# Некоторые аспекты дизайна современного языка программирования общего назначения

Евгений Зуев

Университет Иннополис  
Казань

eugene.zueff@gmail.com

Алексей Канатов

Исследовательский центр Samsung  
Москва

alexey.v.kanatov@gmail.com

**АННОТАЦИЯ**

В данной статье рассматриваются тенденции развития современных языков программирования и предлагаются к обсуждению несколько концепций, которые должны повысить простоту и удобство проектирования и разработки программного обеспечения, в рамках статической проверки правильности программ. В частности дается нетрадиционный подход к базовому строительному блоку программного кода – контейнеру атрибутов и подпрограмм, альтернативная схема наследования, как простого и единственного механизма расширения уже существующего ПО, концепция мультитипа, как расширение наследования и полное решение для проблем нулевых указателей и неинициализированных данных.

**Abstract**

This paper gives an overview of modern trends in programming languages evolution and introduces some concepts targeting to increase the level of suitability and simplicity of design and development of the software with static typing are presented for consideration. In particular, non-traditional approach to the basic building block of the software – attributes and routines container is presented. Alternative scheme of multiple inheritance as simple and unique method of software reuse extended with multi-types is given. Full solution for dealing with null-pointers and non-initialized variables is presented as well.

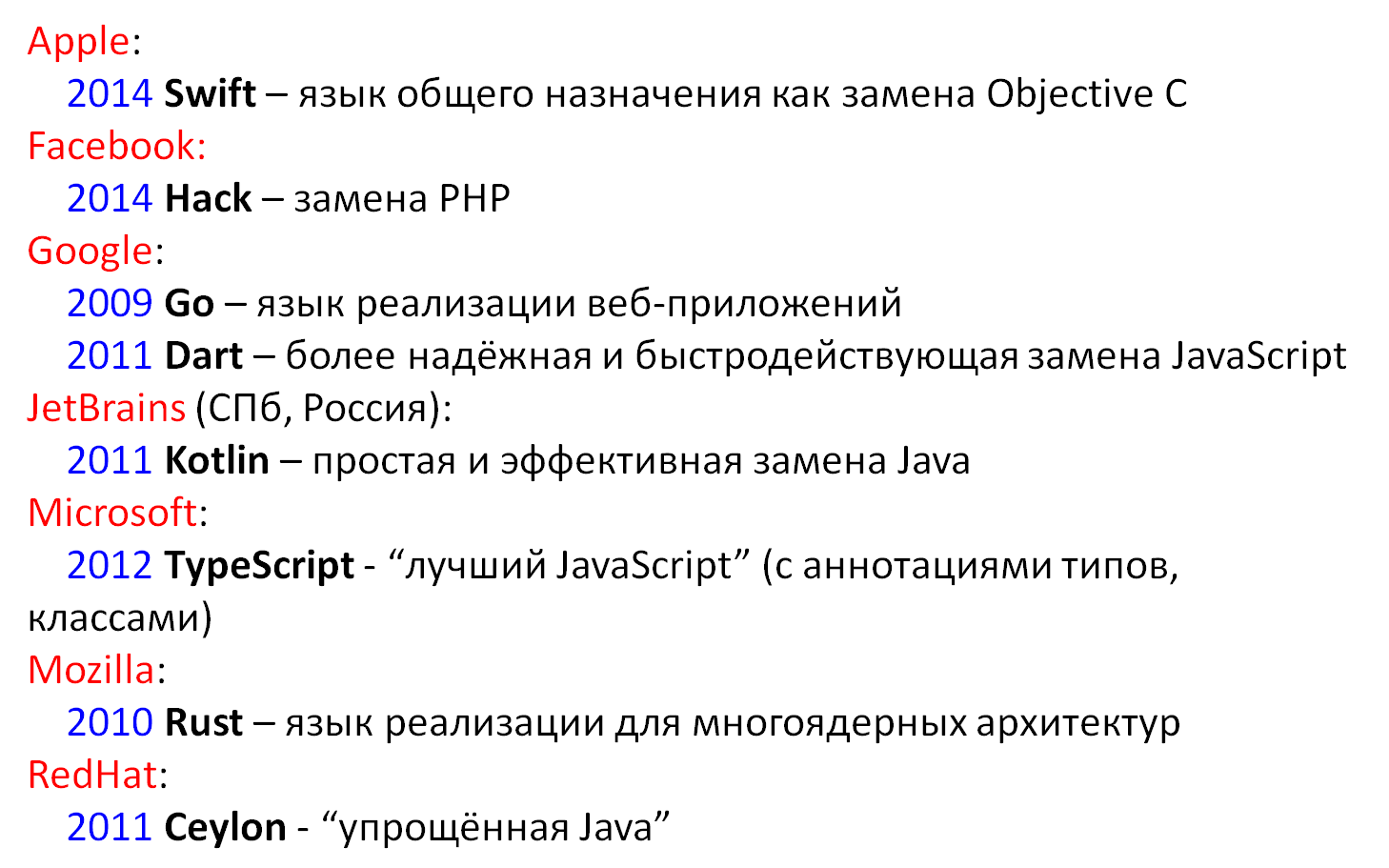
**Ключевые слова**

Языки программирования, модульность, тип, класс, конфигурация программ, наследование, мультитипы, нулевые указатели.

# НОВЫЕ ЯЗЫКИ: ТЕНДЕНЦИИ И ПОТРЕБНОСТИ

Одной из наиболее заметных тенденций последнего времени в сфере инструментальных средств разработки ПО является появление большого числа новых языков программирования (ЯП) и сопутствующих инструментов. В следующей таблице перечислены наиболее заметные языковые проекты последних лет (отсортированные по именам компаний-разрабочиков).

