УДК 681.3

В.В. Кожаев

ИСТРУМЕНТАРИЙ СОЗДАНИЯ ИГРОВОЙ ЛОГИКИ

Проанализированы возможности имеющихся средств декларативного программирования применительно к созданию деловой логики казуальных игр. Разработано средство для создания деловой логики игр на платформе flash с учетом выполненного анализа.

Введение

В последние несколько лет рынок казуальных [1] игр испытывает бурное развитие – конкуренция между производителями игр растет, игры усложняются. В частности, всё более сложными становятся правила поведения игровых сущностей. Задача программирования игры с акторами, поведение которых подчинено определенному набору правил и быстрое их изменение, является не тривиальной, особенно, если это нужно сделать за ограниченное время. Это обуславливает необходимость создания средств, для упрощения программирования таких правил. К сожалению, средства применяемые для задания логики в промышленных программах, требуют серьёзной доработки из-за специфики программирования игр. В данной статье проанализированы существующие подходы к созданию деловой логики и их недостатки для разработки игр, а также предложен подход позволяющий избавиться от ряда недостатков присущих существующим средствам.

Под **бизнес логикой** понимается совокупность правил, принципов, зависимостей поведения объектов предметной области (области человеческой деятельности, которую система поддерживает). Иначе можно сказать, что бизнес-логика – это реализация правил и ограничений автоматизируемых операций. Является синонимом термина "логика предметной области" (англ. domain logic).

В практике программирования часто используется явное разделение программы на логическую и семантическую части. Под **логической частью** программы будем понимать декларативный набор правил, характеризующий поведение актора. Под **семантической частью** – реализацию конкретных действий производимых в процессе работы программы.

Языки, описывающие логическую часть программы, называются языками описания бизнес логики, языками описания предметной области (англ. Domain Specific Language или DSL - языки)

Под **игровой логикой** будем понимать деловую логику характерную для той или иной игры: совокупность правил игры и правил, задающих поведение акторов. Следует отметить, что в понятие игровой логики не содержит указаний на графическую составляющую конкретной игры, или принятую в определенной игре модель физических, или других процессов.

Движок (от англ. *engine* – мотор, двигатель) – выделенная часть программного кода для реализации конкретной прикладной задачи – программа, часть программы, комплекс программ или библиотека, в зависимости от задачи и реализации. Как правило, прикладная часть выделяется из программы для использования в нескольких проектах и/или раздельной разработки/тестирования. Применительно к разработке игр существуют графические (для создания графики), физические (моделирование физических процессов), изометрические (создание игр в изометрической проекции). Существуют так же движки более высокого уровня, представляющие собой композицию нескольких низкоуровневых движков, часто от разных производителей. По сути, движки этого типа являются архитектурными каркасами, которые могут быть превращены в игру только заданием игровой логики.

Под **структурами, используемыми для задания деловой логики**, будем понимать продукционные системы, конеч-

© В.В. Кожаев, 2012

ные автоматы, деревья поиска и другие структуры данных, применяемые в программировании логики.

Как правило, для управления каждой игровой сущностью не создается отдельного потока. Вместо этого, сущности объединяются в списки и обрабатываются последовательно. Цикл в течении итерации которого обрабатывается такой список называется игровым циклом, итерация этого цикла – тиком. Название "тик" обусловлено тем, что между итерациями цикла проходит фиксированный интервал времени. Существуют игры как с несколькими циклами и созданием отдельного потока для каждого, так и одним игровым циклом.

1. Обзор существующих подходов к программированию деловой логики применительно к разработке игр

В промышленном программировании используется множество подходов позволяющих выделить декларативную и семантическую части программы. Однако большинство из них требуют существенной адаптации для применения в разработке игр ввиду того, что производство игр существенно отличается от разработки коммерческого программного обеспечения (подробнее см. [16]). Далее представлен обзор некоторых из существующих подходов и указание их основных недостатков применительно к разработке игр.

