УДК 519.872, 519.688, 681.63

В.В. КОЖАЕВ

ИСПОЛЬЗОВАНИЕ ДЕКЛАРАТИВНОГО ПОДХОДА ДЛЯ КАРКАСА ДВУМЕРНЫХ ИГР

Рассмотрено использование декларативного подхода в создании каркаса двумерных компьютерных игр. Обоснована необходимость применения нескольких языков программирования в рамках одного программного продукта. Предложен язык для задания таблицы переходов конечного автомата.

Вступление. В последние годы сложность решаемых в промышленном программировании задач значительно возросла. В связи с этим активизировался интерес к декларативному программированию. Согласно одному из определений, декларативный стиль подразумевает описание того, что должна сделать данная программа, оставляя при этом за кадром то, каким образом это нужно сделать.

Как правило, языки или системы, декларативные в этом смысле, состоят из двух частей: библиотеки программ, реализующих наиболее распространенные действия в заданной предметной области, и языка или семейства языков, с помощью которых описывается, какие именно действия необходимо выполнить и в каком порядке. Так, декларативным является язык спецификации классов, используемый в известной платформе Spring. Как известно, с помощью этого языка, составляющего подмножество XML, задается то, каким образом экземпляры java-классов будут создаваться и взаимодействовать между собой. При этом непосредственно созданием объектов и связей занимается Spring, программист лишь описывает, каким должно быть это взаимодействие.

Согласно другому определению, языки декларативного программирования относятся к логическим, функциональным или константным. Как уже было сказано, интерес к декларативным языкам в последнее время растет. Согласно рейтингу Tiobe Index (рис. 1), за 2010 г. практически в два раза возросло использование языка программирования Lisp [1], а в TOP 20 вошли языки Erlang и Scala.

РИС. 1. Прогноз популярности языка Lisp согласно рейтингу Tiobe.com

Как и любой другой стиль программирования, декларативный хорош для определенного класса задач. Кроме того, такие распространенные языки программирования как C++, C#, PHP, являются универсальными и плохо подходят для декларативного программирования. Поэтому для разработки программного продукта целесообразны несколько языков программирования, каждый из которых предназначен для решения своего класса задач.

Современное прикладное программное обеспечение разрабатывается в основном на определенной платформе (аппаратно-программное обеспечение плюс операционная система). Это означает, что код программы транслируется не в машинный код, а в IDL-байт-код, исполняемый на физическом или виртуальном устройстве – так называемой машине, поставляемой в составе платформы. Оттранслированный код выполняется совершенно независимо от исходного языка программирования. Таким образом, можно комбинировать несколько языков в рамках одного программного продукта.

В настоящее время основными платформами являются Java, разработанная фирмой Sun Microsystem (недавно выкуплена Oracle), и .NET (фирмы Microsoft). Для этих платформ можно писать программы на десятках языков программирования, часть из них относятся к декларативным. Так, для платформы Java отметим языки Scala и Clojure, а для .NET – F#.

Недостатком данного подхода является отсутствие мобильности, т. е. программы, написанные в рамках платформы, сложно перенести на другие платформы или в иную операционную среду. В то же время данная задача достаточно актуальна, особенно для разработки игровых программ. Причем целесообразны мультиплатформные языки программирования. Программа, написанная на таком языке, транслируется в байт-код для различных платформ, а исходный код программы не изменяется. В качестве такого языка программирования можно назвать язык HaXe, разработанный фирмой Motion-Twin [2]. Язык является входным для собственной платформы, состоящей из трех частей:

- компилятор HaXe-программ;
- поставляемые с компилятором библиотеки;
- NekoVM – виртуальная машина для исполнения байт-кода.

Программы, разработанные на этом языке, можно оттранслировать в байт-код, выполняемый на виртуальной машине NekoVM, в код, исполняемый виртуальной машиной AVM-2 (Flash) либо в исходные коды на языках JavaScript, PHP или C++.

