УДК 681.3

А.Е. Дорошенко, В.А. Иовчев

О МЕТОДЕ ПРОЕКТИРОВАНИЯ АБСТРАКТНОГО ТИПА ДАННЫХ В АЛГЕБРЕ АЛГОРИТМИКИ

Предлагается метод проектирования расширенного абстрактного типа данных и алгебраического класса. Данный абстрактный тип данных является необходимым ключевым звеном между этапами спецификации и проектирования. Рассмотрена проблема полноты расширенного абстрактного типа данных и предложено решение в качестве достаточной полноты.

Введение

К числу актуальных проблем алгебры алгоритмики (АА) [1 – 3] относятся средства описания и реализации рекурсии и параллелизма, а также формализация концепции абстрактного типа данных (АТД). Решение этих задач тесно связано с важными приложениями в современных объектно-ориентированных вычислительных средах. В Украине исследования в данной области восходят к фундаментальным работам В.М. Глушкова по теории систем алгоритмических алгебр (САА) [2, 3].

Современное состояние исследований по применению алгебры алгоритмики связано с решением ряда важных теоретических и практических задач построения сверхвысокого уровня языков проектирования, а также средств автоматизации процесса синтеза (сборки, трансформации) программ в объектно-ориентированных вычислительных средах [4 – 6]. Следует отметить, что множественность указанных языков определяется концепцией алгоритмического клона, системы образующих (СО) которого соответствуют подходящей парадигме программирования (структурной, неструктурной, объектно-ориентированной и др.) [6 – 8].

Цель данной работы – показать возможность развития концепции абстрактного типа данных [2 – 5] в направлении объектно-ориентированной парадигмы в рамках алгебры алгоритмики. В статье показано, как с помощью алгоритмических описаний АА можно провести проектирование объекта менее затратным способом с последующим погружением в произвольно выбранную вычислительную среду с реализацией на объектно-ориентированном языке (Delphi, C++, Java и др.) [6 – 8].

Материал данной статьи подчинён следующей структуре. В разделе 1 дается краткая характеристика алгебры алгоритмики. Раздел 2 посвящен обзору зарубежных публикаций по направлению АТД и технике алгебраического проектирования классов. Раздел 3 описывает стили спецификаций, необходимые для описания метода проектирования. Концепция актуального АТД в АА приведена в разделе 4. Раздел 5 содержит описание метода построения расширенного АТД. Проблема полноты расширенного АТД описывается в разделе 6. Иллюстративный пример процесса построения АТД Стек содержится в разделе 7. Раздел 8 характеризует объектно-ориентированное решение в АА. Раздел 9 содержит описание Класса на основе расширенного АТД. Раздел 10 содержит иллюстративную спецификацию АТД Сорт и эффективного Класса Сорт. В заключении приведены выводы и дальнейшие перспективы.

1. Алгебра алгоритмики

АА положена в основу алгебраической теории алгоритмов, в рамках которой осуществляется разработка средств представления, накопления, конструирования и классификации алгоритмических знаний, относящихся к различным предметным областям, в частности, к задачам символьной мультиобработки (сортировка, поиск, процессирование языков)[1, 4, 5].

Алгебра алгоритмики – это двухуровневая система:

© А.Е. Дорошенко, В.А. Иовчев, 2012

- верхний уровень, ориентирован на проектирование неинтерпретированных схем, здесь используется аппарат теории клонов в связи с построением подходящей алгебры алгоритмов и формированием её алгебраических основ;
- на втором уровне осуществляется конкретизация неинтерпретированных схем и построение прикладных алгебр алгоритмов, ориентированных на выбранные предметные области.

На всех уровнях алгебры алгоритмики применяются метаправила проектирования схем: свёртка (укрупнение), развёртка (детализация), переинтерпретация (сочетание свёртки и развёртки) и трансформация, состоящая в совершенствовании схем на основе применения аппарата тождеств, квазитождеств и соотношений, характеризующих свойства операций алгебры алгоритмов и выбранной предметной области [3 – 5].

С АА связаны три формы представления (спецификации или алгебро-алгоритмические модели) алгоритмов [3]:

- аналитическая – представление алгоритма в виде формулы в выбранной алгебре алгоритмов, удобна для трансформации, в частности для улучшения по выбранным критериям (память, быстродействие и др.);
- текстовая (естественно-лингвистическая) – представление алгоритма на естественном (понятном пользователю) языке в терминах выбранной предметной области, удобна для диалогового проектирования правильных алгоритмов и программ на разных входных языках;
- визуальная (графовая, граф-схемная) – представление алгоритма в виде граф-схем Калужнина, характеризуется в первую очередь наглядностью в процессе диалогового проектирования и, кроме того, является промежуточным звеном при переходе к UML-диаграммам.

Высокоуровневость и взаимодополняемость формализмов предлагаемых АА по сравнению с традиционными языками программирования (ЯП), вместе с теорией клонов и использованием метаправил обеспечивает надежный фундамент для проектирования и синтеза программ в объектно-ориентированных и распределенных (Grid) средах [9].

2. Техника алгебраического проектирования классов

Первые труды по абстрактным типам данных появились в начале 1970-х. Среди них наиболее известны работа Хоара о доказательстве корректности представлений данных [10], в которой было введено понятие абстракции функций, и работа Парнаса по сокрытию информации [11]. Конечно, абстрактные типы данных не ограничиваются вопросами сокрытия информации, хотя многие их элементарные изложения дальше этого не идут.

Собственно АТД были введены Лисков и Зиллес [12]; более глубокие алгебраические представления были приведены в [13 – 14]. Группа ADJ (Гоген, Тэтчер, Вагнер) исследовала алгебраические основания абстрактных типов данных, используя теорию категорий [15].

На основе абстрактных типов данных базируются несколько языков спецификаций. Двумя результатами группы ADJ являются CLEAR [16] и OBJ-2 [17], а также Larch, предложенный Гуттаг, Хорнинг и Винг [18].

