УДК 004.045

С.В. Логанов

К ВОПРОСУ О МНОГООБРАЗИИ ФОРМ НАСЛЕДОВАНИЯ

Нижегородский государственный технический университет им. Р.Е. Алексеева

Анализируются возможные формы наследования и правомерность обоснования их применения в статически типизированных объектно-ориентированных языках. Приводится концепция целей построения иерархий классов наследования. Установлено, что правомерное использование наследования заключается в построении правильных абстракций для обобщения или специализации некоторого поведения, востребованного соответствующими клиентами. Применение наследования просто для повторного использования приводит к построению противоречивых иерархий, которые затруднены или невозможны для повторного использования. Предложена классификация форм наследования, обеспечивающих его адекватное применение в статически типизированных языках Java и C#.

Ключевые слова: объектно-ориентированное программирование, наследование, иерархия классов, повторное использование, принцип LSP, обобщение/специализация.

Описание наследования в объектно-ориентированном программировании в современных авторитетных работах [1,2] включает детализацию всех возможных форм его применения. Приводятся следующие формы применения наследования [1]:

- порождение подклассов для специализации;
- порождение подкласса для спецификации;
- порождение подкласса для расширения;
- порождение подкласса с целью конструирования;
- порождение подкласса для обобщения;
- порождение подкласса для ограничения;
- порождение подкласса для варьирования;
- порождение подкласса для комбинирования.

При порождении подклассов для специализации имеет значение то, что порожденный класс является специализированной версией родительского класса и «удовлетворяет спецификациям родителя во всех существенных моментах» [1]. Таким образом, для этой формы полностью выполняется принцип подстановки Liskov (LSP) и, по мнению автора, «вместе со следующей категорией (наследование для спецификации) специализация является наиболее идеальной формой наследования, к которой должна стремиться хорошая программа» [1]. При порождении подкласса для спецификации родительский класс описывает поведение, отложенное для реализации дочерним классом. Родительский класс для этой формы является абстрактным или даже интерфейсом, а дочерний реализует заданное поведение и, следовательно, также должен удовлетворять принципу LSP. Порождение подкласса для расширения характеризуется тем, что дочерний класс добавляет новые функциональные возможности к родительскому классу, но не меняет наследуемое поведение. В данном случае у дочернего класса появляются собственные клиенты, которые пользуются его расширенным поведением, и он также удовлетворяет принципу LSP, поскольку не изменяет наследуемое поведение.

Порождение подкласса с целью конструирования предполагает, что дочерний класс использует методы, предоставляемые родительским классом, но не является подтипом родительского класса, т.е., реализация методов нарушает принцип LSP. В данном случае даже сам автор делает оговорку, что «дочерний класс не является более специализированной формой родительского класса, так как у нас и в мыслях не будет подставлять представителей дочернего класса туда, где используются представители родительского класса» [1] и что «в языках со

статическими типами данных косо смотрят на порождение подклассов для конструирования» [1]. Такое отступление говорит о том, что обеспечение отсутствия подстановки лишь джентельменское соглашение, следование которому не в силах обеспечить ни один объектно-ориентированный язык, если в нем отсутствует закрытое наследование. Применение закрытого наследования решает данную проблему, но такое наследование более логично отнести к специализированному виду клиентского отношения, при котором времена жизней обоих объектов данных классов совпадают и не считается классическим наследованием.

При порождении подкласса для обобщения подкласс расширяет родительский класс для создания объекта более общего типа. «Например, в систему графического отображения, в которой был определен класс окон Window для черно-белого фона необходимо добавить тип цветных графических окон ColoredWindow. Цвет фона будет отличаться от белого за счет добавления нового поля, содержащего цвет. Необходимо также переопределить наследуемую процедуру изображения окна, в которой происходит фоновая заливка» [1]. Тем не менее, в данном случае класс ColoredWindow должен следовать принципу LSP. В противном случае, если, например, ColoredWindow забудет восстановить цвет фоновой кисти, то все последующие объекты класса Window изменят свой цвет. Такое новое поведение явно будет считаться ошибочным. Поэтому такое поведение подкласса следует считать специализацией, поскольку «нарисовать цветное окно» является специальным случаем просто «нарисовать окно». Кроме этого, признается: «Как правило, следует избегать порождения подкласса для обобщения, пользуясь перевернутой иерархией типов и порождением для специализации. Однако это не всегда возможно» [1]. Единственной причиной является отсутствие исходного кода классов, а, следовательно, расположение этого кода в другом компоненте. Наследование класса из другого компонента приводит к сильной зависимости этих компонентов. А если компоненты различны, то и причины изменений этих компонентов различны. Лучшим решением в данном случае является изоляция этих компонентов с помощью принципа инверсии зависимостей (DIP) [3]. Таким образом, нельзя считать хорошей практикой наследование класса из другого компонента.

