UML Основы

Краткое руководство
по стандартному языку
объектного моделирования

Третье издание

Мартин Фаулер

Санкт-Петербург
2005

9

Прецеденты

Прецеденты – это технология определения функциональных требований к системе. **Работа прецедентов заключается в описании типичных взаимодействий между пользователями системы и самой системой и предоставлении описания процесса ее функционирования.** Вместо того чтобы описывать прецеденты в лоб, я предпочитаю подкрасться к ним сзади и начать с описания сценариев. **Сценарий (scenario) – это последовательность шагов, описывающих взаимодействие пользователя и системы.** Поэтому при наличии онлайнового магазина, основанного на веб-сайте, мы можем использовать сценарий «Покупка товара» (Buy a Product), в котором происходит следующее.

> *Покупатель просматривает каталог и помещает выбранные товары в корзину. При желании оплатить покупку он вводит информацию о кредитной карте и производит платеж. Система проверяет авторизацию кредитной карты и подтверждает оплату товара тотчас же и по электронной почте.*

Подобный сценарий описывает только одну ситуацию, которая может иметь место. Однако если авторизация кредитной карты окажется неудачной, то подобная ситуация может послужить предметом уже другого сценария. В другом случае у вас может быть постоянный клиент, для которого проверка информации о покупке и кредитной карте не обязательна, и это будет третий сценарий.

Так или иначе, но все эти сценарии похожи. Суть в том, что во всех трех сценариях у пользователя одна и та же цель: купить товар. Пользователь не всегда может это сделать, но цель остается. Именно цель пользователя является ключом к прецедентам: прецедент представляет собой множество сценариев, объединенных некоторой общей целью пользователя.

В терминах прецедента пользователи называются актерами. **Актер (actor) представляет собой некую роль, которую пользователь играет по отношению к системе. Актерами могут быть пользователь, торговый представитель пользователя, менеджер по продажам и товаровед.**

Актеры действуют в рамках прецедентов. **Один актер может выполнять несколько прецедентов; и наоборот, в соответствии с одним прецедентом могут действовать несколько актеров.** Обычно клиентов много, поэтому роль клиента могут играть многие люди. К тому же один человек может играть несколько ролей, например менеджер по продажам, выполняющий роль торгового представителя клиента. **Актер не обязательно должен быть человеком. Если система предоставляет некоторый сервис другой компьютерной системе, то другая система является актером.**

На самом деле *актер* – не совсем верный термин; возможно, термин *роль* (*role*) подошел бы лучше. Очевидно, имел место неправильный перевод со шведского языка, и в результате сообщество пользователей прецедентов теперь употребляет термин *актер*.

Прецеденты считаются важной частью языка UML. Однако удивительно то, что определение прецедентов в UML довольно скудное. В UML ничего не говорится о том, как определять содержимое прецедента. Все, что описано в UML, – это диаграмма прецедентов, которая показывает, как прецеденты связаны друг с другом. Но почти вся ценность прецедентов как раз в их содержании, а диаграмма имеет ограниченное значение.

Содержимое прецедентов

Не существует стандартного способа описания содержимого прецедента; в разных случаях применяются различные форматы. На рис. 9.1 показан общий стиль использования. **Вы начинаете с выбора одного из сценариев в качестве главного успешного сценария (main success scenario). Сначала вы описываете тело прецедента, в котором главный успешный сценарий представлен последовательностью нумерованных шагов. Затем берете другой сценарий и вставляете его в виде расширения (extension), описывая его в терминах изменений главного успешного сценария.** Расширения могут быть успешными – пользователь достиг своей цели, как в варианте 3а, или неудачными, как в варианте 6а.

В каждом прецеденте есть ведущий актер, который посылает системе запрос на обслуживание. Ведущий актер – это актер, желание которого пытается удовлетворить прецедент и который обычно, но не всегда, является инициатором прецедента. Одновременно могут быть и другие актеры, с которыми система также взаимодействует во время выполнения прецедента. Они называются второстепенными актерами.

Каждый шаг в прецеденте – это элемент взаимодействия актера с системой. Каждый шаг должен быть простым утверждением и должен четко указывать, кто выполняет этот шаг. Шаг должен показывать намерение актера, а не механику его действий. Следовательно, в прецеденте интерфейс актера не описывается. Действительно, составление прецедента обычно предшествует разработке интерфейса пользователя.

