ПРОГРАММНАЯ ИНЖЕНЕРИЯ ВЧЕРА И СЕГОДНЯ: ЭВОЛЮЦИЯ ИМПЕРАТИВНОГО ПРОГРАММИРОВАНИЯ

Ермаков И.Е., студент физ.-мат. ф-та,
зам. директора ООО ОЦПИ «Метасистемы» (ermakov@metasystems.ru)

> Но существует одно качество, которое нельзя купить, — это надежность. Цена надежности — погоня за крайней простотой. Это цена, которую очень богатому труднее всего заплатить.
>
> Ч-А. Хоар

I. Обзор. Цель данной статьи − краткий обзор состояния, проблем и перспектив современной программной инженерии (ПИ). Под ПИ будем понимать создание технических систем программного обеспечения (ПО). На сегодняшний день в ПИ существует несколько фундаментальных, ортогональных друг другу методологий программирования (МТП). В основе МТП лежит какая-либо модель алгоритма, абстрактная машина. Основные МТП: *императивная* (машина Тьюринга), *функциональная* (лямбда-исчисление), *логическая* (логика предикатов)[1]. Выбор МТП определяет стиль мышления, инструментарий и набор доступных разработчику приемов. Выбор диктуется особенностями задачи, но в целом можно говорить как о достаточной универсальности всех основных МТП, так и об их взаимопроникновении.

Очевидно, что наиболее распространена в индустрии императивная МТП. Однако последние 15 лет острие лабораторных исследований направлено именно на альтернативные МТП – основные надежды (часто чрезмерные) возлагаются на декларативные подходы, в частности, 4GL[2], и может создаться впечатление, что в области императивных 3GL ничего нового не происходит. На самом деле это не так, что мы и постараемся показать далее. Таким образом, мы будем рассматривать современную ПИ с позиций императивной методологии.

II. Эволюция императивного программирования (ИПР). Императивный алгоритм представляется в виде последовательности команд для некоторого исполнителя, которые переводят данный исполнитель из одного состояния в другое. Выполнение алгоритма − это прохождение последовательности состояний, в процессе которого решается задача − вычисление, обработка данных, управление некоторым процессом[3].

Обычный императивный исполнитель имеет ограниченный набор простых команд и работает с неструктурированными двоичными данными. Преимущество языков программирования высокого уровня (ЯВУ) в том, что они позволяют при написании алгоритма оперировать более высокоуровневыми понятиями (*абстракциями*), нежели машинные. В идеале хотелось бы сократить до минимума разрыв между понятиями предметной области (ПрО) и абстракциями, которые предоставляет ЯВУ. Эволюция ИПР – это в первую очередь развитие абстракций ЯВУ.

В этом развитии особую роль сыграли три ключевых нововведения: зарождение ЯВУ (50-е гг.), структурный подход (70-е гг.) и объектный подход

---

[1] Есть и некоторые другие, например, автоматная. См. [3].

[2] 4th Generation Programming Languages – «языки 4-го поколения», позволяют пользователю описывать (в т.ч. графически) требуемый результат без программирования алгоритма для его достижения.
3GL – «языки 3-го поколения» - классические универсальные языки программирования.

[3] Актуальные сегодня параллельные вычисления предполагают взаимодействие многих исполнителей, каждый из которых может проходить свой граф состояний одновременно с другими.

(80-е гг.). После появления каждого из них эффективность ПИ резко возрастала, поскольку в распоряжении разработчиков оказывались некоторые новые абстракции для конструирования программного кода. В первых ЯВУ этими абстракциями стали: **оператор** − минимальная логическая *единица выполнения* (абстрагирует нас от нескольких физических единиц выполнения), **переменная** − *именованная единица данных* (абстрагирует от физического размещения и представления в памяти), **подпрограмма** – *составная единица выполнения* (позволяет явно выделить логически цельную последовательность операторов). Видим, что уже присутствует составная единица выполнения, однако отсутствуют составные единицы данных − насущной потребности в них не было, так как на заре ИТ преобладали вычислительные задачи. Нет никаких абстракций для *управления выполнением*: арифметический IF[1], GOTO и CALL первых ЯВУ эквивалентны соответствующим машинным командам. Построение кода на прямых передачах управления получило название «стиль спагетти» − он порождал огромные трудности при написании больших программ, зачастую делал невозможным их сопровождение и развитие. Низкая производительность программистов и крайне низкая надежность ПО привели к **первому кризису программирования** (60-е гг.). Очередным прорывом явились идеи *структурного программирования* (СП), провозглашенные Э. Дейкстрой, Н. Виртом и Ч. Хоаром (70-е гг.)[2].

