Концепция наследования в современных языках программирования

А.В. Канатов <a.kanatov@samsung.com>
Е.А. Зуев <e.zouev@samsung.com>
Исследовательский центр Samsung,
127018, Россия, г. Москва, ул. Двинцев, дом 12, корпус 1

Аннотация. Статья содержит обзор и анализ реализаций понятия наследования в современных промышленных языках программирования. Исследуются достоинства и недостатки механизмов наследования в таких языках, как C++, Java, C# и Eiffel и других, анализируются их особенности и ограничения моделей наследования, реализованных в этих языках.

На основе проведенного анализа в статье предлагается альтернативный подход к трактовке наследования, который сочетает общность и гибкость множественного наследования и простоту практического применения для целей повторного использования кода. Суть предлагаемого подхода заключается в переносе контроля валидности полного графа наследования на этап обработки обращений к свойствам класса на основе анализа перекрытий (overriding) и контроля подобия (conformance) сигнатур свойств.

Предложенный подход может быть реализован как дополнение к какому-либо существующему языковому инструменту, так и в виде независимой реализации.

Ключевые слова: множественное наследование, переопределение (overriding) конфликт имен и версий, понятие источника свойства (origin and seed).

DOI: 10.15514/ISPRAS-2015-27(6)-12

Для цитирования: Канатов А.В., Зуев Е.А. Концепция наследования в современных языках программирования. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 169-188. DOI: 10.15514/ISPRAS-2015-27(6)-12.

1. Введение

Понятие наследования является одной из фундаментальных концепций в современных ЯП, поддерживающих парадигму объектно-ориентированного программирования.

Понятие наследования служит адекватной концептуальной моделью широкого множества различных схем отношений между сущностями реального мира. С

инженерной точки зрения, языковой механизм, обеспечивающий расширение возможностей классов вместе с заданием полиморфного поведения их свойств в производных классах, служит удобной и надежной основой повторного использования кода при разработке сложных программных систем.

Практически все современные языки программирования реализуют понятие наследования и содержат соответствующие языковые механизмы. В то же время, конкретные реализации наследования в распространенных языках либо заметно ограничены по сравнению с общей теоретической концепцией, которая просто рассматривает наследование как форму отношений между сущностями, не накладывая никаких ограничений, либо следуют определенной модели объектов времени выполнения, вводя в свою очередь решения, продиктованные реализацией (class layout, virtual method table structure).

Также необходимо отметить, что основные проблемы, которые возникают при реализации наследования,– это конфликт имен, неоднозначность версий свойства при полиморфном присваивании, а также согласование статусов видимости 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 (C# [3], Java [2], Kotlin [12]) или «протокола» (Swift [11]).

2. Обзор существующих подходов к поддержке наследования.

2.1 Полная поддержка наследования: язык C++

Наиболее известной реализацией модели множественного наследования служит язык C++. Несмотря на полноту реализации всех аспектов, связанных с организацией множественного наследования, и соответствующим богатством изобразительных возможностей, этот язык подвергается серьезной и обоснованной критике в связи со сложностью реального программирования и недостаточной надежностью создаваемых программ, прямо вытекающих из намерения поддержать все мыслимые потребности прогораммистов в рамках единого языка.

Две схемы ниже иллюстрируют различные аспекты организации множественного наследования в C++. Первый пример (рис. 1) представляет условную схему обычного множественного наследования, при которой в каждом из двух подобъектах базовых классов Man и Woman присутствует, в свою очередь, подобъект базового класса Person.

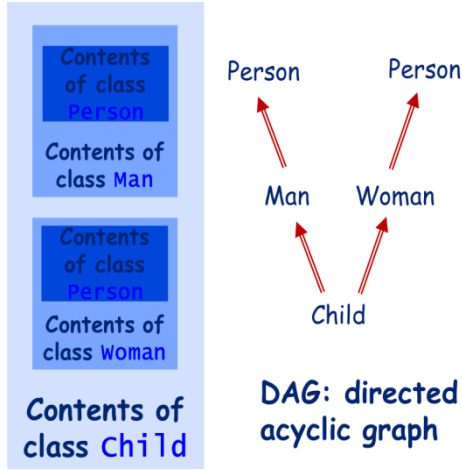

Рис. 1. Пример обычного (невиртуального) наследования в C++

Второй пример представляет модификацию схемы из предыдущего примера, когда подобъект базового класса Vehicle присутствует в объекте производного класса только в одном экземпляре, несмотря на то, что формально оба подобъекта Car и Plane (дважды) наследуют один и тот же класс Vehicle.

Такая композиция иллюстрирует понятие виртуального наследования, а соответствующая схема носит название diamond scheme (из-за ее сходства с кристаллической решеткой алмаза).

