10

Шаблон Состояние

Государство, которое неспособно
видоизменяться, неспособно и сохраниться.

Эдмунд Берк

Все перевороты Нового времени
приводили к укреплению государства.

Альбер Камю

Шаблон проектирования для создания автоматов

Шаблон проектирования Состояние работает в области различных состояний в приложении, переходов между ними и различного поведения для каждого состояния. Даже в простом приложении управления лампами освещения мы можем обнаружить два состояния: «включено» и «выключено». В выключенном состоянии свет не горит, а во включенном – горит. Переход переключателя света из выключенного состояния во включенное происходит с помощью метода, изменяющего состояние такого приложения, – щелчка по выключателю. Переход из включенного состояние в выключенное происходит с помощью другого метода. Интерфейс содержит в себе все такие переходы, и каждое состояние имплементирует их уникальным для себя образом. Каждый такой метод имплементируется по-своему в зависимости от контекста своего использования. То есть метод включения света `illuminateLight()`, например, будет работать одним способом в выключенном состоянии и совершенно по-другому во включенном, даже хотя он будет входить в оба эти состояния.

Ключевые особенности

Следующие ключевые особенности характеризуют шаблон проектирования Состояние:

- Состояния существуют как внутренние характеристики объекта.
- Объекты меняются определенным образом при смене состояния. Может показаться, что изменяется и класс объекта, но на самом деле меняется лишь его поведение, которое является частью его класса.
- Поведение в каждом состоянии зависит от текущего состояния объекта.

Одним из типов приложений, где шаблон Состояние очень популярен, являются симуляторы различных устройств. Многие устройства, которые меняют состояние объекта, сами изменяются со сменой состояния. Так, ручка громкости радио меняет состояние уровня громкости его звука. Примером более сложного симулируемого устройства может служить панель звукового синтезатора, на которой симулируемые регуляторы изменяют различные свои состояния так, что влияют на работу всего объекта (синтезатора) и на получаемый от него звук. Видеоплеер Flash также имеет несколько состояний клавиш управления: проигрывание, запись, добавление, пауза и стоп. Каждое из состояний видеоплеера ведет себя в зависимости от статуса других состояний, равно как и от своего собственного статуса.

Модель шаблона Состояние

Чтобы понять и оценить значение шаблона проектирования Состояние, мы должны кое-что узнать о *конечных автоматах*. Конечный автомат – это общая теоретическая модель состояний, которые вы будете использовать в своем приложении, применяющем шаблон Состояние. Таким образом, если работа вашего видеоплеера зависит от состояния его клавиш, ваше приложение является конечным автоматом. Кроме того, мы должны упомянуть об автоматных движках, с помощью которых реализуются на практике конечные автоматы. (Вы можете считать чертеж вашего автомобиля аналогом схемы конечного автомата, а сам автомобиль – аналогом автоматного движка, реализующего ее.) Программа, используемая нами для движения от одного состояния к другому, является автоматным движком. Структура данных автоматного движка определяет механизмы обработки внешних сообщений и управления контекстом автомата.

Начнем мы не с наших обычных диаграмм для шаблонов проектирования, а с *диаграмм состояний*. На самом базовом уровне диаграмма состояний – это иллюстрация всех состояний приложения и переходов между ними, и по существу она является моделью для конечного автомата и реализующего его автоматного движка. Возьмем в качестве примера простейшее приложение для видеоплеера, в котором есть только

два состояния: Играть (Play) и Стоп (Stop). Когда приложение запускается в первый раз, оно попадает в состояние Стоп и может только перейти в состояние Играть. Рисунок 10.1 показывает нам соответствующую диаграмму состояний.

Данная иллюстрация нарисована на компьютере. Но, поскольку основная идея использования диаграмм состояний состоит в том, чтобы, начав с грубой исходной идеи, постепенно уточнять ее с помощью серии рисунков, такой процесс пойдет быстрее, если использовать карандаш и бумагу, а не программы для рисования.