Покупка товара
Целевой уровень: уровень моря

Главный успешный сценарий:
1. Покупатель просматривает каталог и выбирает товары для покупки.
2. Покупатель оценивает стоимость всех товаров.
3. Покупатель вводит информацию, необходимую для доставки товара (адрес, доставка на следующий день или в течение трех дней).
4. Система предоставляет полную информацию о цене товара и его доставке.
5. Покупатель вводит информацию о кредитной карточке.
6. Система осуществляет авторизацию счета покупателя.
7. Система подтверждает оплату товаров немедленно.
8. Система посылает подтверждение оплаты товаров по адресу электронной почты покупателя.

Расширения:
3а. Клиент является постоянным покупателем.
.1: Система предоставляет информацию о текущей покупке и ее цене, а также информацию о счете.
.2: Покупатель может согласиться или изменить значения по умолчанию, затем возвращаемся к шагу 6 главного успешного сценария.
6а. Система не подтверждает авторизацию счета.
.1: Пользователь может повторить ввод информации о кредитной карте или закончить сеанс.

Рис. 9.1. Пример текста прецедента

Расширение внутри прецедента указывает условие, которое приводит к взаимодействиям, отличным от описанных в главном успешном сценарии (main success scenario, MSS), и устанавливает, в чем состоят эти отличия. Расширение начинается с имени шага, на котором определяется это условие, и предоставляет краткое описание этого условия. Следуйте этому условию, нумеруя шаги таким же образом, что и в главном успешном сценарии. Заканчивайте эти шаги описанием точки возврата в главный успешный сценарий, если это необходимо.

Структура прецедента – это отличный инструмент для поиска альтернатив главного успешного сценария. На каждом шаге спрашивайте: «Что может еще произойти?» и в частности «Что может пойти не так?» Обычно лучше сначала изучить все возможные условия расширения, чтобы потом не увязнуть в трясине работы над последствиями. Таким образом, вы, возможно, обдумаете больше условий, что приведет к меньшему количеству ошибок, которые потом пришлось бы отлавливать.

Сложный шаг в прецеденте можно представить другим прецедентом. В терминах языка UML мы говорим, что первый прецедент включает (includes) второй. Не существует стандартного способа показать в тексте

включение прецедента, но я думаю, что подчеркивание, которое предполагает гиперссылку, работает прекрасно, а во многих инструментах действительно будет гиперссылкой. Так, на рис. 9.1 первый шаг включает шаблон «просматривает каталог и выбирает товары для покупки».

Включенные прецеденты могут быть полезными в случае сложных шагов, которые иначе загромождали бы главный сценарий, или когда одни и те же шаги присутствуют в нескольких сценариях. Однако не пытайтесь разбивать прецеденты на подпрецеденты и использовать их для функциональной декомпозиции. Такая декомпозиция – хороший способ потерять много времени.

Наряду с шагами сценария можно вставить в прецедент дополнительную общую информацию.

- **Предусловие (pre-condition) описывает действия, обязательно выполняемые системой перед тем, как она разрешит начать работу прецедента. Это полезная информация, позволяющая разработчикам не проверять некоторые условия в их программе.**
- **Гарантия (guarantee) описывает обязательные действия системы по окончании работы шаблона ответа. Успешные гарантии выполняются после успешного сценария; минимальные гарантии выполняются после любого сценария.**
- **Триггер (trigger) определяет событие, инициирующее выполнение прецедента.**

При рассмотрении дополнительных элементов относитесь к этому скептически. Лучше сделать слишком мало, чем слишком много. Кроме того, приложите максимум усилий, чтобы сделать прецедент кратким и легким для чтения. Я убедился, что излишне подробный прецедент, который трудно читать, скорее приведет к провалу, чем к достижению цели. Не обязательно записывать все детали; устное общение часто бывает очень эффективным, особенно во время итеративного цикла, когда необходимые условия быстро выполняются запущенной программой.

Степень детализации, необходимая в прецеденте, зависит от уровня риска этого прецедента. Часто детали нужны в начале только немногих ключевых прецедентов, другие можно конкретизировать непосредственно перед их реализацией.