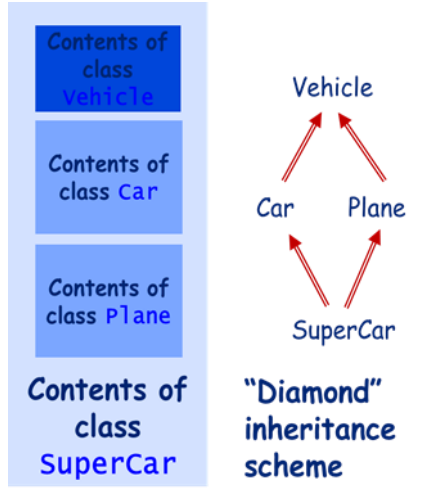

Рис. 2. Пример виртуального наследования в C++

2.2 Oberon и Zonnon

Философия языка Oberon [10] заключается в предельноем упрощении, как самого механизма наследования, так и его реализации. В языке поддерживаются лишь базовые возможности единичного наследования и полиморфизма. С одной стороны, это позволяет весьма эффективно реализовать данный механизм, что необходимо по причине ориентации языка на реализацию встроенных систем с ограниченными ресурсами. В то же время минимализм модели наследования Oberon дает возможность использовать язык для целей обучения.

Язык Zonnon [4], позиционируемый как непосредственный наследник языковой линии Pascal – Modula-2 – Oberon, переосмысливает традиционный взгляд на объектный подход в целом и на механизм наследования, в частности. Вместо привычной схемы – базовый и производный классы, возможно, реализующие некоторые интерфейсы – этот язык предлагает концептуально более чистую модель, согласно которой ключевым понятием и одновременно единственным объектом (множественного) наследования является абстрактное понятие интерфейса, а объекты выступают только как конечные «реализации» определенного интерфейса или группы интерфейсов.

Более подробно, Zonnon определяет следующие базовые единицы программы, на основе которых строится его объектно-ориентированная модель:

- Описание (definition): единый элемент абстракции. Он задает абстрактный интерфейс, а также может уточнять (refine) другое описание.
- Реализация (implementation): предоставляет реализацию некоторого описания «по умолчанию», а также может выступать как независимая коллекция ресурсов, агрегируемая в объекты.
- Объект (object): программно-управляемый ресурс, который реализует абстрактный интерфейс некоторого описания (описаний), а также может определять параллельное поведение.
- Модуль (module): специальный объект, управляемый системой (singleton object). Включает коллекцию ресурсов (данных и функциональностей), в том числе объектов и реализаций.

Рис.3 иллюстрирует модель наследования, реализованную в языке Zonnon.

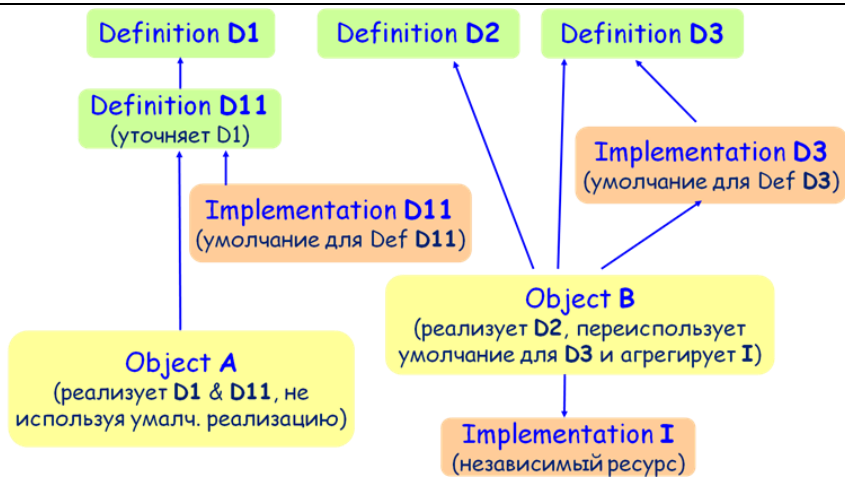

Рис. 3 Модель наследования языка Zonnon

2.3 C#, Java и Scala

Как говорилось выше, единичное наследование является в настоящее время доминирующей реализаций ООП в современных языках, предоставляя разумный компромисс между сложностью реализации и использования множественного наследования и поддержкой важнейших преимуществ наследования как базового принципа проектирования программ.

В качестве частичной компенсации за отказ от «настоящего» множественного наследования такие языки, как Java и C# предлагают в качестве альтернативы понятие интерфейса. Это понятие, будучи существенно легче для понимания и реализации, нежели сходное понятие «абстрактных классов» C++, служит в то же время и адекватной моделью многих реальных отношений.

На следующем рисунке схематически показаны отношения, характерные для таких языков.