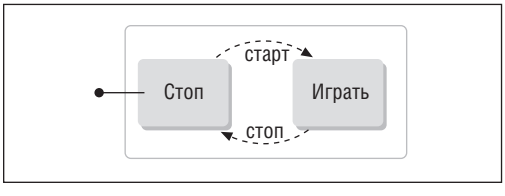

Рис. 10.1. *Простая диаграмма состояний*

Линия, идущая от черной точки к состоянию Стоп, показывает состояние приложения, пока оно не работает, и мы будем полагать, что при запуске оно попадает именно в данное состояние. Это можно было бы проиллюстрировать в виде иерархии состояний работающего и неработающего приложения. Мы могли бы также поместить всю такую иерархию в состояния работающего и выключенного компьютера, но от этого не будет никакой пользы, потому что мы ничего не программируем для этих состояний.

Прежде чем обсуждать переходы из одного состояния в другое, давайте рассмотрим, что каждое из состояний позволяет нам делать. В состоянии Стоп можно только инициировать состояние Играть; остановить воспроизведение нельзя, поскольку оно уже остановлено. А в состоянии Играть можно лишь перейти в состояние Стоп.

Переходы

Переходами в конечных автоматах называют процессы смены состояний. На рис. 10.1 линия, идущая от состояния Стоп к состоянию Играть, может быть методом `startPlay()` такого процесса. А линия от Играть к Стоп может быть методом `stopPlay()`. Если бы состояний было больше, то вы, вероятно, обнаружили бы, что не всегда возможно напрямую перейти из одного состояния в другое, а только пройдя через серию промежуточных состояний. Как вы увидите далее в примере, если вы находитесь в состоянии Стоп, то вы не можете напрямую перейти из него в состояние Пауза. Вам нужно сначала перейти в состояние Играть, и только потом вы сможете попасть в состояние Пауза.

Триггеры

Чтобы инициировать переход в другое состояние, вам нужен какой-либо триггер. Триггер – это любое событие, которое запускает переход из состояния в состояние. Обычно под таким событием мы подразумеваем некоторое действие пользователя приложения, вроде движения мыши или нажатия на кнопку, но при симуляции устройств переход в другое состояние может быть вызван и текущей ситуацией, вроде истощения запасов топлива, заряда батареи или столкновения с объектом. Триггеры зависят от контекста ситуации и могут сработать только при определенных условиях. То есть вы можете использовать кнопку `Играть`, чтобы перейти из состояния `Стоп` в состояние `Играть`, но она не будет инициировать состояние `Играть` из самого этого состояния.

Триггеры часто размещают вместе с переходами на диаграмме состояний. Это помогает определять события, запускающие триггер, и соответствующий переход между состояниями. Рисунок 10.2 показывает нашу уточненную диаграмму состояний, содержащую триггеры и переходы, которые они запускают.

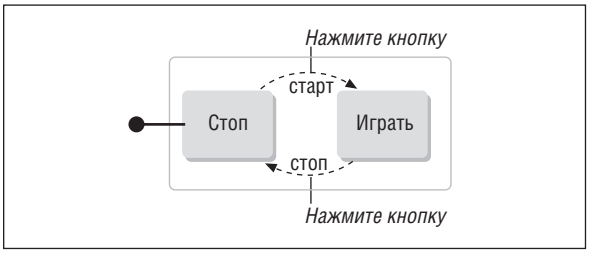

Рис. 10.2. *Диаграмма состояний с переходами и триггерами*

Если вы хотите подробнее изучить тему конечных автоматов, автоматных движков и диаграмм состояний и вопросы реализации работы с состояниями во Flash, то мы рекомендуем вам обратиться к книге «Flash Mx for Interactive Simulation» Джонатана Кайе (Jonathan Kaye) и Дэвида Кастильо (David Castillo) (издательство Thomson, 2003). Хотя в ней и используется Flash, устаревший уже на несколько версий, книга ценна своими концепциями и отличными примерами симуляции некоторых устройств.