Диаграммы прецедентов

Я уже говорил, что язык UML умалчивает о содержимом прецедента, но предоставляет формат диаграммы, позволяющий его отображать (рис. 9.2). Хотя диаграмма иногда оказывается полезной, без нее можно обойтись. **При разработке прецедента не стоит прилагать много усилий для создания диаграммы. Вместо этого лучше сконцентрироваться на текстовом содержании прецедентов.**

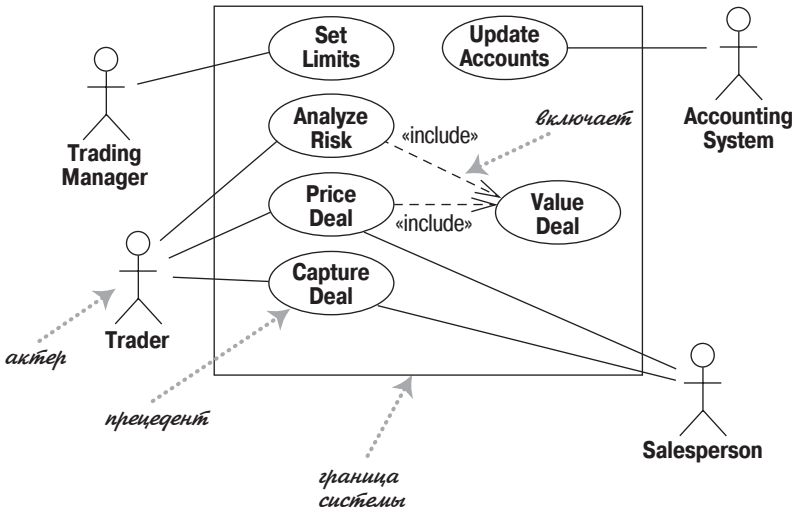

Рис. 9.2. Диаграмма прецедентов

Лучше всего обдумывать диаграмму прецедентов с помощью графической таблицы, показывающей их содержимое. Она напоминает диаграмму контекста, используемую в структурных методах, поскольку она показывает границы системы и ее взаимодействие с внешним миром. **Диаграмма прецедентов показывает актеров, прецеденты и отношения между ними:**

- **Какие актеры выполняют тот или иной прецедент**
- **Какие прецеденты включают другие прецеденты**

В языке UML помимо отношения `«include»` **(включает) есть и другие типы отношений между прецедентами, например отношение** `«extend»` **(расширяет). Я настоятельно рекомендую его избегать. Слишком часто на моих глазах разработчики целыми командами надолго погружались в рассмотрение различных отношений между прецедентами, понапрасну растрачивая силы.** Лучше уделяйте больше внимания текстовому описанию прецедента; именно в этом заключается истинная ценность этой технологии.

Уровни прецедентов

Общая проблема прецедентов состоит в том, что, увлекшись взаимодействием пользователя с системой, можно не обратить внимание на тот факт, что лучшим способом решения проблемы может быть изменение самого бизнес-процесса. **Часто можно слышать упоминание о прецедентах системы и прецедентах бизнес-процессов. Конечно, эта терминология не является точной, но обычно считается, что прецедент системы (system use case) описывает особенности взаимодействия с программ-**

ным обеспечением, тогда как прецедент бизнес-процесса (business use case) представляет собой реакцию бизнес-процесса на действие клиента или некоторое событие.

В книге Кокборна [10] предлагается схема уровней прецедентов. Базовый прецедент находится на «уровне моря». **Прецеденты уровня моря (sea level) обычно представляют отдельное взаимодействие ведущего актера и системы.** Такие прецеденты предоставляют ведущему актеру какой-либо полезный результат и обычно занимают от пары минут до получаса. Прецеденты, которые существуют в системе, только если они включены в прецеденты уровня моря, называются прецедентами **уровня рыб** (fish level). **Прецеденты высшего уровня, уровня воздушного змея (kite-level), показывают, как прецеденты уровня моря настраиваются на более широкое взаимодействие с бизнес-процессами. Обычно прецеденты уровня воздушного змея являются прецедентами бизнес-процессов, а на уровне моря и на уровне рыб находятся прецеденты системы. Большинство ваших прецедентов должно принадлежать уровню моря. Я предпочитаю указывать уровень в начале прецедента, как показано на рис. 9.1.**

Прецеденты и возможности (или пожелания)

Во многих подходах возможности системы применяются для описания требований к системе; в экстремальном программировании (Extreme Programming) **возможности системы называются пожеланиями пользователя.** Общим является вопрос о том, как установить соответствие между возможностями и прецедентами.