1.1. Использование систем, основанных на правилах. Для задания деловой логики в промышленном программировании широко используются разнообразные системы, основанные на правилах. В этих системах правила задаются в виде структуры "Если – то". Для описания правил обычно используются **DSL** языки, которые могут быть как интерпретируемыми, так и компилируемыми. Во втором случае правила компилируются непосредственно в бинарный код или в байт-код, исполняемый на данной платформе. В обоих случаях существует возможность комбинировать декларативные выражения на DSL – языках с вызовами кода на входном для данной платформы языке. В качестве примера систем созданных на платформе Java можно привести Jess [2], Drools [3], Algernon [4], JEOPS [5] и другие. К достоинствам данных систем относится большое количество документации и мощные средства, предоставляемые платформой.

Однако применительно к разработке игр данный подход имеет ряд существенных недостатков, одним из которых является отсутствие единого стандарта для DSL-языков. На сегодняшний день не существует системы портированной на все популярные платформы. Поэтому, при переносе игры, например, с платформы Java на flash придется полностью переписывать логическую часть программы, либо переносить на данную платформу выбранную систему. Кроме того системы основанные на правилах, не позволяют сохранять состояние актора, что неудобно, поскольку при таком подходе приходится каждый раз анализировать большое количество правил, часть из которых являются значимыми только в небольшой промежуток времени. Использование систем с сохранением состояний позволяет избежать анализа всего множества правил при каждом обращении.

В настоящее время данные системы не используются широко в создании игр, но могут быть использованы после адаптации.

1.2. Использование универсальных интерпретируемых языков высокого уровня. Для декларативного задания игровой логики часто используются интерпретируемые такие языки с динамической типизацией, как **JavaScript, Python, Lua**, причем особенно популярен последний. К достоинствам этих языков относятся простота их использования и изменения ввиду безтиповости. Одним из недостатков данных языков является невысокая скорость работы, другим – то что, возможности встраиваемых версий таких языков весьма ограничены. Сложную систему, используя их создать тяжело.

Данный подход используется в создании не казуальных игр, для задания интерфейса пользователя, или логических связок очень высокого уровня.

1.3. Использование дополнительных языков программирования. Более общим, чем предыдущий случай, является подход, в котором деловую логику разрабатывают на универсальном языке, отличном от принятого на данной платформе в качестве основного, без использования логического движка. Как правило, это динамические или функциональные языки. Для платформы "Java" такими языками являются Clojure [6], Scala [7], Groovy [8] и некоторые другие. Для платформы .net – F# [9], Nemerle [10] и так далее. Программный код на этих языках компилируется непосредственно в байт–код исполняемый виртуальной машиной. Ввиду присутствия в данных языках функционального подхода и их высокоуровневости игровую логику на них создавать много проще, чем на языках, которые являются основными для той или иной платформы. Кроме того, данный подход позволяет переносить игровую логику с платформы на платформу в случае, если язык создания логики реализован на обеих платформах.

Недостаток данного подхода к разработке игровой логики состоит во-первых, в отсутствии реализаций данных языков программирования для популярных игровых платформ, в частности для платформы flash. Другой проблемой является малая распространенность указанных языков среди разработчиков игр ввиду сложности указанных языков.

1.4. Использование мультиплатформных языков. При разработке игр (и других приложений, предназначенных для кроссплатформенного использования) является подход, при котором код пишется на кросс-платформенном языке, либо приложение, написанное для одной платформы, конвертируется для запуска на другой. В качестве примера можно привести язык HaXe [11] и технологию Adobe Alchemy [12].

К недостаткам этого подхода относится то, что в различных языках программирования различными являются и библиотеки программ. Библиотеки для решения одной и той же задачи на разных языках могут иметь совершенно разную архитектуру, или вовсе отсутствовать. Кроме того, указанные средства совершенно не упрощают собственно разработку игровой логики. Язык "HaXe" является универсальным, объектно-ориентированным языком и разработка игровой логики с его использованием ничем не проще, чем с использованием Java, C#, C++ или других универсальных языков программирования высокого уровня. Adobe Alchemy представляет собой технологию преобразования кода программ написанных на C++ в промежуточный байт-код AVM2 (виртуальная машина для выполнения flash – приложений).