Структура шаблона Состояние

Еще в бытность одним из шаблонов проектирования, описанным в книге «банды четырех» «Design Patterns: Elements of Reusable Object-Oriented Software», шаблон Состояние был признан шаблоном, область применения которого вышла за свои первоначальные границы, из области конечных автоматов. Во многом сходный с шаблоном Стратегия шаблон проектирования Состояние может быть использован тогда, когда пове-

дение приложения зависит от изменения его состояний во время работы или имеет сложные условные последовательности операций, зависящие от его текущего состояния. Объект, разработанный с помощью шаблона Состояния, меняет свое поведение при изменении своего внутреннего состояния. Рисунок 10.3 показывает нам общую структуру этого шаблона проектирования с помощью диаграммы классов.

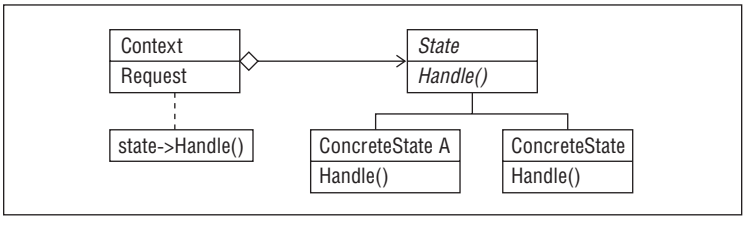

Рис. 10.3. *Шаблон проектирования Состояние*

Ключевые концепции ООП, используемые в шаблоне Состояние

Хотя полиморфизм является основополагающей концепцией ООП, от него не будет пользы, если принципы и задачи его использования поняты не очень хорошо. Полиморфизм – это одна из концепций объектно-ориентированного программирования, применяемая для описания использования различных форм имплементаций объектов. Одной из характеристик полиморфизма, которую вы увидите при изучении шаблона проектирования Состояние, является то, что он достаточно очевиден. Она поможет вам лучше понять, как можно с пользой применять полиморфизм в объектно-ориентированных программах.

Если посмотреть на интерфейс состояния в примере 10.1, то можно увидеть различные методы, которые становятся ядром функциональности для различных состояний. Каждое состояние имеет свой класс. Взглянув на каждый класс, можно заметить, что поведение одних и тех же методов принимает в них различные формы. Налицо работа полиморфизма. Просто посмотрите на различные классы состояний и сравните в них одинаковые по заголовкам методы.

Наверно, одной из самых важных способностей каждого метода состояния является знание своего состояния. Например, в следующих примерах вы увидите, что оба состояния (классы) `Stop` и `Play` имеют у себя метод `startPlay()`. Однако каждый их этих методов ведет себя поразному в своем контексте. Из состояния `Stop` метод `startPlay()` начинает воспроизведение видео. А если его же запустить во время просмотра видео, то он ничего не сделает. Такая пассивность иногда очень важна. Давайте представим, что кто-то смотрел видео и по каким-то причинам нажал на кнопку `Play`. Типичное приложение для воспроизведения ви-

део начало бы при этом заново его показывать. А в этом приложении метод `startPlay()` остается немым, как скала. Полиморфизм позволяет нам иметь в приложении различные формы одного и того же метода, которые знают, как себя вести в различных контекстах. Поэтому в состоянии `Play` рассматриваемый метод знает, что видео уже воспроизводится, и ничего не делает. Пользователь приложения может нажимать на кнопку `Play` сколько угодно, но видео будет просто продолжать показываться. Если пользователь нажмет на кнопку `Stop`, то видео остановится.