Другим интересным решением является продукт «Alchemy» (Алхимия), разработанный фирмой Adobe [3] и позволяющий компилировать C++-код в IDL-байт-код, исполняемый виртуальной машиной AVM-2. Данный продукт позволяет использовать наработанные в течение многих лет исходные C++-коды в веб-программировании. Кроме того, программы, созданные с использованием этой технологии, получаются очень быстрыми ввиду оптимизации байт-кода в процессе компиляции. Известным приложением, перенесенным на flash-платформу с помощью этой технологии, является игра Doom.

К достоинствам такого подхода можно отнести простоту переноса программы с одной платформы на другую, к недостаткам – необходимость учитывать ограниченность поставляемых вместе с компилятором библиотек. Библиотеки, поставляемые вместе с целевой платформой, значительно отличаются друг от друга, так что для их использования программу приходится модифицировать, что сводит на нет преимущества мультиплатформности. Кроме того, как указывалось выше, для различных классов задач гораздо удобнее использовать специализированные языки программирования.

Для обеспечения быстрого переноса программного обеспечения с одной платформы на другую целесообразно разрабатывать логическую часть программы в декларативном виде, на языке, который поддерживается одинаково всеми платформами или использовать интерпретатор. Части, отвечающие за взаимодействие с операционной системой, графический интерфейс пользователя, использование специфических для данной платформы библиотек и т. д. на наиболее распространенном в рамках данной платформы языке.

Поскольку быстрый перенос с платформы на платформу особенно важен в разработке игр, а также в играх особенно важна логическая часть программы, не зависящая от конкретного устройства, для которого разрабатывается игра, рассмотрим применение данного подхода для создания двумерной игры «Охотники за астероидами».

Пример игры «Охотники за астероидами»

В этой игре n космических кораблей-сборщиков движутся по двумерному игровому полю, разбитому на клетки c_{ij}, движение происходит в случайном порядке. Если в клетке c_{ij} находится астероид, то космический корабль k (k=1,..,n) перерабатывает его и движется дальше. Если грузовой отсек корабля k заполнен, он движется в направлении корабля-приемника. У корабля k есть определенный запас горючего, если корабль k его расходует, он останавливается. Поэтому, если запас горючего становится меньше определенного, корабль k движется в направлении к космической станции, заряжается и возобновляет сбор астероидов.

Цель игры состоит в сборе всех астероидов за выделенное время T. Противодействовать поведению кораблей-сборщиков пока не предполагается ввиду отсутствия механизмов планирования стратегии игры и ее нацеленности главным образом на развитие моторных реакций игрока.

Не детализируя подробно игровую ситуацию, рассмотрим реализацию движения кораблей-сборщиков. Корабль k может находиться в таких состояниях: движение в режиме поиска астероидов, переработка астероида, движение к кораблю-сборщику, движение к кораблю-заправщику, разгрузка, заправка. В каждом из указанных состояний корабль-сборщик k осуществляет определенные действия. Так, в состоянии движения координаты корабля-сборщика k изменяются, в состоянии обработки демонстрируется некоторый визуальный или звуковой эффект и т. д. При переходе из одного состояния в другое также выполняются определенные действия, причем какие – зависит как от исходного, так и от целевого состояния. В каждом состоянии корабль находится в течение определенного времени, проверка того, нужно ли изменить состояние, происходит по окончании периода времени.

Уровни сложности этой игры задаются с помощью количества n кораблей-сборщиков, периода времени T и размера клеток c_{ij}. Разумеется, перед исполнением игры можно конфигурировать ее программный код, задавая соответствующие интервалы значений, ассоциированных с уровнями сложности.

Представим состояния корабля-сборщика в виде UML-диаграммы состояний (рис. 2).

Очевидно, что ядром большого количества игр с плоским игровым полем является система размещенных на плоскости акторов, поведение которых описывается конечными автоматами. Поэтому целесообразно создать программный каркас для разработки таких игр. Реализация подобного продукта позволит быстро создавать игры с разнообразными игровыми сценариями.