Таблица 1. Новые языки программирования

 К этому списку можно добавить языки, появившиеся в предшествующие годы, но именно в настоящее время получившие широкое распространение и популярность (например, Scala [3] и Python [2]).

Заметим, что приведенные в таблице примеры – не академические или экспериментальные проекты (которые появляются десятками каждый месяц) – речь идет именно о языках промышленного назначения, предназначенных для создания больших тиражируемых программных комплексов. Серьезность этих проектов подтверждает то обстоятельство, что фирмы и организации, представившие эти языки, как правило, ориентируются на их использование в собственных разработках.

Масштаб современного процесса языкотворчества позволяет говорить именно о *тенденции*, важность которой представляется несомненной. Следует отметить основные причины возникновения этой тенденции. В первую очередь, это говорит о неудовлетворенности существующими «мейнстримными» ЯП: они зачастую слишком сложны с точки зрения большинства разработчиков и требуют больших усилий по освоению. Жизненный цикл программ, создаваемых с использованием современных языков, связан с необходимостью использовать большое число инструментов (компиляторы, редакторы связей, инструменты управления проектами, интегрированные среды), которые зачастую слабо совместимы друг с другом и плохо интегрируются. Многие распространенные языки откровенно устарели и не соответствуют современным требованиям по части надежности, устойчивости и эффективности создаваемых программ.

Другим аспектом современного состояния средств разработки является появление новых задач и возрастание требований к создаваемому ПО. Важные и актуальные потребности (например, создание параллельных и распределенных программных комплексов, интернет-программирование) не поддерживаются на уровне языков и их стандартных библиотек и удовлетворяются за счет привлечения сторонних программных средств.

Дизайн многих существующих ЯП страдает серьёзными недостатками: он архаичен, неуклюж и негибок, основан на устаревших представлениях и концепциях, не учитывает актуальных потребностей, аспектов и технологий программирования, зачастую сформирован под влиянием личных вкусов и пристрастий их создателей и, как следствие, не удовлетворяет ни организации, занимающиеся созданием ПО, ни рядовых разработчиков.

Весь этот комплекс нерешенных проблем, устаревших принципов и подходов, потерявших свою актуальность, вместе с возникновением новых требований к инструментам разработки, и послужили важнейшими причинами «языкотворческого взрыва» последних лет.

В данной работе анализируются некоторые существенные и актуальные, на взгляд авторов, проблемы дизайна языков программирования, и предлагаются концепции и механизмы, которые позволяют преодолеть недостатки, свойственные многим современным ЯП. К числу таких концепций относятся:

* **Контейнер** (unit) как базовое понятие, обобщающее понятия модуля, класса и типа и представляющее основу для построения программ.
* Описывается практичный и надежный механизм решения проблем, связанных с множественным **наследованием**, позволяющий упростить создание комплексных программных систем со сложными семантическими связями. Оригинальный подход к множественному наследованию с конфликтами и множественным переопределением.
* Вводится понятие **мультитипа** как мощного механизма факторизации и расширения функциональности существующего кода без его модификации.
* Предлагается решение проблемы **неинициализированных данных** и, в частности, **нулевых указателей** на основе статических и динамических механизмов типового анализа.

Рассмотренные вопросы в совокупности могут лечь в основу проекта языка программирования, который был бы свободен от большинства недостатков, свойственных многим современным ЯП.

# КОНТЕЙНЕР КАК ОСНОВНАЯ ЕДИНИЦА ПОСТРОЕНИЯ ПРОГРАММ

Важнейшими концепциями, используемыми при разработке программного обеспечения (ПО), служат понятия атрибутов (данных) и подпрограмм (действий). Атрибуты могут изменяться подпрограммами в процессе работы программы; они образуют ее вычислительный контекст, в то время как подпрограммы задают алгоритм решения задачи. Между атрибутами и подпрограммами есть логические связи, и объединяя атрибуты и подпрограммы в единый именованный контейнер, мы просто фиксируем эту связь. Таким образом, понятие *контейнера* (английский термин, выбранный для его наименования,– unit) можно считать простым средством агрегации логически связанных данных и действий в единое целое.

Формальное определение контейнера звучит так. Контейнер (unit) – это поименованная совокупность атрибутов и подпрограмм, которая может быть параметризована либо типом, либо константным выражением перечислимого типа, и может быть использована для задания типов, конструирования новых контейнеров при помощи наследования или для прямого использования атрибутов и подпрограмм данного контейнера в других контейнерах и отдельно-стоящих подпрограммах.

Рассмотрим варианты использования контейнеров. Во-первых, его можно рассматривать как определение множества данных и операций над ними – то есть, как задание некоторого **типа**. Тем самым, можно определить объект, тип которого будет контейнером. Во-вторых, можно предоставить открытое (общедоступное) содержимое контейнера для **использования** в некотором программном коде, то есть, включить его ресурсы в некоторый контекст. Наконец, атрибуты и подпрограммы контейнера могут (пере)использоваться при создании нового контейнера. Такой механизм носит название **наследования**.

Таким образом, различные варианты использования контейнера приводят к понятиям *типа*, *модуля* и *класса*.

В большинстве современных языков программирования все перечисленные варианты композиции реализуются посредством единого понятия класса. Так, в С++ класс, все члены которого являются статическими, по существу, представляет собой простую агрегацию атрибутов и подпрограмм (вариант модуля). Аналогичное решение («статические классы») принято в языке C#. Заметим, что для представления модуля на основе класса приходится привлекать не вполне адекватное (обусловленное историческими причинами) понятие «статических» членов, а гибкое использование такого «модуля» (прежде всего, механизм его включения в определенный контекст) отсутствует.

