут `course_code` (код курса).

Это необходимо, поскольку ассоциация `takes` реализуется с помощью атрибута `takes_crsoff` ("берет" дисциплину) объекта `Student`, введенного как *коллекция* (*collection*), например, `Set[CourseOffering]`. Атрибут `takes_crsoff` не зависит от информации во вложенном объекте AcademicRecord, хотя он безусловно связывает объект `Student` с объектом `Course`.

Понятие "курс" (объект `Course`) включает дисциплину (объект `CourseOffering`)" — это описание агрегации в языке UML (с семантикой "по ссылке"). Каждый объект `CourseOffering` только логически содержится в одном из объектов `Course`. Объект `CourseOffering` может также участвовать в других агрегациях и/или ассоциациях (например, с объектами `Student` и `AcademicInCharge` (ответственный преподаватель)).

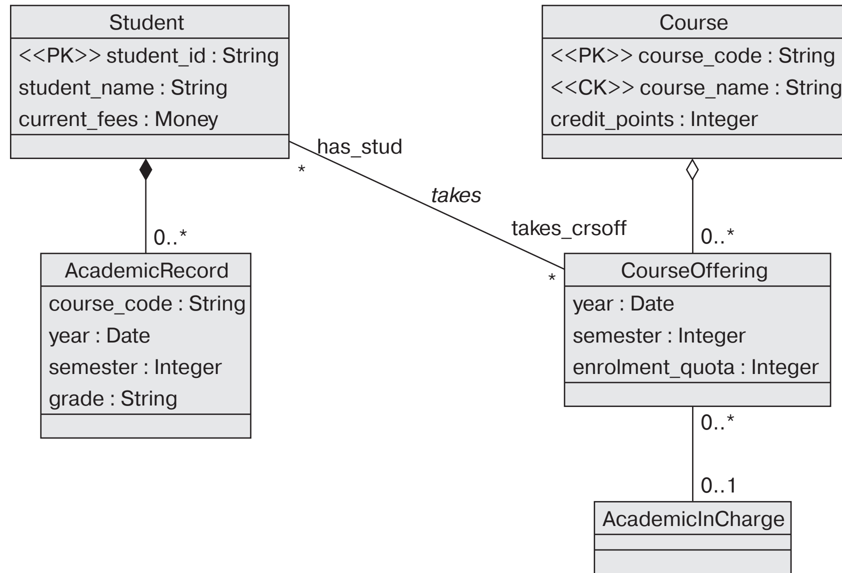

Рис. 4.6. Спецификация агрегаций (Запись на университетские курсы)

4.2.4. Моделирование отношений обобщения

Общие *характеристики* (атрибуты и операции) одного или более классов можно погрузить в более общий класс. Это явление известно как *обобщение*. Отношение обобщения соединяет родовой класс (*суперкласс*) с более специализированными классами (*подклассы*). Обобщение делает возможным наследование (многократное использование) характеристик суперкласса подклассом. В традиционных объектно-ориентированных системах наследование применяется к классам, а не к объектам (наследуются типы, а не значения).

Помимо *наследования* обобщение преследует еще две цели [76].

1. Подставимость.
2. Полиморфизм.

В соответствии с принципом *подставимости* (*substitutability*) объект подкласса является законным значением переменной суперкласса. Например, если переменная объявлена с целью хранения объекта `Fruit` (Фрукт), то объект `Apple` (Яблоко) является допустимым значением.

В соответствии с принципом *полиморфизма* (*polymorphism*) одна и та же операция может иметь разные реализации в разных классах. Вызывающий объект может вызвать операцию, не зная и не заботясь о том, какая из реализаций операции выполнится. Вызываемый объект знает, какому классу он принадлежит и выполняет свою собственную реализацию.

Полиморфизм работает лучше в тех ситуациях, когда он идет “рука об руку” с наследованием. Зачастую полиморфная операция в суперклассе объявляется, а реализация для нее в этом классе отсутствует. Это значит, что операция получила *сигнатуру* (имя и список формальных аргументов), а реализация должна быть обеспечена в каждом из подклассов. Подобная *операция* называется *абстрактной*.

Абстрактную операцию не следует путать с *абстрактным классом*. Последний является классом, у которого отсутствуют непосредственные объекты-экземпляры (однако его подклассы могут обладать объектами-экзеплярами). Экземпляров класса Vegetable (Овощи) может не существовать. Непосредственными экземплярами являются только объекты классов Potato (Картофель), Carrot (Морковь) и т.д.

Фактически, класс, обладающий абстрактной операцией, является абстрактным. Конкретный класс, наподобие Apple, не может иметь абстрактных операций [37]. Хотя абстрактные операции вводятся на уровне спецификаций поведения, абстрактные классы — это сфера спецификаций состояния.

4.2.4.1. Выявление обобщений

Многие суперклассы/подклассы аналитик отмечает еще в процессе формирования первоначального перечня классов. Многие другие обобщения можно обнаружить при определении ассоциаций.

Различные ассоциации (даже принадлежащие одному и тому же классу) может потребоваться связать с классом на различных уровнях обобщения/специализации. Например, класс Course может быть связан с классом Student (Student *takes* Course — студент *берет* курс), кроме того, этот класс может быть связан с классом TeachingAssistant (TeachingAssistant *teaches* Course — ассистент *ведет* курс). Дальнейший анализ может показать, что класс TeachingAssistant является подклассом Student.

При поиске отношения обобщения лакмусовой бумажкой выступают фразы "*может быть*" ("*can-be*") и "*это нечто вроде*" ("*is-a-kind-of*"). При истолковании отношения сверху-вниз по иерархии классов используется фраза "может быть" (например, Student "can-be" a TeachingAssistant — "студент "может быть" ассистентом"). При интерпретации отношения снизу-вверх используется фраза "это нечто вроде" (например, TeachingAssistant "is-a-kind-of" Student — "ассистент "это нечто вроде" студента"). Обратите внимание, что если также справедливо утверждение о том, что "ассистент — это "TeachingAssistant "is-a-kind-of" Teacher", то мы установили *множественное наследование*.

4.2.4.2. Спецификация обобщений

Отношение обобщения между классами показывает, что один класс совместно использует структуру или поведение, определенные в одном или более классов. Обобщение представляется в языке UML сплошной линией со *стреловидным наконечником*, указывающим на суперкласс.

Полная спецификация обобщения включает несколько мощных возможностей. Например, более подробное определение отношения может содержать квалификацию *доступа* к нему, указания на то, предоставляет ли класс *права* другому классу, решая, что необходимо делать в случае *множественного наследования* и т.д. Эти вопросы рассматриваются в главе 5.

4.2.4.3. Пример спецификации обобщений

На рис. 4.7 представлена расширенная модель классов для приложения *Магазин видеопроката*. Иерархия обобщения выражает тот факт, что видеоноситель (VideoMedium) "*может быть*" видеокассетой (VideoTape) или видеодиском VideoDisk. Видеокассета (VideoTape) "*может быть*" кассетой типа BetaTape или типа VHSTape. Видеодиск "*может быть*" диском типа DVDDisk (в настоящий момент видеодиск "*должен быть*" диском

DVDDisk, однако мы предполагаем, что в будущем возможно появление новых подклассов для VideoDisk).

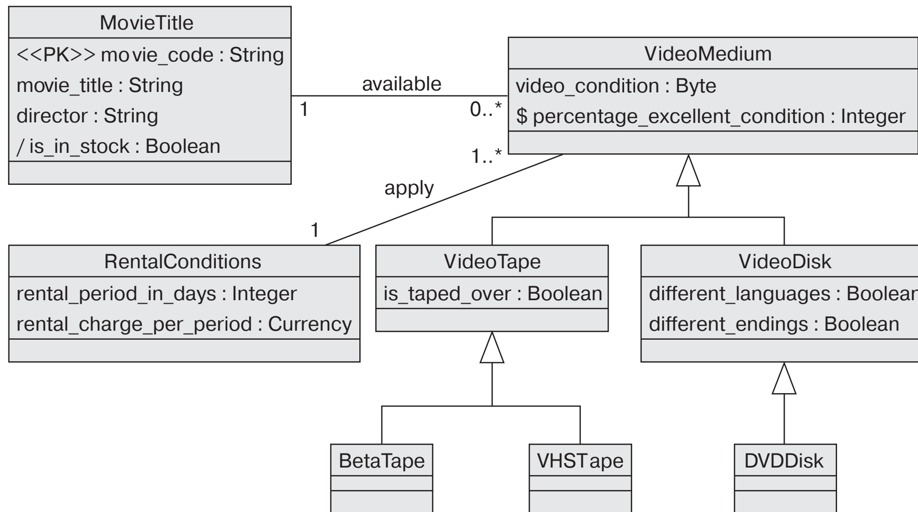

Рис. 4.7. Спецификация обобщения (Магазин видеопроката)

Пример 4.10. Магазин видеопроката

Обратитесь к примерам 4.1 и 4.5. Классы, обозначенные на рис. 4.2 (пример 4.5), заключают в себе иерархию обобщения с корнем в классе VideoMedium. Однако, до сих пор мы не приступили к моделированию различий в состоянии и поведении между разными типами носителей VideoMedium. Предположим, что руководству магазина видеопроката требуется знать, является ли видеокассета (VideoTape) новой марочной кассетой или она уже перезаписана (этот факт можно зафиксировать с помощью атрибута is_taped_over). Также предположим, что емкость памяти видеодиска (VideoDisk) позволяет хранить несколько версий одного фильма, каждая из которых отличается языком или окончанием.

Наша задача заключается в расширении модели, показанной на рис. 4.2, за счет включения в нее отношений между классами и, в частности, спецификации отношения обобщения.