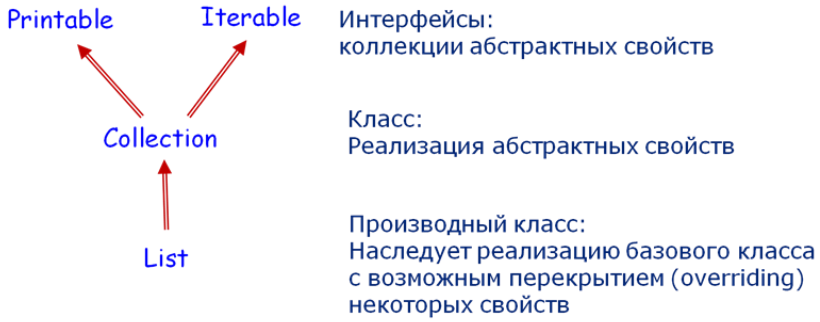

Рис. 4 Модель с единичным наследованием и множественными интерфейсами

Язык Scala [7] сохраняет в целом подход, принятый в Java (единичное наследование и интерфейсы), однако заметно расширяет возможности наследования за счет введения понятия трейта (trait), позволяющего создавать более гибкие и функционально богатые схемы отношений между объектами. Кроме того, в языке имеются возможности гибкого задания различных схем наследования (ковариантность или контравариантость), которые, однако, весьма сложны для практического использования и ориентированы больше на разработчиков прикладных библиотек, нежели на конечных пользователей языка.

2.4 Нишевые модели наследования: Ada и Eiffel

Ada95 и последующие редакции стандарта [8] добавляют объектно-ориентированные возможности к предыдущей версии языка (Ada83) при сохранении максимально возможной обратной совместимости Ада-программ.

Объектная модель Ады основана на расширении традиционного для императивных языков понятия записи (record) введением так называемых тегированных записей (tagged record), «расширяющих» некоторую другую запись. Один из атрибутов тегированной записи (‘Class) предоставляет средства, традиционные для объектного подхода, включая наследование, полиморфное присваивание и др.

В целом, ООП в этом языке, предоставляя достаточно полный спектр механизмов единичного наследования, реализован способом, не слишком привычным для разработчиков, использующих «мейнстримные» C-подобные языки, и потому практическое использование механизма наследования в этом языке представляется для них весьма затруднительным.

Для языка Eiffel характерна полная реализация полного варианта множественного наследования, которая сочетается с поддержкой настраиваемых классов (generics). Характерный для Eiffel механизм предикатов естественным образом интегрирован с механизмом наследования. Eiffel не подразумевает понятие подобъекта (характерное, например, для C++), а интерпретирует каждый метод и атрибут как отдельную сущность, к которой при наследовании могут применяться различные адаптации. Эти адаптации позволяют менять имя свойства, задавать новое тело, делать унаследованный метод абстрактным, изменять его видимость и разрешать неоднозначность при множественном наследовании.

Подобный механизм адаптации для многих программистов, знакомых с C-подобными языками, образует повышенный «барьер вхождения»; эти особенности, а также некоторые другие непривычные свойства языка вызывают определенное отторжение у современного программистского сообщества.

2.5 Нестандартные модели: JavaScript

Другая группа современных ЯП воплощает нестандартные подходы к реализации ООП и, в частности, к понятию наследования. Так, язык JavaScript [5] предлагает альтернативный подход, получивший название «прототипного наследования», при котором отношения между объектами устанавливаются динамически на основе общего для них предопределенного свойства («прототипа»).

На следующем рисунке схематически показано различие в подходах между статической и динамической моделями наследования.

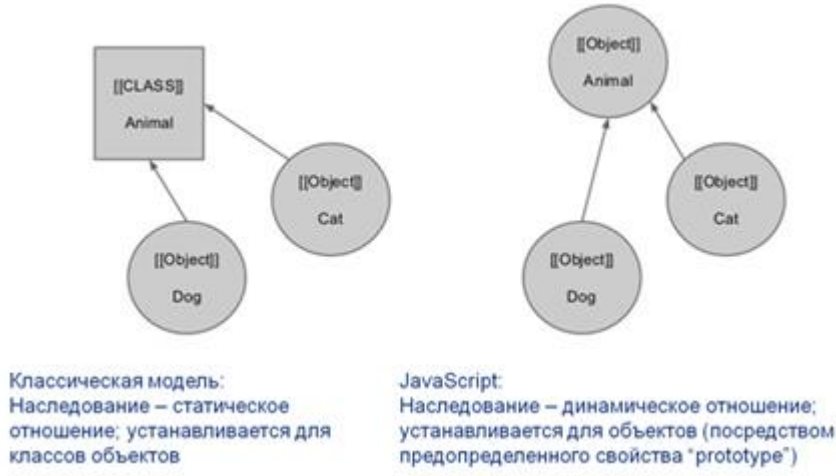

Рис.5 Различные модели наследования

Подобный подход, давая определенные преимущества в сфере построения интерактивных программ, не может считаться универсальным из-за слабых средств статического контроля, легкости внесения ошибок и трудностей в отладке.

В заключение краткого обзора следует упомянуть о языке Python [6], который не вполне вписывается в представленную выше классификацию и представляет в целом удачную попытку совмещения традиционного единичного наследования и динамической типизации, расширяя тем самым привычную парадигму «ООП = Статическая система типов».

3. Предлагаемая модель наследования

На основе анализа существующих моделей реализации наследования в современных языках программирования предлагается подход, который, как представляется авторам, преодолевает недостатки и ограничения существующих схем, сохраняя при этом механизм множественного

наследования со всеми его преимуществами. Для начала рассмотрим несколько известных определений, которые понадобятся для раскрытия сути концепции.