Оценить значение полиморфизма поможет приложение, содержащее гораздо больше состояний, за которыми нужно следить. При очень большом количестве методов нам будет крайне сложно исправлять в них какие-либо ошибки. Без полиморфизма мы бы сильно рисковали получить методы с одинаковыми именами, некоторые из которых делали бы не то, что нам нужно. Например, если вы не хотите, чтобы ваше приложение проигрывало видео с самого начала всякий раз при вызове метода `startPlay()`, то, используя шаблон проектирования Состояние, вы можете так организовать ваше приложение, что оно будет показывать видео с начала только в состоянии `Stop`. Вы можете сделать и так, что данный метод будет воспроизводить видео с начала из состояния `Play`. Вы пишете код данного приложения, значит, вы же контролируете поведение всех методов.

В ходе чтения следующей главы о шаблоне проектирования Стратегия у вас может возникнуть ощущение дежавю. Совпадение текстов этих двух глав вовсе не случайно. Когда вы изучите все примеры из каждой главы, вы точно научитесь различать эти шаблоны, несмотря на то что их структуры очень похожи. Шаблон проектирования Состояние специализируется на состояниях и четко определенных переходах. Это одна из причин, почему мы здесь использовали диаграммы состояний; они помогают просто и ясно показать структуру работы данного шаблона, с акцентом на различные состояния и на то, как они переходят друг в друга. Переходы могут контролироваться в шаблоне Состояние либо самими состояниями, либо классом их общего контекста. Кроме того, поскольку шаблон Состояние создает отдельные классы для каждого состояния (среду поведения), в нем обычно создается больше классов, чем в шаблоне Стратегия. Определение, каким именно поведением воспользоваться, делегируется классам состояний, в то время как шаблон Стратегия инкапсулирует в себе набор алгоритмов и позволяет своим клиентам выбирать между ними внутри структуры, которая их использует.

Минималистский абстрактный шаблон Состояние

При использовании шаблона проектирования Состояние все методы (способы поведения) для одного состояния помещаются в один объект (конкретное состояние), а все переходы между состояниями (в конечном автомате) помещаются в один интерфейс. Каждое состояние им-

плементирует данный интерфейс так, как ему это нужно. Ввиду такой структуры для различного поведения, в зависимости от текущего состояния, не требуются никакие условные ветви. Так что нет необходимости писать сложные условные выражения; каждый из объектов состояний сам определяет, как его методы должны вести себя для данного состояния.

Например, для автомата с двумя состояниями `Play` («играть») и `Stop` («стоп») с помощью следующего псевдокода можно определить поведение, начинающее показ видео в зависимости от текущего состояния нашего автомата.

```
function doPlay():void {
    if(state == Play)
    {
        trace("You're already playing.");
    }
    else if (state == Stop)
    {
        trace("Go to the Play state.");
    }
}
```

Когда у нас только пара различных состояний, написать такой код не слишком трудно. Однако при значительном росте общего числа состояний все становится значительно сложнее, и вы можете увидеть просто море подобных условных выражений, работающих совместно. Альтернативой является установка «контекстного» поведения с использованием схемы шаблона Состояние. Пример 10.1 содержит в себе два различных объекта с разной реализацией поведения из одного интерфейса:

Пример 10.1. State.as

```
// Интерфейс
interface State
{
    function startPlay():void;
    function stopPlay():void;
}
// Состояние «проигрывание»
class PlayState implements State
{
    public function startPlay():void
    {
        trace("You're already playing");
    }
    public function stopPlay():void
    {
        trace("Go to the Stop state.");
    }
}
```

```
// Состояние «проигрывание остановлено»
class StopState implements State
{
    public function startPlay():void
    {
        trace("Go to the Play state.");
    }
    public function stopPlay():void
    {
        trace("You're already stopped");
    }
}
```

Как вы можете видеть, поведение (методы) по-разному имплементируется для различных состояний. Если число состояний увеличить, потребуется только добавить новые правила переходов между ними в интерфейсе и создать новое конкретное состояние (класс), которое будет имплементировать свое поведение. Таким образом, каждое новое поведение просто добавляется к существующим классам состояний.

