ДЕКЛАРАТИВНАЯ И ИМПЕРАТИВНАЯ ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ В ОПИСАНИИ ПОВЕДЕНИЯ СЛОЖНЫХ РАСПРЕДЕЛЕННЫХ ИМИТАЦИОННЫХ МОДЕЛЕЙ

Ю.И. Бродский

Гораздо важнее знать, что делается, чем делать то, что знаешь.
А. Боэций

Под императивным программированием будем понимать самый распространенный подход к написанию программ на языках программирования типа FORTRAN, или семейства C, или на Java, где программа есть последовательность инструкций-приказов, которые должен выполнить компьютер. Таким образом, применяя императивное программирование, мы описываем последовательность действий, достаточную (на взгляд разработчика) для получения результата проекта. Результат при этом не предполагается известным заранее, наоборот, скорее всего целью проекта императивного программирования и является получение этого результата (например, получение численного решения системы уравнений в частных производных, или изучение поведения сложной системы в имитационном эксперименте с ее моделью), а на стадии отладки программ получаемый результат вообще постоянно удивляет и поражает своих создателей.

При декларативном программировании, наоборот, описывается то, каким должен быть известный заранее желаемый результат, например, статическая страничка HTML, или документ в системе LaTeX, или же DVD-проект в системе авторинга Scenarist. При этом выбор последовательности действий, приводящей к описанному в системе декларативного программирования результату, как правило, оставляется на усмотрение соответствующего компилятора или интерпретатора (чему свидетельство – заметная иногда разница представления одного и того же документа HTML в различных браузерах).

Под парадигмой программирования, вслед за Памелой Зейв (*Pamela Zave*), авторы склонны понимать набор представлений о некотором классе программных систем, допускающих реализацию с помощью этой парадигмы «way of thinking about computer systems» [7].

Помимо упомянутых выше императивной и декларативной парадигм, существуют и другие весьма интересные парадигмы программирования, например функциональная, однако наша основная цель – обсуждение путей решения вполне конкретной прикладной задачи, сформулированной в начале работы, а не теория программирования.

Заметим, что, по крайней мере на первый взгляд, эта задача хорошо укладывается в императивную парадигму программирования. В самом деле, отталкиваясь от имеющихся знаний об отдельных компонентах сложной системы, мы беремся построить синтез всей системы, основанный на воспроизведении известного нам поведения каждой ее компоненты, и при этом собираемся наблюдать и изучать поведение системы в целом, неизвестное нам заранее. В основе синтеза многокомпонентной системы лежит идея высказанная еще в работах Н.П. Бусленко – дать каждой из компонент, про которые нам все известно, максимально проявить себя, учитывая при этом и все межкомпонентные связи [13].

Основным инструментом реализации проектов императивного программирования являются объектно-ориентированные языки программирования. В свое время эти языки возникли в значительной мере как ответ на запросы исследователей, занимавшихся имитационным моделированием. Одним из самых первых объектно-ориентированных языков можно считать Симулу-67. Структура сложной системы, состоящей из многих экземпляров различных типов компонент хорошо описывается иерархией классов объектно-ориентированного программирования (ООП). У ООП масса известных всем достоинств, которые и сделали этот подход бесспорным лидером в программировании последних 25 лет. Однако, применительно к нашей задаче есть у ООП и существенные недостатки.

Главный из них (конечно, применительно к классу задач, о которых идет речь в данной работе) – отсутствие у объекта поведения. У объекта есть характеристики, есть масса всяческих умений – это его методы – а поведения, в смысле умения давать ответы на стандартные запросы окружающей среды, у него нет. Конечно, ООП в соединении с соответствующим программным обеспечением промежуточного уровня, позволяет объектам даже в распределенных системах взаимодействовать между собою, наблюдать чужие характеристики, пользоваться чужими методами, однако все это делается в некотором смысле «вручную».

В этом смысле коллектив программистов, приступающий к созданию сложного проекта, и вооруженный набором библиотек самых полезных классов, похож на человека которому выдали несколько компьютеров, каждый из которых оснащен большим набором полезных прикладных программ, однако из системных программ имеющий только загрузчик, и поставили задачу сделать из этого распределенную систему. Понятно, что современный программист возмутился бы страшно: «Так не бывает! Где моя сетевая операционная система? Где, наконец, ПО промежуточного уровня?». Однако, при создании сложной системы средствами ООП дело обстоит именно так – к сложности содержательных вычислений предметной области

добавляется еще и сложность организации поведения объектов а также их связей и взаимодействий между собою.

