УДК 502:004.45 (075.8) https://doi.org/10.15407/pp2019.03.045

Н.А. Сидоров

ОСНОВЫ ПРОГРАММИРОВАНИЯ В КОНТЕКСТЕ ИНЖЕНЕРИИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Рассматривается применение конструктивного подхода к построению текстов программ, который систематически культивируется в инженерии программного обеспечения и стало возможным благодаря ряду фундаментальных результатов, полученных в теории программирования. Во-первых, на основе известной структурной теоремы, аргументировано отказались от использования оператора *go to* и предложили метод структурного программирования, что обеспечило реальный путь к созданию понятных программ. Во-вторых, понятие подпрограммы, хотя и использовалось только для уменьшения рутинной работы в процессе программирования, стало первым средством модульного представления программ. Позднее блок и подпрограмма составили основу блок-ориентированных (процедурных, подпрограммных) языков и метода процедурного (подпрограммного) программирования. В-третьих, для ответа на вопросы, относящихся к определению границ, размеров и устройства модуля ввели понятия связывания частей, составляющих модуль и соединения между модулями; создали конкретные критерии модуляризации; предложили устройство модуля на основе понятия сокрытия информации. Модуль был реализован в языке программирования *Modula*, а позже *Modula-2*, в которых использовался на основе метода модульного (композиционного) программирования. При разработке языка *Simula 67*, были заложены основы объектно-ориентированных языков, которые получили развитие благодаря работам по концепциям наследования, позднего связывания и ссылкам и основы были завершены разработкой объектно-ориентированных языков и методом объектно-ориентированного (классификационного) программирования. Таким образом, была создана база для повторного, многократного использования и компонентной разработки программного обеспечения. Сейчас эти работы развиваются в направлении исследования и создания программного обеспечения как системы систем (system of systems), используя связь системного анализа и инженерии программного обеспечения, и развивая системную инженерию программного обеспечения. В статье, для обучения основам программирования, как средство, которое позволяет уточнить понятие программной конструкции, использовано классификацию, а как классификационный признак - уровень инкапсуляции, который строится на основе принципов инженерии программного обеспечения - инкапсуляции и многоуровневого представления, Применяя принцип инкапсуляции на разных уровнях представления структуры программы, соответствующие различным степеням абстракции программного обеспечения, получено понятие уровня инкапсуляции. Воспользовавшись этим понятием, можно выяснить типы программных конструкций и соответствующие методы программирования (конструирования) программ. На основе конструктивного подхода, и введенных понятий к построению программы, автором создана дидактика основ программирования, которая внедрена путем использования в лекциях для студентов специальности «Инженерия программного обеспечения» (121) и при написании автором учебного пособия для студентов и аспирантов, указанной специальности по основам программирования.

Ключевые слова: обучение, дидактика, основы программирования, языки программирования, инженерия программного обеспечения.

Введение

Функционирование своей машины Ч. Бэббидж основал на принципе программного управления, и вследствие этого появилась необходимость писать программы, процесс – программирование и первые программисты. Впоследствии, с появлением электронных вычислительных машин и их широким распространением, этот процесс стал массовым, а, принимая во внимание его сложность и стоимость, стоимость труда писателей и исполнителей программ еще и дорогим. В первое время к созданию программного обеспечения относились как к созданию «железа», но к 60-ым годам пришло понимание, что программное обеспечение принципиально отличается от «железа». Во-первых, программное обеспечение легко модифицировалось и копировалось. Во-вторых, программное обеспечение не изнашивалось, а его сопровождение отличалось от сопровождения «железа». Программное обеспе-

чение было невидимо, не имело веса, но при этом было очень дорогим. Разработка программного обеспечения была практически неуправляемой и требовала огромного количества спецификаций (в 50 раз больше чем для «железа») [1]. В-третьих, с распространением машин в программировании появилась кадровая проблема – нехватка программистов, которую решали путем тренинга гуманитариев, филологов, социологов и специалистов из подобных отраслей. Изменение отношения отразилось и на программировании, которое стали квалифицировать как процесс относящийся к искусству. Для решения проблем была предложена инженерия программного обеспечения (*Software engineering)* – приложение систематического, дисциплинированного, измеримого подхода к разработке, функционированию и сопровождению программного обеспечения, а также исследованию этого подхода; приложение дисциплины инженерии к программному обеспечению (*ISO/IEC/IEEE 24765-2010*) [2]. Поскольку основным процессом в инженерии программного обеспечения попрежнему оставалось программирование, то первые результаты были получены в этой области. Во-первых, на основе известной структурной теоремы, аргументировано отказались от использования оператора *go to* и предложили метод структурного программирования, что обеспечило реальный путь к созданию понятных программ. Во-вторых, понятие подпрограммы, хотя и использовалось только для уменьшения рутинной работы в процессе программирования, стало первым средством модульного представления программ. Позднее блок и подпрограмма составили основу блок-ориентированных (процедурных, подпрограммных) языков и метода процедурного (подпрограммного) программирования. В-третьих, для ответа на вопросы, относящиеся к определению границ, размеров и устройства модуля ввели понятия связывания (c*ohesion)* частей, составляющих модуль и соединения (*coupling)* между модулями [3]. Затем ввели конкретные критерии модуляризации и предложили устройство модуля на основе понятия сокрытия информации. В 1980 году реализовали язык программирования *Modula*, а позже *Modula-2*, в которых использовалось понятие модуля и модульного (композиционного) программирования [4]. В 1984 году был создан язык программирования *Ada*, в котором такое же понятие реализовано в форме пакета [5]. В 1967 году, используя блок и сокрытие информации при разработке языка *Simula 67*, были заложены основы объектно-ориентированных языков [6]. Эти основы получили развитие благодаря работам по концепциям наследования, позднего связывания и ссылкам [7]. Завершены эти работы были разработкой объектно-ориентированных языков и методом объектно-ориентированного (классификационного) программирования [8]. Таким образом, в основном были завершены работы в области модуляризации, как для композиционных, так и классификационных языков и создана основа для повторного, многократного использования и компонентной разработки программного обеспечения. Сейчас эти работы развиваются в направлении исследования и создания программного обеспечения как системы систем (system of systems), используя связь системного анализа и инженерии программного обеспечения, и развивая системную инженерию программного обеспечения (system software engineering) [9].