VideoMedium, VideoTape и VideoDisk — это *абстрактные классы* (их имена выделены курсивом). Абстрактные классы не материализуют объекты. Объекты VideoMedium материализуются с помощью классов BetaTape, VHSTape и DVDDisk. Поэтому, например, ассоциация available (в наличии) может, фактически, связывать объект MovieTitle с одним или более конкретным объектом класса VideoMedium (т.е. с объектами BetaTape, VHSTape и/или DVDDisk).

То же самое справедливо в отношении ассоциации apply (применяются к). Однако в случае ассоциации apply можно заметить некоторую неэффективность модели. Из требований к приложению *Магазин видеопроката* нам известно, что условия проката одного и того же фильма отличаются для видеокассет и видеодисков. Однако в отношении всех видеокассет с одинаковым фильмом и всех видеодисков с одинаковым фильмом применяются одинаковые условия проката. Это предполагает, что объект Rental

Conditions лучше было бы связать в ассоциацию с конкретными объектами классов VideoTape и VideoDisk. Пока же мы примиримся с этой неэффективностью.

4.2.5. Моделирование объектов

Моделирование касается проблем определения систем. Модель — это не действующая система, и поэтому она не отражает объектов-экземпляров. В любом случае, количество объектов в работающей системе может быть огромным, и представить их графически невозможно. Тем не менее, при моделировании классов мы часто представляем себе объекты и рассматриваем трудные сценарии с использованием примеров объектов.

4.2.5.1. Спецификация объектов

Язык UML позволяет представить *объекты* графически (разд. 2.1.1.1). Мы можем изобразить диаграмму объектов, чтобы проиллюстрировать сложные отношения между классами или продемонстрировать изменения объектов со временем. Объектные модели можно использовать для иллюстрации и исследования способа взаимодействия объектов во время прогона программной системы.

4.2.5.2. Пример спецификации объектов

На рис. 4.8 показана диаграмма объектов, соответствующая модели классов, представленной на рис. 4.6. Из рисунка ясно, что объект-студент Student Don Donaldson физически содержит два объекта AcademicRecord (для курсов COMP224 и COMP326). К настоящему времени Don записался на два курса: COMP225 и COMP325. Изучение одного из курсов COMP325 было предложено провести во втором семестре 2000-го года. За чтение этого курса отвечал профессор Rick Richardson.

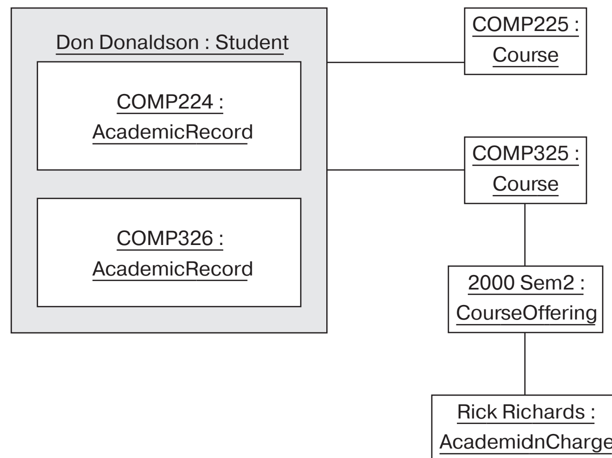

Рис. 4.8. Диаграмма объектов (Запись на университетские курсы)

Пример 4.11. Запись на университетские курсы

Наша задача в этом примере заключается в том, чтобы показать несколько объектов, представляющих классы из модели классов на рис. 4.6 (пример 4.9).

4.3. Спецификация поведения

Поведение системы — так как оно выглядит для внешнего пользователя — изображается в вид *прецедентов* (разд. 2.2.2). Модели прецедентов можно разрабатывать на различных уровнях абстракции. Их можно применить к системе в целом для того, чтобы специфицировать основные функциональные блоки разрабатываемого приложения. Их также можно использовать для фиксации поведения *пакетов* UML, частей пакетов или даже *класса* внутри пакета.

На этапе *анализа* прецеденты вбирают в себя системные требования, концентрируясь на том, *что* делает или должна делать система. На этапе *проектирования* представление проектных решений в виде прецедентов можно использовать для спецификации поведения системы в том виде, как оно должно быть реализовано.

Поведение системы, закрепленное с помощью прецедентов, требует осуществить соответствующие вычисления и обеспечить взаимодействие объектов для выполнения этих прецедентов. *Вычисления* можно смоделировать с помощью диаграмм видов деятельности. *Взаимодействие* объектов можно задать с помощью диаграмм последовательностей или диаграмм кооперации. (В данной главе используются только диаграммы последовательностей, диаграммы кооперации используются далее в главах, посвященных проектированию (разд. 6.2).)

Спецификация поведения позволяет взглянуть на систему с *точки зрения ее функционирования*. Здесь основная задача состоит в том, чтобы определить прецеденты для области приложений и установить, какие классы участвуют в выполнении этих прецедентов. При этом необходимо идентифицировать *операции* классов и *сообщения*, передаваемые между объектами. Хотя взаимодействие объектов инициирует изменения состояния объектов, спецификации поведения дают *функциональный взгляд на застывшее состояние* системы. Изменения в состоянии объектов явным образом находят выражение в *спецификациях изменения состояний*.

Модели прецедентов должны нарабатываться итеративно и параллельно с моделями классов. Классы, введенные в спецификации состояний, подлежат дальнейшей детальной проработке, при этом обозначаются наиболее важные операции. Следует, однако, напомнить, что спецификация состояний определяет только *классы сущностей* (“бизнес-объектов”).

По мере создания моделей поведения появляются еще два уровня классов.

1. Классы, которые обслуживают события, инициируемые пользователями, и представляют бизнес-процессы (*управляющие классы*).
2. Классы, представляющие GUI-интерфейсы (*пограничные классы*) (разд. 5.2.4).

4.3.1. Моделирование прецедентов

Моделирование прецедентов тесно связано с установлением требований (см. главу 3). Требования, изложенные в текстовом виде в документе описания требований, необходимо довести до прецедентов, зафиксированных в документе спецификации требований. Если остальной процесс разработки направляется прецедентами, то процесс называется проблемно-ориентированным (разд. 4.2.1.1.3).

Подобно моделированию классов моделирование прецедентов существенно *итеративный* и *наращиваемый* процесс. Первоначальную диаграмму прецедентов можно определить на основе требований верхнего уровня. Это может быть *модель бизнес-*

прецедентов (разд. 3.5.2). Для дальнейшего уточнения прецедентов следует руководствоваться более детализированными требованиями. Если в течение ЖЦ разработки требования пользователей подвергаются изменениям, эти изменения следует отразить сперва в документе описания требований, а уже затем — в модели прецедентов. Затем изменения в прецедентах доводятся до других моделей [37], [69].

4.3.1.1. Выявление прецедентов

При выявлении прецедентов аналитик должен убедиться в том, что он твердо придерживается сущности концепции прецедентов. Прецеденты представляют следующие компоненты общей модели системы [37], [47], [66], [76].

1. *Завершенный* фрагмент функциональных возможностей (включая *основной поток* логики управления, его любые вариации (*подпотоки*) и исключительные условия (*альтернативные потоки*)).
2. Фрагмент внешне *наблюдаемых функций* (отличных от внутренних фунций).
3. *Ортогональный* фрагмент функциональных возможностей (прецеденты могут при выполнении совместно использовать объекты, но выполнение каждого прецедента независимо от других прецедентов).
4. Фрагмент функциональных возможностей, *инициируемый субъектом* (будучи инициирован, прецедент может взаимодействовать с другими субъектами). При этом возможно, что субъект окажется только на принимающем конце прецедента (может быть, опосредовано), инициированного другим субъектом.
5. Фрагмент функциональных возможностей, который предоставляет *субъекту* ощутимый *полезный результат* (и этот полезный результат достигается в пределах одного прецедента).

Выявление *прецедентов* основано на анализе следующих источников информации.

1. Требования, определенные в документе описания требований.
2. Субъектов и их целей применительно к системе.

Вопросы управления требованиями рассматривались в разделе 3.4. Напомним только, что требования представляются в отпечатанном виде. При поиске прецедентов нас интересуют только *функциональные требования.*

Прецеденты можно определить на основе анализа задач, выполняемых субъектами. Джекобсон (Jacobson) [39] предлагает ответить на ряд вопросов, касающихся субъектов. Ответы на эти вопросы могут позволить обозначить прецеденты. Вот эти вопросы.

- Каковы основные задачи, выполняемые каждым субъектом?
- Должен ли субъект получать доступ к информации в системе или модифицировать информацию?
- Должен ли субъект информировать систему о любых изменениях в других системах?
- Должен ли субъект быть проинформирован о непредвиденных изменениях в системе?

В ходе анализа прецеденты обращаются к личностным потребностям субъектов. В некотором роде это *прецеденты субъектов.* Поскольку прецеденты определяют основные строительные блоки для системы, то существует необходимость в выделении *системных прецедентов.* Системные прецеденты извлекают все то общее, что присутствует в прецедентах субъектов и дают возможность разработать общее решение, примени-

мое (через механизм наследования) к области прецедентов субъектов. “Субъектом” системного прецедента является разработчик/программист, а не пользователь. Системные прецеденты идентифицируются на этапе проектирования.

4.3.1.2. Спецификация прецедентов

Спецификация прецедентов включает графическое представление субъектов, прецедентов и четырех типов отношений, перечисленных ниже [24], [76].

1. Ассоциация.
2. Включение.
3. Расширение.
4. Обобщение прецедента.

