МЕХАНИЗМЪТ НА НАСЛЕДЯВАНЕ В УЧЕБНИЯ КУРС ПО ОБЕКТНО ОРИЕНТИРАНО ПРОГРАМИРАНЕ

Ивайло Дончев

1. Значение на наследяването за обектно ориентирания подход

Наличието на механизъм на наследяване отличава обектно ориентираните езици от обектните. Наследяването се сочи от повечето автори като най-фундаменталната концепция в обектно ориентираното програмиране (ООП) и едновременно с това една от най-проблемните в процеса на обучение [10, с. 113] [7, с. 101] [6]. За да подчертаем важността и за обектно ориентирания подход ще цитираме дефиницията за ООП на Grady Booch:

> **Деф. 1.** "*ООП е метод на имплементиране, при който програмите са организирани като сбор от сътрудничещи си обекти, всеки от които е инстанция на даден клас, а класовете са обединени от наследствени взаимоотношения в йерархия от класове.*" [5, с. 35].

В тази дефиниция откриваме 3 важни части:

(1) Използват се обекти, вместо алгоритми като градивни блокове на програмата.

(2) Всеки обект е инстанция на клас.

(3) Класовете са свързани помежду си с наследствени отношения.

Ако кой да е от тези елементи липсва, програмата не може да се нарече обектно ориентирана. В частност, програмиране без наследяване Booch нарича „програмиране с абстрактни типове данни".

От гледна точка на реализацията на наследяването в езиците за програмиране:

> **Деф. 2.** „Един език (или техника) е обектно ориентиран тогава и само тогава, когато поддържа директно:

(1) ***Абстракция*** *– осигурява форми за дефиниране на класове и обекти.*

(2) ***Наследяване*** *– осигурява възможност за изграждане на нови абстракции от вече създадени.*

(3) ***Динамичен полиморфизъм*** *– осигурява някаква форма на динамично свързване.*" [23].

Наличието на механизъм за наследяване е критерият по който даден език или техника се определят като обектно ориентирани.

Важната роля, определена на наследяването в ООП е съвсем обоснована. Тази конструкция позволява проектиране на много по-високо ниво, в сравнение с начина на проектиране при императивните езици. Програмите, проектирани с интелигентно използване на наследяването се отличават с изчистеност, гъвкавост и лесна поддръжка. Kolling определя наследяването като "швейцарското ножче на езиковите конструкции – гъвкав инструмент, полезен за множество напълно несвързани задачи" [10, с. 114].

2. Същност на наследяването

Поддръжката на наследяване означава възможност за представяне на отношения от вида "генерализация/спецификация". Това е разглеждане на наследяването от гледна точка на класифицирането – основна дейност в ООП, която го разграничава от процедурното програмиране. Taivalsaari [25] и Zhu [27] отделят внимание и на другата гледна точка за наследяването – повторната използваемост на кода, тъй като от нейното неразбиране могат да възникнат сериозни грешки в етапа на проектирането. От тази гледна точка класовете се третират като „обекти, върху които се прилага наследяването" [27]. Така интерес представлява имплементирания в класовете код, а не тяхната класификация. Това може да доведе до трудна за разбиране и модифициране програма. Затова препоръката за правилното разбиране на наследяването е да се изучават първо неговите най-важни аспекти – тези свързани с изграждането на класовете (класифицирането) и отношенията между тях.

Това в никакъв случай не означава пренебрегване на повторната изпплзваемост на код. Eckel [4, с. 291] отбелязва важно преимущество на наследяването, свързано с този аспект – възможността за създаване на нов код без да се допуска внасяне на грешки във вече написания. Това значително улеснява процеса на дебъгването, защото евентуалните грешки се ограничават в рамките на новонаписания код.

Формално наследяването може да се дефинира така [25]:

Деф. 3. $R = P \oplus \Delta R$,

където R е новодефинираният клас; с P са обозначени характеристиките, наследени от съществуващия клас; ΔR са добавените нови характеристики, по които R се различава от P; а с $\oplus$ е обозначена операцията, чрез която по някакъв начин са комбинирани P и. В резултат на тази комбинация R ще притежава всички характеристики на P, с изключение евентуално на тези, предефинирани или заличени в ΔR. Затова R може не винаги да е напълно съвместим с P. В литературата P се нарича **непосредствен предшественик**, или **суперклас**, или **базов клас** на R. Съответно, R се нарича **непосредствен наследник**, или **субклас**, или **производен клас** на P. В обектно ориентираното проектиране е прието да се използват термините субклас и суперклас. Stroustrup [22, с. 303] определя тази терминология като объркваща за тези, които забелязват, че данните в обект на субклас не са подмножество на данните на обект на суперкласа, а точно обратното. Субкласът е "по-обширен" от своя суперклас от гледна точка на това, че съдържа повече данни и предоставя повече функции. Затова в C++ се използват термините производен и базов клас (В [30] и [28] наред с термина базов се използва и "основен" клас – буквален превод от английски). В тази статия термините базов клас, основен клас, суперклас, както и производен клас, субклас ще се използват като синоними.

Релацията наследяване е транзитивна, така че един суперклас може да бъде субклас за друг. Това предполага, че един суперклас съдържа характеристиките на всички свои предшественици, а не само на непосредствените. За илюстриране на йерархичните отношения е допустимо и използването на означения като **прародител** и **потомък**.

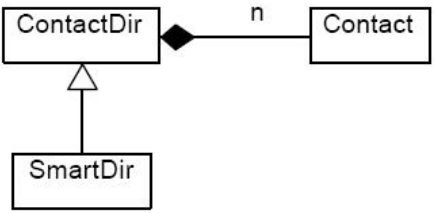
3. Дидактически аспекти

В повечето учебници по програмиране, например [15] [24] [3] [12] [20] [30] са разгледани всички особености на наследяването от гледна точка на реализацията на механизма в конкретния програмен език, но рядко откриваме препоръки за неговото използване – кога е необходимо да се използва наследяване и кога друга релация; как да се изгражда йерархията от класове; в кои случаи е удачно да се използва множествено наследяване, виртуално наследяване? Не липсват и книги, в които са коментирани особеностите на проектирането на наследствената йерархия и са дадени полезни препоръки към начинаещите програмисти. Такива са [19] [11] [17] [26] [18] [8]. Общото за всички е големият обем от съдържанието, отделен за изучаване на наследяването и свързаните с

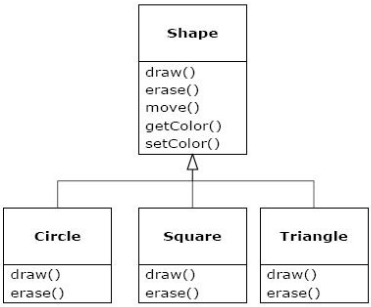
него механизми (виртуални методи, абстрактни класове, динамичен полиморфизъм).

За онагледяване на йерархиите от класове във всички проучени книги се използва графично представяне чрез различни диаграми, като всеки автор сам решава каква нотация да използва. Затова срещаме твърде големи различия, които могат да предизвикат объркване у неопитния читател. За пример ще посочим изобразяването на релацията наследяване. В повечето учебни материали то се изобразява със стрелка, насочена от производния към базовия клас. Но има учебници, например [30], където е избрана точно обратната посока — от базовия към производния клас. Освен това, тъй като наследяването не е единствената релация възниква проблемът как да се изобразяват другите отношения между класовете (например асоциация, композиция, агрегация). Да се използват едни и същи стрелки не е добро решение.

В [19][13][4][8] е възприето използването на UML нотация. Всеки клас е представен с правоъгълник, разделен на 3 части. Най-отгоре е името на класа, членовете-данни, които имат някакво отношение към модела са в средната част и членовете-функции, чрез които се осъществява комуникацията с останалите са в най-долната част. Често само името на класа и `public` функциите се изобразяват на UML диаграмите (липсва средната част). Ако интерес представляват само имената на класовете, долната част също може да се пропусне. Наследяването между два класа се изобразява със стрелка в посока от производния към базовия клас. Линия с ромб в единия край описва композиция. Броят на съдържащите се обекти се изписва (например *n*). Примерни диаграми са показани на фиг. 1 и фиг. 2. На фиг. 1 класовете са представени само чрез имената си, а на фиг. 2 са посочени и методите на класовете, тъй като два от тях – `draw()` и `erase()` са предефинирани в производните класове.



Фиг. 1. UML диаграма, изобразяваща композиция и наследяване



Фиг. 2. UML диаграма, изобразяваща наследяване с предефиниране на методи

Изборът на UML за онагледяване на отношенията между класовете е удачен и от гледна точка на междупредметните връзки. Студентите изучават анализ и проектиране в отделен курс, където основното средство е именно UML.