Поведение компоненты сложной системы есть функциональный аналог операционной системы в области системного ПО, и функциональный аналог бытовой культуры в социальных системах – способность давать стандартные ответы на стандартные запросы окружающей среды. В имитационном моделировании сложных систем очень важно уметь моделировать поведение компонент. Как уже говорилось выше – именно из их поведения естественно синтезировать поведение системы в целом.

В истории развития информатики известно несколько подходов к объектному программированию, где объекты обладают поведением. Подходы эти зародились в среде исследователей, занимавшихся проблемами искусственного интеллекта, и известны как модель акторов [3] и агентное программирование [5, 6]. Однако хотя авторы согласны с главным посылом этих подходов – важностью моделирования поведения агентов системы, детали описания этого поведения в [3, 5, 6], на наш взгляд слишком окрашены спецификой проблематики искусственного интеллекта, т. е., недостаточно универсальны. Так у Шохема [5, 6] поведение – это поведение ментальное, о состоянии которого рассуждают в категориях «убеждения», «обязанности», «способности» и т.д. Акторная модель Хьюитта [3] интересна своей ориентированностью на параллельные вычисления, однако асинхронный обмен акторов сигналами-сообщенями, а также порождение новых акторов представляется нам избыточными конструкциями, затрудняющими реализацию подобных систем. Подробно этот вопрос обсуждался в [12].

Проблема здесь в том, что согласившись на наличие поведения у компонент системы, нужно учиться описывать и реализовывать это поведение. Это поведение с одной стороны достаточно сложно, так как это поведение элементарной сложной системы. Заранее оно не может быть известно, поскольку может зависеть от внешних по отношению к рассматриваемой компоненте условий, на которые она должна уметь реагировать должным образом. Следовательно, такие поведенческие моменты должны вычисляться и, следовательно, алгоритмы их вычисления должны быть описаны на императивных языках программирования. С другой стороны, кое-что о поведении компоненты мы всегда знаем заранее, так как она в силу исходных предположений работы имеет в предметной области моделирования достаточно хорошо изученный прообраз. Поэтому обычно мы заранее можем перечислить весь набор ее «умений»-элементарных действий, а также сказать какое из этих элементарных действий может перейти в какое, и под действием каких обстоятельств – просто уж так устроен прообраз этой компоненты в предметной области моделирования. Вот это-то знание, коррелирующее с пониманием устройства в предметной области, как прообраза отдельной компоненты, так и всей многокомпонентной модели, как раз наличествует заранее, еще до построения модели, и не зависит от хода имитационных экспериментов. Поэтому такое знание об устройстве модели сложной системы вполне может быть выражено на декларативном языке программирования.

Такой подход позволяет получить «ортогональные» (подобным образом «ортогональны», например, описания стиля и наполнения в Word- или HTML-документах) описания модели сложной системы: на декларативном языке описывается то, как устроена система и правила поведения ее составляющих. На императивном языке описываются отдельные законченные элементарные алгоритмы, не взаимодействующие ни с чем в модели, кроме собственных входных и выходных параметров.

Предлагаемое разделение описаний на декларативное и императивное оказывается весьма технологичным: декларативную и императивные части можно отлаживать независимо друг от друга. При этом самая сложная императивная составляющая программы распадается на ряд независимых друг от друга законченных элементарных алгоритмов. При этом она редуцируется настолько, что чаще всего возможности ООП оказываются слабо востребованными: алгоритмически цельную программу с фиксированным набором входных и выходных параметров чаще всего нетрудно реализовать хоть на FORTRANe. И это существенно облегчает отладку системы. Вся объектная ориентированность, идущая от предметной области моделирования, оказывается в декларативном описании модели.