Класс: поименованная совокупность свойств, где свойство может быть либо атрибутом (переменным или константным) или подпрограммой (функцией, методом).

Наследование: направленное отношение между классами, при котором все свойства класса-родителя (базового) переходят к классу-наследнику (производному).

Отношение подобия (conformance): определяется наличием пути в графе наследования от одного класса до другого. Направленность графа наследования совпадает с отношением подобия.

Класс-источник (origin): это класс, в котором данное свойство было описано впервые.

Свойство-источник (seed): это та версия свойства, где это свойство было первый раз описано.

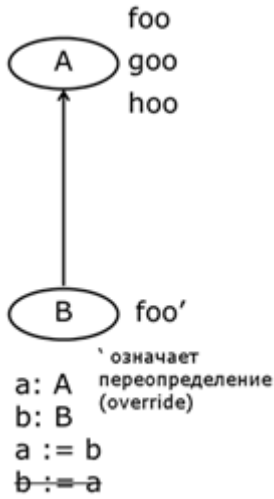

Рис. 6 Основные понятия, связанные с наследованием

На рис. 6 источником для свойства foo из класса B является свойство foo из класса A, где свойство foo было описано первый раз.

Очевидно, что для класса B класс A является родителем или непосредственным предком (базовым классом), а для класса A класс B

является наследником (производным классом). Понятия предка и потомка суть лишь транзитивные отношения от родителя и наследника соответственно.

Отношение подобия задает возможность операции присваивания для объектов типа B или его потомков объектам типа A.

Теперь рассмотрим, какие проблемы приходится решать при наличии множественного наследования в общем случае.

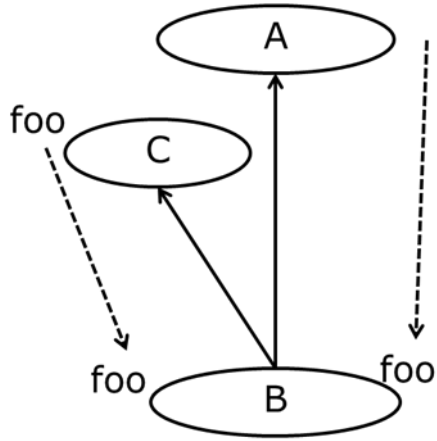

Рис. 7 Конфликт имен при множественном наследовании.

Первая задача заключается в определении того, сколько версий свойства foo будет в наследнике B, если в него наследуется более одной версии свойства с именем foo. В данном случае (см. рис. 7) у нас есть две версии foo из классов A и C, соответственно. Такая ситуация является примером конфликта имен и должна иметь определённое решение.

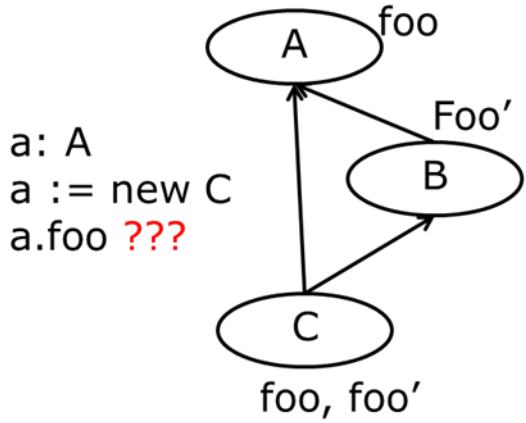

Рис. 8 Неоднозначность версий при множественном наследовании.

Для данного случая возникает неоднозначность между разными версиями свойства foo так как класс C имеет две версии – одна которая была получена непосредственно из класса A и вторая из класса B. И тогда возникает неоднозначность при полиморфном присваивании. Какая версия свойство foo должна быть вызвана при выполнении кода слева. Этот вопрос тоже должен иметь свое решение.

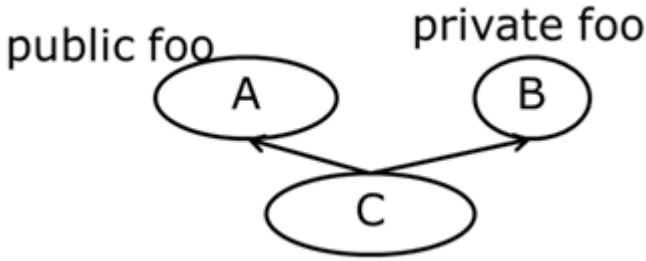

Рис. 9 Видимость свойств при множественном наследовании

Третий случай несет некоторую проблематичность. Если наследуются версии foo с разным уровнем видимости, то если они сливаются в одно свойство, необходимо решить, какая должна быть видимость у этог свойства? Эта проблема гораздо проще, чем первые две, но и она должна иметь свое решение.