1.5. Создание деловой логики с использованием автоматного программирования. Задачи программирования деловой логики как задачи реализации систем с сложным поведением можно успешно решать с помощью **автоматного программировани**я (Этот термин впервые введен в работе [13]). При таком подходе управляющая (логическая) часть программы реализуется в виде конечного автомата. В процессе перехода между состояниями производятся действия (так называемые управляющие), влияющие на акторов или систему в целом (которую можно рассматривать как частный случай актора) и таким образом, осуществляется управление.

Можно выделить следующие виды управляющих действий:

действие входа выполняется при входе в состояние;

действие выхода, выполняется при выходе из состояния;

действие ввода, которое выполняется в зависимости от данного состояния и входного условия;

действие перехода, выполняемое при выполнении определенного перехода.

Действие неизменного состояния, выполняется в системах с дискретным временем, когда ни одно из условий перехода не выполняется и состояние остается неизменным.

К достоинствам автоматного программирования относится следующее:

автомат, полученный при создании деловой логики, является моделью программы. Таким образом, в отличие от традиционных способов программирования

собственно созданию программы предшествует её проектирование;

графическая диаграмма переходов конечного автомата является наглядным и интуитивным средством проектирования. (В ходе рассуждений часто используется людьми без специального образования в виде стрелочной схемы);

позволяет легко тестировать логику: измерить покрытие тестами, сгенерировать виды тестов, создать прототип системы и так далее;

представление логической части программы в виде автомата удобно для строгого доказательства её коррекции;

DSL-языки, используемые при создании автоматов, удобно использовать для частичной генерации семантического кода.

Основным недостатком автоматного программирования является увеличение времени разработки программы. Другими словами, к разработке добавляется время необходимое на оформление кода в автоматном стиле.

1.5.1. Способы описания структуры конечного автомата. Существует большое разнообразие средств позволяющих реализовать автоматный подход (перечисление всех их выходит за рамки данной статьи). Тем не менее, их можно разделить по подходу к заданию структуры конечного автомата на использующие для описания структуры языки полные и не полные по Тьюрингу соответственно.

В первой группе средств часто используются расширения функциональных языков, а во второй – метаязыки на основе XML. Однако это – не правило: существуют реализации средств использующих для описания конечного автомата универсальные, сильно типизированные языки, такие как Java или C# равно как и метаязыки самой причудливой формы.

В качестве полного по Тьюрингу языка можно привести Язык **ASML (Abstract State Machine Language)** [14], который разработан компанией Microsoft в качестве языка реализации управляемых моделей. Является одной из реализаций формализма абстрактных машин состояний, описанных в [7] в виде языка программирования. Это объектно-ориентированный, динамический слабо типизированный язык, по идеологии имеющий некоторое сходство с Python и Visual Basic. Является .net языком и в этом качестве имеет доступ к любому .net совместимому коду и стандартным сервисам .net framework. (существует несколько реализаций данного формализма в виде языка программирования, эта выбрана ввиду большей распространенности).

Основное предназначение языка ASML – разработка выполняемых спецификаций, другими словами разработка моделей программ с целью изучения их поведения и структуры.

Центральными понятиями данного языка являются abstract state (абстрактное состояние) и step.

Под абстрактным состоянием понимают состояние модели системы, которое абстрагировано от несущественных для моделируемых процессов деталей. В языке программирования не существует соответствующей структуры, это скорее некая концепция всего языка: конкретные состояния системы задаются в виде множества пар (имя переменной, значение).

Step – конструкция языка (в отличие от первого понятия), предназначенная для задания действий, которые производятся при переходах между состояниями.

