**УДК 004.045**

**С.В. Логанов**

# ЕЩЕ РАЗ О ПРИНЦИПАХ ПРИМЕНЕНИЯ НАСЛЕДОВАНИЯ

Нижегородский государственный технический университет им. Р.Е. Алексеева

Посвящена решению проблемы правомерности и адекватности применения наследования в объектно-ориентированном программировании. Приводится концепция о целях применения наследования и ее отличия от реализации интерфейсов.

*Ключевые слова:* объектно-ориентированное программирование, наследование, множественное наследование, реализация интерфейсов, отношения “Is-A” и “Has-A”.

Нет сомнения, что наследование – это один из основополагающих принципов объектно-ориентированного программирования. В большинстве авторитетных изданий принцип наследования трактуется как способ повторного использования [1, 2, 3], но при этом часто делается множество оговорок о сложности самой концепции наследования. Именно из-за трактовки наследования как способа повторного использования в [4] вводится отрицательное правило наследования, которое отфильтровывает вполне очевидные случаи неприменимости наследования.

Но как только удовлетворяется отрицательное правило наследования, тут же возникает проблема выбора между отношениями «является» и «имеет». В [4] при выборе между этими отношениями рекомендуется использовать правила «изменений» и «полиморфизма», которые не носят всеобъемлющего характера. Правило «изменений» (опять же отрицательное) не рекомендует использовать наследование, если объект может изменять тип во время выполнения. И лишь правило «полиморфизма» рекомендует использование наследования, если «для сущностей возникает потребность присоединения к объектам различных типов». То есть наследование необходимо, когда возникает потребность объединения сущностей и их подтипов в единые списки, массивы и иные структуры, и другого механизма для этого просто не существует. Но получение универсального списка или иной структуры не является самоцелью, а предполагает обход этих структур с выполнением некоторых действий. Таким образом, появляется класс-клиент, которому требуется выполнение таких действий в обобщённом виде, реализуемых сущностями в универсальном списке.

Следовательно, если подойти к наследованию не как к способу повторного использования, а как к объектно-ориентированному механизму реализации вариации поведения, то все противоречия в трактовке наследования исчезают. В [5] также признается, что применение наследования просто для повторного использования, является не самым хорошим способом использования наследования. К этому необходимо добавить, что такое применение существенно ограничивает возможности расширения и модернизации получаемых решений.

Решение, используемое на рис. 1, имеет практически неограниченные возможности по модернизации и расширению поведения, так как легко преобразуется в структуру, показанную на рис. 2, с неограниченными возможностями вариации поведения как по линии класса A, так и B. Решение же, показанное на рис. 3, ограничивается жесткими рамками наследования, в котором модернизация и расширение поведения возможно только добавлением новых потомков, не имеющих возможности удаления методов родителя (что абсолютно правильно).

Применение наследования для многократного использования (для конструирования нового типа) предполагает отсутствие клиента у базового класса или, по крайней мере, отсутствие возможности попадания наследника на место своего родителя, что не гарантирует

ни один объектно-ориентированный язык, и, следовательно, не может быть рекомендовано как хорошая практика программирования.

Однако в [1] считается вполне нормальным использование наследования для конструирования нового типа, при котором возможно нарушение правила «Is-A» (отрицательного правила наследования), но при этом делается оговорка, что «дочерний класс не является специализированной версией родительского класса, так как у нас и в мыслях не будет подставлять представителей дочернего класса туда, где используются представители родительского класса».