Одной из первых сред программирования, воплотивших описанную выше концепцию разделения описания сложной системы на декларативную и императивную составляющие была разработанная в ВЦ АН СССР инструментальная система имитационного моделирования MISS (Multilingual Instrumental Simulation System). Концепция программирования, лежащая в ее основе описана в работах [8, 9], а полное описание среды моделирования на уровне руководства пользователя – в [10]. В системе MISS была реализована в среде MS-DOS система программирования на специальном декларативном языке описания сложных систем, интегрированная с базой данных, системой поддержки выполнения модели и системой презентации результатов моделирования. В качестве императивных языков программирования, на которых писались вычислительные алгоритма были разрешены две версии языка MODULA-2, а также языки C и C++ в Борландовской реализации. До этого в основном моделирующее сообщество создавало языки моделирования императивного типа.

Впоследствии идея разделения описания сложной системы на декларативную и императивную части применялась неоднократно. Так в спецификации архитектуры верхнего уровня (High Level Architecture – HLA) [4] – средстве создания сложных распределенных моделей – устройство системы, ее компонент и связей между ними описывается специальными шаблонами. В коммерчески успешной отечественной инструментальной системе имитационного моделирования AnyLogic [15] в качестве декларативного языка описания сложной

системы применяется интерактивная графическая система, где на экране размещаются пиктограммы компонент будущей системы и тут же проводятся связи между ними. В качестве императивного языка программирования используется родной для системы AnyLogic язык Java. Наконец, появился и даже стал международным стандартом язык моделирования объектных систем UML [14], однако авторам он не симпатичен в силу своей громоздкости и порой за счет этого неоднозначности. В качестве результатов компиляции транслятор UML может выдавать заготовки модулей для императивных языков.

В настоящее время в ВЦ РАН продолжается развитие концепции разделения описания сложной системы на декларативную и императивную составляющие. Работает макет инструментальной системы распределенного моделирования. Между прочим, определенное затруднение вызывает название описываемой концепции – такие названия, как агент, актор и даже компонента уже заняты, и в программистском сообществе под ними понимаются вполне определенные и отличные от описанного выше подходы. Сами разработчики концепции в работах [1, 2, 8-11] называли свой подход объектным или объектно-событийным, что тоже не совсем верно, так как не отражает главных его моментов – наличия поведения у компоненты и декомпозицию описания сложной системы на декларативную и императивную составляющие.

Опишем теперь кратко предлагаемую концепцию описания модели. Подробное описание можно найти в [12]. В основе концепции лежит понятие компоненты. Компонента – это в некотором смысле «элементарная» сложная система. Основой конструкции компоненты будет объект объектного анализа, в том смысле, что описываемая ниже компонента есть некий тип или класс моделей, а заполняться данными и запускаться на счет будут экземпляры этого класса. Однако главное отличие от объекта в том, что компонента – это объект с поведением. Опишем «устройство» компоненты.

Характеристики. Компонента, как и объект, имеет характеристики. Эти характеристики мы будем разбивать на внутренние и внешние. Внутренние характеристики – это то, что компонента моделирует, внешние – это то, что она знает о внешнем мире.

Процессы. Функциональность компоненты удобно структурировать следующим образом: Считается, что компонента реализует один или несколько параллельно выполняющихся процессов. Процесс состоит в последовательном чередовании элементов – алгоритмически элементарных методов. Если какой-либо процесс какую-то часть модельного времени не выполняется – удобно считать, что в это время он выполняет «пустой» элемент, не меняющий никаких характеристик компоненты.

Элементы. Элементарные, алгоритмически однородные методы, реализующие функциональности компоненты (т. е. то, что компонента умеет делать, получение на основании значений некоторых внутренних и внешних характеристик компоненты, новых значений некоторых ее внутренних характеристик).

По отношению к модельному времени некоторые элементы выполняются мгновенно, это сосредоточенные или быстрые элементы. Быстрыми элементами можно моделировать дискретные характеристики системы.

Выполнение других элементов занимает определенное время. Если при этом элемент для любого промежутка времени $\Delta\tau$, не превосходящего стандартный шаг моделирования Δt выдает некий осмысленный результат, такой элемент называется распределенным или медленным. Распределенные элементы – естественное средство вычисления непрерывных характеристик модели.

Может оказаться и так, что выполнение элемента занимает определенное модельное время, но результат его действия наступает лишь в конце, после полного выполнения элемента, т. е. никаких промежуточных результатов за время меньшее полного времени выполнения нет. Такие элементы называются условно-распределенными. Вообще говоря, условно-распределенные элементы можно не рассматривать как отдельный класс, а моделировать парой: пустой распределенный элемент, за которым идет сосредоточенный, выдающий результат.