Конструктивный подход к устройству программы в контексте инженерии программного обеспечения

В контексте инженерии программного обеспечения, рассматривая программу метафорически с точки зрения теории машин, будем иметь в виду три точки зрения на машину, а именно технологическую, кинематическую и конструктивную [10].

В статье будет применяется последняя точка зрения, конструктивная. Конструктивный подход к рассмотрению программы, систематически культивируется в инженерии программного обеспечения и стал возможным благодаря ряду фундаментальных результатов, полученных в теории программирования, о которых

кратко говорится во введении статьи. Конструктивная теория машин рассматривает их с точки зрения форм и частей целого, статически, разделяя машину на отдельные части и подчеркивая ее конструкцию. В XVIII веке утверждалось, что «многообразные механизмы движения, которыми пользуются для устройства рабочих машин, не должны заново изобретаться каждый раз. Это было необходимо, когда были изобретены паровые и прядильные машины, так как тогда были известны лишь немногие механизмы для преобразования движений. Теперь же известно очень много разнообразных механизмов и всегда можно отыскать такой, который подходит для частного случая. Таким образом, лишь для совершенно необычных условий движения действительно необходимы новые изобретения, и очень ясное и полное знание изобретенных до настоящего времени передаточных механизмов, служащих для устройства рабочих машин, является необычайно важным» [11]. В работе [12], B. Boehm делает аналогичное предположение относительно программного обеспечения, аргументируя повторное использование (reuse и systematic reuse) программного обеспечения как наиболее перспективный подход для повышения продуктивности создания и сопровождения программного обеспечения на ближайшие десятилетия. С годами, это предположение подтвердилось. Очевидно, что продуктивность этого подхода зависит от реализации конструктивной точки зрения на программы. При этом, такой взгляд имеет важное значение не только при проектировании, но и при программировании. К сожалению, компонентный (конструктивный) подход дается студентам поздно, только в дисциплинах проектирования программного обеспечения и совсем не применяется в раннем обучении основам программирования. Студенты, в контексте обучения по специальности инженерия программного обеспечения должны не только учиться писать работающие программы, но программы, которые отвечают ряду дополнительных требований, связанных с реализацией компонентного подхода. Не владея конструктивным взглядом на программы невозможно удовлетворять эти требования.

При обучении основам программирования, как средство, которое позволяет уточнить понятие программной конструкции, предложено использовать классификацию, а как классификационный признак – уровень инкапсуляции, который строится на основе принципов инженерии программного обеспечения – инкапсуляции и многоуровневого представления [13].

Инкапсуляция в целом – это процесс создания оболочки вокруг тех или иных веществ. Оболочка называется капсулой, а вещества в капсуле инкапсулированными и характеризуются высокими внутренними (cohesion) и низкими внешними (coupling) связями. Инкапсуляция в программировании – это процесс, который агрегирует «вещество» программ, кодированного в том или ином языке программирования, образуя капсулы. Они могут использоваться в программировании как единственные в определенном смысле законченные части программ. Этот процесс, как увидим дальше, регламентирован.

В программе роль инкапсулированного «вещества» играют ее компоненты, например, операторы, подпрограммы. Образование капсулы (конструкции) вокруг компонентов программы позволяет достичь следующих, присущих конструктивному взгляду, целей [13]:

- манипулировать при написании, отладке и понимании программ капсулами, рассматривая их как законченные части;
- указать метод программирования, который регламентирует использование капсул;
- указать значения, обрабатываемые капсулой;
- ограничивать допуск к компонентам, размещенным в капсуле;
- скрывать детали реализации компонентов, размещенных в капсуле;
- использовать капсулы для построения других капсул.

Капсула строится агрегированием необходимых частей программы и созданием оболочки и интерфейса, что обеспечивает правильное применение капсулы. При создании или повторном использова-

нии капсул перед программистом возникают три вопроса. Во-первых, в каком случае (для реализации каких проектных решений – действий) может применяться та или иная капсула (какова концепция капсулы). Во-вторых, какой должна быть капсула (какое устройство, конструкция капсулы). В-третьих, как связывается капсула с окружением (контекст капсулы). Программист, создавая или повторно, многократно используя капсулы, применяет их в соответствии с методом программирования и реализует проектные решения, полученные в предыдущих фазах жизненного цикла программы, что составляет часть процессов конструирования программ. Капсулу, которая применяется в соответствии с методом программирования можно называть программной конструкцией. Чтобы охарактеризовать известные ныне программные конструкции, а также выяснить, какие значения они обрабатывают и в чем заключаются методы конструирования (программирования) программ из них, воспользуемся еще одним принципом инженерии программного обеспечения – многоуровневым представлением.