Использование возможностей – это хороший способ разделения системы на блоки при планировании итеративного процесса, в результате чего каждая итерация предоставляет определенное количество возможностей. Отсюда следует, что хотя оба приема описывают требования, их цели различны.

Описать возможности можно сразу, но многие специалисты считают более удобным в первую очередь разработать прецеденты, а уже затем сгенерировать список возможностей. **Возможность может быть представлена целым прецедентом, сценарием в прецеденте, шагом в прецеденте или каким-либо вариантом поведения, например добавление еще одного метода вычисления амортизации при оценке вашего имущества, который не указан в описании прецедента.** Обычно возможности получаются более четко определенными, чем прецеденты.

Когда применяются прецеденты

Прецеденты представляют собой ценный инструмент для понимания функциональных требований к системе. Первый вариант прецедентов должен составляться на ранней стадии выполнения проекта. Более

подробные версии прецедентов должны появляться непосредственно перед реализацией данного прецедента.

Важно помнить, что **прецеденты представляют взгляд на систему *со стороны*.** А раз так, то не ждите какого-либо соответствия между прецедентами и классами внутри системы.

Чем больше прецедентов я вижу, тем менее ценной мне кажется диаграмма прецедентов. Несмотря на то что в языке UML ничего не говорится о тексте прецедентов, именно **текстовое содержание прецедентов является основной ценностью этой технологии**.

Большая опасность прецедентов заключается в том, что разработчики делают их очень сложными и застревают на них. Обычно чем меньше вы делаете, тем меньший вред можете нанести. Если у вас немного информации, то получится короткий, легко читаемый документ, который явится отправной точкой для вопросов. Если информации слишком много, то вряд ли кто-то вообще будет ее изучать и пытаться понять.

10

Диаграммы состояний

Диаграммы состояний (state machine diagrams) – это известная технология описания поведения системы. В том или ином виде диаграммы состояний существуют с 1960 года, и на заре объектно-ориентированного программирования они применялись для представления поведения системы. В объектно-ориентированных подходах вы рисуете диаграмму состояний единственного класса, чтобы показать поведение одного объекта в течение его жизни.

Всякий раз, когда пишут о конечных автоматах, в качестве примеров неизбежно приводят системы круиз-контроля или торговые автоматы. Поскольку они мне слегка наскучили, я решил использовать контроллер секретной панели управления в Готическом замке. В этом замке я хочу так спрятать свои сокровища, чтобы их было трудно найти. Для того чтобы получить доступ к замку сейфа, я должен вытащить из канделябра стратегическую свечу, но замок появится, только если дверь закрыта. После появления замка я могу вставить в него ключ и открыть сейф. Для дополнительной безопасности я сделал так, чтобы сейф можно было открыть только после возвращения свечи. Если вор не обратит внимания на эту предосторожность, то я спущу с цепи отвратительного монстра, который проглотит вора.

На рис. 10.1 показана диаграмма состояний класса контроллера, который управляет моей необычной системой безопасности. Диаграмма состояния начинается с состояния создаваемого объекта контроллера: состояния `Wait` (Ожидание). На диаграмме это обозначено с помощью начального псевдосостояния (initial pseudostate), которое не является состоянием, но имеет стрелку, указывающую на начальное состояние.

На диаграмме показано, что контроллер может находиться в одном из трех состояний: `Wait` (Ожидание), `Lock` (Замок) и `Open` (Открыт). На диаграмме также представлены правила, согласно которым контроллер переходит из одного состояния в другое. Эти правила представлены в виде переходов – линий, связывающих состояния.