В языке Scala [3] предпринята попытка отделить «модульную» часть класса в специальную конструкцию «объект-спутник» (companion object) с тем же именем, что и класс; однако на использование такого объекта накладываются существенные ограничения (в частности, он должен компилироваться в том же контексте, что и класс, спутником которого он является).

Определённым аналогом модуля можно считать механизм пространств имен (namespaces) в языках C++ и C#, а также пакеты (packages) языка Java, однако это крайне слабое средство модуляризации, введенное в языки прежде всего для разрешения конфликтов имен.

Таким образом, преимущества использования единственного понятия класса для задания различных видов программных контейнеров представляются сомнительными как с инженерной, так и с концептуальной точек зрения. В статье [1] с характерным названием «Импорт – это не наследование: Почему нам нужны и модули, и классы» приводятся убедительные обоснования необходимости явного разделения этих понятий. В то же время сосуществование в рамках единой языковой нотации понятий модуля и класса (например, в языках Ada [4] и Oberon [6]) выглядит несколько искусственным – при многих других несомненных достоинствах этих языков.

В данной работе предпринята попытка сохранить преимущества единой нотации задания контейнеров, совместив такую нотацию с явным заданием различных способов использования контейнеров.

Рассмотрим простой пример. Предположим, имеется некоторый контейнер, содержащий библиотеку математических функций.

**unit** Math  
 sin(Real): Real  
 cos(Real) : Real  
 ...  
**end**

По своей природе такой контейнер представляет собой модуль: он должен явно включаться в клиентскую программу, в процессе ее работы он существует в единственном экземпляре, независимо от числа его подключений. В то же время было бы желательно предоставить возможность переиспользования такого модуля, например, расширения его функциональности за счет включения новых математических функций. Такое изменение естественным образом реализуется посредством механизма наследования. Таким образом, контейнер должен обеспечивать как свойства модуля, так и свойства традиционного класса.

Итак, основную идею данного раздела можно сформулировать следующим образом: при сохранении единой формы задания контейнеров произвольной природы (модуль, класс, тип) обеспечить различные способы их использования. Иными словами, сделать так, чтобы один и тот же контейнер мог бы выступать в различных «ипостасях» в зависимости *от контекста его использования*.

Рассмотрим примеры, иллюстрирующие предложенную выше идею. Для упомянутого выше контейнера математических функций естественным (и, возможно, основным) способом использования будет включение его свойств в некоторый программный контекст:

use **Math**

Свойства Math (под *свойствами* контейнера мы будем понимать как его атрибуты, так и подпрограммы) будут использоваться традиционным способом: системой будет создан один объект (экземпляр) Math с собственным состоянием и предоставлять свой сервис всем контекстам, в которых явно задано использование этого юнита. Нотация доступа к свойствам Math допускает использование как простых, так и квалифицированных имен, например, sin(x) или Math.Pi.

Наряду с этим, представляется естественным обеспечить возможность расширения функциональности Math – например, за счет включения новых функций или более эффективных реализаций имеющихся функций. Эта возможность реализуется традиционным для объектно-ориентированного подхода механизмом наследования:

**unit** BetterMath **extends** Math  
 ...  
**end**

Таким образом, в данном контексте контейнер Math используется как обычный класс.

Наконец, в некоторых обстоятельствах Math можно рассматривать как тип и, соответственно, использовать его при определении объектов, например:

**var** m **is** Math

Такая запись вводит в текущий контекст объект с именем m, свойства которого определяются контейнером Math и доступны посредством обычной точечной нотации, например, m.sin(x).

Использование контейнера Math как типа допускает все возможности объектно-ориентированного, подхода, в частности, полиморфизм. Так, объекту m из примера выше можно присвоить объект производного типа и использовать «улучшенные» версии математических функций из этого производного типа:

m := BetterMath()  
...  
m.sin(x) // «Улучшенный» sin из контейнера BetterMath

Разумеется, природа конкретного контейнера не обязательно предполагает его реальное использование всеми тремя показанными способами. В каждом конкретном случае программист имеет возможность применить контейнер так, как требуется для решения конкретной задачи, в том числе и комбинируя способы, описанные выше. Кроме того, на тот или иной способ использования контейнера могут накладываться определенные ограничения. В частности, для того, чтобы контейнер мог использоваться как модуль, он должен иметь процедуру инициализации («конструктор») без параметров.

# КОМПОЗИЦИЯ ПРОГРАММ

## Единицы компиляции

Как правило, реальные программы состоят из многих (десятков или сотен) компонентов, и от правил организации компонентов, образующих программу, в значительной степени зависит ее сопровождаемость, развиваемость и в конечном итоге соответствие заданным требованиям.

Ассортимент компонентов непосредственно определяется сложившейся практикой программирования и должен включать контейнеры (units), предполагающие различные стили использования, подпрограммы (как возвращающие значения, так и обладающие только побочным эффектом), а также неименованные подпрограммы, или блоки: последовательности объявлений и/или операторов, предназначенные для однократного выполнения.