От езикова гледна точка за правилното и ефективно използване на механизма на наследяване е важно в курса по ООП с подходящи примери да се изяснят:

1. Възможностите за използване на производния клас вместо неговия базов клас.

2. Правата на достъп на членовете на производния клас до членовете на базовия.

3. Обсегът на действие на всеки клас в йерархията.

4. Извикването и редът на изпълнение на конструктори на производен и базов клас.

5. Копиране и присвоявания между обекти на производен и базов клас.

6. Препокриване методи в производни и базови класове.

7. Конструкции, които не се наследяват (конструктори и деструктори, приятелски функции, статични членове данни и функции, операция за присвояване).

8. Възможност за съвместно използване на параметризация и наследяване.

Важна особеност, на която в учебниците не се акцентира е **съвместимостта между базовия и производния клас**. Тъй като при наследяването всички методи на базовия клас са достъпни в производния, всяко

съобщение, което може да се изпрати до обект на базовия клас може също така да се изпрати и до обект на производния. Производният клас може да се разглежда като подтип на своя базов клас и затова може да се използва навсякъде, където е допустимо използването на базовия клас. Обратното не е съвсем коректно – базовият клас не може безопасно да замества производния. Конкретно за C++, обект на производен клас може да се третира като обект на своя базов клас, ако се манипулира чрез указател или псевдоним.

При изучаването на наследяването трябва да се изясни значението на ключовата дума `protected`. Членовете на производния клас нямат специални права на достъп до `private` компонентите на своя базов клас. Това може да изглежда странно на някои студенти, но е напълно оправдано, тъй като в противен случай капсулирането на класа ще загуби своя смисъл. Ако такъв достъп все пак е необходим, трябва да се използват `protected`, вместо `public` декларации в базовия клас. При проектирането на клас трябва да се реши дали той ще се използва за в бъдеще чрез наследяване. Ако това е така, е удачно данните му да се декларират като `protected`, вместо `private`.

Добра препоръка относно декларирането на методите откриваме в [26]: "Функциите в базовия клас, които ще се предефинират от производните класове да се декларират като виртуални.". Множество препоръки, относно правилното моделиране и ефективната реализация на наследяването могат да се прочетат в [18], но това по-скоро е предмет на специализиран курс по ООП за напреднали.

Базовият курс по ООП трябва да е съобразен с ограничения брой часове. Като основен проблем на обучението по програмиране в [1] се посочва, че "студентите не са способни да абсорбират целия нужен материал за кратък период от време. Доброто владеене на езика изисква време". За усвояване на тънкостите по използването на композицията и наследяването препоръчвам много самостоятелни упражнения на студентите, както и подходящ примерен код към всяка точка от лекциите, с препратки към свързания с него учебен материал и допълнителна методическа литература.

Наред с преимуществата от използване на наследяването ще спомена и някои слабости:

1. Могат да се получат доста големи в дълбочина и ширина йерархии, които са трудни за разбиране и навигация без подходящо помощно средство (Class Browser).

2. Може да спадне производителността на програмата, особено при използване на множествено наследяване и динамично свързване.

3. Без динамично свързване приложението на наследяването е силно ограничено, както и динамичното свързване е безсмислено без наследяване.

4. Без механизъм на множествено наследяване може да се наложи изкуствено създаване на базови класове и неестествено обвързване между класовете, което в още по-голяма степен усложнява йерархията.

4. Класифициране на видовете наследяване

В литературата по програмиране наследяването най-често се разделя на две категории, в зависимост от броя на базовите класове, които може да има един производен клас:

- **единично (просто)** – всеки клас може да има само един базов
- **множествено** – няма ограничения в броя на класовете, от които един производен клас може да наследява.

В повечето езици за ООП е реализиран директно само механизъм на просто наследяване. Това се дължи както на сложността на реализацията, така и на някои проблеми, свързани с множественото наследяване, които са обсъдени по-долу. Езици, в които е реализирано множествено наследяване са Eiffel, C++, Python, Perl. При простото наследяване се получава йерархия от класове във вид на дърво, докато при множественото – ориентиран ацикличен граф.

В популярните на българския пазар учебници [30] [28] разграничаването на единично и множествено наследяване е единствената класификация. И в двата учебника се говори за "управление на механизма на наследяване", съответно чрез "атрибута на основния клас в декларацията на производния" [28], или "атрибута за област на базовия клас в декларацията на производния" [30], като се визират правата на достъп до членовете на наследената част.