**Рис. 1. Повторное использование на основе отношения «имеет»**

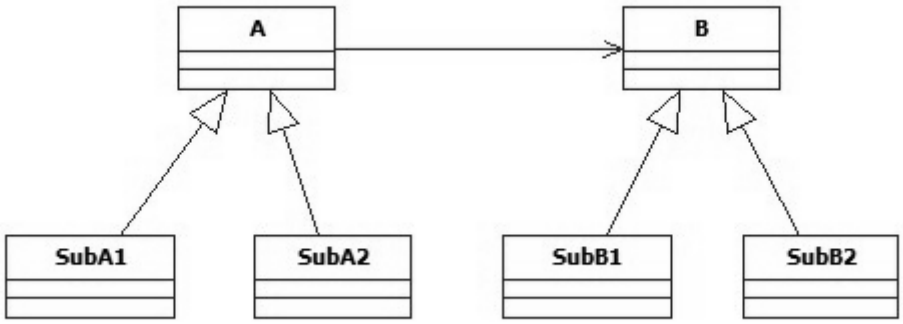


**Рис. 2. Реализация вариативного поведения между классами A и B**

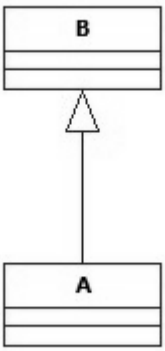


**Рис. 3. Повторное использование на основе отношения «является»**

Если у разработчика такого класса подобных мыслей и не возникает, то у программистов-клиентов этого класса они по определению будут, так как все они изучали принцип LSV (принцип подстановки Лисков). И сконструированный таким образом класс во многих случаях окажется неработоспособным.

Возможность повторного использования кода без внесения в него каких-либо изменений была основополагающим принципом развития программирования на протяжении многих лет, что привело к появлению объектно-ориентированного программирования. Однако классическим способом повторного использования является закрытое наследование, которое появилось в C++ и не предполагает возможность использования наследника вместо своего родителя. Возможность подстановки потомка вместо своего родителя, имеющаяся в открытом наследовании, дает принципиально иной способ построения кода, который предполагает вариативность поведения обобщенной сущности. Таким образом, и применение открытого наследования должно быть направлено на обеспечение вариативного поведения.

В желании использовать наследование в качестве повторного использования подкупает возможность воспользоваться функциями родителя, не написав ни единой дополнительной строчки кода. В то время как использование отношения предполагает написание некоторых функций, делегирующих полномочия соответствующему объекту. Однако, с помощью написания нескольких простых строк для доступа к используемому объекту реализуется инкапсуляция взаимодействия совокупности объектов, позволяющая безболезненно модернизировать их составную функциональность и тем самым обеспечивающая адаптированность кода к постоянно изменяющимся требованиям.

Кроме многократного использования, наследование применяют также для программирования отличий и замены типов [5].

Применение наследования для отличий позволяет запрограммировать только отличия между классом потомком и его родительским классом в рамках отношения «Is-A» (рис. 4). Программирование отличий – достаточно мощное средство. Небольшой объем кодирования и повышенная управляемость кода облегчают разработку программной системы. А поскольку в этом случае приходится писать меньше строчек кода, то уменьшается и количество добавляемых ошибок.

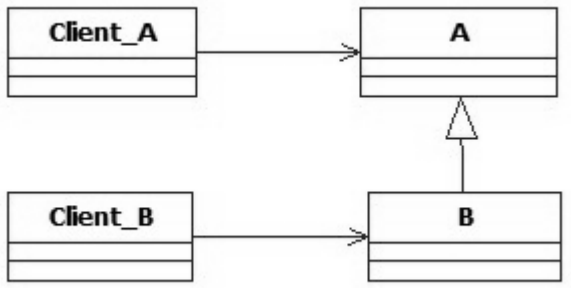


**Рис. 4. Наследование класса B для программирования отличий класса A**

При этом возможная замена родителя на его потомков не приводит к каким-либо последствиям, так как наследник лишь добавляет новые функции, а клиенты класса A продолжают использовать унаследованный код.

Возможность замены – одно из важных понятий в ООП. Поскольку классу-потомку можно посылать те же сообщения, что и классу родителя, то с классом потомком можно обращаться так, как с его родителем. Именно поэтому нельзя удалять поведение при создании класса-потомка, а можно лишь модернизировать его. Как правило возможность замены используется для сбора объектов базового класса и любых его подтипов в единую коллекцию и посылки им некоторого набора сообщений, которые, как правило, обрабатываются полиморфно (рис. 5).