Порождение подкласса для ограничения происходит, когда возможности подкласса более ограничены, чем в родительском классе. «Так же, как и при обобщении, порождение для ограничения чаще всего возникает, когда программист строит класс на основе существующей иерархии, которая не должна или не может быть изменена» [1]. Тогда возникает закономерный вопрос: зачем вообще нарушать стройность существующей иерархии добавлением такого нового класса?

Попадание экземпляров такого нового класса в качестве абстракции, существующей иерархии, для использования соответствующим клиентом приведет к появлению неожиданных сюрпризов в его поведении. Помимо этого, и сам автор признает, что данную форму наследования следует по возможности избегать. Такая возможность всегда существует в виде использования клиентского отношения и отсутствия необходимости наличия отношения заменяемости. «Порождение подкласса для варьирования применяется, когда два класса имеют сходную реализацию, но не имеют никакой видимой иерархической связи между абстрактными понятиями, ими представляемыми. Например, программный код для управления мышкой может быть почти идентичным тому, что требуется для управления графическим планшетом. Теоретически, однако, нет никаких причин, для того чтобы класс Mouse, управляющий манипулятором «мышь», был подклассом класса Tablet, контролирующего графический планшет, или наоборот. В данном случае в качестве родителя произвольно выбирается один из них, при этом другой наследует общую программную часть кода и переопределяет код, зависящий от устройства». Данная проблема является типовой проблемой шаблона GRASP «Защита от изменений» [4] и решается с помощью введения необходимого интерфейса, обеспечивающего изоляцию клиента от конкретных деталей реализации. Применение наследования в данной ситуации обеспечивает слишком сильную взаимозависимость данных классов, имеющую соот-

ветствующие негативные последствия. Появление необходимости управления новым устройством, вероятнее всего, приведет к пересмотру данного решения в целом. Таким образом, не следует использовать наследование просто на том основании, что некоторые классы имеют похожий код.

Порождение подкласса для комбинирования применяется, когда дочерний класс наследует черты более чем одного родительского класса. Это множественное наследование, которое объединяет линии поведения родительских классов и тем самым нарушает принцип единственности обязанности (SRP), поскольку обязанности родительских классов принципиально различны, иначе они не были бы различными классами. Поэтому множественное наследование было исключено из языков, которые считаются типовыми объектно-ориентированными языками и заменено реализацией интерфейсов, которая позволяет объекту выступить в роли другого объекта, не изменяя своей первоначальной сущности. Однако и в этом случае дочерний класс должен соблюдать принцип LSP, чтобы не преподносить сюрпризов клиентам родительских классов. В [2] представлены те же самые формы наследования с той лишь разницей, что опущено наследование для варьирования, а наследование для спецификации названо наследованием для реализации. Однако, при описании наследования для конструирования приводится следующий пример. «Путем наследования класса «Экономичное Окно» конструируется класс Самолет, в котором родительский метод изменить положение () превращается в метод летать (), а от методов свернуть () и развернуть () вообще отказываются». Такое использование наследование вообще нельзя признать более или менее разумным. Это аналогично тому чтобы в реальной жизни пытаться сконструировать мобильный телефон на основе молотка. По словам автора, «конечно, бывают ситуации, когда наследование применимо, даже если тест «является» не пройден» [2]. В [5] однозначно показывается, что не следует использовать наследование, если тест «is-a» не пройден. Желание использовать наследование как механизм повторного использования как можно чаще приводит к тому, что появились такие экзотические формы наследования как для обобщения, ограничения, конструирования и варьирования, которые в объектно-ориентированных языках со статической типизацией приносят лишь проблемы при сопровождении, поддержке и модернизации кода.