Применяя принцип инкапсуляции на разных уровнях представления структуры программы, соответствующие различным степеням абстракции программного обеспечения, получено понятие уровня инкапсуляции [13]. Воспользовавшись этим понятием, можно выяснить типы программных конструкций и соответствующие методы программирования (конструирования) программ. Сейчас можно выделить шесть уровней инкапсуляции и столько же типов программных конструкций (таблица).

Таблица. Уровни инкапсуляции

Уровень инкапсуляции	**Инкапсулируемые части программы**	**Создаваемая конструкция (капсула)**	**Метод программирования (применения)**	**Средства и механизмы**	**Метафора**
Лексический	Символы алфавита языка программирования	Лексема	Правила создания лексем и выражений	Оператор присваивания, операции	Звено
Операторный	Лексемы и выражения из них	Структурный оператор	Структурное программирование	Структурный операторный базис языка программирования	Цепь
Подпрограммный	Описание данных, структурные операторы	Подпрограмма (макрос, процедура, функция)	Процедурное (подпрограммное) программирование	Независимая компиляция	Механізм
Модульный	Описание данных, подпрограммы	Модуль (пакет)	Модульное (композиционное) программирование	Механизмы сокрытия и видимости, раздельная компиляция	Механізм
Классный	Описание данных, подпрограммы	Класс	Объектно-ориентированное (классификационное) программирование	Наследование, полиморфизм, связывание	Механізм
Мегамодульный (система систем)	Онтология, модули, классы	Мегамодуль (система)	Мегапрограммирование	Операционная независимость, независимое управление и поведение, географическая распределенность, эволюционное и адаптивное развитие	Машина

Степень абстрагирования и понимания программы повышается от лексического уровня к мегамодульному. Каждому уровню инкапсуляции соответствует свой тип капсул (программная конструкция), правила образования и дисциплина их использования в конструировании программ (метод программирования).

Программные конструкции. В аспекте конструктивной точки зрения на программу, программная конструкция – это фундаментальное понятие языка программирования. С точки зрения инженерии программного обеспечения каждая часть программы – это программная конструкция, если в ней инкапсулированы другие части программы (нижнего уровня инкапсуляции), она характеризуется конструктивными (системными) свойствами и имеет собственный способ применения [13].

Простейшая программная конструкция, это лексема. Являясь обозначением, она играет важную роль в построении таких конструкций как литерал, константа, переменная. Более сложными программными конструкциями являются выражение, оператор, подпрограмма, модуль, класс, мегамодуль (система систем).

Лексемы. Символы алфавита не играют в языке самостоятельной роли, однако они используются для построения лексем. С одной стороны, лексемы – это простые программные конструкции, которые составляют словарный запас языка. А с другой стороны, лексемы – это капсулы, которые инкапсулируют символы алфавита. «Оболочки» капсул образуют специальные символы, или символы других лексем. Капсулу-лексему можно подать размеченной цепочкой: $S_i*l_1\ l_2...l_n\ *S_j...$, где S_i, S_j, $l_k \in V$; V – алфавит языка; S_i, S_j – пробельные символы или символы других лексем; l_k, ($k = 1, ..., n$) – символы лексемы, при этом l_1 – первый символ, а l_n – последний символ лексемы (если S_i, S_j символы других лексем). В лексеме-капсуле можно установить порядок (последовательность) символов, который дает понятие границ капсулы (первый и последний символы капсулы) и используется лексическим анализатором – программой, входящей в состав транслятора. Этот порядок позволяет говорить о входе в капсулу и выходе из нее. Все лексемы в тексте программы являются обозначениями. Это означает, что лексемы обозначают другие конструкции программы, которые строятся и используются в процессах трансляции и выполнения программы. На лексическом уровне инкапсуляции текст программы конструктивно состоит из последовательности лексем. Когда программа транслируется или выполняется, тогда каждой лексеме ставится в соответствие некоторое значение. Соответствие между лексемой и значением может устанавливать программист или транслятор, или это соответствие может заранее определяться стандартным окружением языка программирования. На рис. 1 показана роль лексемы в конструкции переменной.

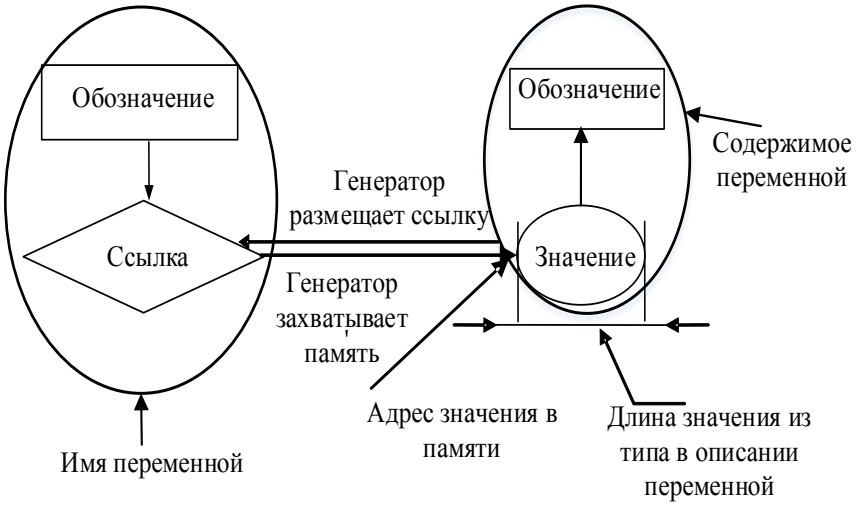

Рис. 1. Конструкция простой переменной

Установление соответствия между лексемой и значением или между обозначением и значением, или еще шире - между двумя объектами программы обеспечивается механизмом связывания (binding).