Частью предлагаемой концепции является жесткая дисциплина работы методов с фазовыми переменными модели: каждый метод имеет право изменять только «свои» переменные. Эта дисциплина основана на принятии предположения о детерминированности и однозначности имитационных вычислений. В рамках предлагаемой концепции, конфликт доступа возникающий, когда методы A и B вычисляют одну и ту же характеристику $X = X_A$ и $X = X_B$, может быть разрешен, например, введением метода C, который получая на входе в качестве параметров X_A и X_B, вычисляет на их основании искомую характеристику X, устраняя тем самым не только конфликт доступа к ней, но и очевидно, имевшую место неоднозначность вычисления упомянутой характеристики. Принятая дисциплина доступа к характеристикам позволяет вызывать параллельно те методы, которые в модельном времени выполняются одновременно.

Наконец, последнее, что следует заметить относительно элементов. Как и всякий метод, каждый элемент имеет входные и возвращаемые параметры. Концепция моделирования предполагает возможность распределенного вычисления элементов, т. е. элемент, вообще говоря, может быть найден разработчиком компоненты на просторах Интернета, поэтому его параметры таковы, какими их сделал его автор, и скорее всего, никак не согласованы с характеристиками компоненты, которые придумал ее разработчик. Поэтому, при описании компоненты обязательно должно быть уделено внимание описанию коммутаций входных параметров

элементов с внутренними и внешними характеристиками компоненты и выходных параметров элементов – с ее внутренними характеристиками.

События. Содержательно, события – это то, что нельзя пропустить при моделировании динамики системы – точки синхронизации различных ее функциональностей, представляемых процессами. Точки, когда получены такие значения характеристик модели и внешних переменных, на которые обязаны отреагировать некоторые процессы компоненты.

Формально событие – функция значений внутренних и внешних переменных в начале шага моделирования. С точки зрения организации имитационных вычислений, событие – это метод, входными параметрами которого является подмножество внутренних и внешних характеристик компоненты, а выходной параметр один – прогнозируемое время до наступления этого события. Если это прогнозируемое время равно нулю – значит, событие уже наступило. События управляют чередованием элементов в процессе.

Для каждой упорядоченной пары элементов процесса $\{A,B\}$, если между ними возможен переход, то ему обязан соответствовать метод-событие $E_{\{A,B\}}$, прогнозирующий время этого перехода. Возможны также события вида $E_{\{A,A\}}$, прерывающее выполнение элемента A, например, если его еще не нужно заканчивать, но он вычислил характеристики компоненты, которые могут повлечь смену элементов других процессов. Процесс перехода должен быть однозначным. Одновременное наступление событий $E_{\{A,B\}}$ и $E_{\{A,C\}}$ говорит лишь о том, что разработчик модели при ее проектировании упустил из своего рассмотрения этот случай, которому, быть может, должен соответствовать переход $\{A,D\}$.

Возможно тем, кто знаком с архитектурой компьютера или с событийно-ориентированными языками программирования такое определение событий покажется странным, и вызовет вопросы типа а где же здесь что-нибудь подобное обработчикам событий? Дело в том, что в моделировании (если, конечно не брать моделирование полунатурное, когда к компьютерному комплексу подключают реальное изделие «в железе») чаще всего наступление события должна определить сама модель – даже если это событие неожиданное, например, случайное, – все равно, в нужный момент модель вполне детерминировано должна запустить генератор случайных чисел, чтобы выяснить не произошло ли оно. А подобный порядок действий вполне адекватно ложится на изложенную выше концепцию событий.

Выполнение компоненты. Компонента – элементарная, но, тем не менее, полноправная модель сложной системы. Поэтому ее можно запустить на выполнение имитационного эксперимента. Конечно, если имеются начальные данные и способ нахождения внешних характеристик компоненты в моменты событий.