Более подробная классификация форм наследования (причем правильных форм наследования, соответствующих отношению «is-a») приведена в [5], где различаются следующие три общие категории (рис. 1):

- наследование модели, отражающее отношения «is-a» между абстракциями, характерными для модели предметной области;
- программное наследование, выражающее отношения между объектами программной системы, не имеющих очевидных двойников во внешней модели;
- наследование вариаций – специальный случай, относящийся как к моделям, так и программному наследованию, служащий для описания класса через его отличия от других классов.

При моделировании внешней системы часто возникает ситуация, когда категория внешних объектов естественно разделяется на непересекающиеся подкатегории, которые являются подтипами этой общей категории (наследование подтипов). При этом утверждается, что родительский класс является абстрактным, «поскольку он описывает не полностью специфицированное множество объектов», т.е., является обобщением, используемым при классификации объектов. В то же время наследник этого родителя может быть, как эффективным, так и отложенным (абстрактным).

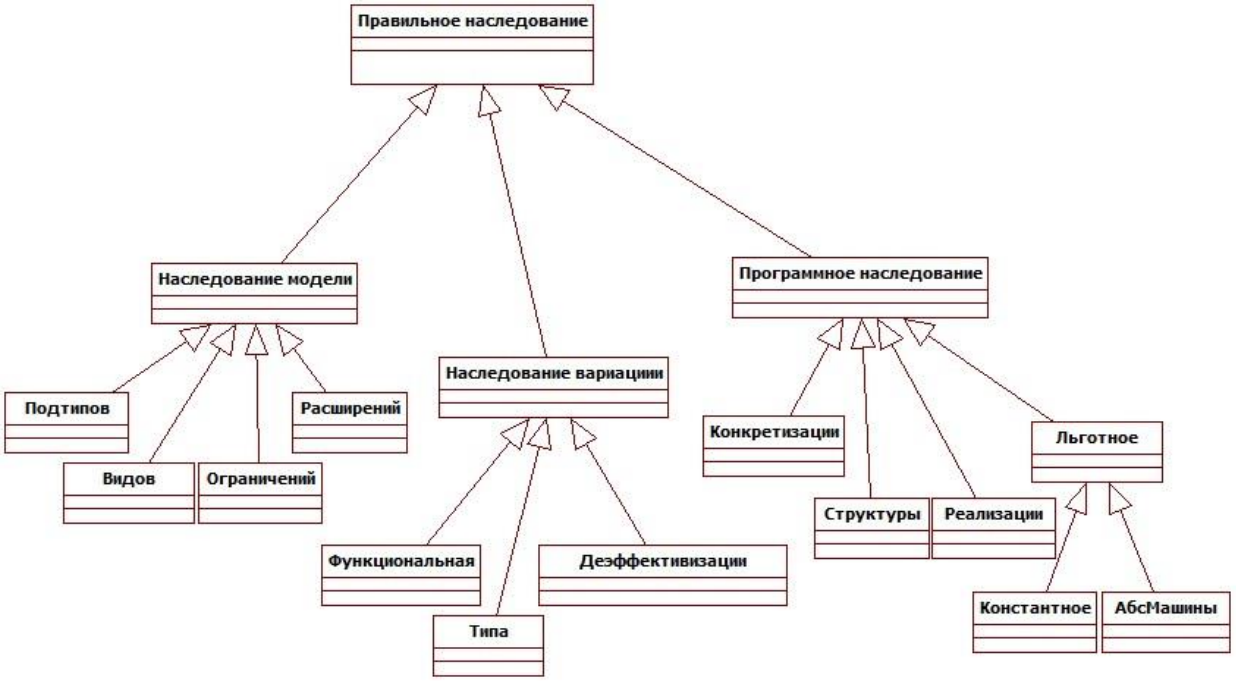

Рис. 1. Классификация форм наследования Б. Мейера

«Наследование подтипов является формой наследования ближайшей к иерархической таксономии в ботанике, зоологии и других естественных науках» [5]. Например, млекопитающие являются наследниками позвоночных и т.д. При этом следует оговорится, что цели моделирования в программной системе должны соответствовать целям классификации в естественных науках иначе ее использование теряет смысл. Также примером данного наследования может служить определение банковского счета, который может быть текущим счетом, сберегательным счетом и т.д. Наследование с ограничением применимо, если экземпляры класса потомка являются экземплярами родительского класса, удовлетворяющими некоторому дополнительному ограничению, выраженному, как часть инварианта потомка, не включенного в инвариант родителя. В качестве типичного примера наследования с ограничением приводится квадрат как наследник прямоугольника. Ограничением является утверждение, что сторона 1 = сторона 2, включаемое в инвариант класса квадрат. Однако, математика – это очень абстрактная область знаний, утверждающая лишь, то что в отдельных случаях у прямоугольников стороны могут оказаться равны. А по словам автора, «другим типичным примером является таксомания, в котором простое булево свойство, такое как пол персоны (или свойство с несколькими фиксированными значениями, такое как цвет светофора), используется как критерий наследования, хотя нет важных вариантов компонентов, зависящих от свойства» [5]. При моделировании объектов реального мира стоит лишь уменьшить погрешность измерений и квадрат тут же перестает быть квадратом. Кроме этого, если, например, потребуется масштабирование моделируемых объектов с различными коэффициентами масштаба по осям, то едва ли пользователи захотят сохранения этого свойства. Таким образом, в каждом индивидуальном случае решение будет достаточно специфическим и опираться на потребности конкретных пользователей. Однако примером данной формы наследования может служить ограниченный банковский счет, являющийся потомком банковского счета, с которого запрещается снятие в день суммы, превышающей заданный порог. При этом клиенты родительского банковского счета должны быть готовы к тому, что со счета может сниматься запрошенная сумма не полностью. Тем не менее, данный пример, как и пример с моделированием геометрии объектов, являются частными случаями наследования подтипов.

Наследование с расширением применимо, когда потомок вводит компоненты, не представленные в родителе и неприменимые к его прямым экземплярам. При этом утверждается,