Отношение *ассоциации* устанавливает каналы связи между субъектом и прецедентом. В качестве стереотипов для отношений “*включает*” (*include*) и “*расширяет*” (*extend*) используются слова `<<include>>` и `<<extend>>`. Отношение *обобщения* позволяет специализировать прецедент посредством изменения любого из аспектов базового прецедента.

Отношение включения `<<include>>` позволяет вынести общее поведение за пределы включаемого прецедента. (Это отношение пришло на смену широко применяемой в более ранних версиях UML концепции отношения `<<uses>>` (использует)). Отношение `<<extend>>` обеспечивает контролируемую форму расширения поведения прецедента с помощью активизации другого прецедента в определенных точках расширения. Отношение `<<include>>` отличается от отношения `<<extend>>` в том, что “включаемый” прецедент является необходимым для завершения “активизирующего” прецедента.

На практике, проект может легко столкнуться с проблемами, если прилагать слишком большие усилия для выявления отношений между прецедентами и установления, какие отношения применимы к определенной паре прецедентов. Кроме того, обобщенные прецеденты имеют тенденцию к таким тесным связям, что взаимосвязи станут превалировать и загромождать диаграмму, смещая акценты с надлежащей идентификации прецедентов в сторону отношений между прецедентами.

4.3.1.3. Пример спецификации прецедентов

Пример 4.12. Запись на университетские курсы

Обратитесь к постановке задачи для системы *Запись на университетские курсы*, приведенной в разделе 2.3.1, а также к требованиям, определенным в примерах 4.1 и 4.4 раздела 4.2.1. Наша задача состоит в том, чтобы установить прецеденты на основе анализа функциональных требований.

На рис. 4.9 показана обобщенная диаграмма прецедентов для приложения *Запись на университетские курсы*. Модель содержит четыре субъекта и четыре прецедента. Каждый прецедент инициируется субъектом и является завершенным, внешне видимым и ортогональным фрагментом функциональных возможностей. Все субъекты, за исключением субъекта `Student`, представляют собой *инициирующих* субъектов. Субъект `Student` получает результаты экзаменов и инструкции по записи на учебные курсы перед тем, как программа обучения в следующем семестре (учебном периоде) может быть введена и проверена.

Прецедент `Provide Examination Results` (Предоставить результаты экзаменов) *может* “расширить” (`<<extend>>`) прецедент `Provide Enrolment Instructions` (Предоставить инструкции по записи). Первый прецедент не всегда расширяет последний прецедент. Например, для новых студентов результаты экзаменов неизвестны. Вот почему отношение моделируется с использованием стереотипа расширения (`<<extend>>`), а не включения (`<<include>>`).

Отношение `<<include>>` было установлено от прецедента `Enter Program of Study` (Ввести программу обучения) к прецеденту `Validate Program of Study` (Проверить программу обучения). Отношение `<<include>>` означает, что первый из прецедентов всегда включает последний. Как только программа изучения введена, она проверяется на предмет конфликтов расписания, специальных согласований и т.д.

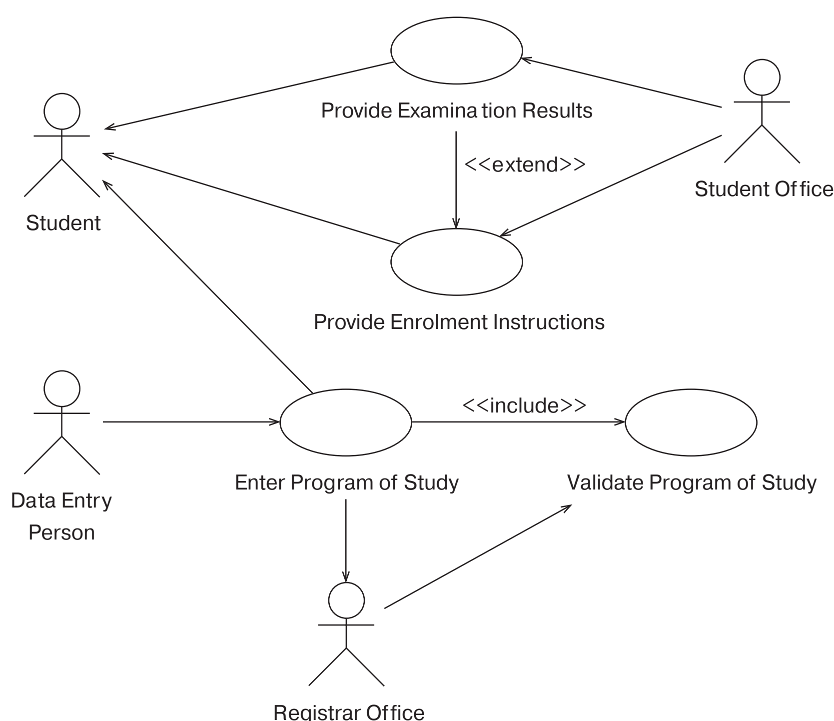

Рис. 4.9. Диаграмма прецедентов (Запись на университетские курсы)

Пример 4.13. Управление контактами с клиентами

Обратитесь к постановке задачи для системы *Управление контактами с клиентами*, приведенной в разделе 2.3.3, а также к требованиям, определенным в примерах 4.3 и 4.6 раздела 4.2.1. Наша задача состоит в том, чтобы установить прецеденты на основе анализа функциональных требований.

На диаграмме прецедентов для приложения *Управление контактами с клиентами* представлены три субъекта и пять прецедентов (рис. 4.10). Эта модель обладает интересной особенностью: субъекты связаны отношением обобщения. Менеджер по обслуживанию клиентов (`Customer Services Manager`) — "*это нечто вроде*" сотрудника по обслуживанию клиентов (`Customer Services Employee`), который в свою очередь "*нечто вроде*" сотрудника (`Employee`). Обобщающая иерархия делает диаграмму более выразительной. Любое мероприятие, выполненное сотрудником, может также выполнить сотрудник по обслуживанию клиентов или менеджер по обслуживанию клиентов. Следовательно, менеджер по обслуживанию клиентов неявно входит в ассоциацию (и может инициировать) любой прецедент.

Прецедент создания задания (`Create Task`) "включает" прецедент планирования события (`Schedule Event`), как следствие того требования, что задание нельзя создать, не запланировав первое мероприятие. Отношение "расширяет" обозначает тот факт, что завершение мероприятия может инициировать изменения деталей, связанных с организацией или контактом.

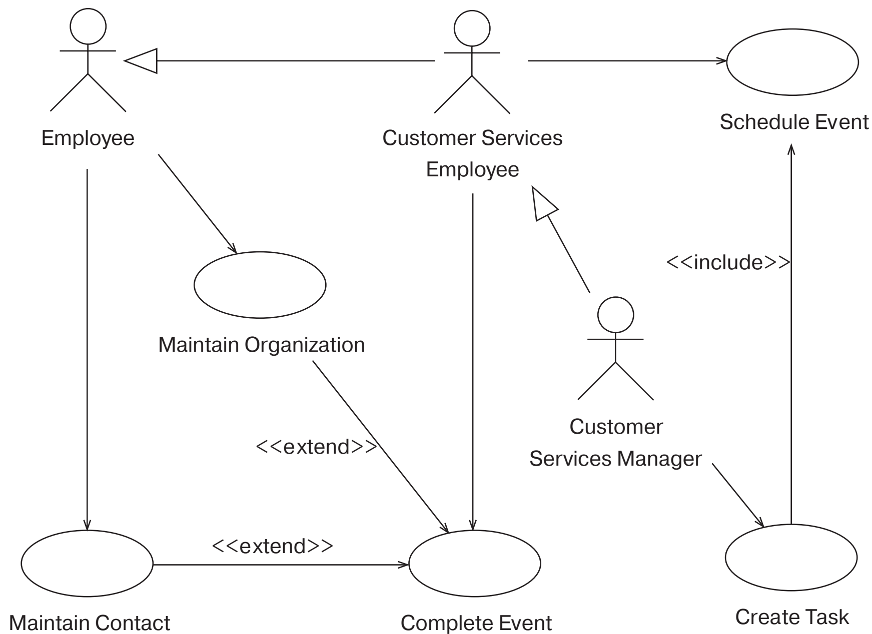

Рис. 4.10. Диаграмма прецедентов (Управление контактами с клиентами)

Пример 4.14. Магазин видеопроката

Обратитесь к постановке задачи для системы *Магазин видеопроката*, приведенной в разделе 2.3.2, а также к требованиям, определенным в примерах 4.2 и 4.5 раздела 4.2.1. Наша задача состоит в том, чтобы установить прецеденты на основе анализа функциональных требований.

Для одного из прецедентов напишите неформальную спецификацию, включающую следующие разделы: "Резюме", "Субъекты", "Предпосылки", "Описание", "Исключения" и "Постусловия".

На диаграмме прецедентов для приложения *Магазин видеопроката* представлены только два субъекта и шесть прецедентов (рис. 4.11). Второстепенные субъекты, наподобие клиента (`Customer`) или поставщика (`Supplier`), не показаны. Эти субъекты не инициируют ни одного прецедента. Все прецеденты инициирует работник (`Employee`). Между субъектами `Scanning Device` (устройство сканирования) установлено отношение "*зависимости*" (*dependency*), которому аналитик присвоил стереотип в виде фразы `<<depends on>>` (зависит от).