Итак, начнем с конфликта имен. Рассмотрим пример

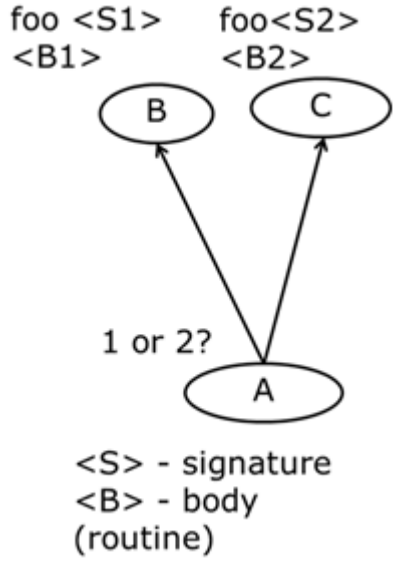

Рис. 10 Частный случай конфликта имен и его разрешение

Из классов B и C в класс A наследуются два свойства с одинаковым именем foo. Разрешение этого конфликта предлагается на основе следующего подхода. Если сигнатуры <S1> и <S2> идентичны, тела <B1> и <B2> идентичны и оба свойства имеют один и тот же источник (origin & seed), то это одно и тоже свойство и, следовательно, оно будет присутствовать в A в единственном экземпляре. Иными словами, если одно и тоже свойство приходит в класс разными путями, то можно считать, что это одно свойство (по сути, это практически то же самое, что виртуальное наследование в C++). А вот если не выполняется хотя бы одно из условий, перечисленных выше то, тогда класс B будет иметь два свойства с именем foo. Таким образом, допускается перекрытие имен (overloading).

Теперь рассмотрим короткий пример использования класса A.

```
a: A

a.foo(<parameters)>
```

Из примера видно, что в зависимости от того, как связаны между собой сигнатуры версий и их источники, обращение к свойству может быть либо однозначно разрешено компилятором, либо квалифицировано им как неоднозначность. В случае, если foo существует в A в единственной версии, то неоднозначности нет. Если же присутствует более одной версии, то в зависимости от типов параметров, которые передаются данному свойству,

компилятор пробует определить, какая версия из класса A должна быть вызвана. Если такое определение свойства дает единственный результат, то неоднозначности не возникает и именно это свойство и должно быть вызвано. В случае неоднозначности компилятор выдает сообщение об ошибке. Таким образом, мы уходим от тотальной проверки правильности всего графа наследования для всех свойств и проверяем правильность обращений к свойствам. Если обращения могут быть однозначно разрешены компилятором, то программа будет считаться корректной.

Рассмотрим еще один пример (рис. 11), когда у нас в классе наследнике есть новая версия свойства – т.е. мы производим переопределение свойств при наследовании (overriding).

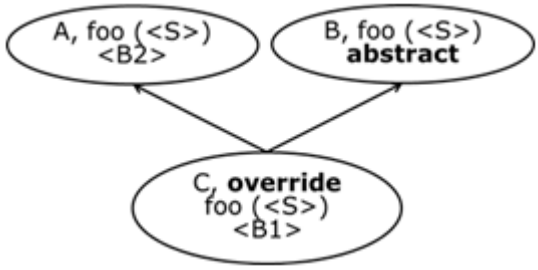

Рис. 11 Простой пример переопределения

Начнём с простого случая, когда все сигнатуры идентичны. Класс C вводит новую версию свойства foo с идентичной сигнатурой S и некоторым новым телом B1 и переопределяет обе версии foo, которые наследуются из A и B соответственно. Хотелось бы обратить внимание, на то что в классе A свойство foo имело реально тело, в то время как в классе B тело отсутствовало – foo являлось абстрактным методом. И свою очередь foo из C может быть как конкретным свойством так и абстрактным.

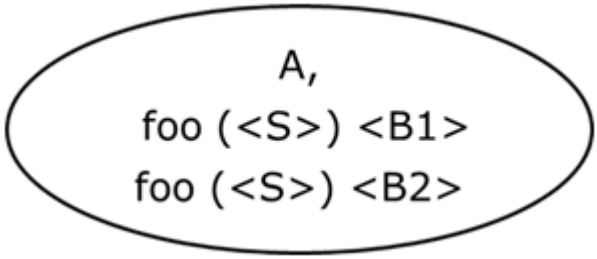

Рис. 12 Перекрытие имен свойств – недопустимая ситуация

Если еще раз обратиться к концепции перекрытия имен, то возникает вопрос, какое перекрытие является допустимым. Предлагается, что в рамках одного

класса два свойства являются различными, если у них разные имена или если у них одно имя и различные сигнатуры. Тем самым в рамках одного класса предлагается трактовать как ошибочную ситуацию, представленную на рисунке 12.

Однако, такого рода ситуация может возникнуть при наследовании и мы ее рассмотрим отдельно.

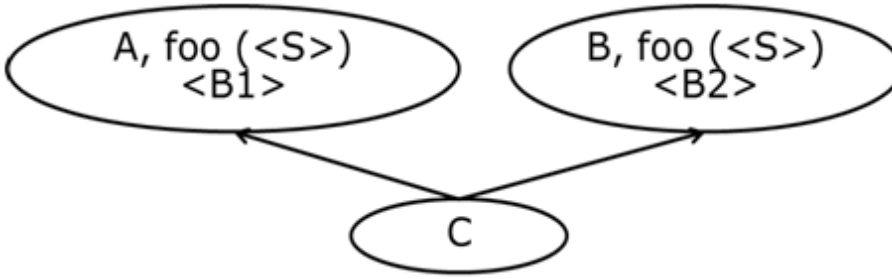

Рис. 13 Неоднозначность при множественном наследовании.