С цел да се акцентира именно върху необходимостта и практическата полза от управление на механизма на наследяване е полезно разграничаването на `public`, `private` и `protected` наследяване, придружено от конкретни препоръки за използването на всеки вид при моделирането на реални ситуации:

- **`public` наследяване**, известно още като наследяване на типа (type inheritance) [17] или тип/подтип наследяване [16]. Производният клас е подтип на базовия клас. Предефинират се всички специфични за типа едноименни

методи от базовия клас. По принцип се използва за представяне на "is-a" взаимоотношения, т.е. производният клас е специализация на по-общия си базов клас.

• **`private` наследяване**, известно още като implementation наследяване [22] [29]. Производният клас не поддържа директно интерфейса на своя базов клас. По-скоро има отношение към повторната използваемост на кода от базовия клас за собствения `public` интерфейс на производния. Затова Kolling [10] нарича този тип наследяване "наследяване за повторно използване на код". Може да се разглежда като синтактичен вариант на композицията ("has-a" взаимоотношение). В [11, с. 155] това отношение между класовете е наречено "is implemented with". Съществено за програмиста ограничение е, че при `private` наследяването, както и при композицията не може указател или псевдоним на производния клас автоматично да се конвертира до указател (псевдоним) на базовия клас (виж примера по-долу).

• **`protected` наследяване**. При него всички `public` членове от базовия клас стават `protected` членове на производния. Това означава, че до тях може да имат достъп само евентуалните наследници на класа, но не и тези извън йерархията. `Protected` наследяването позволява производните на производните класове да „знаят" за наследственото отношение. Предимството е, че подкласовете на `protected` наследен клас могат да използват релацията с `protected` базовия клас. Цената за това е, че `protected` наследен клас не може да променя отношението без потенциална опасност да прекъсне по-нататъшното наследяване в йерархията.

В [9] `private` и `protected` наследяването са обединени в една категория **nonpublic наследяване**, поради факта, че когато производният клас не наследява `public` от своя базов, "is-a" отношението между тях не съществува. Ето един **пример** за това:

```
class Mem_Manager {/*..*/};
class List: private Mem_Manager {/*..*/};
void OS_Register( Mem_Manager& mm);
int main()
{
   List li;
   OS_Register( li );  //Грешка при компилиране!
                       //невъзможно преобразуването на типа
   return 0;
}
```

Класът `List` има `private` базов клас `Mem_Manager`, който се грижи за управлението на паметта, затова е използвано `private` наследяване, което блокира неправилното му използване. Същият ефект може да се постигне чрез композиция на двата класа.

"Преобладаващият модел на наследяване в реалните C++ програми е този на `public` наследяване от единствен базов клас. По правило можем да очакваме мнозинството от нашите проекти също да попадат в този модел." [17, с. 880]. Следователно за уводен курс по ООП от най-голямо значение е усвояването на `public` наследяването. Затова в някои книги подробно се разглежда само то. Lippman [16] определя останалите видове, както и множественото и виртуално наследяване като сложни, за напреднали, елементи на проектирането, които той разглежда в отделна книга. Разбира се, в края на курса по ООП студентите трябва да са придобили умения да управляват ефективно механизма на наследяване като използват най-подходящите за конкретния модел техники, а не само единично `public` наследяване.

Taivalsaari [25] разграничава две допълнителни форми на наследяване:

- **наследяване от завършени имплементации**: това е наследяване от класове, които не са абстрактни. Позволява повторна използваемост и усъвършенстване на съществуващи абстракции (класове);
- **наследяване от частични имплементации**: наследяване от абстрактни класове. Служи като структуриращо средство, позволяващо изграждане на програмата от абстрактни спецификации към по-конкретни представяния.

5. Особености на множественото наследяване

Деф. 4. „В случаите, когато производният клас наследява няколко основни класа, се казва, че има множествено наследяване." [28, с. 168].

Тази кратка дефиниция ни води до един от най-дискутираните въпроси, свързани с езиците за ООП – нужно ли е те да поддържат механизъм за множествено наследяване? Противниците му твърдят, че:

1. Така езикът става ненужно сложен.

2. Всеки модел, който използва множествено наследяване може да се направи и с единично.

3. Усложнява се написването на компилатор за такъв език.

Привържениците на този механизъм признават единствено третия аргумент. Множественото наследяване е просто възможност на езика.