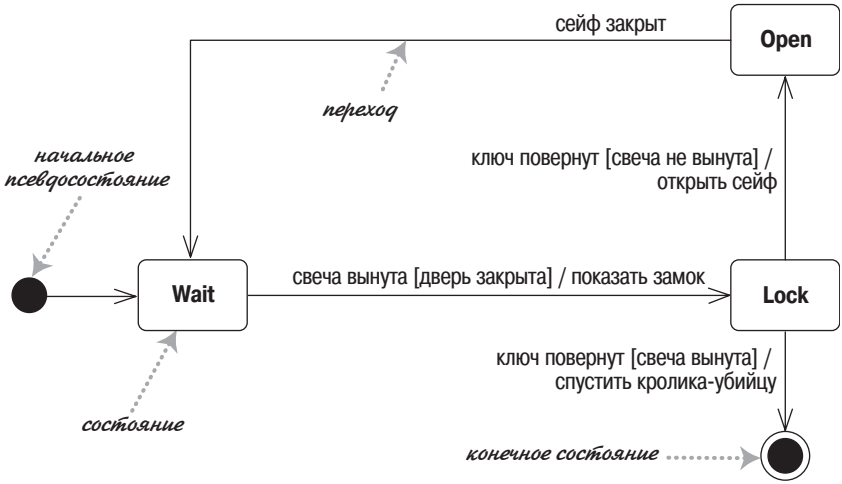

Рис. 10.1. Простая диаграмма состояний (альтернативная реализация)

Переход (transition) означает перемещение из одного состояния в другое. Каждый переход имеет свою метку, которая состоит из трех частей: `триггер-идентификатор [защита]/активность` **(**`trigger-signature [guard]/activity`**). Все они не обязательны. Как правило,** `триггер-идентификатор` **— это единственное событие, которое может вызвать изменение состояния.** `Защита`**, если она указана, представляет собой логическое условие, которое должно быть выполнено, чтобы переход имел место.** `Активность` **— это некоторое поведение системы во время перехода. Это может быть любое поведенческое выражение. Полная форма** `триггера-идентификатора` **может включать несколько событий и параметров. Переход из состояния** `Wait` **(рис. 10.1) в другое состояние можно прочесть как «В состоянии** `Wait`**, если свеча удалена, вы видите замок и переходите в состояние** `Lock`**».**

Все три части описания перехода не обязательны. Пропуск активности означает, что в процессе перехода ничего не происходит. Пропуск защиты означает, что переход всегда осуществляется, если происходит инициирующее событие. Триггер-идентификатор отсутствует редко, но и так бывает. Это означает, что переход происходит немедленно, что можно наблюдать главным образом в состояниях активности, о чем я расскажу в нужный момент.

Когда в определенном состоянии происходит событие, то из этого состояния можно совершить только один переход, например в состоянии `Lock` **(рис. 10.1) защиты должны быть взаимно исключающими. Если событие происходит, а разрешенных переходов нет – например закрытие сейфа в состоянии** `Wait` **или удаление свечи при открытой двери, – событие игнорируется.**

Конечное состояние (final state) означает, что конечный автомат закончил работу, что вызывает удаление объекта контроллера. Так что

для тех, кто имел неосторожность попасть в мою ловушку, сообщаю, что поскольку объект контроллера прекращает свое существование, я вынужден посадить кролика обратно в клетку, вымыть пол и перегрузить систему.

Помните, что **конечные автоматы могут показывать только те объекты, которые непосредственно наблюдаются или действуют.** Поэтому, хотя вы могли ожидать, что я положу что-нибудь в сейф или что-нибудь возьму оттуда, когда дверь открыта, я не отметил это на диаграмме, поскольку контроллер об этом ничего сообщить не может.

Когда разработчики говорят об объектах, они часто ссылаются на состояние объектов, имея в виду комбинацию всех данных, содержащихся в полях объектов. Однако состояние на диаграмме конечного автомата является более абстрактным понятием состояния; суть в том, что различные состояния предполагают различные способы реакции на события.

Внутренние активности

Состояния могут реагировать на события без совершения перехода, используя внутренние активности (internal activities), и в этом случае событие, защита и активность размещаются внутри прямоугольника состояния.

На рис. 10.2 представлено состояние с внутренними активностями символов и событиями системы помощи, которые вы можете наблюдать в текстовых полях редактора UI. Внутренняя активность подобна **самопереходу** (self-transition) – переходу, который возвращает в то же самое состояние. Синтаксис внутренних активностей построен по той же логической схеме события, защиты и процедуры.