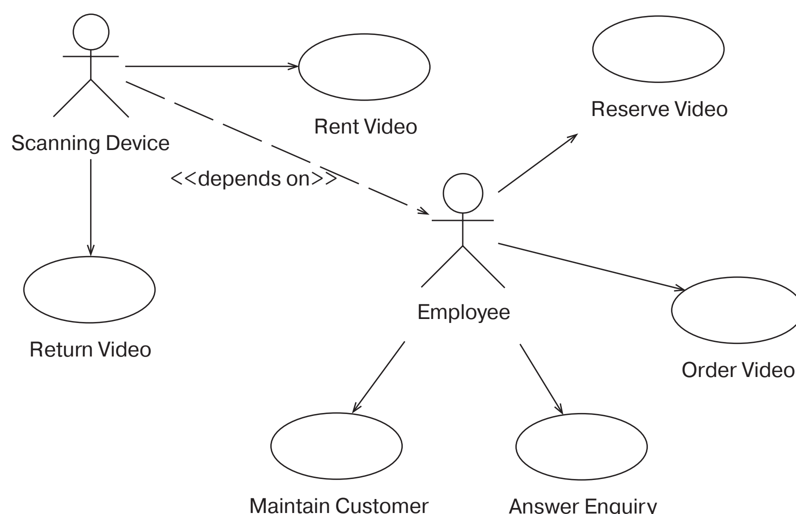

Рис. 4.11. Диаграмма прецедентов (Магазин видеопроката)

Табл. 4.4 является иллюстрацией того факта, что графическое представление прецедентов являет собой лишь один из аспектов полной модели прецедентов. Каждый прецедент диаграммы в дальнейшем должен быть задокументирован в репозитории CASE-средства. В частности, необходимо текстовое описание. В табл. 4.4 используется одна популярная структура для описательной спецификации прецедентов (разд. 2.2.2.4).

Таблица 4.4. Описательная спецификация прецедента "Прокат видео" (Магазин видеопроката)

Прецеденты	***Прокат видео***
Краткое описание	Клиент желает взять на прокат видеокассету или диск, которые снимаются с полки магазина или были предварительно зарезервированы клиентом. При условии, что у клиента нет неоплаченных счетов, сразу после внесения платы кассета выдается напрокат. Если кассета не возвращается вовремя, клиенту отправляется напоминание о просроченном возврате.
Субъекты	Employee (Сотрудник), Scanning Device (устройство сканирования)

Окончание табл. 4.4

Прецеденты	*Прокат видео*
Предпосылки	В наличии имеются видеокассеты или диски, которые можно взять напрокат. У клиентов есть клубные карточки. Устройство сканирования работает правильно. Работники за прилавком знают, как обращаться с системой.
Основной поток	Клиент может спросить работника о наличии видео (включая зарезервированные видеофильмы) или может взять кассету или диск с полки. Видеоноситель и членская карточка сканируются и работнику не сообщается никаких сведений о неплатежах или задержках, так что он не задает клиенту соответствующих вопросов. Если клиент не нарушает правил, тогда он может взять максимум до восьми видеофильмов. Однако если статус клиента определен как "ненадежный", то его просят внести задаток за один период проката каждой кассеты или диска. После получения суммы задатка информация о наличии фильмов обновляется, и кассета или диск передаются клиенту вместе с чеком на прокат. Клиент расплачивается наличными с помощью кредитной карточки или электронного перевода. Каждая запись о прокате хранит (под учетным номером клиента) дату выдачи и срок возврата видеофильма вместе с идентификатором работника. Для каждого видеофильма, сданного напрокат, создается отдельная запись. Прецедент генерирует напоминание о просроченном возврате клиенту, если видеофильм не был возвращен в течение двух дней по истечении даты возврата, а также повторное напоминание по истечении следующих двух дней (и в этот момент клиент помечается как "нарушитель")
Альтернативные потоки	У клиента нет членской карточки. В этом случае прецедент "Maintain Customer" (сопровождение клиентов) может быть активизирован для выдачи новой карточки. Попытка взять напрокат слишком много видеофильмов. Видеофильмы не выдаются из-за нарушения клиентом правил. Видеоноситель или членская карточка не могут быть отсканированы из-за их повреждения. Электронный перевод денег или платеж по кредитной карточке отклоняются.
Постусловия	Видеофильмы сданы напрокат, и база данных соответствующим образом обновлена.

Решение для примера 4.15, приведенное на рис. 4.12, состоит из нескольких прецедентов. Субъекты непосредственно взаимодействуют с тремя прецедентами: планирования и осуществления следующего звонка (`Schedule and Make Next Call`), записи результатов звонка (`Record Call Outcome`) и отображения деталей звонка (`Display Call Details`). По-

следний прецедент может быть в свою очередь "расширен" до отображения подробностей о кампании (`Display Campaign Details`), отображения истории благотворителя (`Display Supporter History`) и отображения подробностей о призе (`Display Prize Details`). Прецедент обновления информации по благотворителю (`Update Supporter`) "расширяет" `Display Supporter History`. Запись заказа на билет (`Record Ticket Order`) и планирование повторного вызова (`Schedule Callback`) может "расширить" `Record Call Outcome`. Предполагается, что субъекты могут независимо связываться с прецедентами расширения, однако каналы связи для этих прецедентов явно не показаны.

Пример 4.15. Телемаркетинг

Обратитесь к постановке задачи для системы *Телемаркетинг*, приведенной в разделе 2.3.4, а также к требованиям, определенным в примерах 3.1, 3.1 и 3.3, а также в примере 4.7. Наша задача состоит в том, чтобы установить прецеденты на основе анализа функциональных требований.

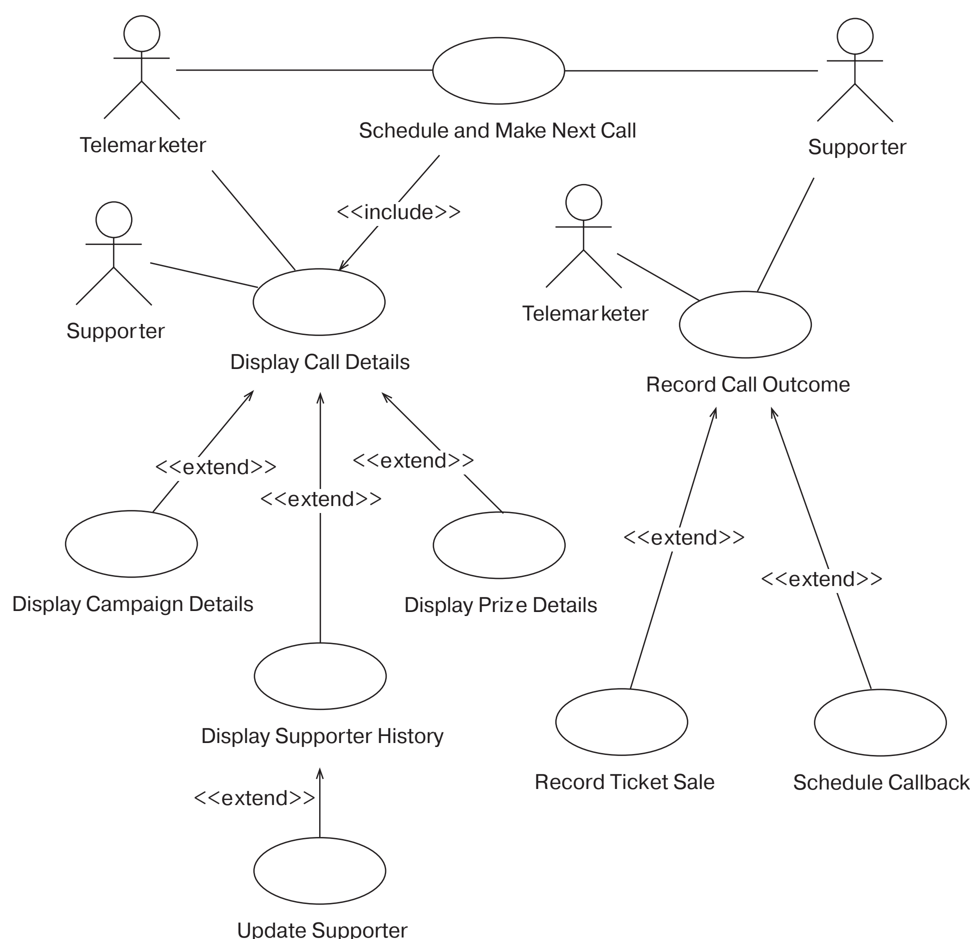

Рис. 4.12. Диаграмма прецедентов (Магазин видеопроката)

4.3.2. Моделирование видов деятельности

Диаграммы видов деятельности были введены в язык UML сравнительно недавно (разд. 2.2.3). Подобно традиционным *потоковым диаграммам* и *структурным схемам*, получившим распространение в рамках структурных методов для разработки процедурно-ориентированных программ, диаграммы видов деятельности представляют поток логики управления в объектно-ориентированных программах (хотя и на более высоком уровне абстракции). Различие же заключается в возможности представления с помощью диаграмм видов деятельности *управления параллельными потоками* наряду с *последовательным управлением*.

Модели видов деятельности широко используются в *проектировании*. Однако они также служат в качестве отличного метода изображения вычислений или технологических процессов на уровне абстракции, который подходит для *анализа*. Графы видов деятельности можно использовать для демонстрации различных уровней детализации вычислений.

Модели видов деятельности могут быть особенно полезны для определения потоков *видов деятельности* в процессе выполнения прецедентов. Поскольку модели видов деятельности не отображают *объектов*, которые осуществляют деятельность, граф видов деятельности можно построить даже в том случае, когда модель классов отсутствует или разрабатывается. В конце концов каждый вид деятельности определяется одной или несколькими операциями в одном или нескольких кооперирующихся классах. Детальная проработка такой кооперации может быть осуществлена с использованием диаграмм кооперации (разд. 6.2.2).