Идея "разделения интересов" является центральной в работах Э. Дейкстры, в частности, в его книге "Дисциплина программирования" [19]. Суть ее состоит в разделении программного решения на отдельные функции, которые минимально перекрываются в функциональности. Можно также сказать, что "Интерес" является синонимом "функции" или "поведения". Данная идея предполагает достижение прогресса за счет модульности и инкапсуляции. Как пример, проектирование информационных систем базируется на разделении "интересов", а конкретнее на уровень представления, уровень бизнес-логики, уровня доступа к данным, уровень базы данных и т. д.

Понятие достаточной полноты впервые опубликовано исследователями Гуттаг и Хорнинг [20].

Поскольку под классом можно понимать пару «тип данных + функции», а в рассматриваемой технике помимо проек-

тирования собственно АТД выполняется проектирование функций работы с ним, то это эту технику можно считать техникой алгебраического проектирования классов (АПК) [20 – 28].

Чтобы получить надлежащие описания объектов, разрабатываемый метод АПК должен удовлетворять трем условиям:

1) описания должны быть точными и недвусмысленными;
2) они должны быть полными - или, по крайней мере, иметь в каждом конкретном случае нужную нам полноту (некоторые детали можно намеренно опускать);
3) они не должны быть «излишне специфицированы».

Основой для третьего условия являются результаты изучения Лиенц и Свонсон стоимости сопровождения [26]. Было установлено, что более 17 % стоимости программного обеспечения (ПО) приходится на изменения в форматах данных. Ясно, что метод, который ставит анализ и проектирование в зависимость от физического представления структур данных, не обеспечит разработку достаточно гибкого ПО [21]. Поэтому при использовании объектов или типов объектов в качестве основы для архитектуры системы требуется найти лучший способ описания, чем конкретное представление.

3. Стиль спецификаций

Выбор конкретного стиля спецификаций несет как ряд преимуществ, так и ряд ограничений. В рамках данной проблемы существует как минимум четыре альтернативы [22, 28]:

- аппликативный последовательный – функциональное программирование без переменных и параллельных вычислений;
- императивный последовательный – программирование с переменными, присваиванием, циклами и т. д., но без параллельных вычислений;
- аппликативный конкурентный – функциональное программирование с параллельными вычислениями;
- императивный конкурентный – программирование с переменными, присваиванием, циклами и т. д. и параллельными вычислениями.

Конкурентные стили больше зависят от дальнейшего представления и реализации проектируемого решения, и аппликативный конкурентный стиль считается неподходящим как базис [22, 28].

Разница между аппликативным и императивным стилями такая же, как и разница между абстрактными и конкретными стилями. Под абстрактностью понимается свойство спецификаций иметь множество открытых альтернативных вариантов формализации. Другими словами, чем меньше проектных решений принимается в схеме, тем больше в ней вариантов и соответственно тем более она абстрактна.

Понятие проектного решения включает в себя такие решения [22, 28]:

- как определить модуль;
- «конкретизация» структур данных;
- «конкретизация» алгоритмов;
- использование переменных;
- используемые образцы данных для обмена информации;
- использование каналов обмена данных.

Таким образом, можно выделить общие категории:

- абстрактно аппликативная – описание, содержащее абстрактные типы и сигнатуры функций с аксиомами, но не явные определения;
- конкретно аппликативная – описание, содержащее конкретные типы и явные определения;
- абстрактно императивная – в описании нет явного определения переменных, вместо их используется абстрактное ключевое слово, присутствуют описания аксиом;
- конкретно императивная – описание, содержащее определения переменных и явные определения функций;
- абстрактно конкурентная – описание, не содержащее явных определений каналов, вместо их используется абстракт-

ное ключевое слово, присутствуют описания аксиом;

- конкретно конкурентная – описание, содержащее явные определения переменных, каналов и функций.

Отметим, что эти различия более относительные, чем абсолютные. Описание может быть в одном смысле абстрактным и конкретным в другом. Данная неопределенность имеет сходство с вышеупомянутой (раздел 2) идеей "разделения интересов" Э. Дейкстры [19].

В итоге, спецификации могут содержать как комбинации разных стилей, так и разные степени абстракций.

4. Концепция абстрактного типа данных в АА

Все языки программирования построены на абстракции [29]. Сложность решения задачи напрямую зависит от типа и качества абстракции.

Концепция абстрактного типа данных (АТД) состоит в фиксации типов (сортов) обрабатываемых данных и средств доступа к ним (логических условий и операторов), допустимых при конструировании алгоритмов и программ обработки данных указанных типов. Для АТД характерен принцип инкапсуляции, в соответствии с которым обработка данных допустима лишь определенными для них средствами доступа [3, 4]. Также данный принцип идентичен инкапсуляции в объектно-ориентированном программировании (ООП).

Различают два уровня представления АТД. На верхнем (абстрактном) уровне перечисляются типы данных и сигнатура АТД, в рамках которой обозначаются средства доступа данных. На нижнем уровне (реализации) разрабатываются представления типов для выбранного (целевого) языка программирования и совокупность оформленных в данном языке процедур, реализующих доступ к обрабатываемым данным соответствии с сигнатурой АТД. Можно сказать, что "видимая" часть абстрактного типа данных – наименования зафиксированных средств доступа, тогда как их реализации "невидимы", недоступны для программиста [3, 4].

Для формализации понятия абстрактного типа данных в АА используются многоосновные (многосортные) алгебраические системы (МАС). Определение МАС представляет собой обобщение понятий модели и алгебры. Следует отметить, что математический фундамент алгоритмики – алгебры алгоритмов – это двухосновные алгебраические системы, ориентированные на формализованное описание и преобразования алгоритмов. А также, основы различных алгебр алгоритмов – множества операторов и предикатов (логических условий), порожденных функциями, входящими в сигнатуру АТД, связанного с выбранным классом задач [3 – 6].

Многоосновная алгебраическая система имеет вид:

$$MAC = < Базис; Сигнатура >$$

где,

$Базис - \left\{ \begin{array}{l} q_{1x} \mid q_{1x} \in Q_1 \\ \ldots \\ q_{nx} \mid q_{nx} \in Q_n \end{array} \right\}$ – совокупность сортов (основ) $Q_i (1 \le i \le n)$,

Сигнатура – это объединение предикатов (логических условий) и операций (операторов), определенных на совокупности основ [3, 4].