В терминологии предлагаемой в работе [13], данный язык предназначен для создания конечных автоматов с активными переходами.

Типичным средством использующим метаязык является **UniMod** [15]. Представляет так называемое Case-средство реализованное в виде plug-in a IDE Eclipse, который включает графический редактор графов состояний, парсер XML – языка описывающего деловую логику и генератор кода по созданной диаграмме состояний. В отличие от предыдущего, средство предназначено для реализации автоматов с пассивными переходами. Прогрессивным является то, что возможна кодогенерация на различных языках программирования (доступны Java и Simbean C). В настоящее время разработка и поддержка этого средства прекращена, что естественно является серьезным

аргументом против использования его в разработке.

Достоинством языков полных по Тьюрингу является то, что они являются исполняемыми, то есть позволяют протестировать программную модель без реализации семантической части программы. Однако, реализация полного по Тьюрингу языка на конкретной платформе является на порядок более сложной задачей, чем создание парсера метаязыка. Поэтому, если предполагается перенос логики на несколько платформ целесообразно использование именно метаязыков. С другой стороны, использование метаязыков более удобно при создании графических редакторов графа перехода состояний.

2. Описание реализованного средства автоматного программирования

Хотя автоматное программирование широко используется при создании промышленных программ, не существует средства, которое повсеместно использовалось бы в производстве игр. В качестве причины можно указать некоторую консервативность разработчиков игр, а тот факт, что игры менее требовательны к отказоустойчивости по сравнению с промышленными программами. Действительно, в случае отказа программы управления атомной электростанцией может случиться техногенная катастрофа, при сбое программного обеспечения банка возникнут ошибки в финансовых операциях – люди не получат деньги. При ошибке же работы игры катастрофы не произойдет – пользователь просто перезапустит программу. Поэтому, в практике разработки игр добиваются исправления лишь часто возникающих сбоев.

Однако необходимость в создании версий одной и той же игры для разных платформ и все усложняющееся поведение игровых сущностей обуславливают необходимость создания такого средства. Чтобы восполнить этот пробел, автором было разработано средство под названием "casual intellect". Это средство создано на платформе "flash" (виртуальная машина - AVM2) с использованием языка программирования ActionScript 3.0.

При разработке поставлены следующие задачи:

средство предназначено для разработки игр с коротким сроком жизни. Это означает, что время в течении которого пользователи активно играют составляет от одного до нескольких месяцев. Особые требования в создании таких игр предъявляются к скорости разработки. Поэтому средство должно быть интуитивно понятным;

должна быть возможность создания редактора графических диаграмм, визуализирующих игровую логику;

должно давать возможность задать все типы управляющих реакций;

соответствовать принятой методологии производства игр;

позволять генерацию программмного кода для разных платформ.

В результате было решено создать архитектурный каркас для реализации конечных автоматов с активными переходами с метаязыком на базе XML. В терминологии [13], средство реализует процедурное программирование с явным выделением состояний. Например, таких как робот.

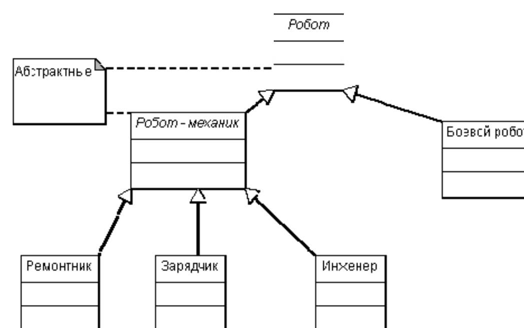

Рис. 1. Диаграмма классов робота игр

2.1. Описание демонстрационной программы. Положим, необходимо реализовать следующую игру. На игровом поле стоит несколько башен стреляющих по противнику. Ремонт башен после разрушения, зарядку пушек и улучшение тактических характеристик башен производят роботы, специализированные для этих операций: роботы ремонтники, заряжающие и инженеры соответственно.