4.3.2.1. Выявление видов деятельности

Каждый прецедент можно моделировать с помощью одного или нескольких графов видов деятельности. Событие, источником которого служит субъект инициирующий прецедент, это то же самое событие, которое запускает выполнение графа видов деятельности. Процесс выполнения последовательно переходит от одного *состояния вида деятельности* к другому. Состояние вида деятельности считается завершенным, когда завершается его вычисление. Внешние инициируемые событиями прерывания, которые могут вызвать завершение состояния вида деятельности, допускаются только в исключительных случаях. Если ожидается, что подобные события могут происходить часто, то следует вместо этого воспользоваться диаграммой состояний.

Виды деятельности лучше всего выявлять на основе анализа предложений неформальной спецификации прецедентов (табл. 4.4). Каждая фраза, содержащая глагол, может рассматриваться как потенциальный вид деятельности. Описание альтернативных потоков вводит в граф видов деятельности ветвление и разделение потоков. Они приводят к исключительным (непредвиденным) состояниям деятельности. Возможны также параллельные потоки управления.

4.3.2.2. Спецификация видов деятельности

После выявления состояний видов деятельности спецификация видов деятельности выглядит как довольно простой процесс соединения этих состояний *линиями переходов*. Параллельные потоки инициируются (*разделяются*) и *сливаются*, что отображается на диаграмме в виде жирной полосы, обозначающей *синхронизацию потоков*. Альтернативные потоки создаются (*разветвляются*) и *объединяются*, что отображается в виде *ромбов ветвления*.

Внешние события на графе видов деятельности обычно отсутствуют. Однако существует графический метод включения внешних событий в граф. Аналогично, суще-

ствуют графические обозначения для состояний потоков объектов для представления объектов, которые являются входными или выходными для вида деятельности.

4.3.2.3. Пример спецификации видов деятельности

Пример 4.16. Магазин видеопроката

Обратитесь к примеру 4.14 и, в частности, к неформальной спецификации прецедента "Rent Video" (табл. 4.4). Наша задача состоит в том, чтобы построить диаграмму видов деятельности для этого прецедента.

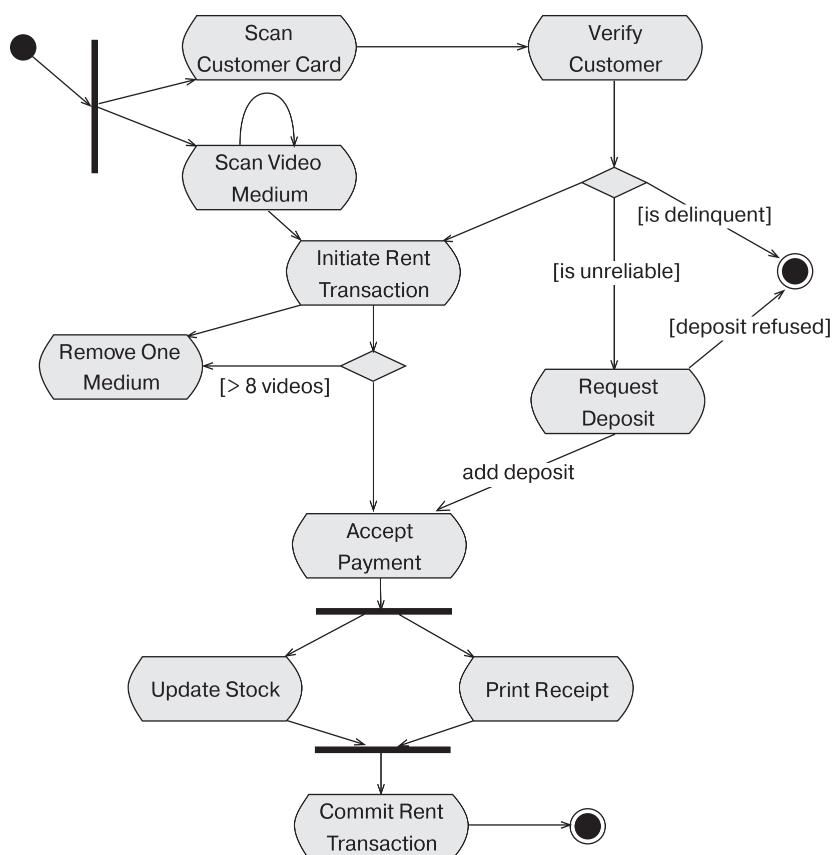

Рис. 4.13. Диаграмма видов деятельности для прецедента "Rent Video" (Магазин видеопроката)

На рис. 4.13 представлена диаграмма видов деятельности для прецедента "Rent Video". Нет ничего удивительного в том, что диаграмма отражает неформальную спецификацию прецедента (табл. 4.4). Обработка начинается со сканирования клубной карточки клиента или видеоносителя. Эти два вида деятельности рассматриваются независимо друг от друга (что показано с помощью символа разделения).

4.3.3. Моделирование взаимодействий

Диаграммы последовательностей и диаграммы кооперации относятся к диаграммам взаимодействия. Они отражают характер *взаимодействия* объектов между собой, которое необходимо для выполнения прецедента, операции или другой поведенческой компоненты.

Диаграммы последовательностей показывают обмен сообщениями между объектами, упорядоченными в виде временной последовательности. Диаграммы кооперации выделяют отношения между объектами, по которым происходит обмен сообщениями. Мы считаем диаграммы последовательностей более полезными для анализа, а диаграммы кооперации — для проектирования.

Поскольку модели взаимодействия ссылаются на объекты, они требуют, чтобы по меньшей мере первая итерация моделирования состояний была завершена, а основные классы объектов определены. Хотя взаимодействия влияют на состояния объектов, диаграммы взаимодействия не отражают в явном виде изменений состояния объектов. Это относится к области спецификации изменения состояний и диаграммам состояний (разд. 2.2.6 и 4.4).

Диаграммы взаимодействия можно использовать для определения *операций* (методов) классов (разд. 2.2.5 и 4.3.4). Любое сообщение, направляемое объекту на диаграмме взаимодействия, должно быть обслужено некоторым методом, определенным в классе этого объекта.

4.3.3.1. Выявление последовательностей сообщений

Выявление последовательностей сообщений является логическим продолжением моделирования видов деятельности. *Виды деятельности*, представленные на соответствующей диаграмме, отображаются на *сообщения* диаграммы последовательностей. Если уровни абстракции, используемые для построения модели видов деятельности и модели последовательностей, совпадают, то осуществить отображение видов деятельности на сообщения довольно просто.

4.3.3.2. Спецификация последовательностей сообщений

При спецификации сообщений полезно проводить различие между сообщением, представляющим собой сигнал, и сообщением, которое являет собой вызов операции [76]. *Сигнал* означает асинхронное взаимодействие объектов. Отправитель может продолжить выполнение потока сразу после отправки сигнального сообщения. *Вызов* означает синхронное обращение к операции с условием возврата управления отправителю. Возвращаемое сообщение может возвращать вызывающему объекту некоторые значения либо просто уведомлять его об успешном завершении операции. В последнем случае, возвращаемое сообщение отображать на диаграмме последовательностей не обязательно.

4.3.3.3. Пример спецификация последовательностей

На рис. 4.14 показана диаграмма последовательностей для приведенного выше сценария. Субъект `Data Entry Person` (Лицо, вводящее данные) инициирует прецедент, отправляя сообщение с запросом объекту граничного класса `ProgramEntryWindow` (Окно программы ввода) для того, чтобы добавить студента (обозначенного аргументом `std`) к списку записавшихся на курс (аргумент `crs`) в данном семестре (аргумент `sem`).

Пример 4.17. Запись на университетские курсы

Обратитесь к примерам 4.9 и 4 12, а также к прецеденту “Enter Program of Study”, показанному на рис. 4.9. Наша задача состоит в том, чтобы построить диаграмму последовательностей для этого прецедента.

Мы не касаемся здесь вопросов проверки корректности введенной программы обучения. Проверке обязательных условий, наличия конфликтов в расписании, специальных разрешений посвящен другой прецедент (“Validate Program of Study”). Наша задача состоит только в том, чтобы проверить, действительно ли в текущем семестре предлагается курс, на который желает записаться студент, и открыта ли еще на него запись (есть ли свободные места).

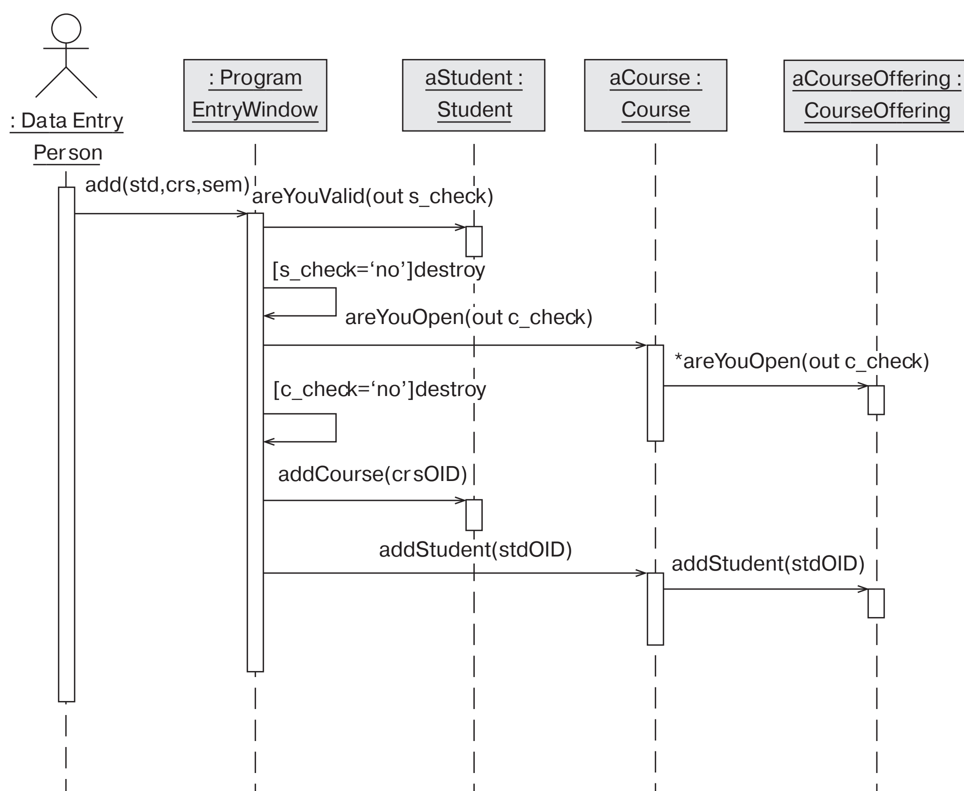

Рис. 4.14. Диаграмма последовательностей для прецедента “Enter Program of Study” (Запись на университетскик курсы)