Описание императивной части каркаса

Для реализации каркаса выбрана платформа Flash, а в качестве входного языка – ActionScript 3.0. Как было сказано, поведение объектов реализуется по таблице состояний конечного автомата. Состояния реализованы в виде классов, поддерживающих интерфейс *IState*. Каждое состояние S_i длится некоторый период времени, причем у каждого состояния имеются методы, запускаемые перед входом в состояние, после его завершения и в период нахождения корабля в состоянии S_i. Также каждое S_i характеризуется такими свойствами как имя и период времени, оставшийся до перехода в новое состояние. Акторы наследуются от абстрактного класса *Agent*, одним из свойств которого является состояние, имеющее вышеописанный тип *IStar*.

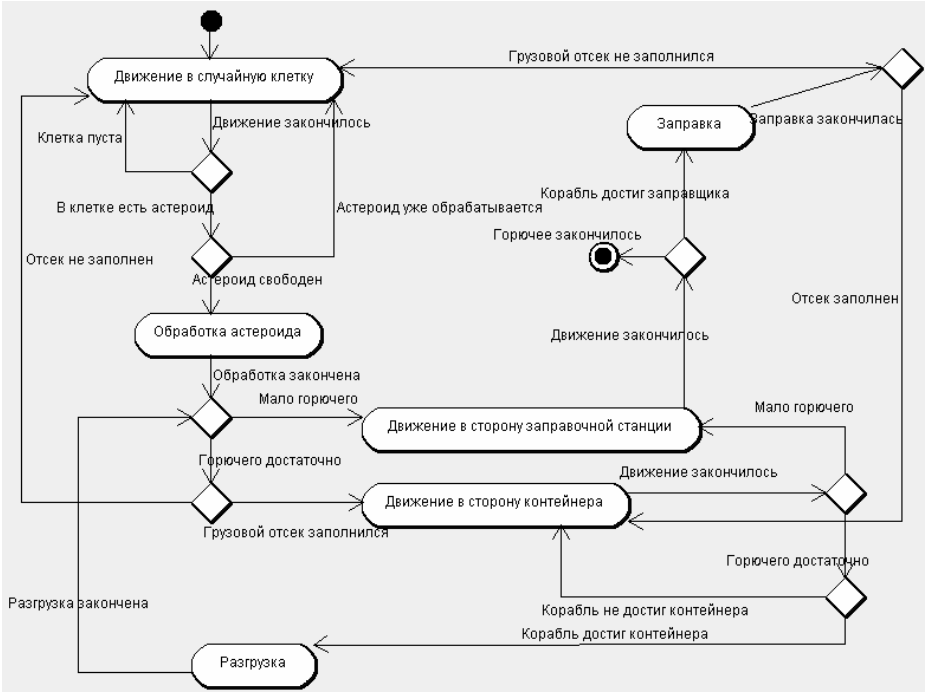

РИС. 2. Диаграмма состояний корабля-сборщика

Как видно из диаграммы состояний (рис. 2), условие перехода из одного состояния в другое является составным. Так, из состояния «Обработка астероида» в состояние «Движение в случайную клетку» корабль переходит в том и только в том случае, если достаточно горючего для движения и контейнер ещё не заполнен. Вместе с тем, из состояния «Разгрузка» корабль переходит в состояние «Движение в случайную клетку», если достаточно горючего, поэтому нецелесообразно проверять, заполнен ли грузовой отсек. Таким образом, очевидно, что целесообразно записать условия в виде логического выражения на некотором языке. Далее функции, выполняемые по завершении одного состояния и в момент перехода в другое, также удобно перечислить списком. Кроме того, если в процессе игры придется добавлять новый тип корабля, например с функциями защиты от врагов, код таблицы состояний, созданной в императивном стиле, придется существенным образом перерабатывать.

Выход из такой ситуации – использование встроенного языка, который позволит гибко описывать условия переходов из состояния в состояние и выполняемые при этом действия.

Описание языка сценариев

Для реализации языка описания конечного автомата выбран Common Lisp, используемый во многих платформах в качестве языка сценариев. Кроме того, он позволяет создавать расширения языка, необходимые для задания формально-логических особенностей алгоритмов для классов задач.