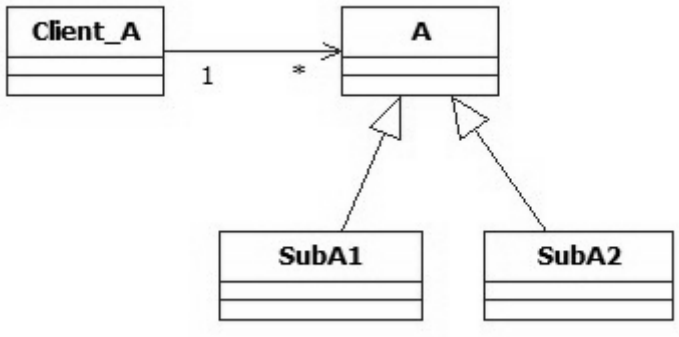


**Рис. 5. Наследование для замены типов**

Наследование – это обобщение, перевернутое с ног на голову. Умение человека делать обобщение – одна из величайших возможностей человеческого интеллекта, которая позволяет объединить и классифицировать сложное поведение с единых позиций. Отсюда множество примеров для применения наследования имеют биологическую природу. Однако обобщение и классификация выполняются с определенной целью. К. Линей разработал свою систему видов для классификации животных по степени их организации. А использование такой системы обобщения и классификации животных, например, для программы отображения на экране окажется совершенно неприемлемой.

Таким образом, особое значение для построения правильной системы обобщения играет ее цель, которая в ПО идентифицируется классом, использующим вершину иерархии и требующим от нее соответствующей вариативности поведения. Поэтому построение иерархии наследования классов без указания ее клиента приводит к непониманию того, зачем и по каким причинам строится такая иерархия.

Умение правильно использовать наследование заключается в умении адекватно идентифицировать обобщенную абстракцию и определить ее обобщенное поведение, которое может конкретизироваться с помощью ее наследников.

Кроме этого, проблемы классификации, рассмотренные в [4], при наличии класса-клиента вообще не имеют основания, так как нет необходимости притягивать к этому процессу какой-либо чужеродный критерий классификации. Поскольку класс является абстракцией, которая выделяет существенные свойства и отбрасывает несущественные, то наличие класса-клиента однозначно диктует соответствующий критерий классификации для ввода новых наследников в иерархию. Таким образом, класс вершина иерархии является моделью обобщенного представления совокупности объектов, обеспечивающего необходимое сложное поведение для класса-клиента. И согласно принципу единственности ответственности каждый класс этой иерархии должен быть неотъемлемой частью, направленной на поддержание данной модели.

Другими словами, наличие класса-клиента однозначно диктует единственный критерий классификации и нет необходимости сочетать противоречивые требования двух и более, казалось бы, возможных критериев, а следовательно, и не должно быть никаких причин для скрытия некоторых методов у классов наследников.

С другой стороны, добавление нового наследника должно реализовывать дополнительную вариацию поведения построенной обобщённой модели, что вполне согласуется с рекомендациями в [4] и однозначно исключает появление пустых классов наследников.

Замечательным примером создания обобщенной абстракции является проектирование визуального редактора документов в [7]. Очевидно, что документ - это организованное некоторым способом множество графических элементов: символов, линий, многоугольников и других геометрических фигур. Заслуга авторов заключается в том, что они представили эти элементы не в графическом виде, а в терминах физической структуры документа и таким образом построили модель предметной области.

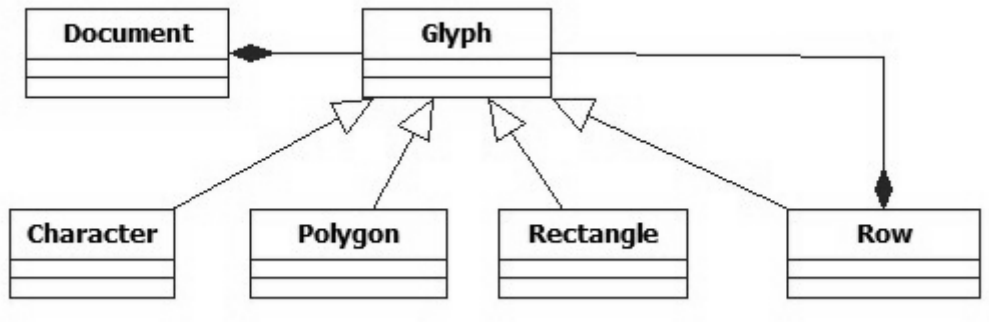


**Рис. 6. Структура графического документа**

Так появилось обобщенная абстракция в виде глифа, на основе которой строится вся структура документа (рис. 6) и которая обеспечивает неограниченную возможность расширения состава элементов визуального редактора.