Операции и предикаты, входящие в сигнатуру рассмотренного АТД, определенные на совокупностях основ, могут рассматриваться в качестве базисных (элементарных) при построении различных алгоритмов обработки последовательностей данных из основ, в частности, алгоритмов сортировки.

5. Расширение абстрактного типа данных

Процесс разработки программного обеспечения проходит этапы известные как анализ или спецификация, проектирование и реализация [20]. Переход от проектирования к реализации – это просто движение от одного явного вида к другому: форма при проектировании более абстрактна и ближе к математическим понятиям, а при реализации более конкретна и ближе к компьютеру, но обе они являются явными. В общих случаях объектная тех-

нология почти стирает различия между проектированием и реализацией [20, 29]. Чего нельзя сказать о переходе от спецификации к проектированию. Главная проблема данного перехода состоит в отсутствии промежуточного звена обладающего рядом свойств:

- спецификация должна быть формальным математическим описанием;
- иметь математическую модель для описания не всюду определенных операций;
- отсутствие императивных состояний присущих программам;
- иметь общий вид для многоразового использования, как для решения конкретной задачи, так и для других, имеющих разные пути решения;
- являть четкий переход от неявного к явному.

Как решение данной проблемы предлагается метод построения расширенного абстрактного типа данных.

Пусть некая *МАС* описывает *АТД*(*T*). Текущего описания *АТД*(*T*) недостаточно для полной спецификации АТД. Возникает проблема полного понимания типа (сорта) данного АТД. Для конкретизации предлагается дополнительная формализация спецификаций, а именно:

- тип;
- функции;
- аксиомы;
- предусловия.

5.1. Тип. Для указания специфицируемых типов следует дать определение типа. Тип – это некое множество, характеризуемое МАС, функциями, аксиомами и предусловиями. Как пример, тип Стек – это множество всех возможных стеков, тип Числовые массивы – это множество всех числовых массивов и т.д.

Указывая:

$$Тип:$$
$$ADT[G]$$

подразумевается, что спецификация относится к одному абстрактному типу данных *ADT* задающему объекты *G*. Другими словами $ADT[G]$ – это не один конкретный объект, а совокупность объектов.

Пусть *ADT(T)* и *ADT'(T)* произвольные абстрактные типы данных тогда:

- любой описанный АТД может входить в базис другого АТД:

$ADT(T):$

$$Базис:\left|\begin{matrix}\{q_i \mid q_i \in Q_i\}\\ \vdots \\ \{q_j \mid q_j \in ADT'(T)\}\end{matrix}\right|, Q_i\,(1 \le i \le n);$$

- вхождение в базис АТД позволяет применяться рекурсивно:

$ADT(T)$:

$$Базис:\left|\begin{matrix}\{q_i \mid q_i \in Q_i\}\\ \vdots \\ \{q_j \mid q_j \in ADT(T)\}\end{matrix}\right|, Q_i\,(1 \le i \le n).$$

Определение. Экземпляром абстрактного типа данных *АТД*(*T*) называется объект, принадлежащий множеству описываемому спецификацией *АТД*(*T*).

5.2. Функции – это понятие, содержащее определение полной, частной функций, а также категорий функций.

Под *полной* функцией понимается $f(x_1, x_2, ..., x_n)$, где переменные x принимают значение из множества D, а функция принимает значение из множества V, реализует отображение из D в V. Множество D – область определения (исходное множество), а V – область значения функции (результирующее множество).

Другими словами, функция – это механизм для получения значения, принадлежащего результирующему множеству, по допустимому входу, принадлежащему исходному множеству.

При описании АТД функции определяются не полностью, вводятся только их сигнатуры. Т. е. списки типов их аргументов и результата. Соответственно, множества D и V входят в описанные типы АТД.

$$f : < x_1, x_2, ..., x_n > \to y,$$
$$x \in D, y \in V.$$

Функция $f(x_1, x_2, ..., x_n)$, из области определения *D*, в результирующее множество *V* называется *частной*, если она определена не для всех элементов *D*. Отсюда, функция, не являющаяся частной, называется полной. Соответственно, областью определения *D* произвольной частной функции $f(x_1, x_2, ..., x_n)$, является подмножество таких элементов *D*, для которых эта функция имеет элементы в результирующем множестве *V*.

$$f : < x_1, x_2, ..., x_n > \not\mapsto y,$$
$$x \in D', y \in V',$$
$$D' \subset D,$$
$$V' \subset V.$$

Примером частной функции является функция обращения действительных чисел *inv*, значение которой на действительной оси *x* равно $inv(x) = 1/x$. Пусть *N* – множество всех действительных чисел, а функция *inv* не определена на $x = 0$, тогда она определяется как частная функция на *N*. Область определения ее является подмножество *R* от *N*, т. е. множество всех действительных чисел, кроме нуля.

Функции предлагается разделить на три *категории*: создающие объекты (конструкторы), возвращающие информацию об объектах (запросы) и изменяющие объекты (команды). В современном ООП эти три категории называются "конструктор", "аксессор" и "модификатор". Конкретизация данных категорий такова:

- функция-конструктор моделирует операцию, использующую аргументы, либо не использующую, которая создает экземпляры *T* из экземпляров других типов;
- функция-запрос моделирует операцию, которая устанавливает свойство *T*, описанное в терминах экземпляров других типов;
- функция-команда моделирует операцию, которая по существующему экземпляру *T* и, возможно экземплярам других типов выдает новые экземпляры типа *T*.

5.3. Аксиомы. Вышерассмотренные данные описываются посредством задания списка функций, применимых к экземплярам АТД. Но описание метода алгебраического проектирования классов подразумевает, что выбор конкретного представления уступает способу описания, т. е., что всякое явное определение обязывает выбрать некоторое представление, а для описания АТД это недопустимо. Допустимы только неявные определения, но вышеописанных деклараций функций явно недостаточно и требуется еще дополнительные определения в виде аксиом. Отметим также, что данные определения не содержат конкретные значения функций в спецификации АТД. По сути, аксиомы являются предикатами (в смысле логики), выражающими истинность некоторых свойств, для всех возможных значений из АТД.