Правила запуска компоненты на выполнение следующие. Во-первых, задается стандартный шаг моделирования Δt. Во-вторых, считается, что в начале шага моделирования известны текущие элементы всех процессов и все внутренние и внешние характеристики модели. Далее,

1. Вычисляются события связанные с текущими элементами процессов. Если есть наступившие события, проверяется нет ли переходов к быстрым (сосредоточенным) элементам, если они есть – выполняются быстрые элементы (они становятся текущими), затем возврат к началу п.1; если нет переходов к быстрым элементам – совершаются переходы к новым медленным (распределенным) элементам, затем возврат к началу п.1.
2. Если нет наступивших событий – из всех прогнозов событий выбирается ближайший $\Delta\tau$.
3. Если стандартный шаг моделирования не превосходит прогнозируемого времени до ближайшего события, $\Delta t \le \Delta\tau$ – моделируем текущие распределенные элементы со стандартным шагом Δt. В противном случае – моделируем их с шагом времени до ближайшего спрогнозированного события $\Delta\tau$.
4. Возвращаемся к началу п.1.

В связи с приведенными правилами выполнения компоненты могут возникнуть вопросы: не может ли выполнение быстрых элементов в п. 1 а также уменьшение шага времени в п. 3 привести к зацикливанию программы за счет возникновения точек накопления системных событий. Подробно этот вопрос анализируется в [12]. К сожалению полную гарантию можно дать лишь для непрерывных систем, описываемых дифференциальными уравнениями с Липшицевой правой частью. Для более широкого класса так называемых Лапласовских моделей [12] можно доказать (неконструктивно) существование нециклящегося набора событий.

Еще заметим, что раз мы требовали от элементов однозначности имитационных вычислений и сумели этого добиться – на выполнение все одновременно выполняющиеся в модельном времени элементы можно запускать асинхронно, т.е. загружать ими имеющиеся ядра процессора или же доступные распределенные компьютеры. Если же этого почему-то добиться не удалось – у системы поддержки выполнения модели есть полная информация о том, кто что меняет (собственно это она должна будет по выполнении элементов обновить соответствующие данные в базе) – и она обязана выдать соответствующую диагностику ошибки времени выполнения.

Комплексы. Компоненты могут объединяться в комплекс, при этом (необязательно) может оказаться, что некоторые компоненты явно моделируют внешние переменные некоторых других компонент. Для того, чтобы полностью описать комплекс, достаточно указать:

- Какие компоненты и в каком количестве экземпляров в него входят.
- Коммутацию компонент внутри комплекса, если она имеет место, т. е., какие внутренние переменные каких компонент являются какими внешними переменными и каких именно компонент комплекса.

При объединении компонент в комплекс, следует иметь в виду, что однозначность вычислительного процесса может быть потеряна. Так может произойти, если по каким-то причинам, несколько компонент вычисляют одну и ту же характеристику моделируемого явления. В таком случае можно ввести в комплекс новую компоненту, которая в качестве внешних переменных получает весь многозначный набор значений упомянутой характеристики, а в качестве внутренней переменной каким-то образом вычисляет единственное ее значение.

Комплекс как компонента. Комплекс, состоящий из многих компонент, вовне может проявляться в качестве единой компоненты.

Введем следующую операцию объединения компонент комплекса:

1. Внутренними переменными комплекса объявляется объединение внутренних переменных всех его компонент.
2. Процессами комплекса объявляется объединение всех процессов его компонент.
3. Методами комплекса объявляется объединение всех методов его компонент.
4. Событиями комплекса объявляется объединение всех событий его компонент.
5. Внешними переменными комплекса объявляется объединение всех внешних переменных его компонент, из которого исключаются все те переменные, которые моделируются явно какими-либо компонентами.

Операция объединения превращает комплекс в компоненту. Этот факт позволяет строить модель как фрактальную конструкцию, сложность которой (и соответственно подробность моделирования) ограничивается лишь желанием разработчика.

Для описания состава и устройства сложной системы разработан специальный декларативный язык ЯОКК (язык описания комплексов и компонент). Он является развитием в сторону упрощения языка MISS [10] и описан в [11,12].

Отметим, что применение декларативного программирования в имитационном моделировании не ограничивается описанием устройства сложной системы. Если система и в самом деле достаточно сложна, всегда возникает вопрос рациональной организации ее данных, т. е. проектирования и описания базы данных. Общепринятым средством для этого является язык SQL, который применительно к задаче описания баз данных является декларативным, хотя может быть и императивным, если речь пойдет о выборках и обновлении данных.