Таким образом, документ состоит лишь из множества объектов одного типа, но может иметь сколь угодно сложную структуру, допускающую расширение как за счет введения новых примитивных типов, так и их объединения в более сложные структуры.

В отличие от наследования реализация интерфейса имеет принципиально иное назначение. Реализация интерфейса - это наделение объекта возможностью исполнить некоторую роль, не изменяя при этом своей первоначальной сущности. Например, получая водительское удостоверение, человек получает право стать участником дорожного движения (исполнить роль водителя) на своем автомобиле, но при этом он не становится его составной частью. С точки зрения разработки программного обеспечения исполнение классом некоторой роли позволяет участвовать его объектам в алгоритмах, реализованных в других классах.

Например, глифы графического редактора документа могут быть упорядочены по занимаемой площади (или по какому-либо другому признаку) с помощью реализации интерфейса ISortable, который требуется классу, непосредственно выполняющему сортировку. При этом глиф не изменяет своей первоначальной сущности – элемента графического документа. А реализация дополнительного интерфейса ISortable лишь позволяет экземплярам глифа стать участниками одного конкретного алгоритма сортировки или же множества алгоритмов, для которых достаточно возможностей, предоставляемых интерфейсом ISortable.

Дополнительные проблемы в проектировании объектно-ориентированных программ вызывает использование множественного наследования. Приведем определение множественного наследования из [3]: «Как видно из названия, множественное наследование позволяет тому или иному классу наследовать более чем от одного класса. … При этом существует много реальных примеров множественного наследования. Родители — хороший пример такого наследования. У каждого ребенка есть два родителя — таков порядок. Поэтому ясно, что вы можете проектировать классы с применением множественного наследования.»

На основании того, что в реальной жизни имеются примеры множественного наследования, делается вывод о возможности применения такого наследования и в программной системе. При этом автор никак не задумывается о кардинально иной сути наследования от двух родителей в реальности и в программировании. Если бы программное наследование использовалось для реального ребенка, то он рождался бы с двумя головами, четырьмя руками и четырьмя ногами. Таким образом реальная суть термина наследования ускользнула от автора и надо признать приведенный пример с двумя родителями крайне неудачным.

Если множественное наследование рассматривать для повторного использования, т.е. когда у базовых классов отсутствуют клиенты (рис. 7), то оно обладает теми же недостатками, что и для одиночного наследования, но умноженными многократно (по количеству линий наследования).

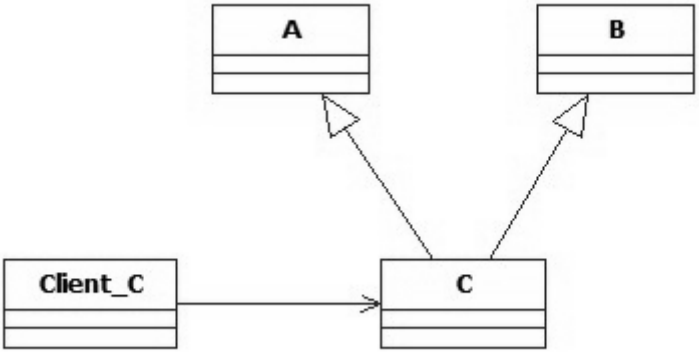


**Рис. 7. Множественное наследование для многократного использования**

Кроме того, наличие у классов A и B разнородных обязанностей (а иначе они были бы одним классом) приводит к тому, что класс C также обладает разнородными обязанностями и, следовательно, не соответствует принципу единственности ответственности со всеми вытекающими из этого негативными последствиями.

Если множественное наследование рассматривать как механизм реализации обязанности одной ветки наследования за счет другой (рис. 8), то такое использование наследования обедняет возможности поддержки и расширения функциональности всей исходной системы.