что класс родителя должен быть эффективным. Рассмотрим пример с абстрактным классом Shape, который декларирует операции перемещения и масштабирования. От данного класса можно унаследовать класс Rectangle, имеющий диагональ и способный вычислять ее длину. Данный пример является примером наследования с расширением, в котором родитель является абстрактным. Таким образом, опровергается то, что родитель должен быть всегда эффективным. Далее описывается следующий парадокс наследования. «Присутствие обоих вариантов – расширения и сужения (ограничения) – является одним из парадоксов наследования. Расширение применяется к компонентам, в то время как ограничение (понимаемое как специализация) применяется к экземплярам. Проблема в том, что добавляемые компоненты обычно включают атрибуты. Так что при наивной интерпретации типа (заданного классом) как множества его экземпляров отношение между классом и наследником (рассматриваемых как множества) «быть подмножеством» становится полностью ошибочным» [5]. Далее рассматривается пример, в котором экземпляры родительского класса образуют одноэлементное множество, записываемое как <n>, где n целое. А экземпляры дочерних классов – как пару, содержащую целое и вещественное, записываемое как <n, f>. На этом основании утверждается, что множество пар дочерних экземпляров не является подмножеством одноэлементного множества родительских экземпляров. А верно обратное: отношение «быть подмножеством» имеет место в обратном направлении. Однако это – чисто математическая трактовка данных множеств. Проблема заключается в том, данный пример рассматривается в отсутствие клиента родительского класса, для которого множество парных экземпляров может быть представлено как множество одноэлементных экземпляров. При этом экземпляры как одноэлементного множества, так и двухэлементного множества различны для клиента, а, следовательно, множество дочерних экземпляров является подмножеством родительского типа. Таким образом, не существует никакого парадокса наследования. Далее и сам автор решает данную проблему, применив для этого другие математические абстракции. Тем не менее, наследование с расширением также следует отнести к частному случаю наследования подтипов.

Наследование вида используется там, где единый критерий классификации кажется ограничительным. В этом случае предлагается использовать «приемы множественного и особенно дублирующего наследования». Далее приводится класс EMPLOYEE в системе управления персоналом, для которого существуют два различных критерия классификации служащих по типу контракта (временные или постоянные работники) и по типу исполняемой работы (инженерная, административная, управленческая). Поэтому для соблюдения «идей использования наследования для классификации, следует ввести промежуточный уровень (два вида служащих), описывающий конкурирующие критерии классификации» [5]. «При наследовании подтипов предполагается, что экземпляры наследников принадлежат непересекающимся подмножествам множества, заданного родителем. При наследовании видов: различные наследники некоторого класса представляют не непересекающиеся подмножества его экземпляров, а различные способы классификации экземпляров родителя» [5]. Наличие двух критериев классификации в наследовании видов говорит о том, что абстракция EMPLOYEE используется двумя различными способами, а, следовательно, содержит две линии поведения (обязанности), что противоречит принципу единственности ответственности со всеми вытекающими негативными последствиями. Поэтому рекомендовать данную форму наследования как хорошую практику программирования вряд ли целесообразно. Кроме того, сам автор не рекомендует применять данную форму наследования, особенно новичкам. «Прежде всего, должно быть ясно, что, подобно дублируемому наследованию, наследование видов не является механизмом для новичков. Правило осмотрительности, введенное для дублируемого наследования, справедливо и здесь: если ваш опыт разработки ОО-проектов измеряется несколькими месяцами, избегайте наследования видов» [5].