Еще одна область имитационного моделирования, ожидающая плодотворного применения декларативных описаний – область подготовки презентаций результатов моделирования. Здесь авторам не известны языки описания подобных презентаций. Тем не менее, необходимость в них явно имеется. Самое простое, но тем не менее, весьма полезное, что можно было бы здесь сделать – это по таблице – выборке из базы данных строить диаграммы, так как это делают мастера диаграмм в MS Excel или в OpenOffice Calc. Второе – использование популярных в последнее время геоинформационных систем – отображение пиктограммами на интересующей карте динамики перемещения моделируемых объектов. Третье – использование простейшей анимации в стиле Flash – морфинг форм и перемещение объектов с возможным изменением размеров. Более серьезная анимация пусть пока остается прерогативой компьютерных игр.

Основной вывод – модель сложной системы сложна не только сложностью своих имитационных вычислений (хотя, несомненно, эти вычисления могут быть весьма сложны) – на долю собственно императивных вычислений остается не более четверти сложности всего проекта. Не менее сложны также организация данных модели, организация устройства модели (состав, связи, поведение компонент), а также организация отображения результатов моделирования. Последние три задачи обычно допускают декларативное описание.

Работа выполнена при финансовой поддержке РФФИ, грант № 10-07-00176.

ЛИТЕРАТУРА:

1. Brodsky Yury I. Simulation Software //System Analysis and Modeling of Integrated World Systems – Volume 1, Oxford: EOLSS Publishers Co. Ltd., 2009, P. 287-298.
2. Brodsky Yury I., Tokarev Vladislav V. Fundamentals of simulation for complex systems. //System Analysis and Modeling of Integrated World Systems – Volume 1, Oxford: EOLSS Publishers Co. Ltd., 2009, P. 235-250.
3. Hewitt Carl Viewing Control Structures as Patterns of Passing Messages //Journal of Artificial Intelligence. June 1977.
4. Kuhl F., Weatherly R., Dahmann J. Creating Computer Simulation Systems: An Introduction to the High Level Architecture NY: Prentice Hall PTR, 1999. – 212 p.
5. Shoham Y. Agent-oriented programming //Artificial Intelligence, vol. 60, 1993, P. 51-92.

6. Shoham Y. MULTIAGENT SYSTEMS: Algorithmic, Game-Theoretic, and Logical Foundations Cambridge: Cambridge University Press, 2010, 532 p.
7. Zave, P. A compositional approach to multiparadigm programming. IEEE Software, 6(5): 15—25, September 1989.
8. Бродский Ю.И., Лебедев В.Ю., Огарышев В.Ф., Павловский Ю.Н., Савин Г.И. Общие проблемы моделирования сложных организационно-технических систем //Вопросы кибернетики. Проблемы математического моделирования и экспертные системы, М.: Научный совет АН СССР по комплексной проблеме «Кибернетика», 1990, С. 42-48.
9. *Бродский Ю.И., Лебедев В.Ю.* Инструментальная система для построения имитационных моделей хорошо структурированных организационно-технических комплексов //Вопросы кибернетики. Проблемы математического моделирования и экспертные системы, М.: Научный совет АН СССР по комплексной проблеме «Кибернетика», 1990, С. 49-64.
10. *Бродский Ю.И., Лебедев В.Ю.* Инструментальная система имитации MISS М.: ВЦ АН СССР, 1991, 180 с.
11. *Бродский Ю.И., Павловский Ю.Н.* Разработка инструментальной системы распределенного имитационного моделирования. //Информационные технологии и вычислительные системы, №4, 2009, С. 9-21.
12. *Бродский Ю.И.* Распределенное имитационное моделирование сложных систем М.: ВЦ РАН, 2010, 156 с.
13. *Бусленко Н.П.* Моделирование сложных систем М.: Наука, 1978, 400 с.
14. *Буч Г., Рамбо Дж., Джекобсон А.* Язык UML. Руководство пользователя— 2-е изд. — М., СПб.: ДМК Пресс, Питер, 2004. — 432 с.
15. *Осоргин А.Е.* AnyLogic 6. Лабораторный практикум Самара: ПГК, 2011, 100 с.