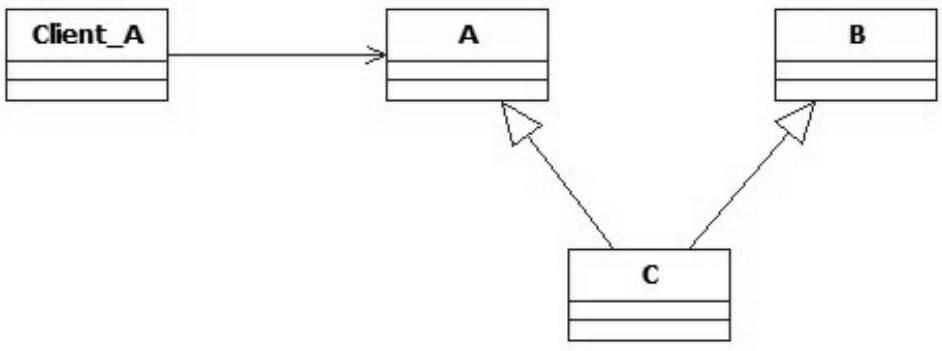


**Рис. 8. Множественное наследование для реализации обязанности одной ветви с помощью другой**

Об этом свидетельствует пример, приведенный в [4] для построения библиотеки Vision. Первый проект библиотеки Vision для платформенно-независимой графики столкнулся с общей проблемой учета зависимости от платформы. Первое решение использовало множественное наследование следующим образом: типичный класс, задающий, например, окна, имел двух родителей - одного, описывающего общие свойства, независящие от платформы, и другого, учитывающего специфику конкретной платформы (рис. 9).

Класс GENERAL_WINDOW и ему подобные, например GENERAL_BUTTON, являются отложенными: они выражают все, что может быть сказано о соответствующих графических объектах и применимых операциях без ссылки на особенности графической платформы. Такие классы, как PLATFORM_WINDOW, обеспечивают связь с графической платформой, например Windows, OS/2 Presentation-Manager или Unix Motif. Они дают доступ к механизмам, специфическим для конкретной платформы.

Такие классы, как WINDOW, комбинируют свойства родителей, реализуя отложенные компоненты GENERAL_WINDOW механизмами, обеспечиваемыми PLATFORM_WINDOW.

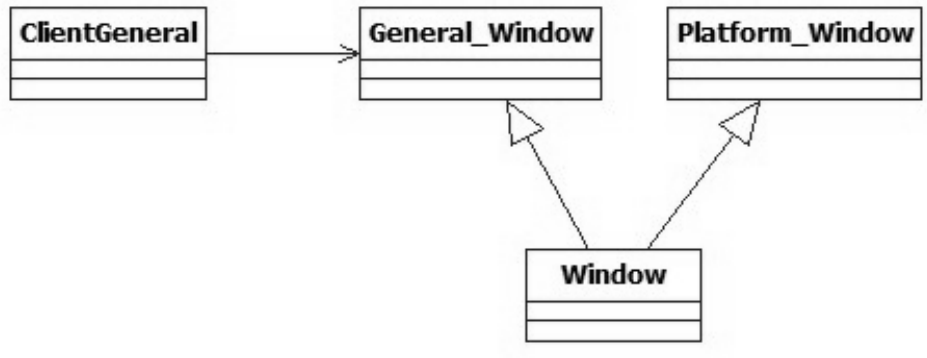


**Рис. 9. Первоначальное решение для библиотеки Vision**

Класс PLATFORM_WINDOW (как и другие подобные классы) должен присутствовать в нескольких вариантах - по одному на каждую платформу. Эти идентично именуемые классы могут храниться в различных каталогах, а специальный инструментарий при компиляции выбирает подходящий.

Это решение работает, но его недостаток в том, что понятие WINDOW становится тесно связанным с выбранной платформой. Иначе говоря, окно, став однажды окном Motif, всегда им и останется. Это не слишком печально, поскольку трудно вообразить, что однажды, достигнув почтенного возраста, окно Unix вдруг решит стать окном OS/2. Картина становится менее абсурдной при расширении определения платформы - при включении форматов Postscript или HTML. В этом случае графический объект может изменять представление, становясь то документом печати, то Web-документом.

Чтобы реализовать новое решение для расширенного толкования платформы, необходимо отказаться от наследования класса, представляющего конкретную платформу, и заменить его на клиентское отношение. Что и было сделано в следующей версии библиотеки Vision (рис. 10).