Выражение описывает правило обработки значений, содержащихся в программных объектах, входящих в его состав. В результате выполнения правила образуется значение, которое является значением выражения. Таким образом, текст выражения – это обозначение, и после выполнения выражения – это обозначение связывается с значением. Поэтому, выражение является программной конструкцией наряду с литералом, константой и переменной.

Операторы. Структурные операторы составляют структурный операторный базис языков программирования, которого достаточно для записи любой программы. Именно структурные операторы и структурное программирование – метод их применения – в свое время стали фундаментальными концепциями в языках программирования. Лексемы выступают как части, которые образуют структурную капсулу, а оболочка капсулы и интерфейс реализуются через организационные ограничения. Во-первых, что касается оболочки, запрещается произвольный доступ к конструкциям внутри капсулы, и произвольный выход из капсулы (рис. 2). Это запрет применения операторов go to, break, exit или continuous. Во-вторых, что касается интерфейса, указанные ограничения технически можно преодолеть, но если их учитывать, то в любой структурной капсуле предполагается только один вход и один выход (рис. 2), с помощью которых она соединяется с другими капсулами, образуя пары (звенья цепи) и цепи (таблица).

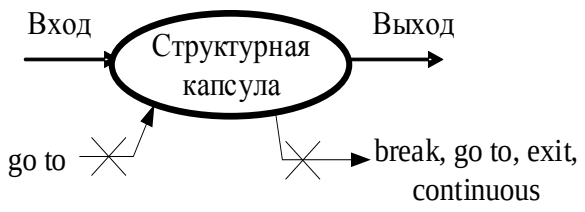

Рис. 2. Структурная капсула

Применение структурного программирования ведет к созданию программных текстов, которые легко читаются и модифицируются. В рамках структурного программирования рассматриваются преобразования программ, которые могут выполняться вручную или автоматически.

Рассмотрим простой пример. В технологическом аспекте необходимо создать подпрограмму («механизм»), которая осуществляет обмен значений целого типа, swap (int a, int b). Имеем следующую конструкцию (рис. 3, 4):

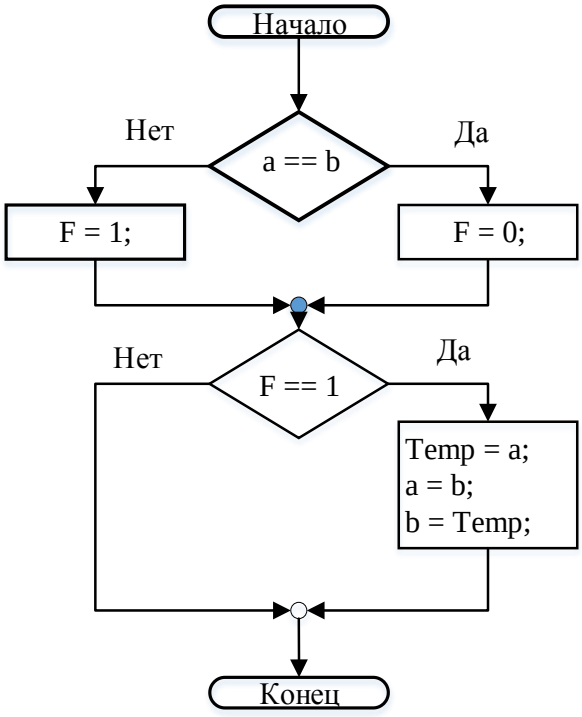

Рис. 3. Конструкция программы

```
void swap(int &a, int &b)
{
 int F;
 int Temp;
 if a == b
    F = 0;
 else
    F = 1;
 if F == 1
    {
     Temp = a;
     a = b;
     b = Temp;
    }
}
```

Рис. 4. Текст программы

– две «цепочки» типа условный оператор в форме альтернативы и выбора;

– три «звена» типа преобразовательного оператора (оператор присваивания);

– один механизм, полученный путем последовательного соединения «цепочек».

Таким образом, процесс конструирования подпрограммы, это использование конструкций-«звеньев», сочетание «звеньев» (построение «цепочек») и построение «механизма» путем последовательного соединения «цепочек». «Механизм» – это «замкнутая» последовательность «цепочек» в смысле выполняемой функции, которая, как известно у модуля должна быть одна.

Очевидно, что полученная конструкция проста для понимания и модификации, что является требованиями компонентного подхода.