Управление всеми состояниями: работа контекстного класса

Для управления всеми состояниями и переходами между ними вам потребуется какой-то специализированный объект – автоматный движок, реализующий схему вашего конечного автомата. На рис. 10.3 рамка с названием `Context` («контекст») является абстракцией для такого автоматного движка. Этот контекстный класс управляет различными состояниями, создавая тем самым заданный конечный автомат. На рис. 10.4 показана более конкретная диаграмма того, что нам нужно реализовать.

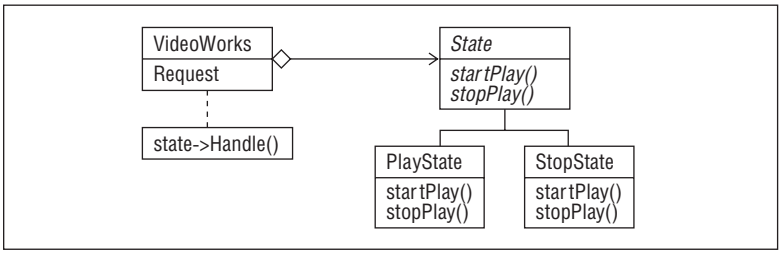

***Рис. 10.4.** Шаблон проектирования Состояние, примененный к видео*

Создание контекстного класса

Для нашего примера реализации простого видеоплеера нам нужен контекстный класс, который будет работать с различными состояниями. Так что сейчас мы им и займемся. Его следует сохранить в файле *Video-*

Works.as. Сначала посмотрите на этот класс в примере 10.2, потом мы обсудим, что в нем происходит.

Пример 10.2. VideoWorks.as

```
1 package
2 {
3     // Контекстный класс
4     class VideoWorks
5     {
6         var playState:State;
7         var stopState:State;
8         var state:State;
9         public function VideoWorks()
10        {
11            trace("Video Player is On");
12            playState = new PlayState(this);
13            stopState = new StopState(this);
14            state=stopState;
15        }
16        public function startPlay():void
17        {
18            state.startPlay();
19        }
20        public function stopPlay():void
21        {
22            state.stopPlay();
23        }
24        public function setState(state:State):void
25        {
26            trace("A new state is set");
27            this.state=state;
28        }
29        public function getState():State
30        {
31            return state;
32        }
33        public function getPlayState():State
34        {
35            return this.playState;
36        }
37        public function getStopState():State
38        {
39            return this.stopState;
40        }
41    }
42 }
```

В начале, в строчках 6–8, данный сценарий создает три объекта класса `State`, по одному для рассмотренных нами состояний (`PlayState` и `StopState`) и еще один (`state`), который будет служить переменной для хране-

ния текущего состояния. Поскольку работа нашего конечного автомата начинается в состоянии `Stop`, переменной `state` присваивается состояние `Stop`. (Это очень похоже на работу выключателя света до того, как вы измените его состояние с «Выключено» на «Включено».)

Далее два метода из интерфейса состояния `State` специфицируются в терминах текущего контекста состояния (строки 16–23). Мы еще создадим код для двух классов состояний, с которыми будет работать этот контекстный класс, но пока можно просто представить себе, что произойдет в этих двух состояниях при вызове их методов. Например, в состоянии `Play` метод `startPlay()` не делает ничего, а в состоянии `Stop` он переключает состояние в `Play`.

В конце добавляются методы получатели и установщики (строки 24–40). Нам потребуется целых шесть методов, по паре этих методов для каждого из трех состояний. Установщики не будут ничего возвращать, а получатели будут возвращать объект `State`.

Завершение и тестирование абстрактного конечного автомата

Чтобы все заработало, нам нужно просмотреть все классы состояний и включить в них ссылки на класс контекста `VideoWorks`. Сохраните пример 10.3 под именем *StopState.as*.