Рассмотрим пример описания таблицы состояний на реализованном расширении языка программирования Common Lisp [4]. Каждый элемент списка содержит название состояния, список правил, по которым из данного состояния осуществляется переход в другие состояния. Каждое правило содержит логическое выражение, атомами которого являются имена функций-условий, список методов, выполняемых, если правило истинно, и имя нового состояния.

```
(setq x '((state_a (((and acond_11 acond_12) (a_do11 a_do12) stateb)
                    ((or acond_21 acond_22) (a_do21 a_do22) statec))
                     ...
  )
          (state_b (((not bcond_11) (b_do11) stateb)
                    ((or bcond_11 bcond_12 bcond_13) () statec)))
...
)
```

Функции-условия реализованы на языке ActionScript 3.0 (в текущей реализации – это статические методы определенных классов). Интерпретатор вычисляет их и подставляет значение в вышеописанное логическое выражение. Если оно истинно, последовательно выполняются необходимые методы и объект переходит в новое состояние.

В реализованном программном продукте использован интерпретатор Common Lisp, разработанный Nate Lokers [5].

Разработанный каркас позволяет реализовать любые двумерные (плоские) игры, для которых поведение акторов (в рассмотренном примере астероидов, кораблей-сборщиков, кораблей-приемников) задается конечным автоматом. Логическую часть игры можно без изменений перенести на другую платформу, например Google Android, IPhone.

Конечный автомат для спецификации поведенческого аспекта сущностей-акторов впервые использован в методе Шлеер – Мелора инженерии требований к разрабатываемому программному продукту [6, 7]. В этом методе предложено сразу две альтернативные нотации для фиксации динамических аспектов требований как поведения классов объектов. Первая графическая называется диаграммой переходов в состояния (ДПС), а вторая табличная – таблицей переходов в состояния (ТПС). Обе нотации основаны на автомате Мура. Если выбирать между ДПС и ТПС, то аргумент в пользу ДПС – ее наглядность в определении действий, а ТПС позволяет зафиксировать все возможные комбинации состояний-событий и обеспечить полноту и непротиворечивость формулировок требований. Поэтому для спецификации поведения акторов целесообразна ДПС, а ее преобразование в ТПС позволит верифицировать игру.

Заключение. Ввиду большого преимущества декларативного программирования вообще и использования нескольких языков программирования в рамках одного программного продукта, а также возможности переносить программы между платформами, большой интерес вызывает разработка программного каркаса (framework) для быстрого создания игровой логики с языком сценариев Common Lisp и распространенными игровыми алгоритмами. Реализация каркаса позволит быстро разрабатывать логическую часть игры, учитывая кроссплатформность программного продукта, обладающего мобильностью для класса платформ.

В.В. Кожаєв

ВИКОРИСТАННЯ ДЕКЛАРАТИВНОГО ПІДХОДУ ДЛЯ КАРКАСА ДВОВИМІРНИХ ІГОР

Розглянуто використання декларативного підходу у створенні каркасу двовимірних комп'ютерних ігор. Обґрунтована потреба застосування кількох мов програмування в рамках одного програмного продукту. Запропоновано мову подання таблиці переходів скінченного автомата.

V.V. Kozhaev

USING THE DECLARATIVE APPROACH TO FRAMEWORK OF TWO-DIMENSIONAL GAMES

The use of a declarative approach to creating the framework of two-dimensional computer games is considered. The necessity to use multiple programming languages within a single software product is proved. A language to specify a transition table for a finite automaton is proposed.

1. http://www.tiobe.com
2. http://haxe.org/
3. w.w.w.adobe.com
4. *Хювенен Э., Сеппянен Й.* Мир Лиспа: В 2-х т. – М.: Мир, 1990. – Т. 1. – 458 с.; Т. 2. – 332 с.
5. http://www.solve-et-coagula.com/
6. *Шлеер С., Меллор С.* Объектно-ориентированный анализ: моделирование мира в состояниях. – Киев: Диалектика, 1993. – 472 с.
7. *Jacobson I., Griss M., Jonsson P.* Software Reuse. – New York: Addison-Wesley, 1997. – 376 p.

Получено 14.12.2010

Об авторе:

Кожаев Владимир Викторович,
аспирант Института кибернетики имени В.М. Глушкова НАН Украины.