На рис. 10.2 показаны также специальные активности: входная и выходная активности. Входная активность выполняется всякий раз, когда вы входите в состояние; выходная активность – всякий раз, когда вы покидаете состояние. Однако внутренние активности не инициируют входную и выходную активности; в этом состоит различие между внутренними активностями и самопереходами.

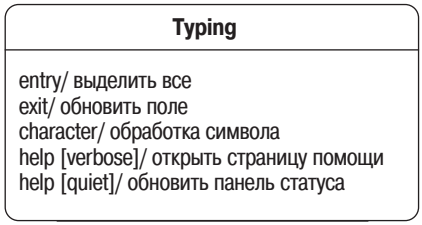

Рис. 10.2. Внутренние события, показанные в состоянии набора текста в текстовом поле

Состояния активности

В состояниях, которые я описывал до сих пор, объект молчит и ожидает следующего события, прежде чем что-нибудь сделать. **Однако возможны состояния, в которых объект проявляет некоторую активность.**

Состояние `Searching` (Поиск) на рис. 10.3 является таким состоянием активности (activity state): ведущаяся активность обозначается символом `do/`; отсюда термин do-activity (проявлять активность). После того как поиск завершен, выполняются переходы без активности, например показ нового оборудования (`Display New Hardware`). Если в процессе активности происходит событие отмены (cancel), то do-активность просто прерывается и мы возвращаемся в состояние `Update Hardware Window` (Обновление окна оборудования).

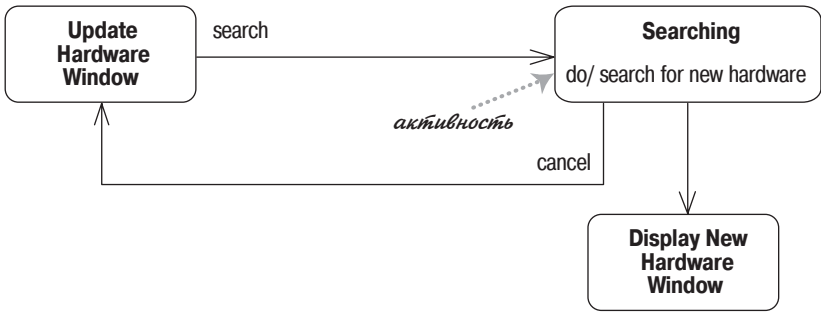

Рис. 10.3. Состояние с активностью

И do-активности, и обычные активности представляют проявление некоторого поведения. Решающее различие между ними заключается в том, что обычные активности происходят «мгновенно» и не могут быть прерваны обычными событиями, тогда как do-активности могут выполняться в течение некоторого ограниченного времени и могут прерываться, как показано на рис. 10.3. Мгновенность для разных систем трактуется по-разному; для систем реального времени это может занимать несколько машинных инструкций, а для настольного программного обеспечения может составить несколько секунд.

В UML 1 обычные активности обозначались термином **action** (действие), а термин **activity** (активность) применялся только для do-активностей.

Суперсостояния

Часто бывает, что несколько состояний имеют общие переходы и внутренние активности. В таких случаях можно их превратить в подсостояния (substates), а общее поведение перенести в суперсостояние (superstate), как показано на рис. 10.4. Без суперсостояния пришлось бы рисовать переход cancel (отмена) для всех трех состояний внутри состояния `Enter Connection Details` (ввод подробностей соединения).

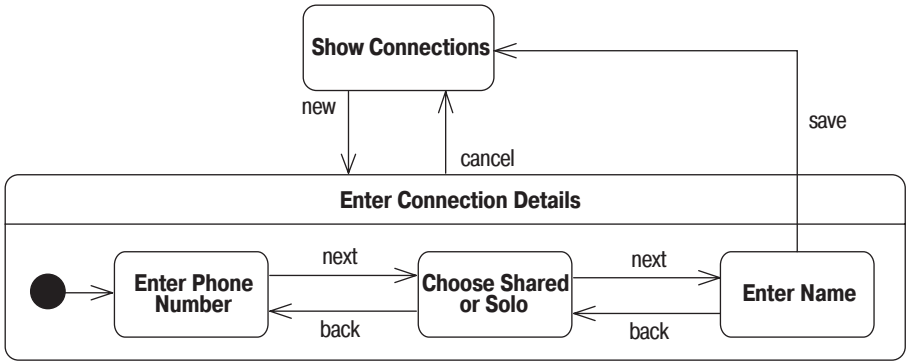

Рис. 10.4. Суперсостояние с вложенными подсостояниями

Параллельные состояния

Состояния могут быть разбиты на несколько параллельных состояний, запускаемых одновременно. На рис. 10.5 показан трогательно простой будильник, который может включать либо CD, либо радио и показывать либо текущее время, либо время сигнала.