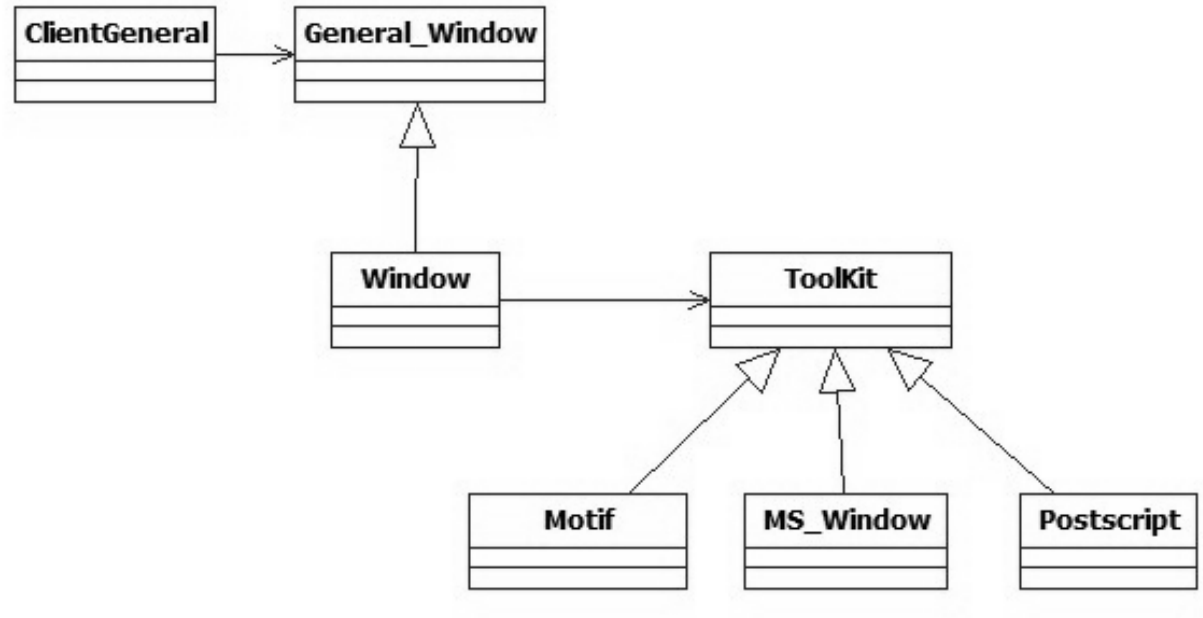


**Рис. 10. Новое решение для зависимости от платформы**

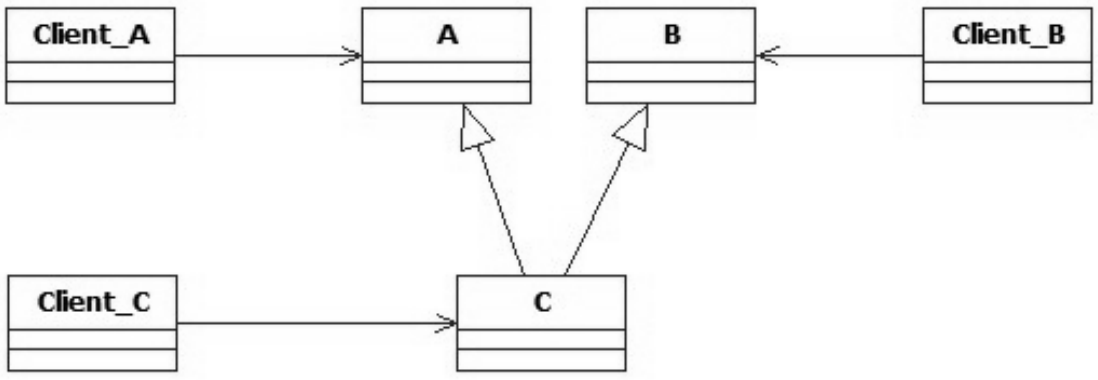


**Рис. 11. Применение множественного наследования для отличий**

Полученное решение не сложнее с точки зрения объема программирования, чем первоначально принятое решение. Однако, если взамен традиционного использования наследования для повторного использования было бы изначально принято клиентское отношение, то проведенная модернизация не потребовалась бы.