Подпрограммы. Программная конструкция на подпрограммном уровне инкапсуляции состоит из операторов и называется подпрограммой. Из подпрограмм формируются библиотеки подпрограмм, которые когда-то стали основой систем программирования. Сначала подпрограммы рассматривались как повторяющиеся части программ, которые предварительно описываются, хранятся отдельно от программы или непосредственно в ней и многократно используются путем встраивания вызовов подпрограмм в программу. Таким образом, подпрограммы служили средством сокращения текстов программ. Сейчас благодаря процедурной абстракции и абстракции управления подпрограмма играет очень важную роль в модульной организации программ и их проектировании и является фундаментальной концепцией любого языка программирования, средством абстрагирования и проектирования. Подпрограмма — это повторяющаяся часть программы, которая важна с двух точек зрения: во-первых, как средство сокращения затрат на запись программы, во-вторых, как средство проектирования программ. На подпрограммном уровне инкапсуляции содержимое капсулы, составленное из структурных операторов методом структурного программирования, не интересует программиста, поэтому основной интерес представляет оболочка капсулы и, в частности ее интерфейс. Оболочка капсулы открытой подпрограммы не существует после ее вызова, поэтому она «непрочная», а содержимое капсулы, видно программисту. Оболочка закрытой подпрограммной капсулы существует и после вызова, хотя ее содержимое также видно программисту, но она более «прочная», чем оболочка капсулы структурного оператора. Степень «прочности» определяется оператором go to. Если для капсулы структурного оператора переход в нее и выход из нее оператором go to запрещены организационно, то в подпрограммной капсуле (закрытая подпрограмма) это невозможно уже конструктивно. Поэтому часть оболочки подпрограммной капсулы должна играть роль интерфейса (рис. 5).

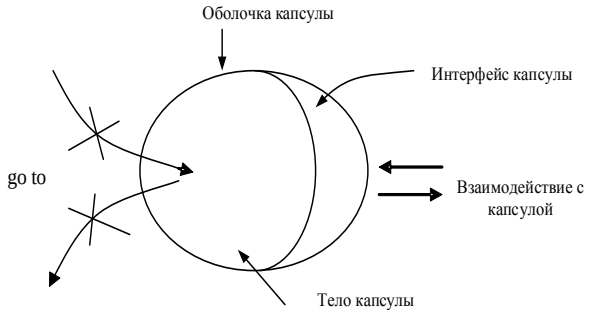

Рис. 5. Капсула подпрограммы

Интерфейс обеспечивает связь внешнего окружения капсулы с компонентами, размещенными в середине капсулы. По определению инкапсуляции подпрограммного уровня следует, что подпрограммная капсула – это последовательность структурных операторов, «помещенных» в оболочку. Порядок размещения операторов в капсуле соответствует алгоритму решаемой капсулой задачи. Отражая сущность процедурной абстракции, понятие подпрограммной капсулы позволяет программисту-читателю абстрагироваться от того, как решает задачу капсула, и сосредоточиться на том, какую задачу она решает.

Подпрограммную капсулу можно рассматривать как «черный ящик», на вход которого передается управление, а на выходе управление возвращается. В результате передачи управления и выполнения операторов капсулы можно получить полезный эффект, который найдет отражение в векторе состояния программы, в точке после вызова подпрограммы. Такой взгляд на подпрограмму составляет сущность абстракции управления. Таким образом, закрытая подпрограмма – это не столько средство сокращения записи текста программы (как считалось ранее), сколько средство разложения (декомпозиции) программы на логически связанные, законченные и, в определенном смысле, закрытые компоненты. Перед началом разработки программы программист имеет более или менее точную формулировку задачи в терминах некоторого домена и инструмент – язык программирования. Говорят, что между сформулированной задачей и языком программирования существует концептуальноя расстояние, которое преодолевается с помощью программирования [14]. Абстрактные типы данных – это фундаментальная концепция программирования, которая направляет как замыслы разработчиков языков программирования, так и повседневные действия программистов. Для первых, абстрактный тип данных – это та концепция, которую должен эффективно реализовать язык программирования, а для вторых – это, тот продукт, который должен создавать программист в любом домене. Подпрограммный уровень инкапсуляции позволяет реализовывать абстрактные типы данных, но очень неэффективно. Именно это является причиной того, что требуется применение возможностей классного или модульного уровней инкапсуляции. Именно на эффективную реализацию абстрактных типов данных и направлены возможности этих уровней инкапсуляции. Класс и модуль – компоненты, которые обычно представляют абстрактный тип данных.

Модуль. На сегодня известно две схемы реализации пост подпрограммного уровня инкапсуляции – композиционная и классификационная. Обе существуют одновременно. Первая используется на модульном уровне инкапсуляции, а вторая - на классном. Обе схемы строятся на основе понятия отношения между компонентами схемы. Различия проявляются в разной интерпретации понятия отношение в схемах как при организации сред (библиотек, коллекций) из программных компонентов (модулей или классов), так и при их использовании. При разработке программ, которые состоят из большого количества частей, основное требование разложения программы на составные части приобретает первостепенное значение. Разложив большую задачу на части, можно получить задачи-компоненты с таким количеством деталей, когда ее можно реализовать, а каждый компонент можно реализовывать отдельно и независимо. Такое разложение, которое реализуется на основе независимой компиляции, абсолютно необходимо, если в разработке программы участвует более одного программиста. Образующиеся в результате разложения задачи реализуются частями программы, которые называются в процедурном программировании – подпрограммы, в композиционном программировании – модули или пакеты. Модуль – это капсула, которая инкапсулирует данные и подпрограммы. В ограниченном виде модуль ввел Н. Вирт в языке Pascal, а наибольшего развития он приобрел в языках Modula-2, Modula-3 и Ada. В языке Ada модуль называется пакетом.