Основные принципы СП: **декомпозиция** – разбиение решаемой задачи на набор подзадач, их *независимая* реализация и затем организация взаимодействия; **блочность** – организация кода в виде отдельных блоков для каждой подзадачи, каждый блок – это подпрограмма (**процедура**) с одним входом и одним выходом, минимальное использование глобальных переменных − сокрытие данных внутри процедур; **структурная организация кода** − отказ от GOTO, использование для управления выполнением исключительно *составных операторов*. СП вводит абстракцию для управления выполнением – **составной оператор**. Вводятся три составных оператора: *последовательный*, *выбора*, *повторения*. Любой алгоритм может быть записан в терминах этих и только этих операторов. В этом случае любой блок кода имеет ровно один вход и один выход. Можно определить *предусловия*/*постусловия*, истинность которых ожидается на входе/выходе, и *инварианты* – условия, которые истинны на протяжении выполнения данного блока. Это дает возможность математически доказывать правильность алгоритмов. Такое доказательство трудоемко и не всегда оправдано, но его облегченный вариант постоянно выполняется опытным программистом в уме, и это позволяет строить действительно надежные программы.

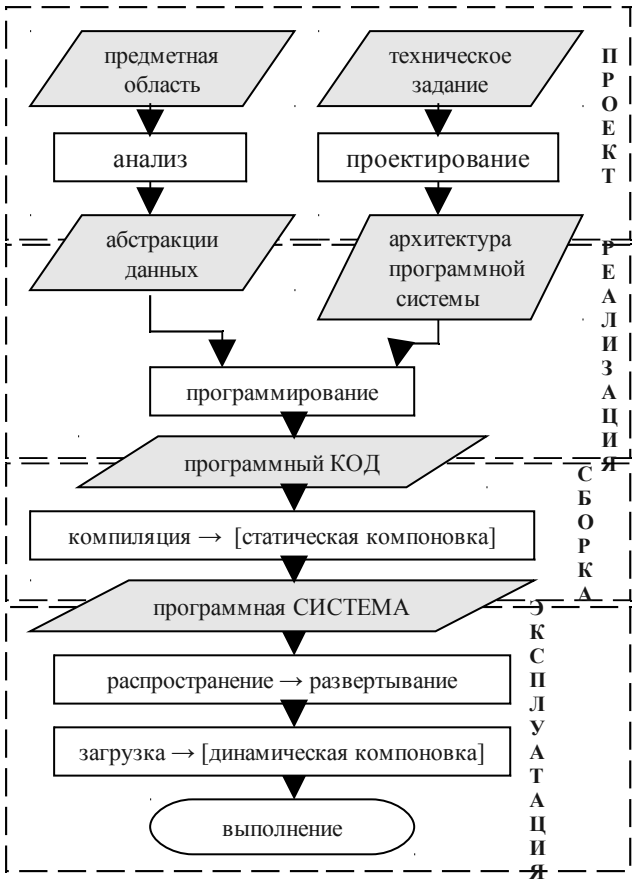
Еще одним китом СП становятся **структуры данных**. Как мы уже отмечали выше, ранее в ЯВУ отсутствовали составные единицы данных, однако к концу 60-х специфика разнообразных прикладных задач потребовала их введения. Структурные языки позволяют конструировать составные **типы данных** из элементарных и таким образом непосредственно выражать в коде

[1] Арифметический IF передавал управление по одной из 3-х возможных веток в зависимости от значения числового параметра: $x < 0$, $x=0$, $x>0$.

[2] Эти идеи были реализованы сначала в Алголе (1968 г.), а затем в завоевавшем всемирное признание Паскале Н.Вирта (1970 г.).Появившийся в то же время «высокоуровневый ассемблер» С в основных аспектах также ориентирован на структурный стиль программирования.

понятия из ПрО. Программная система в СП представляется как набор процедур, обрабатывающих и передающих друг другу **потоки структурированных данных**. Одним из важнейших принципов СП является **сильная типизация** – отсутствие любых неявных преобразований типов, контроль за корректностью использования переменных как на этапе компиляции, так и на этапе выполнения – без этого невозможно гарантировать надежность ПО[1].