Применение множественного наследования для отличий (рис. 11) также не может считаться хорошей практикой, так как наследник нескольких классов содержит множество линий поведения, нарушая тем самым принцип единственности ответственности и шаблон GRASP высокого зацепления. Как следствие такого нарушения– полученный класс труден для понимания, дальнейшего сопровождения и поддержки.

Использование множественного наследования для полиморфного поведения (рис. 12) возможно, но также нарушает принцип единственности ответственности и шаблон высокого зацепления, так как родительские классы явно выполняют разнородные задачи. Поэтому такое применение множественного наследования не может считаться хорошей практикой. По крайней мере, необходимо разбить класс наследник на несколько классов, каждый из которых реализует собственную линию поведения, а необходимая информация доставляется с помощью клиентского отношения (рис. 13).

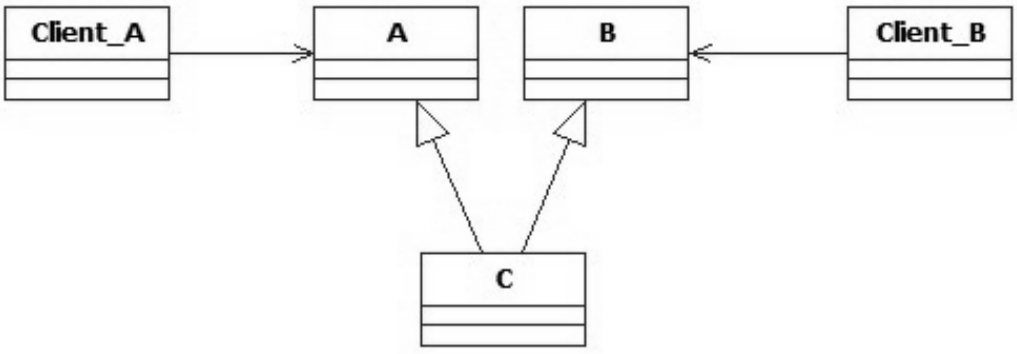


**Рис. 12. Использование множественного наследования для полиморфного поведения**

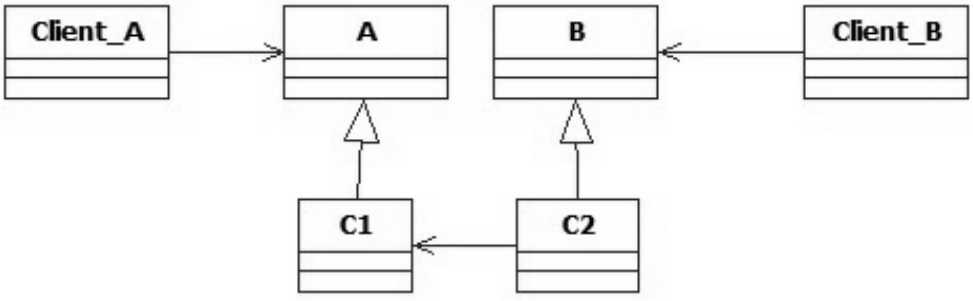


**Рис. 13. Разделение наследника на несколько классов**

При этом, отказавшись от множественного наследования, классы C1 и C2 будут соответствовать принципу единственности ответственности и шаблону высокого зацепления.

Приведенные рассуждения показывают, что применение множественного наследования не приводит к сколь-нибудь ощутимым преимуществам, а лишь затрудняют ясность понимания и расширения функциональности в процессе поддержки класса наследника. Поэтому в таких современных языках, как Java и C# разработчики отказались от концепции множественного наследования. Таким образом, разработчики современных объектно-

ориентированных языков неявно признали неэффективность, даже вредность, использования концепции множественного наследования, так как потомок множества классов обладает разнородными обязанностями.

Однако множество объектов реальной жизни могут использоваться в роли других объектов, не меняя при этом своей сущности. Например, человек может играть роль гражданина, врача, педагога, водителя, оставаясь человеком. Аналогичную концепцию в объектно-ориентированной программе можно реализовать с помощью использования интерфейсов.