Таким образом, если рассмотреть все достоинства и недостатки данного наследования, а также возможности альтернативных решений, вероятнее всего, даже опытный разработчик откажется от наследования видов. Соответственно, акцентирование внимания на решении

конкретных задач разработчика намного важнее, чем построение стройной иерархии ради самой иерархии и принесет более весомую выгоду особенно для долговременной эксплуатации и поддержке программного кода. Наследование вариаций применяется, если класс потомок переопределяет некоторые компоненты класса родителя. «Класс потомок при этом не должен вводить никаких новых компонентов за исключением тех, что непосредственно необходимы переопределяемым компонентам, что обеспечивает отличие данной формы наследования от наследования с расширением» [5]. Наследование вариаций функций переопределяет только тела компонентов, а при наследовании вариаций типа все переопределения являются переопределениями сигнатур. Наследование вариаций применимо, когда существующий класс родителя задает некоторую абстракцию, полезную саму по себе, но обнаруживается необходимость представления подобной, хотя и не идентичной, абстракции, имеющей те же компоненты, но с отличиями в сигнатуре или реализации.

При этом автор настаивает, чтобы оба класса были либо эффективными, либо отложенными. «Такое наследование не рассматривает эффективизацию компонентов, когда речь идет о переходе от абстрактной формы к конкретной» [5]. Эффективизацию компонентов автор отнес чисто к программной форме наследования – к наследованию с овеществлением. Однако и при моделировании реальных объектов такая эффективизация может понадобиться. Это возможно, например, при моделировании транспортных средств, имеющих функцию перемещения, которая, вероятнее всего, будет отложенной, поскольку наземные, воздушные и водные транспортные средства перемещаются совершенно по-разному. Таким образом, наличие отложенного метода перемещения и отсутствии необходимости более детальной классификации транспортных средств приводит к тому, что класс транспортных средств будет абстрактным (отложенным), а его потомки – эффективными. Наследование вариаций функции также следует отнести к наследованию подтипов, т.е., когда имеется некоторое множество объектов, обладающих более специфическим поведением. А наследование вариаций типа следует отнести к наследованию с расширением, так как новые компоненты с измененной сигнатурой у родительского класса недоступны, и у дочернего класса должны быть собственные клиенты, которые имеют доступ к этим компонентам, иначе такое наследование было бы бессмысленным. К данной группе наследования принадлежит и категория деэффективизации, в которой некоторые эффективные компоненты становятся отложенными. По утверждению автора, такая форма наследования может быть законной в двух случаях.

В первом случае при множественном наследовании сливаются компоненты, наследуемые от двух различных родителей. «Если один из них отложенный, а другой эффективный, то слияние произойдет автоматически при условии совпадения имен (возможно после переименования), эффективная версия будет определять реализацию. Но если обе версии эффективны, следует провести потерю эффективизации одной версии, отдавая предпочтение другой версии» [5]. Однако в современных статически типизированных языках автоматическое слияние произойдет, если отложенный класс является интерфейсом и соответственно наследование в данном случае лучше назвать эффективизацией интерфесов. Если же оба родительских класса будут эффективными, такое наследование становится принципиально невозможным.

Во втором случае: «Хотя абстракция соответствует потребностям, но повторно используемый класс слишком конкретен для наших целей. Отмена эффективизации позволит удалить нежеланную реализацию» [5]. Даже сам автор оговаривает, что перед использованием этого решения следует рассмотреть альтернативу – реорганизовать иерархию наследования, сделав более конкретный класс наследником нового отложенного класса. «По понятным причинам, это не всегда возможно, например, из-за отсутствия доступа к исходному коду» [5]. Но если родительский класс принадлежит другому компоненту, и, тем более, если у него отсутствует исходный код, использование такой сильной связи как наследование приведет лишь к дополнительным проблемам сопровождения и поддержки кода. Взаимодействие различных физических компонентов лучше разделить с помощью принципа DIP, обеспечивающего их взаимо-

заменяемость. Кроме этого, даже автор настороженно относится к данной форме наследования: «Отмена эффективизации не является общим приемом и не должна им быть. Основная идея этого способа противоречит общему направлению, так как обычно ожидается конкретизация потомка своего более абстрактного родителя» [5].