Опции CD/радио и текущее время/время сигнала являются параллельными. Если бы вы захотели представить это с помощью диаграммы непараллельных состояний, то получилась бы беспорядочная диаграмма при необходимости добавить состояния. Разделение двух областей поведения на две диаграммы состояний делает ее значительно яснее.

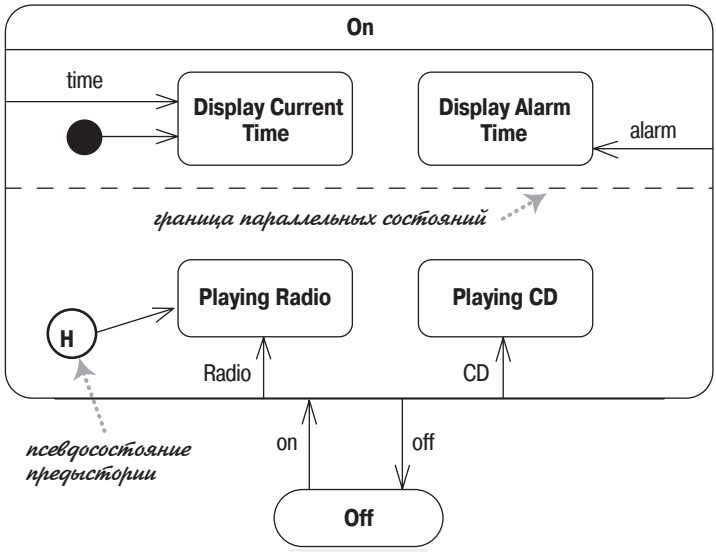

Рис. 10.5. Параллельные состояния

Рис. 10.5 включает также состояние предыстории (history pseudo-state). Это означает, что когда включены часы, опция радио/CD переходит в состояние, в котором находились часы, когда они были выключены. Стрелка, выходящая из предыстории, показывает, какое состояние существовало изначально, когда отсутствовала предыстория.

Реализация диаграмм состояний

Диаграмму состояний можно реализовать тремя основными способами: с помощью вложенного оператора switch, паттерна State и таблицы состояний. Самый прямой подход в работе с диаграммами состояний – это вложенный оператор switch, такой как на рис. 10.6.

```
public void HandleEvent (PanelEvent anEvent) {
   switch (CurrentState) {
      case PanelState.Open :
         switch (anEvent) {
            case PanelEvent.SafeClosed :
               CurrentState = PanelState.Wait;
               break;
         }
         break;
      case PanelState.Wait :
         switch (anEvent) {
            case PanelEvent.CandleRemoved :
               if (isDoorOpen) {
                  RevealLock();
                  CurrentState = PanelState.Lock;
               }
              break;
         }
         break;
      case PanelState.Lock :
         switch (anEvent) {
            case PanelEvent.KeyTurned :
               if (isCandleIn) {
                  OpenSafe();
                  CurrentState = PanelState.Open;
               } else {
                  ReleaseKillerRabbit();
                  CurrentState = PanelState.Final;
               }
               break;
         }
         break;
     }
  }
}
```

Рис. 10.6. Вложенный оператор switch на языке C# для обработки перехода состояний, представленного на рис. 10.1

Хотя этот способ и прямой, но очень длинный даже для этого простого случая. Кроме того, при данном подходе очень легко потерять контроль, поэтому я не люблю применять его даже в элементарных ситуациях.

Паттерн «Состояние» (State pattern) [21] представляет иерархию классов состояний для обработки поведения состояний. Каждое состояние на диаграмме имеет свой подкласс состояния. Контроллер имеет методы для каждого события, которые просто перенаправляют к классу состояния. Диаграмма состояний, показанная на рис. 10.1, могла бы быть реализована с помощью классов, представленных на рис. 10.7.