Возможность объекта поучаствовать в нескольких алгоритмах, выполняемых другими объектами, или выступить в роли другого объекта достаточно легко осуществляется с помощью исполнения им требуемых интерфейсов. При этом данный объект не изменяет своей первоначальной сущности и не становится членом других иерархий. Такая возможность приводит к тому, что некоторые объекты могут реализовывать свои функции над любыми объектами, которые выполняют определенные условия, реализуя соответствующий интерфейс. Например, в современных системах нет смысла реализовывать алгоритм сортировки для работников фирмы, достаточно обеспечить объекты данного класса реализацией соответствующего интерфейса. Таким образом существенно расширяются возможности повторного использования.

Возможность любого класса появиться во множестве ролей существенно расширяет его внешний интерфейс, что не очень хорошо для клиентов этого класса и не имеющих к этим интерфейсам никакого отношения. Поэтому в C# существует возможность явной реализации интерфейса, которая позволяет избежать данного недостатка.

## Выводы

Наследование является механизмом реализации обобщенной абстракции, сочетающим совокупность программирования отличий, замены типов с полиморфным поведением, а также использования элементов классификации, обеспечивающих различные виды поведения этой обобщенной абстракции. Наследование не должно применяться просто для повторного использования, так как для этого имеется клиентское отношение, которое не препятствует превращению классов клиентов в обобщенные абстракции с возможностью безболезненного расширения их поведения.

В связи с этим, важное значение приобретает указание клиента обобщённой абстракции, так как именно класс-клиент определяет цели и задачи, реализуемые этой абстракцией, и диктует критерии классификации ее наследников.

Дополнительные действия с объектами, членами иерархии обобщенной абстракции, должны выполняться на основе реализации соответствующих интерфейсов без изменения их первоначальной сущности.

## Библиографический список

1. **Бадд, Т.** Объектно-ориентированное программирование в действии: [пер. с англ.] / Т. Бадд. – 3-е изд. – СПб.: Питер, 1997. – 304 с.
2. **Дейтел, П.** Как программировать на Visual C# 2012: [пер. с англ.] / П. Дейтел, Х. Дейтел. – 5-е изд. – СПб.: Питер, 2014. — 864 с.
3. **Вайсфельд, М.** Объектно-ориентированное мышление: [пер. с англ.] / М.Вайсфельд. – СПб.: Питер, 2014. – 304 с.
4. **Мейер, Б.** Объектно-ориентированное конструирование программных систем: [пер. с англ.] / Б. Мейер. – М.: Русская редакция, 2005. – 768 с.
5. **Синтес, А.** Освой самостоятельно объектно-ориентированное программирование за 21 день: [пер. с англ.] / А. Синтес. – М.: Вильямс, 2002. – 672 с.

6. **Ларман, К.** Применение UML2.0 и шаблонов проектирования. Введение в объектно-ориентированный анализ и проектирование: учеб. пособие: [пер. с англ.] / К. Ларман. – М.: Вильямс, 2008. – 736 с.
7. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма [и др.]. – СПб.: Питер-ДМК, 2001. – 366 с.

*Дата поступления*
*в редакцию 10.08.2017*

**S.V. Loganov**

## AGAIN ON THE PRINCIPLES OF USING INHERITANCE

Nizhny Novgorod state technical university n.a. R.E. Alekseev

**Purpose:** The solution to the problem of the propriety and adequacy of the use of inheritance in object-oriented programming. Is the concept of the purpose of use of inheritance and the difference of inheritance from implementation of interfaces.
**Design/ methodology/ approach**: The use of inheritance for reuse is accompanied by many side effects.
**Findings:** Inheritance should be used for the variable behavior of generalized abstractions. Special value gets the client a generalized abstraction that defines the boundaries of the required functionality of a generalized abstraction.
**Research limitations/implications:** Inheritance is not recommended to use for a reuse, if we are not talking about supporting legacy code.
**Originality/ value:** The use of inheritance for variable behavior of the generalized abstraction eliminates the possibility of errors. Applying the relation of using allows you to safely convert entities to the generalized abstraction.

*Key words*: object-oriented programming, inheritance, multiple inheritance, interfaces implementation, relations “Is-A” and “Has-A”.