Несколько слов о правилах разрешения имен. Дополнительно к квалификации имени некоторого свойства контейнера именем этого контейнера, предлагается свободное использование *точечных имен*. Идея заключается в том, чтобы имя любого компонента самого внешнего уровня могло представлять собой последовательность произвольного числа идентификаторов, разделенных точками. Такое правило, избавляя от необходимости введения дополнительной конструкции (namespace или package), дало бы возможность систематического именования компонентов внутри программного проекта, исключающего конфликты имен. Например, имя контейнера могло бы включать информацию о разработчиках:

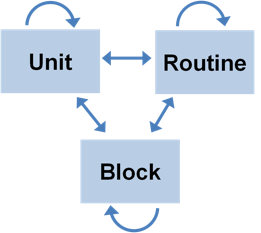
**unit** CompanyName.DepartName.LabName.MATH  
 ...  
 **end**

Использование единицы с длинным именем может допускать его сокращенный вариант, например

**use** CompanyName.DepartName.LabName.MATH **as** MATH

Важную роль в случае создания программных комплексов со сложной архитектурой играют правила вложенности компонентов. Практически все распространенные языки программирования накладывают те или иные ограничения на вложенность. Ярким примером может служить отсутствие вложенности функций в языке С и многих его потомках (на фоне этого запрета странно выглядит допущение локальных классов в С++).

Представляется, что общее правило должно быть максимально простым: любой компонент может содержать компоненты другого вида, в том числе и того же вида. Из этого правила следует возможность задания локальных и вложенных контейнеров и вложенных функций. Допустимые отношения вложенности могут быть наглядно представлены следующей схемой:



**Рис.1 Отношения вложенности единиц компиляции**

Представляется, что любое исключение из общих правил вложенности компонентов ограничивает возможности построения архитектуры ПО, адекватной требованиям, не продиктовано какими-либо концептуальными соображениями дизайна, но имеет реальной причиной желание упростить создание компилятора или других языковых утилит.

Продолжая эту логику, естественно считать, что компонент любого из указанных видов может служить *единицей компиляции*, то есть, допускать раздельную компиляцию. Отсутствие ограничений способствует созданию композиции программы, адекватной требованиям и особенностям ее использования. Программа может представлять собой, по сути, произвольную композицию единиц, от простого набора взаимодействующих подпрограмм до сложной комбинации контейнеров различных видов. В предельном случае программа может представлять собой единственную единицу – простую последовательность операторов. Если необходимо написать несколько строк кода, которые будут служить реакцией на нажатие клавиши мыши в некотором средстве просмотра Интернет-страниц, то нет необходимости писать подпрограмму: достаточно просто задать последовательность операторов, которая выполнит нужное действие. Если же решение задачи предполагает более сложную логику, то результат можно получить, комбинируя отдельные подпрограммы, контейнеры и блоки. Таким образом, единая языковая нотация может быть использована для решения максимально широкого класса задач.

Если говорить о формальном определении, то понятие единицы компиляции будет звучать так. Единица компиляции – это либо последовательность операторов, либо отдельно-стоящая подпрограмма, либо описание контейнера. Единицы компиляции могут находиться в одном файле/источнике (source) вместе с директивами включения (use) других контейнеров.

Рассмотрим несколько примеров. Известная классическая программа может быть записана в одну строчку:

StandardIO.put("Hello world!\n")

Контейнер StandardIO содержит подпрограмму консольного вывода put, которая вызывается с аргументом строкового типа. Контейнер StandardIO используется как модуль: при первом обращении к нему системой поддержки времени выполнения создается единственный экземпляр этого контейнера, из которого производится вызов заданной подпрограммы.

Следующий пример представляет альтернативное использование того же контейнера:

**use** StandardIO  
put("Hello world!\n")  
ch **is** getChar

Директива **use** задает контекст, в котором интерпретируются последующие вхождения имен. Так, обращения к подпрограмме put могут производиться без префикса с именем контейнера.

В последней строке примера создается и инициализируется локальный атрибут ch. Его начальным значением становится результат вызова функции getChar из контейнера StandardIO. Тип атрибута явно не задается, а выводится автоматически компилятором, исходя из описания функции getChar (на основе механизма, получившего название type inference).

## Общая конфигурация программных систем

Чем больше размер программы и сложнее ее внутреннее устройство, тем большее значение приобретают проблемы, связанные с управлением ее конфигурацией. Сюда относятся как вопросы установления, контроля и управления отношениями между компонентами (зависимостями), так и поддержание актуальности соответствия между исходными текстами и кодовыми образами компонентов.

Для большинства распространенных языков программирования характерны крайне архаичные подходы к решению подобных задач, использующие явное установление отношений между файлами компонентов и прямые текстовые включения интерфейсов (так называемые «include-файлы»). Многочисленные проблемы, связанные с такими подходами, отлично известны любому разработчику.

Представляется, что в современной системе программирования необходимо использовать более гибкие и надежные схемы конфигурирования. Принципиальным решением на этом пути может служить *полное отделение* вопросов конфигурирования программной системы и вопросов разработки ее компонентов. Единица компиляции в принципе не должна содержать никакой технической информации о своем окружении: имена файлов, содержащих используемые компоненты, пути доступа к ним и т.д. Семантика единицы компиляции не должна зависеть от того, где находится используемая ею единица. Расположение исходного текста единицы компиляции в каком-либо определенном источнике (например, в некотором дисковом файле) не должно влиять на правила установления отношений этой единицы с другими единицами – вне зависимости от того, находится ли используемая единица в тот же источнике, расположена ли она текстуально до или после данной.