Отметим, что имеются два вида «неявности»:

- метод определяет неявно некоторое множество объектов, задавая применимые к ним функции. Но, из этого определения не следует, что перечислены все операции. Т.е. в процессе построения и/или использования могут быть добавлены и другие;
- сами функции также определены неявно. Свойства данных функций задают аксиомы. Т. е. утверждения о полноте нет, и в процессе проектирования они могут приобрести дополнительные свойства.

Эта неявность и является важным ключевым аспектом АТД и, как следствие, их воплощения в классы ООП. Такая неявность предполагает открытость определений. Т.е. всегда можно добавить новые свойства АТД или класса. В ООП подобным механизмом расширения является наследование.

5.4. Предусловия. Процесс расширенной формализации спецификаций АТД неизбежно сталкивается с проблемой необходимости частных функций. В п. 5.2, в функциях при их объявлении используются перечеркнутые стрелки, таким образом, указывая, что эти функции являются частными. В процессе проектирования ПО

очевидно, что не каждая операция применима ко всем объектам (т. е. является частной) и что они часто являются источником ошибок. Даже если описание частной функции $f(x)$ корректно, и если x принадлежит D, это не дает гарантии что на x из D функция $f(x)$ определена. Для этого спецификация АТД должна содержать заданные области для частных функций. В этом заключен смысл предусловий.

Предусловие p функции f – это характеристическая функция области f. Характеристической функцией подмножества A' множества A называется полная функция $p(x)$ истинная, если $x \in A'$, и ложная в противном случае.

Каждая функция имеет условия, которым должны удовлетворять аргументы функции, чтобы входить в ее область.

Булевское выражение, которое определяет область функции, называется предусловием соответствующей частичной функции.

В итоге, *расширенным АТД* является система $< T, B, S, F, A, P >$, где

T – описываемый тип;
B – совокупность основ (сортов);
S – объединение базисных предикатов и операций, определенных на совокупности основ;
F – функции для обработки основ;
A – аксиомы;
P – предусловия.

6. Полнота АТД

Формализация спецификаций АТД является неявной и неполной [20 – 28]. Так вышеприведенная спецификация выражает все, что нужно знать об объекте, но и не включает ничего, что бы относилось к конкретным реализациям. Методом построения определяется некое множество элементов (объектов) со свойствами и функциями. Так же, нет как формального, так и неформального эталонного документа для определения полноты спецификации АТД. Отметим, что в логических системах, идея полноты относится к возможности доказать все истинные утверждения [20]. Применение этой идеи для спецификаций АТД позволяет определить только достаточную полноту. Достаточная полнота позволяет охватить АТД и не оставить вне спецификации никакое важное свойство.

Определение. Пусть C – схема содержащая одну или более функций некого АТД. Эта схема является *корректной* тогда и только тогда, когда все функции (по рекурсии) имеют правильное число аргументов соответствующих типов и их значения удовлетворяют предусловиям, если они имеются.

Определение. Спецификация *АТД*(T) $< T, B, S, F, A, P >$ является *непротиворечивой* тогда и только тогда, когда для корректно простроенной схемы ее аксиомы позволяют вывести не более одного значения.

Определение. Спецификация *АТД*(T) $< T, B, S, F, A, P >$ является *достаточно полной* тогда и только тогда, когда:

- на каждой основе $\{B_1, ..., B_n\}, B_i (1 \le i \le n)$ определена совокупность операций $S = \{Сигн(o) \cup Сигн(n)\}$, и нет операций из S не определенных на этих основах;
- аксиомы $A = \{@_1, ..., @_n\}, @_i (1 \le i \le n)$ и предусловия $P = \{p_1, ..., p_n\}, p_i (1 \le i \le n)$ позволяют определить корректность схемы C использующей *АТД*(T);
- спецификация АТД является непротиворечивой.

Отметим, что по аналогии, такое решение проблемы полноты используется в разных методах построения алгебраического класса [12, 13, 15, 16, 20, 21].

При дальнейших описаниях возникнет потребность в конкретизации типов, и могут добавиться другие, в зависимости от пути представления (например, разные языки программирования или разные среды выполнения). Функции из неявных определений и аксиом приобретут дополнительные свойства.

Описанные спецификации формируют общую модель на соответствующих структурах данных. Определенные функции дают возможность строить посредством операции суперпозиции [3] более

сложные выражения, а аксиомы в свою очередь упрощают понимание сложных выражений и позволяют получать более простые результаты.

Также следует отметить, что спецификация АТД является формальным математическим описанием и не описывает явных изменений, т. е. является аппликативной. Все свойства АТД моделируются как математические функции, включая конструкторы, запросы и команды.

7. Построение АТД Стек

Проиллюстрируем описанную концепцию. Одним из хорошо изученных примеров является описание типа стек. Стек служит для того, чтобы накапливать и извлекать другие элементы в режиме "последним пришел – первым ушел" (LIFO). Элемент, помещенный в стек последним, будет извлечен из него первым. Стеки присутствуют в дидактических представлениях абстрактных типов данных настолько часто, что Э. Дейкстра как-то заметил, что "абстрактные типы данных являются прекрасной теорией, целью которой является описание стеков" [19, 21].

Далее будет рассматриваться в качестве примера физическое представление стека МАССИВ_ВВЕРХ [21].

Данный стек представляется посредством массива *Представление* и целого числа *Счет*, с диапазоном значений от 0 (для пустого стека) до *Емкость* – размера массива *Представление*, элементы стека хранятся в массиве и индексируются от 1 до *Счет*.