Силы противника состоят из наземных и воздушных боевых единиц. Воз-

душные подразделяются на бомбардировщики и вертолеты, наземные – на танки и картечницы. За каждый уничтоженный робот противника игроку начисляются очки, за которые можно построить новых роботов. Цель игры – найти оптимальную стратегию постройки роботов.

2.1.1. Поведение робота игрока. Типы поведения роботов игрока можно представить в виде диаграммы наследования (см. рис. 1). Как видно из диаграммы, тип "робот" является базовым поведенческим типом для боевых роботов и механиков.

к ней, обслуживает и переходит в состояние покоя до тех пор, пока в нем не появится необходимость. Поскольку по роботу ведется стрельба, рано или поздно наступает момент, когда его жизненные ресурсы исчерпываются и он переходит в состояние агонии из которого следует переход в конечное состояние – смерть.

Спецификация языка UML не позволяет раскрыть все особенности поведения роботов. Невозможно указать, что управляющие воздействия при переходах из состояния в состояние в общем случае зависят не от выходного состояния I или

```
<!element object_logic (consts?, states?)>
<element variables (varialble+)>
<!variable @name>
<!element states state+>
<!element state (rooles?,methods?,extends?)>
<!element rooles roole+>
<!element roole (condition, roole_methods?)>
<!condition #equation>
<!roole_methods method+>
<!extends #statename)> (methods_before?,methods_after?,methods_in_process?)>
<!methods_before method+>
<!methods_after method+>
<!methods_in_process method+>
<!method #methodname>
<!roolemethod method>
```

Листинг 2.1. XSD схема языка программирования

Поведенческий тип "робот – механик" базовый для типов ремонтник, зарядчик и инженер.

Поведение роботов задается конечным автоматом с помощью управляющих воздействий. Рассмотрим диаграмму состояний графа управляющего роботом (рис. 2). Изначально робот находится в состоянии покоя, если возникает необхо димость в обслуживании башни движется

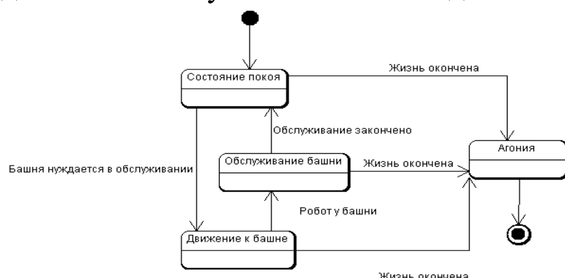

Рис. 2. Диаграмма состояний робота – механика

входного J, а от вектора (I,J) Кроме того, отсутствует возможность наследования автоматов и переопределения управляющих воздействий без изменения графа переходов состояний. Разработанный DSL язык имеет указанные возможности.

2.2. Описание реализованного DSL языка. Поскольку разработанный DSL язык является XML – языком его спецификация задана в виде DTD – схемы (см. Листинг 2.1.).

Проверка условий перехода, и переход в новое состояние (если соответствующее условие выполняется) производится единожды за тик. При переходах между состоянием над актором выполняются соответствующие методы, реализованные на языке программирования являющимся входным для данной платформы (ActionScript 3.0 в данной реализации).

Каждое состояние имеет методы,

выполняемые после входа в него актора, перед выходом, и методами выполняемыми если состояние актора не изменилось. Эти методы задаются тегом *methods* который может содержать вложенные теги *methods_before, «methods_after» и «methods_in_process»* для задания множества методов выполняемых перед входом в состояние, перед выходом и если состояние не изменилось, соответственно.

Вышеописанные теги содержат в себе один или несколько тегов *«method»* для задания списков выполняемых методов.