Объект класса `ProgramEntryWindow` проверяет с помощью объекта `aStudent`, имеет ли право студент записаться (например, внес ли студент `s_check` соответствующую плату). Выходной аргумент возвращает значение “`yes`” или “`no`” объекту `:ProgramEntryWindow`. Если возвращается значение “`no`”, запись не может продолжаться, и объект `:ProgramEntryWindow` отправляет *автосообщение* самому себе, чтобы уничтожить (`destroy`) себя (операция деструктора). Запись условия в квадратных скобках говорит о том, что операция деструктора является необязательной.

Использование автосообщения для указания на необходимость уничтожения объекта является просто сокращенной записью. В действительности объект-экземпляр не может уничтожить себя. Сообщение об уничтожении (destroy) должно быть отправлено объекту-классу. Кроме того, в языке UML имеется специальное обозначение для операции уничтожения объекта — большой знак **X**, помещенный на жизненной линии объекта в точке, в которой объект должен быть уничтожен.

Если студент aStudent может быть записан, нам требуется выяснить, открыта ли запись на курс aCourse. Для обслуживания этого требования объект aCourse должен запросить свой текущий объект aCourseOffering (на который указывает агрегативная связь от Course к CourseOffering (рис. 4.6)). Если в текущем объекте aCourseOffering свободных мест уже нет, запись не может продолжаться и объект :ProgramEntryWindow снова отправляет себе автосообщение, чтобы разрушить себя.

Если процесс записи может продолжаться, объект :ProgramEntryWindow требует от объекта aStudent добавить себе объект aCourseOffering, а затем требует, чтобы объект aCourseOffering добавил объект aStudent к себе. Последовательность двух операций объекта :ProgramEntryWindow может быть изменена на обратную, если программа гарантирует ссылочную целостность между aCourseOffering и aStudent (т.е. если студент записался на предлагаемый курс, то в список студентов по данному курсу должен быть внесен данный студент).

Заметим, что если для хранения объектов Student и CourseOffering используется ПО СУБД, то объект :ProgramEntryWindow мог бы отправлять только одно из этих двух сообщений, т.е. отправить либо сообщение addCourse, либо сообщение addStudent. После того, как объект aStudent добавлен к объекту aCourseOffering, ПО СУБД берет на себя ответственность за поддержание ссылочной целостности и должно зафиксировать существование связи от aStudent к aCourseOffering и наоборот.

Заметим также, что фактические аргументы, включенные в сообщения addCourse и addStudent, представляют собой идентификаторы объектов (OID), содержащие значения OID (*дескрипторы*), указывающие на объект aStudent и aCourseOfferin, соответственно. Эти значения OID были определены, когда объект :ProgramEntryWindow получил доступ к объектам aStudent и aCourse с помощью сообщений areYouValid и areYouOpen, представляющих собой запросы на возможность записи и наличие мест на курсе.

4.3.4. Моделирование открытых интерфейсов

Открытый интерфейс (*public interface*) класса определяется набором *операций*, предлагаемых классом в качестве услуг другим классам в системе. Подобные операции объявляются с открытой видимостью. Объекты могут кооперироваться для выполнения прецедентов только посредством открытых интерфейсов.

Открытые интерфейсы впервые определяются ближе к окончанию этапа анализа, когда спецификации состояний и поведения в значительной степени определены. В ходе анализа мы определяем только *сигнатуру* каждой открытой операции (имя операции, список формальных аргументов, тип возвращаемого значения). В ходе проектирования мы даем определение алгоритмам для метода, реализующего операцию.

4.3.4.1. Выявление операций классов

Операции классов лучше всего определять на основе диаграмм последовательностей. Каждое *сообщение* в модели последовательностей, отличное от возвращаемого

значения, должно быть обслужено операцией в объекте-приемнике (разд. 2.2.5.2). Если модели последовательностей полностью построены, определение открытых операций представляется автоматической задачей.

Однако на практике диаграммы последовательностей — даже если они разработаны для всех прецедентов — могут не обеспечить достаточного уровня детализации, позволяющего выявить все открытые операции. Кроме того, диаграммы последовательностей для операций, которые пересекают границы прецедентов, могут отсутствовать (например, когда деловые операции охватывают более одного прецедента).

Поэтому при выявлении операций могут оказаться полезными вспомогательные методы. Один из таких методов проистекает из того соображения, что объекты отвечают за собственную "судьбу" и поэтому они должны поддерживать четыре элементарных операции.

1. "Создать" (Create).
2. "Читать" (Read).
3. "Обновить" (Update)4.
4. "Удалить" (Delete).

Этот набор операций известен как CRUD-операции (разд. 3.5.2.1). CRUD-операции позволяют другим объектам отправлять сообщение объекту с требованием выполнить следующие действия.

1. Создать новый экземпляр объекта.
2. Получить доступ к состоянию объекта.
3. Модифицировать состояние объекта.
4. Объекту уничтожить самого себя.

4.3.4.2. Спецификация операций классов

На этом этапе ЖЦ разработки ПО необходимо модифицировать диаграмму классов, чтобы ввести в модель *сигнатуры операций*. Здесь можно также определить область действия операций. По умолчанию предполагается использование *области действия экземпляра* (операция применяется к *объектам-экземплярам*). *Область действия-класс* (*статическая*) должна быть объявлена явно (указанием символа "$" перед именем операции). Область действия-класс устанавливает, что операция применима к *объектам-классам* (разд. 2.1.6).

Другие свойства операции, такие как параллельность, полиморфное поведение и спецификация алгоритма, задаются позже в процессе проектирования.

4.3.4.3. Пример спецификации операций классов

Пример 4.18. Запись на университетские курсы

Обратитесь к примерам 4.9. и 4.17, а также классам `Course` и `CourseOffering`, приведенным на рис. 4.6. Объекты этих двух классов использовались в диаграмме последовательностей, показанной на рис. 4.14.

Наша задача заключается в том, чтобы получить операции на основе диаграммы последовательностей и ввести их в классы `Course` и `CourseOffering`.

Операции, заданные в расширенной модели классов на рис. 4.15, получены непосредственно из диаграммы последовательностей, показанной на рис. 4.14. Пиктограммы перед атрибутами означают, что атрибуты обладают *закрытой* областью действия. Пиктограммы перед операциями говорят о том, что они *открытые*.

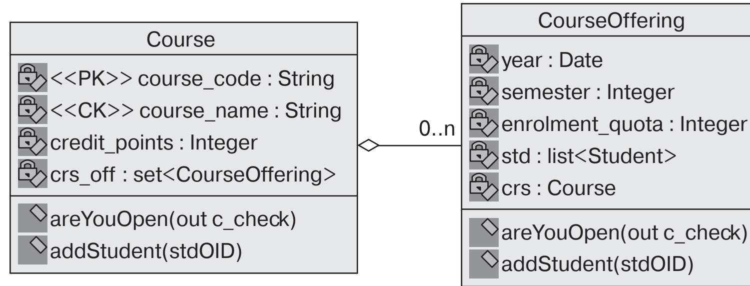

Рис. 4.15. Спецификация операций классов (Запись на университетские курсы)

Вот перечень операций: `Course.areYouOpen`, `CourseOffering.areYouOpen`, `Course.addStudent` и `CourseOffering.addStudent`. Предполагается, что объект `Course` получает список студентов с помощью доступа к своим составным объектам класса `CourseOffering`.

Хотя атрибуты отношения обычно не приводятся в моделях анализа (они неявно входят в связь, обозначающую отношение), мы включили три атрибута отношения, необходимых операциям. Вот эти атрибуты: `Course.crs_off`, `CourseOffering.std` и `CourseOffering.crs`. Заметим, что атрибуты `crs_off` и `std` обладают *параметризованными типами*; они являются коллекциями (`set` и `list`) объектов.

4.4. Спецификация изменения состояний

Состояние объекта в определенный момент времени определяется значениями его атрибутов, включая атрибуты отношения. Спецификация состояний, помимо прочего, определяет атрибуты класса. Спецификация поведения определяет операции класса, некоторые из которых дают *побочные эффекты* (т.е. они изменяют состояние объекта). Однако для понимания того, каким образом объект может изменять свое состояние во времени, нам необходим более целенаправленный взгляд на систему. Подобный взгляд может обеспечить спецификация изменения состояний.

Значение спецификации изменения состояний изменяется в зависимости от области приложения. Моделирование изменения состояний для бизнес-приложений значительно менее критично, чем инженерных приложений и приложений реального времени. Многие инженерные приложения и приложения реального времени полностью определяются изменением состояний объектов. При моделировании подобных систем необходимо с первого дня сосредоточиться на изменении состояний. Что случится, если температура окажется слишком высокой? Что будет, если клапан не закроется? Что произойдет, если контейнер переполнится? И т.д.