7.1. Базис. Пусть *Базис* – это множество основ *Базис*(q), где q = 1,...,n, а множество $Базис(q) = \{A(q,x) \mid x \in X\}$ – основа q, состоящая из элементов типа (сорта) X. Следовательно, определим базис стека:

$$\left| \begin{array}{l} \{Счет \mid Счет \in Числа\} \\ \{Представление \mid Представление \in Числа\} \\ \{e \mid e \in E2 = \{Истинно, Ложно\}\} \\ \{v \mid v \in Элемент\} \end{array} \right|.$$

Отметим, что в качестве элементов стека может выступать сам стек. В этом случае необходимо добавить:

$$\left|\{v' \mid v' \in Стек\}\right|.$$

7.2. Сигнатура. На множествах *Базис* определим сигнатуру предикатов и операций, входящих в качестве элементарных логических условий и операторов в схему Стек.

Сигн(р):

$Счет < Емкость$ – предикат, истинный, если *Счет* не достиг *Емкость*;

$Счет == 0$ – предикат, истинный, если *Счет* равен 0;

Сигн(о):

$Записать(v, Счет, Представление)$ – записать элемент в массиве *Представление* с позицией *Счет*;

$Читать(Счет, Представление)$ – прочитать элемент в массиве *Представление* с позицией *Счет*;

$Удалить(Счет, Представление)$ – удалить элемент в массиве *Представление* с позицией *Счет*;

$Увеличить(Счет)$ – увеличить *Счет* на единицу типа;

$Уменьшить(Счет)$ – уменьшить *Счет* на единицу типа.

Таким образом, спецификация относится к одному абстрактному типу данных – стек, задающему стеки объектов произвольного типа S.

7.3. Функции.

- *Поместить* – функция-команда, возвращает новое состояние стека с новым элементом, помещенным в его вершину.

$$Поместить: S \times v \to S.$$

- *Извлечь* – функция-команда, возвращает новое состояние стека с вытолкнутым верхним элементом, если таковой был. Объяснение, как учесть возможность пустого стека, из вершины которого нечего удалять, последует далее.

$$Извлечь: S \nrightarrow S.$$

- *Элемент* – функция-запрос, возвращает верхний элемент стека, если таковой имеется.

$$Элемент : S \not\rightarrow v.$$

- *Пустой* – функция-запрос, выявляет пустоту стека, ее результатом является логическое значение из множества *E2* (истина или ложь).

$$Пустой : S \rightarrow e.$$

- *Новый* – функция-конструктор, создает пустой стек.

$$Новый : Стек[S] \rightarrow S.$$

Функция *Поместить* в качестве аргумента принимает пары вида (*S,v*), в которой *S* – экземпляр типа *Стек*[*S*], а *v* – экземпляр типа *Элемент* и возвращает в качестве результата экземпляр типа *Стек*[*S*]. В сигнатуре таких функций как *Извлечь* и *Элемент* под перечеркнутой стрелкой понимается, что эти функции применимы не ко всем элементам множества входов. Описание функции *Новый* сокращенно в виду того, что результат один, а аргументов нет.

7.4. Аксиомы. Формализация их для АТД Стек такова:

$$@1 : Элемент(Поместить(S, v)) = v;$$

$$@2 : Извлечь(Поместить(S, v)) = S;$$

$$@3 : Пустой(Новый) = e_0;$$

$$@4 : Пустой(Поместить(S, v)) = e_1;$$

где

$e_0, e_1 \in E2,$

$e_0 = Истинно,$

$e_1 = Ложно.$

Аксиомы @1 и @2 выражают основные свойства стеков LIFO, первая – вершина содержит последний помещенный элемент *m*, вторая, – после удаления элемента *m* из *s* получаем *s* который был до помещения в него *m*. Аксиома @3, любой стек, полученный в результате выполнения *Новый* пустой. И аксиома @4, любой стек, полученный после выполнения *Поместить*, другими словами, полученный после помещения элемента в существующий стек, не является пустым.

7.5. Предусловия. Для всякого выражения, содержащего частичные функции, необходимо проверять, что их аргументы удовлетворяют соответствующим предусловиям. В данном случае, существуют две, не полностью определенные функции:

- *Элемент* – у пустого стека нет верхнего элемента;
- *Извлечь* – нельзя удалить элемент из пустого стека.

В разделе функции при их объявлении использованы перечеркнутые стрелки, таким образом, указывая, что эти функции являются частными.

*p*1*: Элемент*(*S*) если *НЕ Пустой*(*S*)

*p*2*: Извлечь*(*S*) если *НЕ Пустой*(*S*).

7.6. Полная спецификация АТД Стек.

АТД *Стек*:

1. Тип:

   Стек[*S*].

2. Базис:

$$\left| \begin{array}{l} \{Счет \mid Счет \in Числа\} \\ \{Представление \mid Представление \in Числа\} \\ \{e \mid e \in E2 = \{Истинно, Ложно\}\} \\ \{v \mid v \in Элемент\} \end{array} \right|.$$

3. Сигнатура:

$$Сигн(o) = \left\{ \begin{array}{l} Записать(v, Счет, Представление), \\ Читать(Счет, Представление), \\ Удалить(Счет, Представление), \\ Увеличить(Счет), \\ Уменьшить(Счет) \end{array} \right\},$$

$$Сигн(p) = \left\{ \begin{array}{l} (Счет == 0), \\ (Счет < Емкость) \end{array} \right\}$$

4. Функции:

$Новый : Стек[S] \rightarrow S,$

$Поместить : S \times v \rightarrow S,$

$Извлечь : S \not\rightarrow S,$

$Элемент : S \not\rightarrow v,$

$Пустой : S \to e.$

5. Аксиомы:

$@1 : Элемент(Поместить(S, v)) = v,$

$@2 : Извлечь(Поместить(S, v)) = S,$

$@3 : Пустой(Новый) = e_0,$

$@4 : Пустой(Поместить(S, v)) = e_1,$

где

$e_0, e_1 \in E2,$

$e_0 = Истинно,$

$e_1 = Ложно.$

6. Предусловия:

*p*1*: Элемент*(S) если *НЕ Пустой*(S)
*p*2*: Извлечь*(S) если *НЕ Пустой*(S)

8. Объектно-ориентированное решение в АА

При возникновении проблемы построения модульной структуры, основанной на типах объектов, АТД представляет гибкое решение содержащее механизм описания высокого уровня, при этом не связанный с особенностями реализации. Что в свою очередь является подходящим фундаментом для развития объектно-ориентированного направления в рамках АА.