Другое требование – сокрытие деталей реализации модулей приводит к тому, что модуль обычно подается двумя конструкциями. В первой – интерфейсе – описываются ресурсы (сервисы), которые представляет модуль (в зависимости от типа модуля). Во второй – описываются реализации ресурсов, поданных интерфейсом модуля. Таким образом, любая капсула-модуль состоит из интерфейса и инкапсулированных конструкций (рис. 6). Механизм экспорта и импорта действует через перечень экспортируемых ресурсов (типы, объекты и подпрограммы), описанных в модуле определения и перечень импортируемых ресурсов, объявленных в модуле реализации. Обычно, при обучении основам программирования различают модули,

которые поставляют с языком программирования и реализуют, например, ввод/вывод.

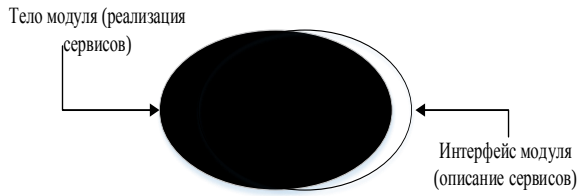

Рис. 6. Капсула модуля

Модульное программирование – это метод создания программ из программных конструкций модулей, который применяется на модульном уровне инкапсуляции. Он основывается на механизме раздельной компиляции, который наряду с указанным устройством модуля обеспечивает полное сокрытие информации о реализации модуля. Суть механизма заключается в том, что обе части модуля (интерфейс и реализация) компилируются раздельно. Благодаря этому в случае изменения реализации ресурсов (с сохранением ее интерфейсов) в перекомпиляции нуждаются только модули реализации и не требует программа, в которой применены соответствующие модули определения. Модульное программирование называется еще композиционным программированием. Альтернативой ему является классификационное программирования, реализацией которого является объектно-ориентированное программирование. Выделение программного модуля влияет на программирование в трех аспектах:

- понятность – программа понятна, если она состоит из модулей;
- эффективность – программа работает более эффективно, поскольку модуль позволяет рационально организовать структуру текста;
- тестируемость – программа легче тестируется, если она модульная.

Классы. Концепция объектно-ориентированного программирования исторически тесно связана с концепцией универсального языка программирования, которая интенсивно разрабатывалась в 70х. Основными принципами объектно-ориентированного программирования являются наследование и полиморфизм. Для реализации полиморфизма в объектно-ориентированном программировании кроме раннего связывания – механизма, который обеспечивает установление связи между интерфейсом программной конструкции и одной из реализаций (форм) конструкции, предусмотренных для нее, во время компиляции, вводится позднее связывание – установление указанной связи во время выполнения программы. Кроме основных могут применяться дополнительные принципы: параметризация, многократное и повторное использование. Многократное использование (systematic reuse) – принцип, обеспечивающий создание, так называемых типовых программных конструкций (компонентов), которые можно использовать многократно. Этим принципом в модульном и объектно-ориентированном программировании руководствуются при проектировании модулей и классов. Повторное использование (reuse) – принцип, который обеспечивает использование наследуемого программного обеспечения в новых разработках. Любой класс как программная конструкция в целом, также как модуль состоит из интерфейса и тела (рис. 7). Интерфейс описывает ресурсы, которые предоставляются классом. Тело состоит из конструкций, описывающих реализацию ресурсов, которые предоставляет класс. В зависимости от конкретного языка программирования строение класса может отличаться.

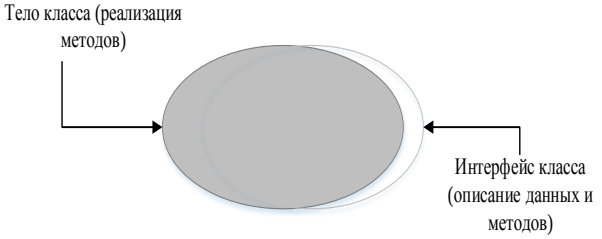

Рис. 7. Капсула класса

Поскольку класс, как правило – это абстрактный тип данных, то тело класса и также, как и модуля должно содержать описание следующих компонентов:

- значений, предоставляемых абстрактным типом данных (классом)
- подпрограмм, обеспечивающих обработку значений абстрактного типа данных (класса).

Важно, что класс, в отличии от модуля не реализует важный принцип модуляризации – сокрытие информации. Он реализуется либо схемой, имитирующей раздельную компиляцию (подобно модульному сокрытию), либо путем введения дополнительных понятий в язык программирования.

Мегамодули. Системы систем. Понятие мегамодуля введено в работе [15], и в отличие от модуля инкапсулирует не только данные и операции, но и поведение и доменные знания, отражающие, в частности, языки, культуры и традиции. При этом, несколько мегамодулей могут входить в мегапрограмму, которая пишется на языке мегапрограммирования. Позднее, было введено понятие системы систем (system of systems), которое по сути является мегапрограммой [16]. Так как программирование системы систем значительно сложнее основ программирования, этот класс модулей и составление из них систем должен изучаться в отдельной дисциплине, например, при подготовке магистров.