В этом учебнике мы преимущественно сосредоточили свое внимание на бизнес-приложениях, для которых изменения состояния наблюдаются менее часто. Поэтому моделирование изменения состояний обычно осуществляется ближе к окончанию этапа анализа (и продолжается со значительно большей глубиной на этапе проекти-

рования). Многие спецификации изменения состояний определяют исключительные условия в системе. Естественно, что исключения из обычного поведения системы моделируются после спецификации нормального поведения.

4.4.1. Моделирование состояний объектов

Моделирование состояний объектов осуществляется с помощью диаграмм состояний. Граф состояний (автомат) — это граф состояний и переходов. Модели состояний строятся для каждого класса, проявляющего интересное для аналитика динамическое поведение. Не все классы диаграммы классов относятся к этой категории.

Диаграммы состояний можно также использовать для описания динамического поведения других моделируемых элементов, например, прецедентов, коопераций или операций. Это, однако, встречается нечасто, и некоторые CASE-средства могут не поддерживать подобные функциональные возможности.

4.4.1.1. Выявление состояний объектов

Процесс исследования состояний объектов основан на анализе содержимого атрибутов классов и выявлении того, какие из этих атрибутов представляют особый интерес с точки зрения прецедентов. Не все атрибуты влияют на изменения состояний.

Например, модификация телефонного номера клиентом не изменяет состояния объекта `Customer`. У клиента по-прежнему имеется телефон, а номер телефона не существенен. Однако удаление номера телефона может трактоваться как изменение состояния с точки зрения некоторых прецедентов. Больше связаться с клиентом по телефону нельзя.

Аналогично, изменение номера телефона, которое включает изменение кода региона может представлять интерес как изменение состояния, поскольку указывает на то, что клиент сменил свое географическое местоположение, и это изменение может быть необходимо зафиксировать и отразить в диаграмме состояний.

4.4.1.2. Спецификация состояний объектов

Основные обозначения языка UML, касающиеся спецификации диаграмм состояний, были введены в разделе 2.2.6. Для успешного применения этой системы обозначений аналитик должен понимать взаимосвязи между различными понятиями, какие сочетания понятий являются неестественными или недопустимы и какие сокращения возможны в рамках предлагаемой нотации. Применение CASE-средств может быть связано с определенными ограничениями в представлении диаграмм состояний, но вполне возможны и некоторые любопытные расширения.

Переходы между состояниями активизируются при появлении определенных событий *или* выполнении определенных усилий. Это означает, например, что линия перехода не нуждается в метке, указывающей *имя события*. Само *условие* (записываемое в квадратных скобках) активизирует переход всякий раз, когда *вид деятельности* в состоянии завершен и условие принимает значение “истина”.

В типичной ситуации переход запускается сигнальным событием или событием вызова. *Сигнальное событие* устанавливает явную, асинхронную однонаправленную связь между двумя объектами. Событие вызова устанавливает синхронную связь, в которой вызывающий объект ожидает ответа. Существует еще два типа пусковых событий: *событие изменения* и *временное событие*. В частности, временное событие, запускающее переход, основано на абсолютном или относительном понятии времени и является очень полезным в некоторых моделях.

Еще одно соображение, касающееся моделирования состояний, связано с возможностью спецификации входных действий внутри пиктограммы состояния или во входящем переходе. Аналогично, выходное действие можно поместить внутри пиктограмм состояний или в исходящих переходах. Хотя это и не затрагивает семантики, выбор используемого метода может сказаться на удобстве восприятия модели [74].

4.4.1.3. Пример спецификации диаграммы состояний

Пример 4.19. Магазин видеопроката

Обратитесь к примеру 4.9 и рассмотрите класс `MovieTitle` (рис. 4.7). Наша задача состоит в спецификации диаграммы состояний для класса `MovieTitle`.

Диаграмма состояний для класса `MovieTitle` приведена на рис. 4.16. Диаграмма иллюстрирует различные способы спецификации переходов. Переходы между состояниями `Available` (В наличии) и `Not in Stock` (Нет в запасе) задают параметризованные имена событий наряду с именами действий и защитными условиями. С другой стороны, переход из состояния `Reserved` (Забронировано) в `Not Reserved` (Не забронировано) лишен явного пускового события. Этот переход запускается посредством завершения вида деятельности в состоянии `Reserved`, в результате которого условие отказа в бронировании `[no more reserved]` принимает значение “истина”.

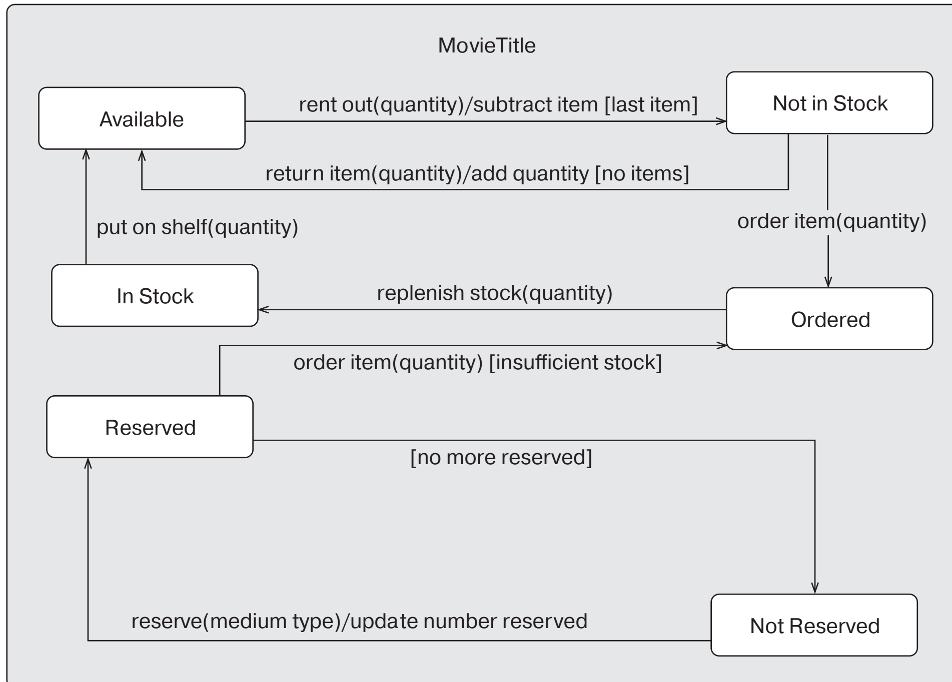

Рис. 4.16. Диаграмма состояний для класса “Movie Title” (Магазин видеопроката)

Резюме

Эта глава является ключевой для всей книги — здесь показан UML в действии. Разбор примеров, введенных в главе 2, послужил в качестве иллюстративного материала. В дело были пущены все часто используемые UML-модели. Девиз этой главы мог бы звучать так: в сфере моделей анализа ИС однозначных решений типа "черное-белое", "нуль-единица", "истина-ложь" не существует. Для каждой проблемы существует множество потенциальных решений. Фокус состоит в том, чтобы придти к решению, которое удовлетворяет требованиям заказчика и способно работать.

- Спецификация состояний описывает мир ИС со статической точки зрения *классов*, содержания их *атрибутов* и их *отношений*. Существует много методов выявления классов, но ни один из них не дает единого рецепта. Практическим ответом на проблему выявления классов служит комплекс методов, соответствующий знаниям и опыту аналитика. В UML классы задаются с помощью *диаграмм классов*. Диаграммы позволяют наглядно представить классы и три типа отношений между ними: *ассоциации*, *агрегации* и *обобщения*.
- Спецификация поведения описывает мир ИС с *операционной* точки зрения (для того, чтобы не использовать перегруженный термин — *функциональная* точка зрения). Движущей силой спецификации поведения и, конечно, анализа требований и проектирования системы в целом выступают *прецеденты*. *Диаграммы прецедентов* дают только наглядное представление — истинная сила прецедентов заключается в их *неформальных спецификациях*. Остальные поведенческие диаграммы являются производными от моделей прецедентов. Они включают *диаграммы видов деятельности*, *диаграммы взаимодействия* и введение *операций* в классы.
- Спецификации изменения состояний описывают мир ИС с *динамической* точки зрения. Объекты "бомбардируются" событиями, и некоторые из этих событий вызывают изменения *состояний* объектов. Диаграммы состояний позволяют моделировать изменения состояний.

Вопросы

В1. Обсудите, каким образом функциональные требования и требования к данным, выявленные на этапе установления требований, связаны с моделями состояний, поведения и изменения состояний, разрабатываемыми на этапе спецификации требований.

В2. Приведите доводы за и против использования CASE-средств для спецификации требований.

В3. Объясните, в чем заключаются основные различия четырех подходов к выявлению классов.

В4. Обсудите и сравните процессы, связанные с выявлением и спецификацией атрибутов классов и операций классов. Как следует моделировать атрибуты и операции: в параллель или по отдельности? Почему?

В5. Обратитесь к модели классов на рис. 4.7 (пример 4.10). Объясните, почему атрибут `MovieTitle.is_in_stock` моделируется как производный атрибут, а атрибут `VideoMedium.percentage_excellent_condition` — как статический.

В6. Использование тернарных и производных ассоциаций в моделях анализа нежелательно. Почему? Приведите примеры.

В7. Обратитесь к модели классов на рис. 4.5 (пример 4.8). Рассмотрите верхнюю часть модели, в которой определяются классы и ассоциации между классами `PostalAddress`,

CourierAddress, Organization и Contact. Предложите альтернативные пути моделирования этих же требований. Можно ли придать модели большую гибкость, чтобы ее можно было приспособить к возможным изменениям в требованиях? Приведите доводы за и против альтернативных решений.