Таким образом, новое объектно-ориентированное решение строится (на уровне анализа, проектирования и реализации) как совокупность взаимодействующих, с разной степенью описания, АТД.

Конструирование объектно-ориентированного решения в АА – это описание решения как структурированной совокупности полной и/или частичной реализации абстрактных типов данных со следующей структурой:

- в основе лежит понятие АТД;
- для конструирования программ нужны классы – реализации АТД;
- реализации не обязаны быть полными;
- в основе структурирования лежат отношения между классами и наследование.

Отсюда следует, что переход от спецификации к проектированию – это идентификация каждой абстракции. На рисунке показано переходы, в процессе конструирования объектно-ориентированного решения, между формами и их экземплярами.

9. Класс

Определение. *Класс* – это абстрактный тип данных, с описанной частичной или полной реализацией.

Под ***классом*** понимается система $Класс = (T, B, F, I)$, где:

- T – тип, характеризуемый абстрактным типом данных, либо (в случае множественной реализации) множеством абстрактных типов данных.

$$T : \{АТД_1, ..., АТД_n\},$$
$$АТД_j\,(1 \le j \le n).$$

- B – множество, полученное в результате объединения базиса из $АТД_i$ и конкретизации (привязки к целевой платформе, или «реализации») базиса.

$$B = Базис^{АТД} \cup Базис.$$

- F – описание алгоритмов функций посредством *Сигнатуры* из АТД в строгих рамках аксиом и предусловий. Как пример, в статье в качестве описания выбрана Алгебра Дейкстры [3,19].

$$f ::= S,$$

где S – структурная схема

- I – множество, характеризующее интерфейс класса с точки зрения ООП. Содержит декларации функций, представляемые открыто классом. Отметим, что функции описанные в АТД, но не содержащиеся в I, следует рассматривать как внутренние или «приватные».

$$I : \begin{Bmatrix} f_0 \in Функции \Rightarrow T \\ \ldots \\ f_n \in Функции \Rightarrow T \end{Bmatrix}.$$

АТД может иметь разную степень описания, а класс – разную степень реализации. По сути, реализация это уже сформированная, конкретная привязка к некой платформе. Полностью реализованный класс называется ***эффективным***. Частично реализованный – называется ***отложенным***. Любой класс является либо отложенным, либо эффективным [21].

Для получения эффективного класса необходимо описать:

- спецификации АТД;
- выбор представления (View);
- отображение из АТД в View в виде множества компонентов (features), каждый из которых реализует одну из функций в рамках представления, и при этом строго удовлетворяет аксиомам и предусловиям.

Следует отметить, что данные компоненты в процессе реализации могут сформироваться как в процедуры и/или функции, так и в качестве полей данных или атрибутов.

Часто при разработке программного обеспечения объектно-ориентированная система в завершенном виде не содержит данных о проектировании, разработке и реализации. Тем, кто будет обслуживать такую систему (расширять, переносить, отлаживать), придется полностью изучить ее, потратив на это время и ресурсы. В качестве решения предлагается обеспечить систему описанными АТД и/или классами.

Отложенные классы служат для классификации групп связанных типов объектов, описывают важные многократно используемые модули высокого уровня, фиксируют общие свойства поведения. Именно они играют ключевую роль в полиморфизме, а также обеспечении децентрализации и расширяемости программной архитектуры.

Если спецификации АТД являются аппликативными, то в классах аппликативная точка зрения на функции отбрасывается, и команды переопределяются как операции, которые могут изменять объекты. Такое изменение четко отражает императивную парадигму, преобладающую при разработке ПО. Это в свою очередь влечет изменение в аксиомах АТД.

10. Построение АТД Сорт

Проиллюстрируем абстрактный тип данных *Сорт* с описанием эффективного класса *Сорт*. В качестве примера в часть функции взяты алгоритмы сортировки *Раствор*, *Пузырек* и *Шейкер* [3 – 5].

Первым этапом является построение абстрактного типа данных *Сорт*:

АТД *Сорт:*

1. Тип:

Сорт[*C*].