Условия перехода из данного состояния задаются с помощью тега *«rools»*, который содержит в себе список правил («*roole*»). И тега *«roole_methods»*, содержащего методы выполняемые, если условие истинно. Правило состоит из условия

```
...
<states>
        <state name="move_to_tower">
          <!--секция задающая состояния в которые можно перейти из текущего-->
                <usecases>
      <!-- Состояние зарядка -->
                        <usecase name="charging">

                                <roole><!--Переход происходит, если выполняется условие-->
                                 ((not energyIsLow) and
                                 (not needRepairTower))
                                 or (goalIsGetted)
                                </roole>
                              <!--Перед переходом выполняются
                                        управляющие воздействия-->
                                <methods>
                                <!--В частности endMoving-->
                                        <method name="endMoving"/>
                                </methods>
                        </usecase>
                        <!--Состояние "агония"-->
                        <usecase name="agony">
                                <roole>
                                        energyIsLow
                                </roole>
                                <methods>
                                        <method name="showAnger"/>
                                </methods>
                        </usecase>
                </usecases>
        <!--Секция методов выполняющихся всегда-->
        <state_methods>
          <!--Перед входом в состояние-->
                <methods_before>
                        <method name="beforeMoving"/>
                </methods_before>
                <!--Перед выходом-->
                <methods_after>
                        <method name="afterMoving"/>

                </methods_after>
                <!--Каждый игровой тик, пока актор находится в состоянии-->
                <methods_in_process>
                        <method name="move"/>
                </methods_in_process>
        </state_methods>
 </state>
...
```

Листинг 2.2. Декларативное задание состояния «Движение к башне»

перехода «*condition*» представляющих собой логическое выражение с атомами – функциями языка программирования, возвращающими логическое значение. Вычисление этого выражения осуществляется с помощью перевода в обратную польскую запись.

При обработке состояния производится проход по списку, содержащемся в «rools». Для каждого элемента *«roole»* определяется истинно — ли условие перехода содержащееся в condition. Если это так, производится последовательное выполнение методов содержащихся в элементе *«roole_methods»*. Затем выполняются методы, содержащиеся в теге «*methods_after*», который в свою очередь содержится в теге *«methods»* текущего состояния. После этого выполняются методы, содержащиеся в теге *«methods-_before»* нового состояния.

Если ни одно из условий перехода из данного состояния не является истинным, выполняются методы, содержащиеся в теге *«methods_in_process»* текущего состояния.

В качестве примера рассмотрим реализацию состояния «Движение к башне» показанном в фрагменте программного кода 2.2.

Из данного состояния можно перейти в состояние «обслуживание башни», либо в состояние «агония» если жизненные силы истощились. Переход происходит, если истинны логические выражения заданные тегом rules. Перед переходом выполняются управляющие воздействия с именами «*endMoving*» в случае перехода в состояние «обслуживание» или *"showAnger"* если происходит переход в состояние «Агония».

2.3. Спецификация методов – обработчиков. Одна и та же декларативная часть программы может быть интерпретирована по-разному, в зависимости от способа её обработки императивной частью.

В качестве примера рассмотрим реализацию поведения роботов механиков. Данные роботы управляются одним автоматом, разнятся лишь управляющие воздействия. Так для робота – ремонтника в состоянии «обслуживание башни» управляющим воздействием будет «ремонт». Для робота – заряжающего «зарядка орудия», а для робота – инженера – «улучшение».

В разработанном программном средстве управляющие воздействия ассоциируются в императивном стиле, с помощью входного языка программирования ActionScript 3.0 (см. Листинг 2.3).

```
//Инициализируем обработчик состояний
var _statesProcessor:StatesProcessor;
_statesProcessor = StatesProcessor.statesProcessor;
//Создаем контейнер операций
var opContainer:OperationsContainer = new OperationsContainer();
//Ассоциируем итендификаторы управляющих воздействий с конкретными методами
opContainer.addFunction("finishMoving", _robotController.endMoveCondition);
opContainer.addFunction("move", _ robotController .move)
opContainer.addFunction("beforeMoving", _  robotController .beforeMoving)
opContainer.addFunction("afterMoving", _ robotController .afterMoving);
opContainer.addFunction("endWating", _ robotController .endWating);
opContainer.addFunction("movingIsFinished", _ robotController .endMoveCondition);
opContainer.addFunction("watingIsFinished", _ robotController .watingIsFinished);
opContainer.addFunction("beforeWating", _ robotController .beforeWating);
opContainer.addFunction("afterWating", _ robotController .afterWating);
opContainer.addFunction("wait", _ robotController.wait);
//передаем обработчику состояний контейнер операций
_statesProcessor.operationsContainer = opContainer;
```