Наследование с овеществлением (конкретизации) применимо, если родительский класс задает структуру данных общего вида, а дочерний – представляет ее частичную или полную реализацию. Родительский класс в данной форме наследования является отложенным, а класс потомок может быть, как эффективным, так и отложенным. Приведенный ранее пример с транспортными средствами также применим и для данной формы наследования. Кроме этого, при необходимости дальнейшей специализации наземные, воздушные и водные транспортные средства уже могут быть также отложенными. Данную форму наследования также следует отнести к наследованию подтипов, так как принципиальной разницы для построения иерархии программных объектов или объектов предметной области не существует.

Структурное наследование применяется, когда родительский отложенный класс представляет общее структурное свойство, а потомок, который может быть отложенным или эффективным, представляет некоторый тип объектов, обладающих этим свойством. «Например, родительский класс может быть классом COMPARABLE, представляющим объекты с заданным отношением полного порядка. Класс, которому необходимо отношение порядка, становится наследником класса COMPARABLE» [5]. Разница между овеществлением и структурным наследованием заключается в том, что при овеществлении дочерний класс представляет собой то же понятие, что и родительский, отличаясь большей степенью реализации. А при структурном наследовании дочерний класс представляет собственную абстракцию, для которой родительский задает лишь один из аспектов, реализация которого позволяет ему, например, поучаствовать в алгоритмах упорядочивания, выполняемых некоторыми классами. Такая ситуация может существовать не только между программными объектами, но и между объектами реальной предметной области. Например, служащий может оказаться в роли члена профсоюза или в роли спортсмена, не переставая при этом быть собственно служащим в возможной иерархии наследования служащих. Таким образом, данная форма наследования в статически типизированных языках относится к наследованию интерфейса.

Наследованием реализации является «брак по расчету», основанный на множественном наследовании, где один из родителей обеспечивает спецификацию, а другой – предоставляет реализацию. В качестве примера приводится стек, основанный на массиве ARRAYED_STACK. При этом наследование от ARRAY выполняется закрытым. Это вынуждает клиентов класса ARRAYED_STACK использовать соответствующие экземпляры только через компоненты стека. Для статически типизируемых языков закрытое наследование в принципе невозможно, а, следовательно, невозможно гарантировать неиспользуемость стека как массива. Преимущество по сравнению с клиентским отношением, которое дает данная форма наследования – это возможность использовать функции родительского класса без использования имени переменной. Это является сомнительным преимуществом, если принять во внимание, что за его использование отрезается возможность расширения гибкости клиентского класса за счет применения принципа открытости/закрытости (OCP). Это и было продемонстрированно автором приведением примера с техникой описателей (handle). Наследование возможностей (льготное) является схемой, в которой родитель представляет коллекцию полезных компонентов, предназначенных для использования его потомками. Оно применяется, если родительский класс существует единственно в целях обеспечения множества логически связанных компонентов, дающих преимущества его потомкам.

Первой формой наследования возможностей является наследование констант, при котором компоненты родительского класса все являются константами или однократными функциями, описывающими разделяемые объекты. В качестве примера приводится класс ASCII, который является хранилищем множества константных атрибутов, описывающих свойства

множества ASCII и унаследованный от него класс лексического анализатора TOKENIZER, ответственный за идентификацию лексем входного текста. «Лексемами текста, написанного на некотором языке программирования, являются целые, идентификаторы, символы и так далее и TOKENIZER, необходим доступ к кодам символов для их классификации на цифры, буквы и т. д. Такой класс воспользуется льготами и наследует эти коды от ASCII» [5]. Однако в данном случае класс ASCII не реализует правильную абстракцию и не имеет функций по определению принадлежности символа к цифрам, буквам, управляющим символам и т.д. Приведение класса ASCII к такой абстракции и использование клиентского отношения позволяет развивать гибкость новой абстракции с помощью принципа OCP и добавления новых функций, непосредственно полезных лексическому анализатору. Кроме этого, при переходе на юникодные символы использование наследования констант приведет к необходимости полного переписывания кода класса TOKENIZER и не способствует повторному использованию кода.