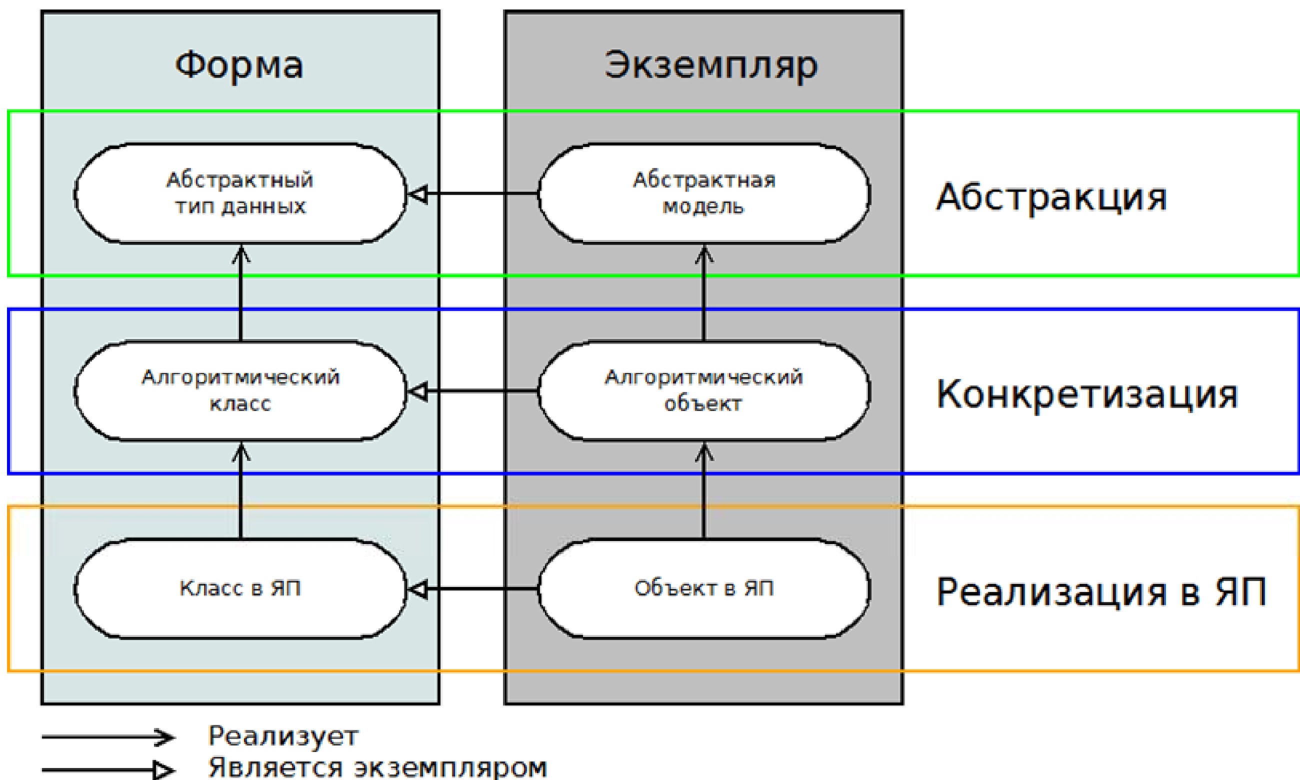

Рисунок. Формы и их экземпляры

2. Базис:

$$\left|\begin{array}{l}\{Массивы^{отсорт} \mid Массивы^{отсорт} \in Массивы\} \\ \left\{\begin{array}{l}Массивы^{не_отсорт} \mid Массивы^{не_отсорт} \\ \in Массивы\end{array}\right\} \\ \{массив^{от} \mid массив^{от} \in Массивы^{отсорт}\} \\ \{массив^{не} \mid массив^{не} \in Массивы^{не_отсорт}\} \\ \{l, r \mid l, r \in массив = массив^{от} \cap массив^{не}\} \\ \{M \mid M \in Маркеры_массива\} \\ \{У \mid У \in Указатели_массива\}\end{array}\right|.$$

3. Сигнатура:

$$Сигн(o) = \left\{\begin{array}{l}Трансп(l, r \mid У), \\ Уст_на_маркер(У, M), \\ Уст_на_указатель(У, У), \\ Сдвиг_вправо(У), \\ Сдвиг_влево(У), \\ E\end{array}\right\},$$

$$Сигн(p) = \left\{\begin{array}{l}Достиг_маркера(У, M), \\ Достиг_указателя(У, У), \\ (l > r \mid У), \\ Упорядочен(массив), \\ Упорядочен_отрезок(У, У)\end{array}\right\},$$

где E – пустой оператор.

4. Функции:

$Новый : Сорт[C] \to C;$

$Раствор : массив^{не} \to массив^{от};$

$Пузырек : массив^{не} \to массив^{от};$

$Шейкер : массив^{не} \to массив^{от}.$

5. Аксиомы:

$@1 : Раствор(массив^{не}) \to массив^{от};$

$@2 : Пузырек(массив^{не}) \to массив^{от};$

$@3 : Шейкер(массив^{не}) \to массив^{от}.$

Таким образом, описанный абстрактный тип данных ориентирован на обработку массивов. В базисном разделе описываются необходимые для обработки данные. Сигнатура содержит необходимый набор операторов и предикатов, информационным множеством которых выступает базис. Далее перечисляются функции, моделирующие соответствующие операции абстрактного типа данных *Сорт*. И завершает описание раздел аксиом. Следует отметить, что данный пример не содержит раздел предусловий в виду отсутствия частично определенных функций.

Следующим и завершающим этапом является построение *эффективного* Класса *Сорт* на основе вышепостроенного АТД *Сорт*:

Эффективный Класс *Сорт*:

1. Тип:

АТД *Сорт.*

2. Базис:

$Базис \cup$

$$\left|\begin{array}{l}\left\{\begin{array}{l}Массивы \mid \\ Массивы = \{Числовые_массивы\}\end{array}\right\} \\ \left\{\begin{array}{l}Массивы \mid \\ Массивы = \{Символьные_цепочки\}\end{array}\right\} \\ \left\{\begin{array}{l}Маркеры_массива \mid \\ Маркеры_массива = \{H, K\}\end{array}\right\} \\ \left\{\begin{array}{l}Указатели_массива \mid \\ Указатели_массива = \{У1, \ldots, УN\}\end{array}\right\}\end{array}\right|.$$

3. Функции:

Раствор ::= {[*Достиг_маркера*(*У, К*)]
([*l*>*r* | *У*] *Трансп*(*l,r* | *У*)*
Уст_на_маркер(*У, Н*), *Сдвиг_вправо*(*У*))}

Пузырек ::= {[*Упорядочен*(*массив*)]
{[*Достиг_маркера*(*У1, К*)]([*l*>*r* | *У1*]
Трансп(*l,r* | *У1*), *Е*)* *Сдвиг_вправо*(*У1*)}*
Уст_на_маркер(*У1, Н*)}

Шейкер ::=
{[*Упорядочен_отрезок*[*У4,У2*]]
{[*Достиг_указатель*(*У1,У2*)]
([*l*>*r* | *У1*] *Трансп*(*l,r* | *У1*)*
Уст_на_указатель(*У3, У1*), *Е*)*
Сдвиг_вправо(*У1*) }*
Уст_на_указатель(*У2, У3*)*
Уст_на_указатель(*У1, У2*)*
([*Упорядочен_отрезок*(*У4, У2*)] *Е,*

{ [*Достиг_указатель*[*У1,У4*]
([*l*>*r* | *У1*] *Трансп*(*l,r* | *У1*)*
Уст_на_указатель(*У3,У1*), *E*)*
Сдвиг_влево(*У1*) }*
Уст_на_указатель(*У4,У3*)*
Уст_на_указатель(*У1,У4*))}

4. Интерфейс:

$$Интерфейс = \begin{Bmatrix} Новый, \\ Раствор(массив), \\ Пузырек(массив), \\ Шейкер(массив) \end{Bmatrix}.$$

Заключение

Таким образом, авторами предложен метод построения расширенного АТД и алгебраического класса. Преимуществами данного метода являются:

- полученная спецификация АТД является формальным математическим описанием, а не текстом программы;
- модель не несет императивных состояний;
- математическая модель для описания не всюду определенных операций;
- аксиомы и предусловия выражают семантику АТД;
- гибким механизмом перехода от анализа и спецификации к проектированию и реализации;
- разработка класса происходит в наиболее общем виде, что необходимо для повторного использования;
- сокращение времени анализа и разбора программного решения для стороннего программиста.