Листинг 2.3. Императивное задание методов

Как видно из представленного кода, за хранение ассоциированных методов отвечает класс *OperationsContainer*, с помощью метода *addFunction* методы – обработчики ассоциируются с идентификаторами. Таким образом, ассоциировав один и тот – же идентификатор метода *shoot* с разными обработчиками можно изменить характеристики выстрела оставив при этом неизменным общий характер поведения робота.

```
<direction>
    <extends parent=”airplane”/>
    ....
</direction>
```

Листинг 2.4. Использование тега extends

```
<!--Унаследовано от состояния airplane-->
        <extends parent=”airplane”/><!--Константы-->
        <variables>
                <variable name=”speed”>5</variable>
                        <variable name=”armour”>3</variable>
        </variables>
        <!--Измененное состояние-->
        <states>
                <state name="moving">
                        <usecases>
                                <usecase name="bombarding ">
                                  <!--Конкретно изменено текущее правило-->
                                        <roole>hasGroundTargets </roole>
                                                <methods>
                                                        <method name="endMoving "/>
                                                </methods>
                                         </usecase>
                        </usecases>
                        <!—Методы выполняемые -->
                        <state_methods>
                                <!—Перед входом в состояние -->
                                <methods_before>
                                        <method name="beforeMoving"/>
                        </methods_before>
                                <!—Перед выходом из состояния -->
                                <methods_after>
                                        <method name="afterMoving"/>
                                </methods_after>
<!—На каждый тик -->
                                <methods_in_process>
                                        <method name="move"/>

                                </methods_in_process>
                        </state_methods>
                 </state>
                <!--Запрещенное состояние-->
                <state name="shooting">
                        <forbidden/>
                </state>
                <state name="bombarding">
        </state >
        </states
</direction>;
```

Листинг 2.5. Декларативная часть программы, управляющая бомбардировщиком

2.4. Наследование декларативных частей программы. Для декларативных частей программы, реализованных с помощью DSL–языка, реализовано наследование. Как и наследование классов в традиционных языках, оно позволяет расширить, изменить или запретить свойства предка.

Указать, что управляющий элемент программы наследуется от другого, можно с помощью тега “extends”, как показано в листинге 2.4.

```
...
<variables>
    <variable name=”speed”>5</variable>
    <variable
name=”armour”>3</variable>
     ...
</variables>
...
```

Листинг 2.6. Пример создания переменных

В качестве примера использования рассмотрим два таких робота как самолет и наследующий его поведение бомбардировщик. У первого из состояния полета производится переход в состояние стрельбы по наземным целям. У второго же – в режим бомбардировки, при этом состояния стрельбы по наземным целям не существует. Кроме того, у бомбардировщика скорость передвижения будет отличаться, от самолета.

```
package org.casualintellect.statemachineclases
{
public interface IStateObject
{
   //Геттер и сеттер имени состояния
  function get state():String;
  function set state(val:String):void;

  //Итендификатор конкретного класса
  //реализуюцего данный  интерфейс
  function get className():String;
}
}
```

Листинг 2.7. Интерфейс «состояние»

Таким образом, необходимо изменить значение константы *“speed”*, добавить состояние *bombarding*, изменить состояния, из которых актор-предок может перейти в состояние “shoot” и удалить состояние “shoot” актора-наследника. Наследование декларативной части программы показано в листинге 2.6 (детали для простоты опущены). Изменено состояние moving базового класса и добавлено состояние *bombarding*.