Дидактика основ программирования

В 2006 г. постановлением Кабинета Министров Украины было открыто новое направление обучения 6.050103 «Программная инженерия». Сейчас, это 121 специальность «Инженерия программного обеспечения». Был разработан соответствующий стандарт для подготовки бакалавров [17]. Новое направление обучения, очевидно требует собственного учебно-методического обеспечения, поскольку, например, от направления «Компьютерные науки» его отличает четкая инженерная направленность. Дисциплины учебного плана этого направления должны как можно больше учитывать указанную направленность. К сожалению, в основном из-за нехватки грамотных, в смысле инженерии программного обеспечения, педагогических кадров, в университетах сложилась ситуация, которая характеризуется тем, что не только не выдерживается соответствующий стандарт «в большом», но и «в малом», особенно в дисциплинах, которые перешли, например, из Компьютерных наук. Это относится и к дисциплине «Основы программирования», которая преподается как правило на первом курсе университета. Поэтому не удивительно то, что отрасль по-прежнему недовольна уровнем подготовки студентов, и как следствие несет огромные затраты по «доводке» студентов до требуемого уровня [18]. Принимая во внимание, то, что в отрасли катастрофически не хватает грамотных программистов, дидактика дисциплины «Основы программирования» приобретает особое значение.

На основе конструктивного подхода к программе, изложенного в статье построена дидактика основ программирования, которая реализована в учебном пособии автора [19].

Известно два подхода к преподаванию основ программирования [20]:

- императивный (imperative-first) – традиционный подход, в котором изложение основ осуществляется, начиная с императивных аспектов языка программирования: выражения, управляющие структуры, процедуры и функции и другие основные элементы процедурных языков программирования. Объектно-ориентированное программирование рассматривается позже в другой дисциплине;
- объектно-ориентированный (objects-first) – с самого начала обучения главное внимание уделяется объектно-ориентированному программированию, объясняя понятие класса, объекта и наследования, затем предлагают студентам написать объектно-ориентированные программы. Позже вводятся императивные аспекты объектно-ориентированного языка программирования.

Традиционно, в Украине используется только первый подход при обучении основам программирования. Это ведет к тому, что, когда начинается обучение объектно-ориентированному программи-

рованию студенты с трудом усваивают такие концепции как класс, объект, наследование, полиморфизм, виртуальные функции, и еще труднее даётся понимание специальных конструкций языков (интерфейсы, делегаты), ориентированных на реализацию модульности.

Автор делал попытку, применения второго подхода при обучении. Опыт показал, что такая подача конструкций программ также затрудняет обучение. Небольшой начальный опыт в объектно-ориентированном программировании с последующим переключением на процедурное программирование только запутывает студента. Вероятно, проблем бы не было или было меньше, если изучать «чистый» объектно-ориентированный язык подобный *SmallTalk,* но это не привлекает студентов в Украине, в перспективе трудоустройства.

Многолетний опыт автора в использовании изложенного в статье конструктивного подхода показал, что он понятен студентам и позволяет на общей основе осваивать как сложные конструкции императивной части – переменная, ссылка, указатель, подпрограммы, и механизмы – разыменование, приведение типов, так и конструкции модульной части – модули, классы, объекты, и механизмы – сокрытие, полиморфизм, наследование. Конструктивный подход знакомит студентов не только с традиционно распространённым в Украине объектно-ориентированном программировании, но с менее известным у нас – модульным программированием. Кроме этого, студенты с первых занятий осознанно настраиваются на применение повторного и многократного использования, понимая цели и задачи компонентной инженерии программного обеспечения.

Дидактика основ программирования, основанная на конструктивном подходе, базируется на понятии конструкции и структурирует изложение учебного материала в соответствии с уровнями инкапсуляции (табл.). Для объяснения устройства конструкций широко используются графические схемы подобные тем, что были введены в работах [21, 22] (например, рис. 1).

Выводы

Известно два подхода к преподаванию основ программирования:

- императивный – традиционный подход, в котором изложение основ осуществляется, начиная с императивных аспектов языка программирования;
- объектно-ориентированный – с самого начала обучения главное внимание уделяется объектно-ориентированному программированию.

В контексте инженерии программного обеспечения для обучения студентов основам программирования целесообразно использовать конструктивный подход, который подготовит студентов к созданию и сопровождению программного обеспечения методами многократного и повторного использования в рамках парадигмы компонентной разработки.

Литература

1. Boehm B., 2006, A View of 20th and 21st Century Software engineering [Text]. ICSE'06. May 20–28. China. 2006. P. 12–29.
2. Report on a conference sponsored by the NATO science committee, Garmisch, Germany, 7th to 11th October 1968, Editors: Peter Naur and Brian Randell.
3. Segmentation and Design Strategies for Modular Programming." In T. O. Barnett and L. L. Constantine (eds.), Modular Programming: Proceedings of a National Symposium. Cambridge, Mass.: Information & Systems Press, 1968.
4. Wirth N. Programming in Modula-2. Springer-Verlag, Heidelberg, New York, 1982.
5. Jean Ichbiah (October 1984). «Ada: Past, Present, Future – An Interview with Jean Ichbiah, the Principal Designer of Ada». *Communications of the ACM*. **27** (10). P. 990–997. doi:10.1145/358274.358278.
6. Dahl O.-J., Myhrhaug B., Nygaard K. SIMULA67, Common base language/-Oslo. 1968. 96 p.
7. Hoare C.A.R. An axiomatic basis for computer programming, Comm. Of ACM, 12(1969). P. 576–580.