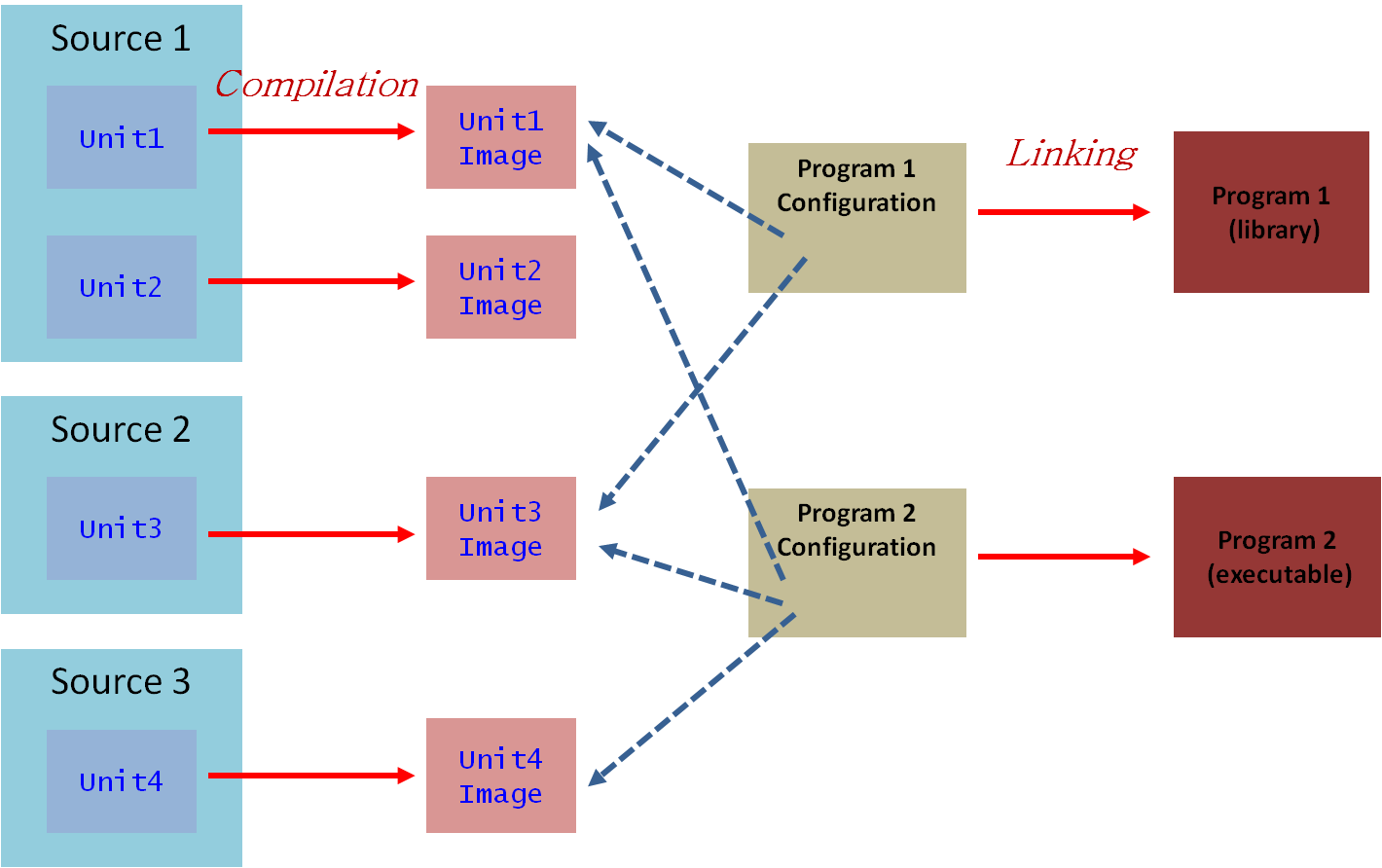
Более того, само понятие «исходного текста» программы или единицы компиляции не должно фиксироваться и, тем более, не должно включаться в определение языка. Вообще, нет никакого смысла явно указывать, откуда в системе появилась та или иная единица компиляции: ее текст может располагаться как в традиционном дисковом файле, так и появиться из любого другого источника – с консольного ввода, по сети связи, быть сгенерирован «на лету» некоторой утилитой, и т.д.

Один источник может содержать произвольное число единиц компиляции. Единственное ограничение заключается в том, что единица компиляции должна целиком содержаться в одном источнике.

Совокупность перечисленных свойств подразумевает наличие специального понятия – *файла конфигурации*, в котором будет сосредоточена минимально необходимая информация, «привязывающая» компоненты программы к операционному окружению,- прежде всего, соответствие имен компонентов информации о месте их хранения. Файл конфигурации может содержать также информацию о версиях компонентов, опциях компиляции и другие данные, нужные для поддержания актуального состояния программной системы.

Таким образом, файл конфигурации может служить своего рода «внешним представлением» программы, в то время как собственно исходные тексты и коды компонентов образовывать «программную библиотеку». Для одной библиотеки можно определить несколько различных файлов конфигурации, которые определяют различные программные системы, или версии одной системы.

Описанный подход можно иллюстрировать следующей схемой.



**Рис.2 Роль файла конфигурации**

Из приведенной схемы видно, что один и тот же компонент может включаться в различные программные конфигурации.

В заключение добавим, что описанный подход к конфигурированию предполагает отсутствие понятия «точки входа» в программу. Выполнение программы может начинаться с вызова любой подпрограммы (как одиночной, так и состоящей в контейнере) или с выполнения некоторого блока. Информация о точке входа также может храниться в файле конфигурации. Передача аргументов для выполнения программы, если они требуются, может происходить посредством запросов к операционному окружению.