В данном случае на рис. 13 свойства foo из A и foo из B не должны иметь общего источника и тогда в классе C у нас есть две перегруженные версии с идентичными сигнатурами и разными телами. И теперь давайте рассмотрим следующие обращения. Если обращение текстуально находиться в C или его наследниках, то обращения A.foo (<exprS>) и B.foo (<exprS>) являются полностью валидными и очевидно понятными – в первом случае вызывается версия foo из класса A и из B во втором. Обращение к свойству foo просто по имени является неоднозначным и при такой попытке - foo (<exprS>) – компилятор выдает сообщение об ошибке. Аналогичная ситуация происходит, если обращение идет из внешнего кода по отношению к классу, как в примере ниже

c: C

c.foo (<exprS>)

Компилятор не может однозначно определить версию foo и выдаёт соответствующее сообщение об ошибке.

Если рассмотреть общий случай как разрешается конфликт имен при помощи перекрытия и/или переопределения, то получается следующая схема:

Пусть есть два метода (подпрограммы, функции) с именем foo и сигнатурами <S1> и <S2> и телами <B1> и <B2> соответственно, которые наследуются в некоторый класс, то в случаях если

1. <S1> = <S2> - сигнатуры совпадают
    a. <B1> = <B2> - это один и тот же метод и он присутствует в единственном экземпляре

b. <B1> != <B2> - это два разных метода. Ситуация неоднозначности и разрешается она при попытке обращение, как было описано выше.

2. <S1> != <S2> - сигнатуры не совпадают – это два разных метода и при обращении к ним компилятор либо однозначно определяет версию исходя из статических типов передаваемых параметров или выдает сообщение о неоднозначности конкретного обращения.

a. Если оба метода происходят из одного источника, то необходимо разрешить неоднозначность, выбрав ту или другую версию для поддержки полиморфных присваиваний.

3. <S1> != <S2> - сигнатуры не совпадают и присутствует переопределение с сигнатурой <S3>, которая подобна обеим сигнатурам <S1> и <S2>. Неоднозначности нет, если только одна версия для foo.

Пусть есть два атрибута с именем attr и типами T1 и T2, которые наследуются в некоторый класс, то

1. T1 = T2 - это один и тот же атрибут и он присутствует в единственном экземпляре

2. T1 != T2 - это два разных атрибута. Ситуация неоднозначности и разрешается компилятором исходя из вида использования атрибута.

a. Если оба атрибута происходят из одного источника, то необходимо разрешить неоднозначность, выбрав ту или другую версию для поддержки полиморфных присваиваний.

3. Если присутствует переопределение с типом T3, который подобен обоим типам T1 и T2, то неоднозначности нет, так как есть одна версия для attr.

Таким образом, существует решение для конфликта имен при наличии перекрытий имен и переопределении версий при наследовании. Причем надо отметить, что оно существует как для методов, так и для атрибутов и принципиально схема не различается для них.

Если обобщить схему на произвольное количество свойств, то общая схема выглядит, как показано на рис. 14.

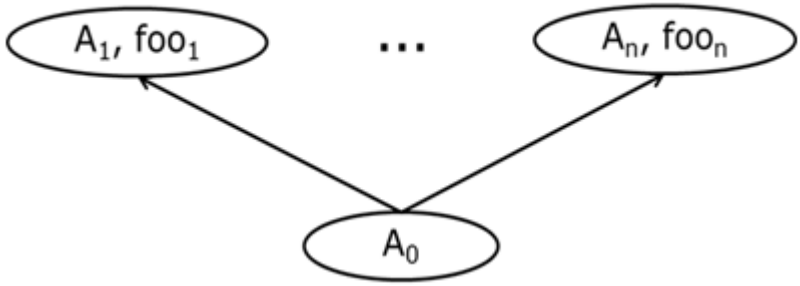

Рис. 14 Общая схема перекрытия при множественном наследовании

Есть класс A, который наследует n классов A_1 .. A_n и каждый из них содержит свойство foo. Тогда множество свойств foo в классе A имеет потенциально 4 составляющие

- Если в A0 для свойства foo из A1 применена операция абстрагирования (синтаксически она может выглядеть как A1.foo is abstract), то среди всех версий foo из классов A2 .. An должна быть версия fooi из класса Ai, которая подобна foo1. Другими словами версия foo1 становится абстрактной и заменятся версией fooi.
- Если в A0 свойство foo из A2 задается как переопределяющее (синтаксически это может выглядеть как override A2.foo), то эта версия переопределяет все версии из множества A3 .. An которым она подобна.
- Пусть свойства foo3 .. fook – это одно и тоже свойство. Тогда все они «сливаются» в одно свойство в классе A0.
- Остается множество свойств fook+1 .. foon, которое определяет различные перекрытые версии foo.