III. Модульное ПО. В конструировании любой техники важнейшим является принцип модульности: система разрабатывается и собирается из набора независимых модулей. Модули взаимодействуют друг с другом только через небольшое количество «каналов» (в ПИ набор таких «каналов» называется *интерфейсом* модуля) по строго определенным правилам. Это позволяет нескольким группам конструкторов работать над разными модулями одновременно. Более того, в дальнейшем это облегчает эксплуатацию и обслуживание техники, так как модули разных производителей, даже различающиеся по своему внутреннему исполнению, будут взаимозаменяемы (самый яркий пример – архитектура IBM PC). Другая сторона модульности – это **инкапсуляция**, представление модулей «черными ящиками», обладающими определенным интерфейсом и поведением, но полностью скрывающими свое внутреннее устройство от доступа извне.



Этапы жизненного цикла ПО изобразим следующим образом (см. рис.)[2]. На этапе анализа выделяются сущности ПрО, которые формулируются в типах данных (ТД) ЯВУ (если информация о внутреннем устройстве ТД скрывается внутри модуля, то он называется *абстрактным* ТД). Процессы обработки данных ПрО формулируются в терминах процедур ЯВУ. На этапе проектирования принимается решение об архитектуре, выделяется набор модулей, описываются их интерфейсы и протоколы взаимодействия. На основе полученных **абстракций** и **архитектуры** выполняется непосредственно программирование.

Неоспоримым преимуществом является возможность проектировать и

[1] Н. Вирт: «Исключительного использование сильно типизированного языка является фактором, в наибольшей степени определяющим возможность проектирования сложной системы в короткий срок». Идеально сбалансированный структурный язык – Модула-2 (Вирт), наследница Паскаля (активно используется в российских военно-космических проектах). Богатейший набор типов предоставляет язык Ада (стандарт военной отрасли США). Оба языка сильно типизированы. Линейка С-образных языков придерживается обратного подхода – «все, что не запрещено, то разрешено», что приводит к низкому качеству и большим затратам на отладку – тестирование – сопровождение программных систем.

[2] Отметим, что схема отражает техническую сторону процесса ПИ, но не организационную – организационно выделенные этапы обычно проходят не последовательно, а одновременно и/или циклически.

реализовывать систему в терминах одного и того же языка. Современный ЯВУ должен позволять непосредственно выражать архитектурные решения. Именно этот критерий в настоящее время является главным, т.к. в плане «программирования в малом» все современные императивные ЯВУ приблизительно одинаковы. Эталоном модульного языка стала Модула-2 (1980). В ней модуль является единицей ПО во всех отношениях – единицей написания кода, инкапсуляции, компиляции, импорта другими модулями. В языке Ада модули носят название пакетов. В С-семействе (включая C#) в принципе нет единой абстракции модуля[1]. В совокупности с трудночитаемым синтаксисом это затрудняет проектирование в терминах ЯВУ. Вводятся дополнительные нотации проектирования, например, Буча или UML, однако переход от одного представления к другому отнимает время и вносит ошибки[2]; контроль соответствия кода архитектуре нельзя переложить на компилятор.

В то же время удачность единой абстракции модуля позволила Вирту в языке Оберон-1 (1989) и одноименной ОС сделать прорыв в новое измерение – к **компонентному программированию** (КП). Модуль становится единицей не только разработки, но и распространения, развертывания и загрузки в память, двоичным кирпичиком как ПО, так и ОС в целом. Фаза статической компоновки модулей была исключена, Модули динамически загружаются, связываются, выгружаются, заменяются в любой момент прямо во время работы ПО. Механизм **метапрограммирования** позволяет выполнять динамический анализ модулей и работать с типами, переменными, процедурами, неизвестными на этапе компиляции. Для КП обязательна строгая типизация, динамический контроль типов и границ массивов, сборщик мусора – в противном случае организовать надежное взаимодействие многих модулей от разных производителей просто невозможно[3]. На основе идей Оберона была создана Java (1994). Сегодня мы имеем три промышленных компонентных языка: Java, Компонентный Паскаль (Оберон-2) и C#.