Пример 10.3. StopState.as

```
1 package
2 {
3     // Состояние «проигрывание остановлено»
4     class StopState implements State
5     {
6         var videoWorks:VideoWorks;
7         public function StopState(videoWorks:VideoWorks)
8         {
9             trace("--Stop State--");
10            this.videoWorks=videoWorks;
11        }
12        public function startPlay():void
13        {
14            trace("Begin playing");
15            videoWorks.setState(videoWorks.getPlayState());
16        }
17        public function stopPlay():void
18        {
19            trace("You're already stopped");
20        }
21    }
22 }
```

Добавив объект VideoWorks, мы получили доступ к методам установщикам и получателям для каждого состояния. Например, в строке 15 он вызывается для изменения состояния на Play.

Далее мы повторим все то же самое с состоянием Play, как показано в примере 10.4. Сохраните следующий код как *PlayState.as*.

Пример 10.4. PlayState.as

```
1 package
2 {
3     // Состояние «проигрывание»
4     class PlayState implements State
5     {
6         var videoWorks:VideoWorks;
7         public function PlayState(videoWorks:VideoWorks)
8         {
9             trace("--Play State--");
10            this.videoWorks=videoWorks;
11        }
12        public function startPlay():void
13        {
14            trace("You're already playing");
15        }
16        public function stopPlay():void
17        {
18            trace("Stop playing.");
19            videoWorks.setState(videoWorks.getStopState());
20        }
21    }
22 }
```

Чтобы завершить разработку нашего конечного автомата, необходимо создать для него интерфейс, а поскольку у него есть всего два состояния и два действия, сделать это очень просто. Если вернуться к нашей диаграмме состояний для данного примера, на ней можно увидеть только два перехода между состояниями: один начинает показ видео, другой заканчивает его. Поэтому нам нужны только две абстрактные функции для их описания. Сохраните следующий сценарий из примера 10.5 как *State.as*.

Пример 10.5. State.as

```
1 package
2 {
3     // Интерфейс автомата состояний
4     interface State
5     {
6         function startPlay():void;
```

```
7         function stopPlay():void;
8     }
9 }
```

Все действия переходов описаны в строчках 6 и 7. Позднее мы добавим сюда больше переходов по мере роста нашего проекта. Теперь нам нужно создать *fla*-файл с ActionScript, который будет работать как наш конечный автомат.

Чтобы протестировать наше абстрактное приложение и показать возможности шаблона проектирования Состояние, наш тест должен вызывать класс `VideoWorks` и его два состояния `Play` и `Stop`, используя переходы (методы) `startPlay()` и `stopPlay()`. Фактически оба состояния нужно вызвать дважды. Из состояния `Stop` (исходное состояние нашего приложения) нужно сначала перейти в состояние `Play`. Потом нужно вызвать переход еще раз, чтобы убедиться, что состояние понимает новый контекст. То же самое нужно сделать и после перехода обратно в состояние `Stop`. Сохраните пример 10.6 как *TestState.as* в том же каталоге, что и остальные файлы.

Пример 10.6. TestState.as

```
1 package
2 {
3     // Тестируем состояния
4     import flash.display.Sprite;
5     public class TestState extends Sprite
6     {
7         public function TestState():void
8         {
9             var test:VideoWorks = new VideoWorks();
10            test.startPlay();
11            test.startPlay();
12            test.stopPlay();
13            test.stopPlay();
14        }
15    }
16 }
```

Поскольку наше приложение на данной стадии только печатает текстовые сообщения, чтобы их увидеть, вам нужно будет использовать Flash Test (Control → Test or Control → Test Project). Откройте новый документ Flash и в поле класс документа введите `TestState`. Вы должны увидеть следующие результаты работы нашего приложения в окне `Output`:

```
Video Player is On
--Play State--
--Stop State--
Begin playing
A new state is set
You're already playing
```

```
Stop playing.
A new state is set
You're already stopped
```

Поскольку классы VideoWorks, PlayState и StopState содержат в себе выражения trace() для индикации создания их представителей, они проявляются сразу после начала тестирования нашего приложения. Так как начальным состоянием приложения является Stop, оно меняется на Play при первом вызове метода startPlay(). В связи с тем что вы меняете состояние, выражение trace() из класса VideoWorks сообщает о таком изменении. Когда второй метод startPlay() вызывается второй раз, этот же метод в новом контексте понимает, что он уже находится в состоянии Play и просто сообщает об этом. При нажатии кнопки Stop вы перемещаетесь в состояние Stop, но при повторном ее нажатии та же ее функция понимает, что вы уже находитесь в состоянии Stop, и просто сообщает об этом факте.