Сформулирована достаточная полнота расширенного АТД как решение проблемы полноты.

В дальнейшем авторы намереваются провести апробацию данного метода на разработке WEB-сервиса и реализацию в виде инструментария АА.

1. *Ноден П., Китте К.* Алгебраическая алгоритмика (с упражнениями и решениями). – М.: Мир, 1999. – 720 с.
2. *Глушков В.М., Цейтлин Г.Е., Ющенко Е.Л.* Алгебра. Языки. Программирование. 3-е изд., перераб. и доп. – Киев: Наук. думка, 1989. – 376 с.
3. *Цейтлин Г. Е.* Введение в алгоритмику. – К.: Сфера, 1999. – 720 с.
4. *Андон Ф.И., Дорошенко А.Е., Цейтлин Г.Е., Яценко Е.А.* Алгеброалгоритмические модели и методы параллельного программирования. – Киев: Академпериодика, 2007. – 634 с.
5. *Цейтлин Г.Е.* Алгебраическая алгоритмика: теория и приложения // Кибернетика и системный анализ. – 2003. – № 1. – С. 8 – 18.
6. *Цейтлин Г.Е., Иовчев В.А., Мусихин А.А.* Ментальные аспекты методов символьной мультиобработки // Проблеми програмування. – 2008. – № 1. – С. 60 – 67.
7. *Иовчев В.А., Мохница А.С.* Инструментальные средства алгебры алгоритмики на платформе WEB 2.0 // Проблеми програмування. (матеріали конф. УкрПрог-2010). – 2010. – № 2 – 3. – С. 547 – 556.
8. *Иовчев В.А., Мохница А.С.* Формальный метод генерации программ в инструментальных средствах алгебры алгоритмики // матеріали конф. TAAPSD'2010.
9. *Дорошенко А.Е., Алистратов О.В., Тырчак Ю.М., Розенблат А.П., Рухлис К.А.* Системы Grid-вычислений – перспектива для научных исследований // Проблеми програмування. – 2005. – № 1. – С. 14 – 38.
10. *Dahl O.-J., Dijkstra E. W. and Hoare C.A.R.*. Structured Programming. Academic Press. 1972.
11. *Parnas D.L.* On the Criteria To Be Used in Decomposing Systems into Modules. December 1972. http://www.cs.umd.edu/class/spring 2003/cmsc838p/Design/criteria.pdf
12. *Barbara Liskov*, Programming with Abstract Data Types, in Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages, Santa Monica, California. – 1974. – P. 50 – 59.
13. *Robert T. Johnson, James B. Morris*: Abstract Data Types in the Model Programming Language. Conference on Data: Abstraction, Definition and Structure. – 1976. – P. 36 – 46.
14. *John Guttag.* Abstract Data Types and the Development of Data Structures. – 1977.Bern, June 1997, 2001.

15. *Gougen J.A., Thatcher J.W., Wagner E.G.:* An initial algebra approach to the specification, correctness, and implementation of abstract data types. In: Current Trends in Programming Methodology, Prentice-Hall, Englewood Cliffs. – 1978. – P. 80 – 149.
16. *Rod M. Burstall, Joseph A. Goguen:* The Semantics of CLEAR, A Specification Language. Abstract Software Specifications. – 1979. – P. 292 – 332.
17. *Futatsugi K. et al.*, Principles of OBJ2, 12th POPL, ACM. – 1985. – P. 52 – 66.
18. *J. Guttag et al*, The Larch Family of Specification Languages, IEEE Trans Soft Eng 2(5). – Sep 1985. – P. 24 – 365.
19. *Dijkstra E.W.* A Discipline of Programming, Prentice-Hall Series in Automatic Computation. – 1976.
20. *Guttag J. V. and Horning J. J.* The algebraic specification of abstract data types., Acta Informatica. – 1978. – Vol. 10. – 27 p.
21. *Bertrand Meyer*. Object-Oriented Software Construction, Second Edition, Prentice Hall. – 1997.
22. *The* RAISE Method Group. The RAISE Development Method, – 1999. http://users.iptele com.net.ua/~agp1/arts/book.pdf.
23. *Birrell N.D., Martyn A.Ould.* A Practical Handbook for Software Development. – February 1988. – 272 p.
24. *Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides*. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley. – 1995.
25. *Jamie Shield*. Towards an Object-Oriented Refinement Calculus. – 2001. Thesis.
26. *Bennet P., Lientz E., Burton Swanson*. Problems in Application Software Maintenance. Commun. ACM. – 1981. – P. 763 – 769.
27. *Пискунов А.Г.* Формализация парадигмы обьектно-ориентированного программирования: критика определения Гради Буча, – 2007. http://i.com.ua/~agp1/ru/oopFormalizm .html
28. *Пискунов А.Г.* The RAISE Method Group: Алгебраическое проектирование класса, 2007, http://www.realcoding.net/article/view/ 4538
29. *Bruce Eckel.* Thinking in Java, 4th edition, 2006.

Получено 14.07.2011

Об авторах:

Дорошенко Анатолий Ефимович,
доктор физико-математических наук, профессор, заведующий отделом теории компьютерных вычислений,

Иовчев Владимир Александрович,
младший научный сотрудник.

Место работы авторов:

Институт программных систем
НАН Украины,
проспект Академика Глушкова, 40.
03680, Киев-187.
Тел. (044) 526 1538
e-mail: dor@isofts.kiev.ua, iovchev.v@gmail.com