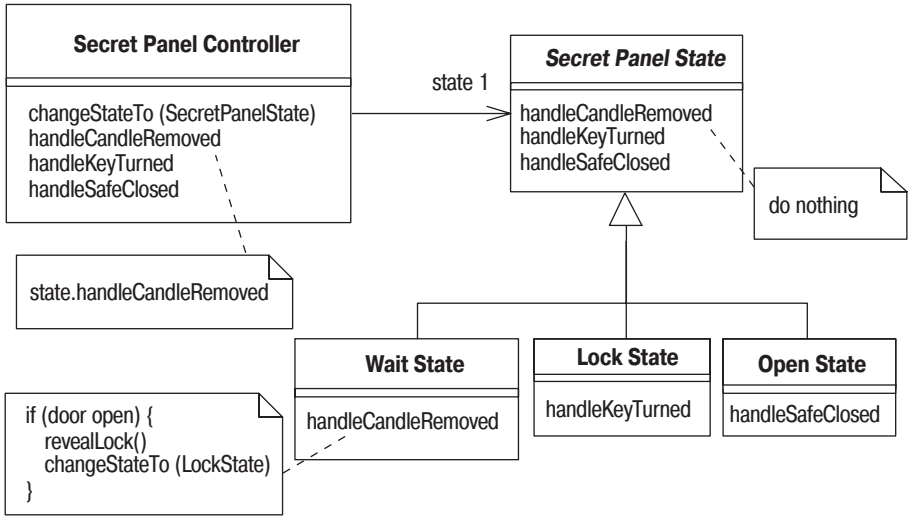

Рис. 10.7. Паттерн «Состояние», реализующий диаграмму на рис. 10.1

Вершиной иерархии является абстрактный класс, который содержит описание всех методов, обрабатывающих события, но без реализации. Для каждого конкретного состояния достаточно переписать метод-обработчик определенного события, инициирующего переход из состояния.

Таблица состояний представляет диаграмму состояний в виде данных. Так, диаграмма на рис. 10.1 может быть представлена в виде табл. 10.1. Затем мы строим интерпретатор, который использует таблицу состоя- ний во время выполнения программы, или генератор кода, который порождает классы на основе этой таблицы.

Очевидно, большая часть работы над таблицей состояний проводится однажды, но затем ее можно использовать всякий раз, когда надо решить проблему, связанную с состояниями. Таблица состояний времени выполнения может быть модифицирована без перекомпиляции, что в некотором смысле удобно. Шаблон состояний собрать легче, и хотя для каждого состояния требуется отдельный класс, но размер кода, который при этом надо написать, совсем невелик.

Таблица 10.1. Таблица состояний для диаграммы на рис. 10.1

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

Приведенные реализации практически минимальные, но они дают представление о том, как применять диаграммы состояний. В каждом случае реализация моделей состояний приводит к довольно стереотипной программе, поэтому обычно для этого лучше прибегнуть к тому или иному способу генерации кода.

Когда применяются диаграммы состояний

Диаграммы состояний хороши для описания поведения одного объекта в нескольких прецедентах. Но они не очень подходят для описания поведения, характеризующегося взаимодействием множества объектов. Поэтому имеет смысл совместно с диаграммами состояний применять другие технологии. Например, диаграммы взаимодействия (глава 4) прекрасно описывают поведение нескольких объектов в одном прецеденте, а диаграммы деятельности (глава 11) хороши для показа основной последовательности действий нескольких объектов в нескольких прецедентах.

Не все считают диаграммы состояний естественными. Понаблюдайте, как специалисты работают с ними. Вполне возможно, что члены вашей команды не думают, что диаграммы состояний подходят для их стиля работы. Это не самая большая трудность; вы должны не забывать совместно использовать различные приемы работы.

Если вы применяете диаграммы состояний, то не старайтесь нарисовать их для каждого класса системы. Такой подход часто применяется в целях формально строгой полноты, но почти всегда это напрасная трата сил. Применяйте диаграммы состояний только для тех классов, которые проявляют интересное поведение, когда построение диаграммы состояний помогает понять, как все происходит. Многие специалисты считают, что редактор UI и управляющие объекты имеют функциональные средства, полезные при отображении с помощью диаграммы состояний.