И, наконец, второй вариант: имеется переопределение в A_0.

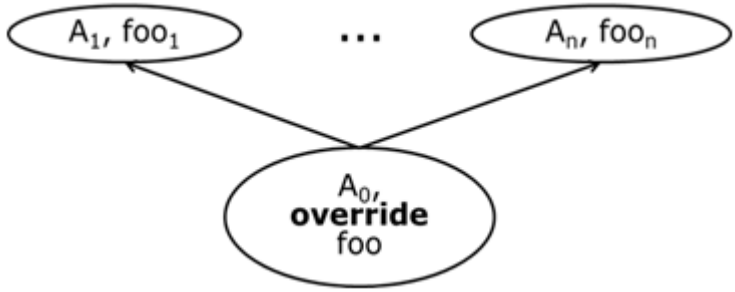

Рис. 15 Общая схема переопределения при множественном наследовании

Тогда опять часть версий может быть сделана абстрактными (как описано выше), а оставшиеся делятся на два множества (потенциально пустые).

- foo_1 .. foo_k – переопределены версией foo из A0.

- foo_{k+1} .. foo_n – наследованные и перекрытые версии foo, которые теперь являются частью A_0.

Есть еще один случай, который является ахиллесовой пятой ковариантного переопределения при наличии полиморфного присваивания – это так называемый cat calls. К сожалению, авторам неизвестен адекватный русский перевод термина, так что оставим англоязычный вариант. Суть проблемы заключается в нарушении статической типизации при следующей схеме наследования.

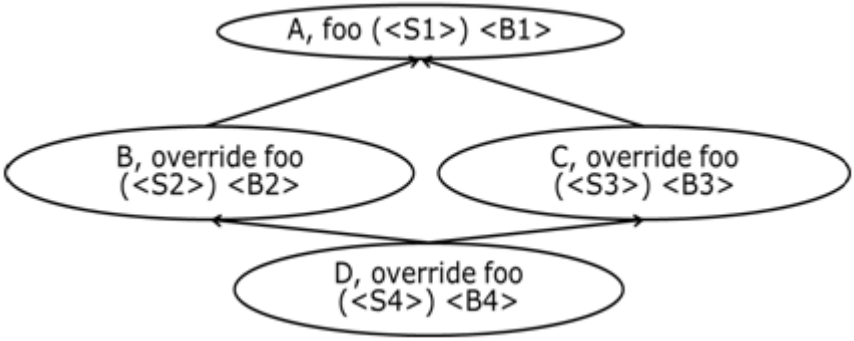

Рис. 16 Пример нарушения системы типизации при множественном наследовании

Данная схема наследования валидна, если сигнатуры <S1> и <S2> различны, и следующие сигнатуры попарно подобны: <S2>→<S1>, <S3>→<S1>, <S4>→<S2> & <S4>→<S3>, где знак → означает подобие. Рассмотрим вариант полиморфного присваивания

a: A **is new** D()

Иными словами, некоторая сущность a была описана как имеющая статический тип A, а ей при выполнении программы будет присвоен объект типа D. И тогда обращение вида

a.foo (<parameters>)

будет работать правильно если типы <parameters> подобны сигнатуре <S4>, и приводить к нарушению системы типов, если они подобны сигнатуре <S1>. Эта проблема не нова, и лобовое ее решение заключается в выполнении проверки правильности обращения в контексте всей программной системы, а

не одного класса (нечто схожее с концепцией LTO – только тут будет производиться консервативная проверка семантики при наличии информации о всей системе). Одно из возможных решений состоит в запрете полиморфного присваивания при ковариантном переопределении, но это несколько сужает возможности применения множественного наследования.

Последняя проблема, которую мы рассмотрим, заключается в возможном конфликте областей видимости при наследовании. Обратимся к рис. 17.

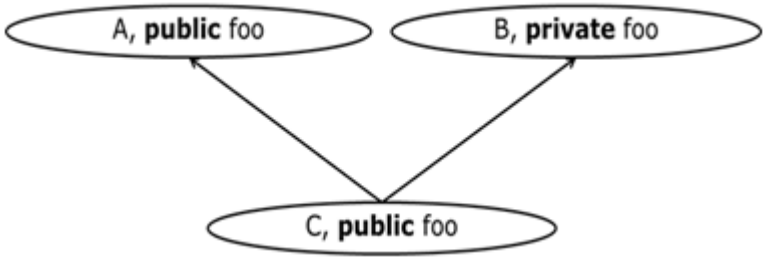

Рис. 17 Области видимости при множественном наследовании

Вне зависимости от того, каким способом (из тех что были описаны выше) мы добиваемся того, что в классе C у нас остается свойство foo в единственном экземпляре, область его видимости не может заужаться. Это можно выразить следующей схемой.

private … private => public или private

public … private = > public

4. Заключение

В статье представлены подходы к реализации концепции наследования в современных языках программирования. Предложена альтернативная концепция, которая ставит во главу угла особый вид множественного наследования с возможностью перегрузки методов и атрибутов, их переопределения при наследовании. Вводится понятие неоднозначности обращений к свойствам классов, которые либо однозначно разрешаются компилятором при проверке всех обращений, либо отвергаются им как неразрешимые.