# АЛЬТЕРНАТИВНЫЙ ПОДХОД К НАСЛЕДОВАНИЮ

Наследование как вид отношений между контейнерами дает возможность расширять возможности новых контейнеров, используя уже имеющиеся и доступные свойства существующих. Основные проблемы, которые возникают при реализации наследования,– это конфликт имен, неоднозначность версий свойства при полиморфном присваивании, а также согласование статусов экспорта свойств. Отмеченные проблемы возникают в основном при множественном наследовании, поэтому во многих современных языках возможности наследования ограничивают единичным наследованием, предлагая в качестве паллиативного решения понятие интерфейса (С#, Java) или «протокола» (Swift [7]).

Можно, однако, сохранить механизм множественного наследования со всеми его преимуществами, если не ставить задачей контроль правильности всего графа наследования для всех свойств всех контейнеров - как делается, например в языке Eiffel [5], котрый имеет мощный, но непривычный аппарат адаптации свойств при наследовании. Вместо этого можно ограничиться решением более простой задачи: проверки на наличие однозначного разрешения *обращения к свойству* контейнера (validity of the call).

Данный раздел содержит обсуждение преимуществ предложенной альтернативы. Рассмотрим схематический пример контейнера, в котором заданы две одноименные подпрограммами с различными сигнатурами.

**unit** A  
 foo(*signature1*)  
 foo(*signature2*)  
**end** А

Общая задача может быть сформулирована так: является ли правильным контейнер А? Иными словами, при каком соотношении сигнатур он непротиворечив? Вместо этой общей задачи поставим более конкретную задачу, которая исходит из *использования* контейнера в качестве типа. Пусть имеется обращение к подпрограмме foo:

a **is** A  
a.foo(*arguments*)

Мы хотим проверить, является ли корректным обращение к a.foo с некоторыми аргументами. Такое обращение будет правильным, если в А есть свойство или несколько свойств с именем foo, и типы аргументов вызова имеют однозначное разрешение среди сигнатур всех свойств foo. Если типы всех аргументов конформны только сигнатуре *signature1*, то перед нами обращение к первой версии foo, если только сигнатуре *signature2* – то ко второй foo. Если же аргументы вызова конформны обеим сигнатурам, то фиксируется неоднозначность, что трактуется компилятором как ошибка. При этом сам контейнер A с набором указанных подпрограмм не считается некорректным. Что же касается самого контейнера A, то единственная проверка, которая представляется для него необходимой,– контроль двух сигнатур на идентичность.

Механизм перегрузки (overloading) может использоваться также для разрешения конфликтов по именам при наследовании. Рассмотрим, как это может работать, на следующем примере.

**unit** B **extends** A  
 foo(*signature3*)  
 **override** foo (*signature4*)  
**end** B

Контейнер В вводит *новое* свойство foo с сигнатурой *signature3*. Единственное ограничение на *signature3* заключается в требовании неидентичности с *signature4*, определенной в этом же контейнере.

В то же время наличие префикса **override** для foo с сигнатурой *signature4* приводит к перекрытию (overriding) *обеих* foo из базового контейнера А, при условии, что *signature4* конформна обеим сигнатурам *signature1* и *signature2*. Если *signature4* конформна только одной foo из базового контейнера, то другая подпрограмма наследуется в В.

Иными словами, основой механизма разрешения конфликтов перекрытия служит конформность сигнатур. Рассмотрим несколько более сложный пример.

**unit** T1 ... **end**  
**unit** T2 **extends** T1 ... **end**  
**unit** T3 **extends** T1 ... **end**  
**unit** T4 **extends** T2, T3 ... **end**

**unit** A  
 foo(arg1: T1; arg2: T2)  
 foo(arg1: T1; arg2: T3)  
**end** А

**unit** B **extends** A  
 foo (arg1: T4)  
 **override** foo (arg1: T4; arg2: T4)  
**end** B

В контейнерах А и В имеются по две подпрограммы foo с различными сигнатурами. Рассмотрим различные случаи использования этих подпрограмм[[1]](#footnote-1).

**var** a **is** A

Создается объект а, типом которого считается контейнер A. При создании производится вызов процедуры инициализации из A.

a.foo(T1(),T2())

Разрешение однозначно: перед нами обращение к первой foo из А.

a.foo(T1(),T3())

Однозначное обращение ко второй foo из А.

a.foo(T1(),T4())

Неоднозначность: могут быть вызваны обе версии A.foo.

a := B()  
a.foo(T1(),T2())

Однозначное обращение к первой foo. При выполнении будет вызвана версия foo из В с сигнатурой (T4,T4). Представленный случай может служить иллюстрацией так называемых «cat calls»: нарушением типизации вследствие ковариантного переопределения сигнатур и полиморфного присваивания. При проведении полной проверки корректности данного вызова (на уровне всей собираемой системы) он будет отвергнут компилятором.

a.foo(T1(),T3())

Это однозначное обращение ко второй foo из А. При выполнении, как и в примере выше, будет вызвана версия foo из контейнера В с сигнатурой (T4, T4), перекрывающая «базовую» версию.

a.foo(T1(),T4())

Неоднозначность: обе версии foo конформны аргументам.

В заключение данного раздела отметим, что предложенный механизм разрешения неоднозначностей в случае множественного наследования, основанный на анализе полиморфного использования, позволяет не решать общую теоретическую задачу проверки правильности графа наследования, а просто решать конкретные маленькие задачи, с которыми сталкивается программист-практик, оставаясь только с одной концепцией наследования для построения повторно используемого ПО.