Состояние *shooting* удалено из автомата с помощью тега forbidden.

2.5. Константы. Поведение акторов зависит не только от их реакции на определенные события, но и от таких начальных условий, как, например, скорость движения, количество выстрелов в секунду, толщина брони и так далее.

В первом приближении можно считать, что начальные условия задаются с помощью численных значений констант, используемых в императивной части программы. Изменив их, можно специфицировать поведение акторов, управляемых одним и тем же конечным автоматом.

В DSL–языке константы задаются с помощью тега *variables*. Название тега подчеркивает тот факт, что константы можно менять в течении игрового процесса. Константы подлежат переопределению в ходе наследования, для этого нужно объявить константу заново. При этом тип значения константы диаграммы предка должен быть совместим с типом константы родителя.

2.6. Детали реализации каркаса. Реализованный каркас состоит из трех частей: парсера XML – языка задающего машину состояний, интерпретатора условий перехода и машины состояний. В начале работы программы парсер строит по XML – текстам конкретные реализации машин состояний управляющих акторами. При этом, условия перехода преобразовываются в обратную польскую запись и вычисляются уже в процессе работы программы с помощью интерпретатора условий. Каркас поддерживает архитектуру «модель – вид – контроллер»: обрабатывающие методы являются методами контроллера. Эти методы должны принимать экземпляр класса реализующего интерфейс IStateObject. Другими словами все классы акторов обрабатываемые с помощью реализованной машины состояний должны реализовывать данный интерфейс.

Как видно из листинга 2.7, объект принимаемый методами контроллера должен иметь возможность изменить имя состояния (функция сеттер), вернуть его (функция геттер) и вернуть имя обрабатываемого класса.

Выводы и перспективы дальнейшего развития

Реализованный каркас является одним из первых применений автоматного программирования к разработке казуальных игр, что обуславливает научную новизну (других применений автору не известно). Декларативную часть программы можно легко перенести на другие платформы, поскольку задается в виде XML.

В перспективе развития программы стоит реализация графического редактора декларативной части и генерации по ней семантической части.

1. *Статья* в википедии об автоматном программировании ru.wikipedia.org/wiki/Автоматное_программирование
2. *Домашняя* страница проекта Jess http://www.jessrules.com/
3. *Домашняя* страница проекта Drools http://www.jboss.org/drools/
4. *Домашняя* страница проекта Algernon http://algernon-j.sourceforge.net/
5. *Домашняя* страница проекта Jeops http://www.jeops.org/
6. *Официальный* сайт языка Clojure http://clojure.org/
7. Официальный сайт языка Scala http://www.scala-lang.org/
8. *Официальный* сайт языка Groovy http://groovy.codehaus.org/
9. *Официальный* сайт языка F# http://msdn.microsoft.com/ru-ru/fsharp/default%28en-us%29.aspx
10. *Официальный* сайт языка Nemerlie http://nemerle.org/Main_Page
11. *Домашняя* страница проекта Haxe http://haxe.org/
12. *Домашняя* страница проекта Alchemy http://labs.adobe.com/technologies/alchemy/
13. *Автоматное* программирование. Н. И. Поликарпова, А. А. Шалыто М. 2008
14. *Спецификация* языка ASML http://research.microsoft.com/en-us/groups/foundations/AsmL/
15. *Домашняя* страница проекта UniMod http://unimod.sourceforge.net/
16. *Кожаев В.В.* Разработка и тестирование игр // Компьютерная математика. – 2011. – № 1. – С. 79 – 85.

Получено 08.08.2012

Об авторе:

Кожаев Владимир Викторович,
аспирант.

Место работы автора:

Институт кибернетики
имени В.М. Глушкова НАН Украины,
03680, Киев-187,
проспект Академика Глушкова, 40.
Тел. (044) 526 3603.
dep145@gmail.com