IV. Модульность + ООП: Оберон-парадигма. Суть объектно-ориентированного (ОО) подхода – в рассмотрении ПрО как множества взаимодействующих **объектов**. Объект обладает состоянием (полями) и поведением (методами). **Класс** − множество объектов со сходной структурой и поведением. Чистые ОО-языки (Smalltalk) не предоставляют никаких абстракций кроме классов и объектов («все есть объект»). Однако многие механизмы ПО лучше выражаются в процедурном, нежели в ОО-стиле. Общепринятый подход − надстройка структурных ЯВУ абстракциями класса, объекта и метода. Но (по мнению Вирта) в ООП нет почти ничего нового по сравнению со СП, кроме названий, вносящих путаницу: класс = тип данных, объект = запись, метод = процедура. Модула-2 позволяет писать программы в «почти ОО» стиле. Единственно новое в ООП – это наследование, которое Вирт называет

---

[1] В C++ пространства имен являются всего лишь логической единицей, единицей написания – текстовые файлы, единицей компиляции и импорта – объектные файлы (сборки в .NET), единицей инкапсуляции – классы. Это порождает путаницу и не позволяет разделить систему на полностью непересекающиеся части.

[2] Б. Мейер: «От модели UML до реального программирования дистанция огромного размера». Н. Вирт: «Это даже шаг назад - на заре программирования было принято записывать программы в виде графов. Такая схема приводит к слишком большому количеству ошибок, как только программа усложняется».

[3] Новейший процессор Эльбрус e2k (разработан в России), благодаря использованию аппарата тегов, поддерживает в полной мере защищенное программирование (для модулей единого адресного пространства).

расширением типа. В Обероне-1 ООП реализовано на основе расширяемых записей, полиморфизм процедур – на основе динамической проверки типа (x IS SomeType), виртуальные методы – на основе полей процедурного типа (это позволяет каждому экземпляру типа иметь особое поведение). В Обероне-2 добавлены **связанные процедуры**, которые позволяют оформлять код в привычном стиле ООП (object.Do). Аналогично вводится ООП в Ада-95. При таком подходе идеология и синтаксис языка становятся гораздо проще, стройнее и логичней. К тому же в языках типа C++ из-за отсутствия модулей на класс возлагается двойная нагрузка – он становится и ТД, и единицей инкапсуляции, «модулем» ПО. Однако модуль и ТД – принципиально различные вещи. ТД – это абстракция ПрО, модуль – элемент архитектуры ПО. Для системного ПО типичны группы классов, которые должны взаимодействовать напрямую *в обход инкапсуляции*. Модульность дает элегантное решение проблемы − под некоторую группу тесно связанных ТД выделяется один «кукловод»-модуль, который инкапсулирует их поведение и взаимодействие, при этом внутри модуля нет сокрытия информации между типами[1]. Оберон-парадигма гармонично сочетает ООП, структурный стиль и модульность: ПО строится как набор модулей, связанных процедурными шинами, по которым передаются объектно-ориентированные данные.

V. В условиях очередного **кризиса программирования** очевидно, что основной целью является **надежность** ПО, и достичь ее можно только стремлением к предельной простоте и стройности систем. В противовес американской ПИ, попавшей в тупик избыточной сложности и неуправляемого калейдоскопа бизнес-технологий, в традициях европейской и российской школ программирования – стремление к системному мышлению и поиску простых решений. Огромным потенциалом идей для преодоления кризиса ПИ обладает швейцарская школа Н. Вирта, в частности Оберон-направление.

Литература

1. http://oberon2005.ru − Европейский центр программирования.
2. http://blackbox.metasystems.ru – русский портал по BlackBox и Оберонам.
3. Одинцов И.О. Профессиональное программирование. Системный подход − 2-е изд-е, − СПб.: БХВ-Петербург, 2004.
4. Szyperski C. Component Software. Beyond Object-Oriented Programming, Addison Wesley Longman, 1998.
5. Pfister C., Component Software − Русский перевод: Ермаков И.Е., http://blackbox.metasystems.ru, раздел “Статьи”.

[1] В системном ОО-программировании много тесно связанных групп классов, для которых приходится обходить инкапсуляцию – в этом одна из причин того, для написания ОС рекомендуется чистый C, а не C++. В то же время на Оберонах в ОО-стиле реализовано много ОС, в том числе две – жесткого реального времени.