Настоящий видеоплеер с состояниями

Пока что в качестве результатов вы видели только сообщения от функции trace(), которые позволяли судить, как работают шаблон проектирования Состояние и наш конечный автомат. Чтобы выводилось что-то более интересное, нам надо включить в наше приложение объект NetStream и строку с названием *flv*-файла. Эта строка нужна нам только при проигрывании видео, поскольку мы можем его остановить, просто закрыв представителя NetStream. Следующие четыре сценария позволяют нашему конечному автомату реально воспроизводить видео и останавливать его показ. Все выражения trace() оставлены в них на месте.

Для имплементации приложения, способного действительно показывать видео, нам необходимо импортировать все необходимые для этого части. Поскольку класс NetStream используется в интерфейсе и в двух классах состояний, каждый из этих файлов нуждается в импорте этого класса. Однако, хотя класс VideoWorks использует оба класса состояний, ему класс NetStream для работы не нужен, поскольку он уже импортирован в эти классы.

Следующие пять листингов кода из примеров с 10.7 по 10.11 следует ввести в файлы ActionScript и сохранить под именами из их заголовков в одном каталоге.

Пример 10.7. State.as

```
package
{
    // Интерфейс состояния
    import flash.net.NetStream;
    interface State
    {
```

```
        function startPlay(ns:NetStream,flv:String):void;
        function stopPlay(ns:NetStream):void;
    }
}
```

В следующем классе `StopState`, показанном в примере 10.8, вы заметите, что метод `startPlay()` имплементирован так, чтобы действительно воспроизводился указанный ему *flv*-файл. Такой переход к состоянию `Play` означает не начало показа видео, а скорее установку такого состояния приложения, в котором происходит воспроизведение видео.

Пример 10.8. StopState.as

```
package
{
    // Состояние «проигрывание остановлено»
    import flash.net.NetStream;

    class StopState implements State
    {
        private var videoWorks:VideoWorks;
        public function StopState(videoWorks:VideoWorks)
        {
            trace("--Stop State--");
            this.videoWorks=videoWorks;
        }
        public function startPlay(ns:NetStream,flv:String):void
        {
            ns.play(flv);
            trace("Begin playing");
            videoWorks.setState(videoWorks.getPlayState());
        }
        public function stopPlay(ns:NetStream):void
        {
            trace("You're already stopped");
        }
    }
}
```

Обратите внимание, что в классе `PlayState` из следующего примера 10.9 метод `startPlay()` разумно ничего не делает, кроме печати сообщения пользователю, что видео уже показывается. В режиме тестирования приложения это сообщение появлялось в окне `Output`. Но в обычном режиме работы такое сообщение не показывается; продолжение показа видео говорит само за себя.

Пример 10.9. PlayState.as

```
package
{
    // Состояние «проигрывание»
```

```
    import flash.net.NetStream;

    class PlayState implements State
    {
        private var videoWorks:VideoWorks;
        public function PlayState(videoWorks:VideoWorks)
        {
            trace("--Play State--");
            this.videoWorks=videoWorks;
        }
        public function startPlay(ns:NetStream,flv:String):void
        {
            trace("You're already playing");
        }
        public function stopPlay(ns:NetStream):void
        {
            ns.close();
            trace("Stop playing.");
            videoWorks.setState(videoWorks.getStopState());
        }
    }
}
```

Контекстный класс из примера 10.10 собирает для данного шаблона проектирования все вместе. Обратите внимание, что в нем нет импорта объекта `NetStream`**. Однако его листинг ясно показывает, что такой объект является одним из параметров для функций** `startPlay()` **и** `stopPlay()`**. Если вы посмотрите внимательнее, то увидите, что обе эти функции берутся из классов** `PlayState` **и** `StopState`**, в которых класс** `NetStream` **уже был импортирован.**