В8. Приведите примеры каждого из типов агрегаций: *Безраздельно обладает*, *Обладает*, *Включает* и *Участник*.

В9. Обратитесь к модели классов на рис. 4.6 (пример 4.9). Объясните, как изменится семантика модели, если класс AcademicRecord, связанный с ассоциацией takes, моделировать как "ассоциацию в роли класса"?

В10. Наследование без полиморфизма возможно, но не очень полезно. Почему? Приведите примеры.

В11. Какие из моделей UML полезны при спецификации поведения? Объясните, в чем заключаются сильные стороны каждой из этих моделей и каким образом они взаимосвязаны при определении поведения системы?

В12. Объясните, в чем состоит различие прецедентов и деловых функций (или деловых транзакций)?

В13. Приведите пример отношений <<include>> и <<extend>> между прецедентами. В чем их основное отличие?

В14. Приведите пример класса с несколькими атрибутами. Обсудите, какие из атрибутов могут инициировать переход между состояниями, а какие из атрибутов индифферентны к изменениям состояний?

Упражнения

Дополнительные требования. Запись на университетские курсы

Рассмотрите следующие дополнительные требования к приложению *Запись на университетские курсы*.

1. Университет разделен на факультеты. Факультеты разделены на кафедры. Преподаватели работают на одной из кафедр.
2. Присвоение большинства ученых степеней является прерогативой одного факультета, но некоторые ученые степени находятся в совместном ведении двух и более факультетов.
3. Новые студенты получают уведомление о приеме и инструкции по записи на курсы по почте. Студенты, продолжающие учебу, которые имеют право перезаписаться на курсы, получают (также по почте) инструкции по записи вместе в уведомлением об экзаменационных результатах.
4. Инструкции по записи на курсы включают расписание аудиторий для предлагаемого курса.
5. В период записи на курсы студенты могут получить консультацию у научного руководителя на факультете, на котором они числятся, по вопросу составления программы обучения.
6. Студенты не ограничены в изучении только тех курсов, которые предлагаются их факультетом, и в любой момент могут сменить факультет, на котором они числятся (заполнив форму изменения программы, которую можно получить в деканате).
7. Чтобы взять определенные курсы, студент должен сначала пройти обязательные курсы, получив требуемые оценки (простого прохождение курса может быть недостаточно). Студент, который не смог удовлетворительно пройти курс, может попы-

таться сделать это вновь. Для записи на курс в третий раз или в случае требования освобождения от обязательных курсов необходимо получить согласие представителя соответствующего деканата.

8. Если студенты записываются на те учебные курсы, которые в совокупности дают за год менее чем 18 условных очков, эти студенты относятся к категории студентов "вечерней" формы обучения.
9. Для того, чтобы получить возможность записаться на программу курсов, которые в данном семестре дают в совокупности более 14 условных очков, требуется специальное разрешение.
10. За каждым курсом обучения закреплен ответственный преподаватель, однако к нему могут быть привлечены дополнительные преподаватели. В каждом семестре за курс может отвечать другой преподаватель, а каждый курс в течение семестра могут вести разные преподаватели.

Дополнительные требования. Магазин видеопроката

Рассмотрите следующие дополнительные требования к приложению *Магазин видеопроката*.

1. За кассеты и диски, возвращенные позже срока, взимается дополнительная плата за период, превышающий срок проката. Каждый видеоноситель обладает уникальным идентификационным номером.
2. Фильмы заказываются у поставщика, который, в общем случае, может поставить кассеты и диски в течение одной недели. Обычно один заказ делается на несколько фильмов.
3. Забронировать можно те фильмы, которые заказаны у поставщика и/или все копии которых находятся в прокате. Можно также забронировать те фильмы, которых нет в запасе и которые не заказаны у поставщика; при этом с клиента требуется задаток за один период проката.
4. Клиент может также сделать несколько предварительных заказов, однако для каждого забронированного фильма готовится отдельный запрос на бронирование. Бронирование может быть отменено из-за отсутствия реакции со стороны клиента, более точно, в течение одной недели с момента, когда клиенту было сообщено о возможности взять фильм напрокат. Если за фильм был уплачен задаток, он записывается на счет клиента.
5. База данных хранит обычную информацию о поставщиках и клиентах, т.е. адреса, телефонные номера и т.д. В каждом заказе поставщику указываются заказываемые фильмы, их количество, форматы кассеты/диска, а также дата ожидаемой доставки, отпускная цена, возможные скидки и т.д.
6. Когда кассета возвращается клиентом или поступает от поставщика, вначале удовлетворяются предварительные заказы. Работники магазина устанавливают контакт с клиентами, сделавшими предварительный заказ. Для правильной обработки бронирования фильмов информация, связанная с бронированием, обновляется дважды: после установления контакта с клиентом, когда ему сообщается, что "забронированный фильм пришел", и после сдачи фильма клиенту напрокат. Эти шаги гарантируют правильное проведение операции бронирования.
7. Клиент может взять несколько кассет или дисков, однако каждому взятому видеоносителю ставится в соответствие отдельная запись. Для каждого выдаваемого напрокат фильма фиксируются дата и время выдачи, установленный и фактический срок возврата. Позже запись о покате обновляется, чтобы отразить факт возврата видеофильма и факт окончательного платежа (или возврата денег). Кроме того, запись хранит информацию о продавце, отвечающем за прокат фильма. Детальная информация о клиенте и по прокату хранится в течение года, чтобы можно было легко оп-

ределить уровень доверия к клиенту. Старая информация по прокату сохраняется в течение года в целях проведения аудита.

8. Все операции выполняются с использованием наличности, электронного перевода денег или кредитных карточек. От клиентов требуется внести плату за прокат при выдаче кассет/дисков.
9. Если кассета/диск возвращены позже установленного срока (или не могут быть возвращены по каким-либо причинам) плата снимается либо со счета клиента, либо принимается непосредственно от клиента.
10. Если кассета/диск задержаны более чем на два дня, клиенту отправляется уведомление о задержке. После отправки двух уведомлений о задержке одной и той же кассеты/диска, клиент предупреждается о том, что он является “нарушителем” и при следующем его обращении в магазин руководство рассматривает вопрос о снятии с него статуса “нарушителя”.

У1. *Запись на университетские курсы* — обратитесь к приведенным выше дополнительным требованиям и к примеру 4.1 (раздел 4.2.1.1.7).

Какие новые классы можно получить на основе анализа расширенных требований?

У2. *Магазин видеопроката* — обратитесь к приведенным выше дополнительным требованиям и к примеру 4.2 (раздел 4.2.1.1.7).

Какие новые классы можно получить на основе анализа расширенных требований?

У3. *Запись на университетские курсы* — обратитесь к приведенным выше дополнительным требованиям, упражнению У1 и примеру 4.9 (раздел 4.2.3.3).

Предложите, каким образом следует расширить модель классов, приведенную на рис. 4.6 (пример 4.9), чтобы учесть расширенные требования. Покажите классы и отношения.

У4. *Магазин видеопроката* — обратитесь к приведенным выше дополнительным требованиям, упражнению У2 и примеру 4.10 (раздел 4.2.4.3).

Предложите, каким образом следует расширить модель классов, приведенную на рис. 4.7 (пример 4.10), чтобы учесть расширенные требования. Покажите классы и отношения.

У5. *Запись на университетские курсы* — обратитесь к приведенным выше дополнительным требованиям, упражнению, примеру 4.12 (раздел 4.3.1.3).

Предложите, каким образом следует расширить модель прецедентов, приведенную на рис. 4.9 (пример 4.12), чтобы учесть расширенные требования.

У6. *Магазин видеопроката* — обратитесь к приведенным выше дополнительным требованиям и примеру 4.14 (раздел 4.3.1.3).

Исследуйте неформальную спецификацию для прецедента `Rent Video`, приведенную в табл. 4.4 (пример 4.14). Пропустите последний абзац раздела “Основной поток” таблицы (этот абзац относится скорее к прецеденту `Maintain Customer`). Разработайте отдельную диаграмму прецедентов для отображения дочерних прецедентов `Rent Video`.

У7. *Запись на университетские курсы* — обратитесь к приведенным выше дополнительным требованиям и примеру 4.17 (раздел 4.3.3.3).

Предложите, каким образом следует расширить диаграмму последовательностей, приведенную на рис. 4.14 (пример 4.17), чтобы включить проверку обязательных условий записи на курсы — студент записывается на курс только в том случае, если он прошел обязательные курсы.

У8. *Запись на университетские курсы* — обратитесь к примеру 4.18 (раздел 4.3.4.3) и решению к приведенному выше упражнению У7.

Воспользуйтесь расширенной диаграммой последовательностей для введения новых операций в релевантные классы, включая два класса, приведенные на рис. 4.14 (пример 4.18).

У9. *Управление контактами с клиентами* — обратитесь к примеру 4.8 (раздел 4.2.2.3).

Разработайте диаграмму состояний для класса `Event`.

У10. *Телемаркетинг* — обратитесь к примеру 4.7 (раздел4.2.1.2.3).

Добавьте ассоциации классов, пропущенные на рис. 4.4. Определите кратность каждой ассоциации.

У11. *Управление контактами с клиентами* — обратитесь к примеру 4.8 (раздел 4.2.2.3).

Рассмотрите классы `Organization`, `Contact`, `PostalAddress` и `CourierAddress`. В качестве расширения модели, приведенной на рис. 4.5, предусмотрите возможность введения иерархической структуры для организации, т.е. организация может состоять из более мелких организаций. Усильте модель классов за счет использования обобщения и одновременно расширьте ее, чтобы учесть иерархическое строение организации.