Второй формой наследования возможностей является наследование абстрактной машины, в котором компоненты родительского класса являются подпрограммами, рассматриваемыми в качестве операций абстрактной машины. Иными словами, родительский класс представляет абстракцию выполнения некоторого алгоритма над объектами дочернего класса, которые реализуют соответствующие абстрактные функции. Преимуществом данной формы является возможность переименования функций родителя, которое позволяет дать им более благозвучное название, соответствующее назначению потомка. Поскольку переименование функций родителя в языках Java и C# невозможно, теряется и данное преимущество. Поэтому в данном случае более уместно определение контракта (интерфейса) для соответствующих абстрактных функций, реализацию которого должны обеспечить объекты, участвующие в выполнении необходимого алгоритма. Построение объектно-ориентированной программной системы – это моделирование как самой предметной области, так и программных задач, возникающих по ее визуальному представлению, сохранению и т.п. Модели для решения задач предметной области концептуально не отличаются от моделей решения программных задач, но они должны быть различны, так как различны сами задачи и тенденции их временного развития. Невозможно построить универсальную модель, пригодную на все случаи жизни, поэтому человечество давно научилось пользоваться принципом «разделяй и властвуй» с помощью построения упрощенных различных моделей для решения различных задач.

Таким образом, для таких статически типизированных языков, как Java и C#, правомерное использование наследования заключается в безусловном соблюдении правила «is-a» и применении следующих форм наследования (рис. 2).

Наследование подтипа подразделяется на наследование вариаций, ограничений и расширений. К наследованию вариаций относятся те случаи, когда потомки родительского класса переопределяют его поведение. При этом класс родитель реализует некоторую абстракцию и существует клиент, который использует поведение данной абстракции. Количество уровней потомков и отложенность родительской абстракции зависят от степени ее обобщения. К данной форме наследования относятся наследование транспортных средств, реализующих функции перемещения или фигур, реализующих функции рисования.

Рис. 2. Правомерные формы наследования для статически типизированных языков

При наследовании ограничений потомки класса родителя вводят дополнительные ограничения при реализации функций родителя. При этом потомки не могут и не должны скрывать сами эти функции, но результат их работы может быть частично положительным или отрицательным. К такому поведению должен быть готов клиент базового класса и корректно обрабатывать соответствующие ситуации, т.е., обеспечивается принцип LSP. К данной форме наследования относится, например, потомок банковского счета с ограничением снятия дневной суммы, который при выполнении операции снятия может снять как всю запрошенную сумму, так и часть этой суммы вплоть до нулевой. В крайнем случае, при наследовании ограничений класс потомок может генерировать исключение, которое должно быть специфическим видом исключения родительской абстракции. При этом клиент родительской абстракции должен знать и уметь обрабатывать такие исключения. Поскольку в настоящее время механизм исключений достаточно тяжеловесен, данная практика не может быть рекомендована к широкому применению. При наследовании расширений потомок родительского класса предоставляет дополнительные возможности, которые отсутствуют у класса родителя. При этом у такого потомка обязан существовать собственный клиент, которому необходимы дополнительные функциональные возможности. В противном случае увеличение функциональной возможностей потомков бесполезно, так как их невозможно использовать.

Все формы наследования подтипа не являются взаимоисключающими, т.е., потомки могут одновременно иметь вариации поведения, вводить ограничения или же реализовывать дополнительные возможности. При наследовании роли класс выполняет определенный контракт (реализует интерфейс) для обеспечения возможности появления его объектов в роли других объектов, над которыми выполняются определенные действия или которые сами способны выполнить некоторый набор действий. При этом класс не изменяет своей первоначальной сущности и не становится участником другой иерархии. К данной форме наследования следует отнести, например, банковский счет, реализующий интерфейс IComporable, позволяющий ему поучаствовать в алгоритмах сортировки, если таковая необходима, или класс банк, содержащий множество своих счетов и реализующий интерфейс IEnumerator, и, тем самым, выполняющий перечисление своих счетов. При наследовании реализации класс потомок выполняет реализацию интерфейса, объявленного в другом компоненте с целью обеспечения его независимости. Примером данного вида наследования является реализация принципа DIP.