# МУЛЬТИТИПЫ

Наследование является мощным механизмом повторного использования кода, однако на практике для него имеются некоторые ограничения. Так, однажды реализованный граф наследования не может быть изменен без изменения исходного кода программы (который может отсутствовать) и без перекомпиляции клиентского кода (что может быть неприемлемо или невозможно). Тем самым, создание общего программного кода для работы с объектами различных типов приводит к существенным затруднениям.

В качестве дополнения к механизму наследования, которое способно решить подобные проблемы, предлагается понятие **мультитипа**. Введение в язык этого понятия вместе с соответствующим механизмом контроля может заменить правила структурной эквивалентности типов без нарушения принципов статической типизации.

Рассмотрим простой пример. Определим атрибут number как объект, тип которого представляет *пересечение* нескольких типов:

number: Integer | Real | Double

Таким образом, атрибуту number можно присваивать сущности типов Integer, Real, Double или их наследников. Соответственно, можно обращаться к тем свойствам мультитипа, которые присутствуют во всех трех типах, например:

number := number1 + number2

С формальной точки зрения, мультитип считается корректным тогда и только тогда, когда все пары типов, входящих в мультитип, не являются конформными друг другу[[2]](#footnote-2). В примере выше свойство сложения, которое обозначается инфиксной операцией +, должно присутствовать в типах Integer, Real и Double. Кроме того, вызов вида number.+(number) должен быть правильным для всех видов сочетаний Integer.+(Integer), Real.+(Real) и Double.+(Double).

# НЕИНИЦИАЛИЗИРОВАННЫЕ ПЕРЕМЕННЫЕ И НУЛЕВЫЕ УКАЗАТЕЛИ

Проблема, связанная с неконтролируемым использованием нулевых указателей («пустых» или «повисших» ссылок), является одной из наиболее распространенных в практике программирования, а также одной из самых опасных по своим последствиям с точки зрения обеспечения надежности программ. В то же время, контроль доступа по таким указателям не имеет удовлетворительного решения в традиционных языках.

В данной работе предлагается рассматривать проблему пустых указателей не как самостоятельную проблему, а трактовать ее как часть более общей проблемы некорректной работы с *неинициализированными атрибутами*. Если рассматривать пустую ссылку как разновидность неинициализированного атрибута, то необходимо дополнить язык механизмами, которые бы строго ограничили те случаи, когда нам действительно нужны неинициализированные атрибуты, от ситуаций, когда всякая сущность должна иметь определенное значение. Кроме того, следует обеспечить надежный механизм перехода от потенциально неинициализированных атрибутов к инициализированным – своего рода мостик от «опасного» мира в «безопасный».

Основные моменты предлагаемого подхода лучше всего показать на примерах. Рассмотрим контейнер Integer, который содержит некоторый ассортимент операций над объектами целочисленных типов, а также может определять собственно целочисленный тип. Программное определение такого контейнера имеет следующий вид:

**val** **unit** Integer  
 ...  
**end**

Префикс val перед объявлением контейнера говорит о том что по умолчанию все объекты этого типа являются *значениями*, а не ссылками. В то же время объявление вида

ri: **ref** Integer

задает *ссылку* на объект типа Integer.

Эти предварительные комментарии приведены, чтобы подчеркнуть то обстоятельство, что понятие инициализированности атрибута применяется как для ссылок, так и для значений (самих объектов) и, разумеется, не только для типа Integer, но для любых типов. Теперь собственно сам пример.

attr1: Integer **is** 5

Как правило, компилятор не пропустит отсутствие инициализации **is** 5: считается, что локальный атрибут attr1 должен получить значение при его описании.

attr2: Integer

Пусть attr2 – атрибут контейнера, и инициализировать его удобнее не непосредственно в описании, а при создании объекта-контейнера, то есть, в теле процедуры инициализации. Семантика языка предполагает, что факт получения этим атрибутом некоторого значения после завершения работы процедуры инициализации будет проверен – либо статически, в процессе компиляции, либо системой времени выполнения. Таким образом, общее правило звучит так: «обычное» объявление всегда приводит к созданию объекта, имеющего значение.

Наряду с этим, имеется альтернативная форма объявления:

attr3: ?Integer

Атрибут attr3, объявленный таким образом, *не имеет значения*. Ни компилятор, ни система времени выполнения не будут контролировать факт получения этим атрибутом значения при инициализации. Однако, с таким атрибутом нельзя выполнять никаких действий, кроме передачи его в качестве аргумента или присваивания ему значения.

Таким образом, мы можем свободно работать с атрибутами attr1 и attr2, присваивая их значения друг другу или другим атрибутам, объявленным аналогичным образом. Кроме того, допустимым является присваивание attr3 := attr2 (в результате этой операции атрибут attr3 получает значение), однако присваивание attr1 := attr3 считается некорректным; допущение такого действия означало бы неконтролируемое распространение неинициализированного значения.

Если же логика программы требует выполнить подобное присваивание, то для его корректности следует предварительно проверить, имеет ли атрибут attr3 в данной точке программы некоторое значение. Отправным моментом такой проверки служит идея, согласно которой *только инициализированный атрибут имеет тип*. Тем самым, требуется узнать, является ли attr3 правильным объектом типа Integer. Таким образом, мы сводим проверку на наличие значения *к динамической проверке типа* атрибута.