Список литературы

[1]. International Standard: ISO/IEC 14882:2011(E) Information technology – Programming Languages – C++

[2]. J.Gosling, B.Joy, G.Steele, G.Bracha, A.Buckley, The Java Language Specification, 2015-02-13, https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

[3]. C# Language Specification, Version 5.0, Microsoft Corporation, https://www.microsoft.com/en-us/download/details.aspx?id=7029
[4]. Gutknecht J., Romanov V., Zueff E. The Zonnon Project: A .NET Language and Compiler Experiment, in V.Skala, P.Nienaltowski (Eds.) .NET Technologies 2005 Conference Proceedings, May 30 – June 1, 2005, University of West Bohemia, Plzen, Czech Republic, ISBN 80-86943-01-1.
[5]. International Standard: ISO/IEC 16262:2011(E) Information technology – Programming Languages, their environments and system software interfaces – ECMAScript language specification.
[6]. The Python Language Reference, https://docs.python.org/3.3/reference/.
[7]. Martin Odersky, Lex Spoon, and Bill Venners: Programming in Scala, Second Edition, Artima Press, 2010.
[8]. International Standard: ISO/IEC 8652:2012 Information technology – Programming Languages – Ada.
[9]. Bertrand Meyer: Object-Oriented Software Construction, Second Edition. Prentice Hall. ISBN 0-13-629155-4.
[10]. N.Wirth: The Programming Language Oberon, http://www.inf.ethz.ch/personal/wirth/Oberon/Oberon.Report.pdf
[11]. The Swift Programming Language Reference: https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/AboutTheLanguageReference.html.
[12]. The Kotlin Language Reference. http://kotlinlang.org/docs/reference/

The Concept of Inheritance in Modern Programming Languages

A. Kanatov <a.kanatov@samsung.com>
E. Zouev <e.zouev@samsung.com>
Samsung R&D Institute Russia, Dvintsev 12, 127018 Moscow, Russia.

Abstract. The paper gives a brief overview of existing approaches to inheritance implemented in some mainstream and experimental programming languages including Ada, Eiffel, C++, Java, Scala, Oberon, and Zonnon.Advantages and limitations of the approaches are analyzed and discussed.The paper claims that multiple inheritance model is one of the most important features that should be supported by any modern programming language because it reflects some fundamental relationships in the real world. However, only a few industrial languages do support the modelwherein such a support is far from being comfortable for users and leads to many troubles in development and maintenance. An alternative approach in the design of inheritance support is presented. The suggested approach keeps multiple inheritance as the general, powerfuland flexible mechanism for software design and reuse and is based on overloading and overriding with conflicts resolution at call sites based on conformance instead of full validity of the system inheritance graph. All typical problems in multiple inheritance including name clashes, name resolution,

signature conformance, covariance and contravariance are carefully consideredand discussed. The detailed explanation of how the problems are solved within the approach is presented. The paper describes a completely research project and is not supported by Samsung R&D Institute.

Keywords: multiple inheritance, overriding, name clashes, version conflict, origin and seed.

DOI: 10.15514/ISPRAS-2015-27(6)-12

For citation: Kanatov A., Zouev E. The Concept of Inheritance in Modern Programming Languages. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 169-188 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-12

References

[1]. International Standard: ISO/IEC 14882:2011(E) Information technology – Programming Languages – C++

[2]. J.Gosling, B.Joy, G.Steele, G.Bracha, A.Buckley, The Java Language Specification, 2015-02-13, https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

[3]. C# Language Specification, Version 5.0, Microsoft Corporation, https://www.microsoft.com/en-us/download/details.aspx?id=7029

[4]. Gutknecht J., Romanov V., Zueff E. The Zonnon Project: A .NET Language and Compiler Experiment, in V.Skala, P.Nienaltowski (Eds.) .NET Technologies 2005 Conference Proceedings, May 30 – June 1, 2005, University of West Bohemia, Plzen, Czech Republic, ISBN 80-86943-01-1.

[5]. International Standard: ISO/IEC 16262:2011(E) Information technology – Programming Languages, their environments and system software interfaces – ECMAScript language specification.

[6]. The Python Language Reference, https://docs.python.org/3.3/reference/.

[7]. Martin Odersky, Lex Spoon, and Bill Venners: Programming in Scala, Second Edition, Artima Press, 2010.

[8]. International Standard: ISO/IEC 8652:2012 Information technology – Programming Languages – Ada.

[9]. Bertrand Meyer: Object-Oriented Software Construction, Second Edition. Prentice Hall. ISBN 0-13-629155-4.

[10]. N.Wirth: The Programming Language Oberon, http://www.inf.ethz.ch/personal/wirth/Oberon/Oberon.Report.pdf

[11]. The Swift Programming Language Reference: https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/AboutTheLanguageReference.html.

[12]. The Kotlin Language Reference. http://kotlinlang.org/docs/reference/