Наследование является самым сильным видом зависимости между классами. Поэтому его применение для повторного использования является необоснованным, поскольку повышает совокупную зависимость. Таким образом, правомерное использование наследования в статически типизированных языках Java и C# заключается в построении для некоторого клиента хорошо определенной абстракции, обладающей гибкостью поведения и возможностью его расширения, а также реализацией контрактов, обеспечивающей разделение алгоритмов от участвующих в них объектов или независимость компонентов.

Кроме того, наличие одного клиента у корневого класса построенной иерархии наследования позволяет иметь четкие критерии классификации классов, входящих в иерархию. Появление дополнительных клиентов приводит к тому, что корневая абстракция должна сочетать две или более линий поведения, требующихся соответствующим клиентам и появлению дополнительных критериев классификации, формирующих иерархию наследования. Наличие нескольких линий поведения противоречит принципу SRP, приводит к более частым изменениям кода и ухудшает возможности его повторного использования, к которому так стремятся ярые поклонники повсеместного использования наследования. Наличие дополнительных критериев классификации, которые часто являются противоречивыми, приводит лишь к нарушению стройности иерархии наследования и дополнительным требованиям к языку программирования в виде возможности потомков сокрытия методов своих родителей. Например, если имеются два клиента, одному из которых необходимы транспортные средства для перемещения грузов из одной точки в другую, а второму – расчет налогов на различные транспортные средства, нет необходимости для этого строить единую иерархию наследования, так как критерии их классификации принципиально различны.

Если у корневой абстракции отсутствует клиент, проектирование иерархии наследования выполняется с целью повторного использования, требующего максимального учета всех возможных вариаций и попытки построения универсальной структуры классов для решения множества задач. При этом, чем более сложную и масштабную систему требуется построить, тем более призрачными являются надежды построить такую универсальную структуру. Таким образом, если у корневой абстракции отсутствует клиент (для объектов не возникает потребности присоединения к объектам различных типов или объединения их в единый список для выполнения некоторых задач (нарушение правила полиморфизма)), то, в первую очередь, необходимо рассмотреть возможность клиентского отношения. И лишь при наличии веских причин (а не возможности при написании кода опускать имя переменной) в ущербности такого отношения следует применять наследование.

Выводы

Адекватное использование наследования заключается в построении правильных абстракций для обобщения или специализации некоторого поведения, востребованного соответствующими клиентами. Применение наследования просто для повторного использования приводит к построению противоречивых иерархий, которые затруднены или невозможны для повторного использования. Помимо этого, наследование необходимо для реализации контрактов, обеспечивающих отделение алгоритмов от участвующих в них объектов или независимость компонентов.

Предложена классификация форм наследования, обеспечивающих его свободное применение в таких статически типизированных языках, как Java и C#.

Библиографический список

1. **Бадд, Т.** Объектно-ориентированное программирование в действии / Т. Бадд. – СПб.: Питер, 1997. – 464 с.
2. **Орлов, С.А.** Теория и практика языков программирования: учебник для вузов / С.А. Орлов. – СПб.: Питер, 2013. – 688 с.
3. **Мартин, Р.** Чистая архитектура. Искусство разработки программного обеспечения / Р. Мартин. – СПб.: Питер, 2018. – 352 с.
4. **Ларман, К.** Применение UML2.0 и шаблонов проектирования. Введение в объектно-ориентированный анализ и проектирование: учеб. пособие / К. Ларман. – М.: Вильямс, 2008. – 736 с.
5. **Мейер, Б.** Объектно-ориентированное конструирование программных систем / Б. Мейер. – М.: Русская редакция, 2005. – 768 с.

Дата поступления
в редакцию: 01.06.2019

S.V. Loganov

ON THE VARIETY OF INHERITANCE CATEGORIES

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

Purpose: The article is dedicated to discussing possible categories of inheritance, their adequate and unreasonable use in statically typed object-oriented languages. The concept of the purpose of construction inheritance class hierarchies is given.
Design/ methodology/ approach: The use of inheritance for reuse leads to the construction of conflicting hierarchies that are very difficult to reuse.
Findings: The valid use of inheritance is to build the abstractions with extendable behavior require by the relevant clients.
Research limitations/implications: Statically typed object-oriented languages.
Originality/ value: Inheritance is not recommended for reuse because it is a very strong coupling between classes.

Key words: object-oriented programming, inheritance, class hierarchy, reusability, LSP principle, generalization/concretization.