Вот как выглядит переход от потенциально неинициализированной переменной к безопасной:

**if** attr3 **typeof** Integer **then**  
 attr3 := attr3 + 5  
 attr1 := attr3  
**end**

Внутри then-части условного оператора динамический тип атрибута attr3 гарантированно есть Integer (или его наследник), и потому обращение к свойствам attr3 или присваивание его другим инициализированным атрибутам является заведомо корректным. Однако, после завершения условного оператора attr3 не считается гарантированно инициализированным, и опять действуют ограничения на работу с ним. Следует иметь ввиду, что наряду с операцией получения значения, имеется и «симметричная» операция потери значения (или отмены инициализации), и она может быть выполнена над attr3 в любой момент выполнения:

?attr3

Такая операция, очевидно, корректна только для атрибутов, объявленных как неинициализированные. Поэтому, например, конструкция ?attr1 приведет к ошибке времени компиляции.[[3]](#footnote-3)

Раздел завершается примером более практического характера, который показывает использование описанного механизма совместно с типовой параметризацией.

// Создаем список из одного элемента  
list1 **is** List[Integer](5)  
// Пустой список  
list2 **is** ?List[Integer]   
list2 := list1

**if** list2 **typeof** List[Integer] **then**  
 // Добавляется еще один элемент  
 list2.add(128)  
 **while** list2 **typeof** List[Integer] **loop**  
 // Цикл работает, пока очередной list2  
 // имеет значение  
 StandardIO.put(" ",l2.item)  
 // Перейти к следующему элементу списка  
 list2 := list2.next   
 **end**  
**end**

**unit** List[G]  
 item **is** G  
 next **is** ?List[G]  
 setItem(other: G)  
 item := other  
 **end**  
 add(other: G)  
 next := List[G](other)  
 **end**  
  
 **init**(element: G)  
 setItem(element)  
 ?next  
 **end**  
**end**

Представляется, что предложенное решение – более полное и практически более удобное, нежели сходные решения в других языках программирования (C#, Kotlin [8], Eiffel и другие).

# ЗАКЛЮЧЕНИЕ

Предлагаемые в статье языковые понятия и конструкции могут лечь в основу достаточно мощного языка программирования, применимого как для разработки небольших фрагментов кода (аналогичных скриптовым языкам), так и для масштабных программных решений, где необходимо надежно решать вопросы интеграции различных программных компонент. В статье вводятся понятия последовательности операторов как единицы компиляции, дается нетрадиционный подход к базовому строительному блоку программного кода – контейнеру атрибутов и подпрограмм, альтернативная схема наследования, как простого и единственного механизма расширения уже существующего ПО, концепция мультитипа, как расширение наследования, а также предлагается полное решение для проблем нулевых указателей и неинициализированных данных. Кроме того, представляется подход к явному разделению программного кода и описания конфигурации ПО.

Также следует отметить, что концепции, обсуждаемые в статье, рассматриваются авторами как некоторая основа (контейнеры + наследование), которая может быть использована в качестве базы для создания нового языка или как возможное расширение одного из существующих языков; данная тема не является предметом и целью статьи.

Заметим, что нотация, использованная для демонстрации концепций, не является фиксированной и была выбрана таким образом, чтобы быть понятной для читателей, знакомых с языками C/C++/Java/Pascal/Ada.

Наконец, подчеркнем, что за рамками данной статьи остались многие другие концепции – в частности, автоматический вывод типов (type inference), атрибуты только для чтения, лямбда-функции, обобщённое программирование (generics), предикаты, параллельное программирование на базе контейнеров - так как перечисленные концепции уже присутствуют в современных языках программирования. В то же время эти концепции, безусловно, рассматриваются в рамках проекта, представленного в данной статье. И, конечно, детали реализации – способ исполнения программ (традиционная компиляция с последующим исполнением двоичного кода, интерпретация, JIT-компиляция) и области применения не являются ограничивающими факторами для применения представленных концепций.

# ССЫЛКИ

1. Clemens A. Szyperski: Import is Not Inheritance. Why We Need Both: Modules and Classes, ECOOP 1992.
2. The Python Language Reference, https://docs.python.org/3.3/reference/.
3. Martin Odersky, Lex Spoon, and Bill Venners: Programming in Scala, Second Edition, Artima Press, 2010.
4. International Standard: ISO/IEC 8652:2012 Information technology – Programming Languages – Ada.
5. Bertrand Meyer: Object-Oriented Software Construction, Second Edition. Prentice Hall. ISBN 0-13-629155-4.
6. N.Wirth: The Programming Language Oberon, http://www.inf.ethz.ch/personal/wirth/Oberon/Oberon.Report.pdf
7. The Swift Programming Language Reference: <https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/AboutTheLanguageReference.html>.
8. The Kotlin Language Reference. http://kotlinlang.org/docs/reference/

1. В последующих примерах нотация вида T() обозначает создание нового объекта данного контейнера (то есть, в данном случае он рассматривается как тип). При этом вызывается процедура инициализации контейнера без параметров, если она имеется. В такой нотации можно опускать пустые круглые скобки. Кроме того, перед именем контейнера можно поставить служебное слово **new**, чтобы явно подчеркнуть создание объекта (экземпляра), например **new** T. Наконец, допускается конструкция явного вызова процедуры инициализации экземпляра: T.**init**(*arguments*). Все эти варианты синтаксиса считаются семантически эквивалентными, однако здесь для простоты используется только вид нотации, указанный первым. [↑](#footnote-ref-1)
2. Понятие конформности определяется следующим образом: тип А конформен типу В, если в графе наследования существует путь от А к В. Соответственно, обращение к некоторому свойству foo из объекта типа мультитип является правильным тогда и только тогда, когда обращение к foo для мультитипа является правильным для всех типов, входящих в мультитип. [↑](#footnote-ref-2)
3. Проведенные рассуждения носят высокоуровневый характер; конкретная реализация может использовать разные механизмы для достижения описанной семантики. [↑](#footnote-ref-3)