8. Goldberg A., Robson D. SmallTalk 80 The language and its implementation, Addison-Wesley, New-York, 1983.
9. Maier M.W. Architecting principels for systems-of-systems, Systems engineering, 1, 4(1998). P. 267–284.
10. Энгельмейер П.К. Философия техники, М. 1912.
11. Redtenbacher F., Der Maschinbau, Mannheim, 1862.
12. Boehm B.W. Improving Software Productivity. Computer. 1987. Vol. 20, N 9. P. 43–57.
13. Сидоров Н.А. Применение принципов программной инженерии в преподавании основ программирования. *Управляющие системы и машины*. 1998. № 4. С. 50–59.
14. Дал У., Дейкстра Э., Хоор К. Структурное программирование = Structured Programming. 1-е изд. М.: Мир, 1975. 247 с.
15. Widerhold G., Wegner P., Ceri S., Toward megaprogramming. *Communication of the ACM*. 1992. Vol. 35, N 11. P. 89–99.
16. System of systems engineering, ed.by Jamshidi M., John Wiley&Sons, 2009, 591 p.
17. Сидоров Н.А. Инженерия программного обеспечения – учебная дисциплина или подготовка бакалавра? *Управляющие системы и машины*. 2006. № 2. С. 25–34.
18. It Ukraine from a to z, http://www.uadn.net/files/ua_hightech.pdf
19. Сидоров М. Основи програмування. Київ, 2018, 435 с.
20. Bennedsen J., Teaching and Learning Introductory Programming, – A Model-Based Approach. 327 p.
21. Линдси Ч., ван дер Мюйлен С., Нефор-мальное введение в Алгол 68, М., Мир, 1973.
22. Баррон Д., Введение в языки программирования, М., Мир, 1980. 189 с.

References

1. Boehm B., 2006, A View of 20th and 21st Century Software engineering[Text]. ICSE'06.-May 20–28 China. 2006. P. 12–29.
2. Report on a conference sponsored by the NATO science committee, Garmisch, Germany, 7-th to 11-th October 1968, Editors: Peter Naur and Brian Randell.
3. Segmentation and Design Strategies for Modular Programming." In T. O. Barnett and L. L. Constantine (eds.), Modular Programming: Proceedings of a National Symposium. Cambridge, Mass.: Information & Systems Press, 1968.
4. Wirth N. Programming in Modula-2. Springer-Verlag, Heidelberg, New York, 1982.
5. Jean Ichbiah (October 1984). «Ada: Past, Present, Future — An Interview with Jean Ichbiah, the Principal Designer of Ada». *Communications of the ACM* **27** (10): 990–997. DOI:10.1145/358274.358278
6. Dahl O.-J., Myhrhaug B., Nygaard K. SIMULA67, Common base language/-Oslo, 1968, 96 p.
7. Hoare C.A.R. An axiomatic basis for computer programming, Comm. Of ACM, 12 (1969). P. 576–580.
8. Goldberg A., Robson D. SmallTalk 80 The language and its implementation, Addison-Wesley, New-York, 1983.
9. M.W.Maier Architecting principels for systems-of-systems, Systems engineering, 1, 4(1998). P. 267–284.
10. Engelmeyer. P. C. Philosophy of technology. M., 1912.
11. Redtenbacher F., Der Maschinbau, Mannheim, 1862.
12. Boehm B.W. Improving Software Productivity. Computer. 1987. Vol. 20, N 9. P. 43–57.
13. Sydorov M. Using the software engineering principals in basics programming education, USIM. 1998. N 4. P. 50–59.
14. Dijkstra E.W. Structured Programming. 1975. 247 p.
15. Widerhold G., Wegner P., Ceri S. Toward megaprogramming. *Communication of the ACM*. 1992. Vol. 35, N 11. P. 89–99.
16. System of systems engineering, ed.by Jamshidi M., John Wiley&Sons, 2009, 591 p.
17. Sydorov M. Is the software engineering education subject or postgraduate, USIM. 2006. N 2. P. 25–34.
18. It Ukraine from a to z, http://www.uadn.net/files/ua_hightech.pdf
19. Sydorov M., Basics of programming, Kiev 2018, 435 p.
20. Bennedsen J. Teaching and Learning Introductory Programming, – A Model-Based Approach. 327 p.

21. Lindsey C., van der Meulen S., Informal introduction to ALGOL 68, London, 1971.
22. Barron D.W. An introduction to the study of programming languages, Cambridge university press, London, 1977.

Получено 18.07.2019

Об авторе:

Сидоров Николай Александрович,
доктор технических наук,
профессор.
Количество научных публикаций в украинских изданиях – 118.
Количество научных публикаций в зарубежных изданиях – 12.
http://orcid.org/0000-0002-3794-780X

Место работы автора:

Национальный Технический Университет Украины «Киевский политехнический институт имени Игоря Сикорского», факультет информатики и вычислительной техники, кафедра автоматизированных систем обработки информации и управления, АСОИУ, профессор.
02000, Киев,
ул. Политехническая, 41.
Моб. тел.: 067 7980361.
Тел.: 044 2343600.
E-mail: nikolay.sidorov@livenau.net,
sna@nau.edu.ua