Пример 10.10. VideoWorks.as

```
package
{
    import flash.net.NetStream;
    // Контекстный класс
    class VideoWorks
    {
        private var playState:State;
        private var stopState:State;
        private var state:State;
        public function VideoWorks()
        {
            trace("Video Player is on");
            playState = new PlayState(this);
            stopState = new StopState(this);
            state=stopState;
        }
        public function startPlay(ns:NetStream,flv:String):void
        {
```

```
            state.startPlay(ns,flv);
        }
        public function stopPlay(ns:NetStream):void
        {
            state.stopPlay(ns);
        }
        public function setState(state:State):void
        {
            trace("A new state is set");
            this.state=state;
        }
        public function getState():State
        {
            return state;
        }
        public function getPlayState():State
        {
            return this.playState;
        }
        public function getStopState():State
        {
            return this.stopState;
        }
    }
}
```

В дополнение к классам, входящим в шаблон проектирования Состояние, вам нужны классы кнопок `NetBtn` и `BtnState` из примеров 10.11 и 10.12. На самом деле файлы с этими классами потребуются вам для всех примеров из данной главы, причем должны находиться в тех же каталогах, что и остальные файлы. Мы не включили данные файлы в другие листинги примеров данной главы, потому что они абсолютно одинаковы для всех примеров. (Значит, если вы ввели их один раз, то больше с ними ничего делать не нужно.) Поэтому держите эти файлы под рукой и следите, чтобы они были в каталогах с вашими остальными классами.

Пример 10.11. NetBtn.as

```
package
{
    // Кнопка для переключения триггеров
    import flash.display.Sprite;
    import flash.display.SimpleButton;
    import flash.display.Shape;
    import flash.text.TextFormat;
    import flash.text.TextField;
    import flash.text.TextFieldAutoSize;
    public class NetBtn extends SimpleButton
    {
```

```
        public function NetBtn (txt:String)
        {
            upState = new BtnState(0xfab383, 0x9e0039,txt);
            downState = new BtnState(0xffffff,0x9e0039, txt);
            overState= new BtnState (0x9e0039,0xfab383,txt);
            hitTestState=upState;
        }
    }
}
```

Пример 10.12. BtnState.as

```
package
{
    // Состояния для кнопок переключения
    import flash.display.Sprite;
    import flash.display.Shape;
    import flash.text.TextFormat;
    import flash.text.TextField;
    import flash.text.TextFieldAutoSize;
    class BtnState extends Sprite
    {
        public var btnLabel:TextField;
        public function BtnState (color:uint,color2:uint,btnLabelText:String)
        {
            btnLabel=new TextField  ;
            btnLabel.text=btnLabelText;
            btnLabel.x=5;
            btnLabel.autoSize=TextFieldAutoSize.LEFT;
            var format:TextFormat=new TextFormat("Verdana");
            format.size=12;
            btnLabel.setTextFormat (format);
            var btnWidth:Number=btnLabel.textWidth + 10;
            var bkground:Shape=new Shape;
            bkground.graphics.beginFill (color);
            bkground.graphics.lineStyle (2,color2);
            bkground.graphics.drawRect (0,0,btnWidth,18);
            addChild (bkground);
            addChild (btnLabel);
        }
    }
}
```

Для тестирования нашего приложения вам потребуется *flv*-файл с именем *test.flv*. Вы можете конвертировать в него любой существующий видеофайл (например, *avi*-файл, *mov*-файл) или использовать любой другой *flv*-файл, изменив его имя. Поместите этот файл в тот же каталоге, что и остальные файлы. Наконец, вам потребуется сценарий для тестирования вашего приложения. Поэтому откройте новый файл ActionScript, введите в него код из листинга примера 10.13 и сохраните его как *TestVid.as*.