Всеки, който чувства, че може да се справи и без него не е задължен да го използва. Повишеното ниво на сложност не е добър аргумент, тъй като същата критика може да се отправи и към другите възможности на езика като шаблони, предефиниране на операции, обработка на изключения, и т.н. Ще цитираме мнението на Booch по въпроса:

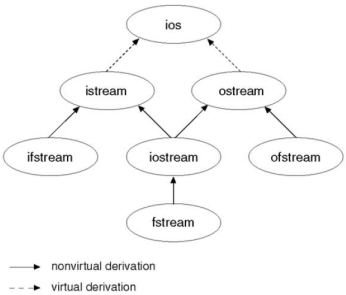
„Изхождайки от нашия опит, можем да сравним множественото наследяване с парашут: Не ви е необходим винаги, но когато ви потрябва сте наистина щастливи да го имате под ръка." [5, с. 122].

Основно предимство на множественото наследяване е възможността моделът да е по-близък до реалността, която представя. Единичното наследяване е твърде рестриктивно при моделирането на реални ситуации. То притиска програмиста да избира измежду няколко еднакво подходящи базови класа, а след това да дублира кода от тези класове, които не могат да бъдат наследени.

За да се убедят студентите в полезността на множественото наследяване, е необходимо този модел да се илюстрира с подходящи примери от тяхното ежедневие – работата с компютъра. Например, факс-модемът обединява функциите на две устройства – факс и модем. По същия начин класът `FaxModem` се дефинира като производен както на класа Fax, така и на `Modem`. Този модел е по-добър и по-естествен от съответния модел с единично наследяване:

```
class Fax {/*……..*};
class Modem {/*……*/};
class FaxModem : public Fax, public Modem {/*…*/};
```

Подобни примери могат да се дадат с други устройства, например All-In-One, обединяващи функциите на принтер, скенер и копир. Добър е и примерът, използван в [17, с. 884] с `iostream` библиотеката на C++ за поточен вход/изход:



Фиг. 3. Множествено наследяване в библиотека на C++

Като най-непоклатим аргумент в подкрепа на множественото наследяване в [9] се посочва, че някои модели просто не могат да бъдат реализирани по друг начин, например шаблоните за проектиране `Наблюдател (Observer)`, `Адаптер (Adapter)` и `Мост (Bridge)` [29].

Singh [122] класифицира начините за използване на множественото наследяване в 4 категории:

1. **Множествени независими протоколи** (Multiple Independent Protocols): Включва ситуациите, при които един клас е създаден чрез комбинирането на няколко напълно независими суперкласа.

2. **Няколко класа, създадени специално, за да бъдат комбинирани** (Mix and Match). В някои езици директно се поддържат такива класове, наречени mixins. Такива са Clos, Perl, Python и Ruby.

3. **Субмодулност** (Submodularity): Покрива ситуациите, когато при създаването на програмата се отделя внимание на модулността на нейните части, като критерий за добро проектиране.

4. **Разделяне на интерфейса от имплементацията** (известно още като „брак по сметка"). В този случай се създава един абстрактен клас, който дефинира интерфейса, а втори клас капсулира детайлите по реализацията. Lippman [17, с. 885] определя този начин на използване на множественото наследяване като най-често срещан.

Използването на множествено наследяване поражда и някои проблеми. В [28] и [30] тези проблеми не са класифицирани, а са разгледани с примери и препоръки за тяхното преодоляване в програмите на C++. В [9] [17] [5] [18] е отделено повече внимание като на възникването на проблемите, така и на подходите за справяне с тях в различните езици за ООП.

Проблемите при множественото наследяване могат да се разделят в две основни категории:

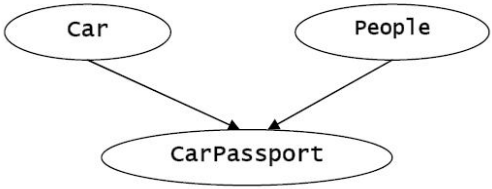
1. Колизии на имената (name collisions, name clash problem).

2. Повторно наследяване (многократно наследяване, repeated inheritance, replication problem, DDD (Dreadful Diamond of Derivation) problem).

Колизии на имената са възможни, ако в интерфейсните части на два или повече базови класа има членове (променливи или функции) с еднакви имена. Тогава се получава неяснота в производния клас. С най-голяма сила това важи за наследените операции (методи). Наличието на две операции с едно и също име внася двусмисленост в поведението на производния клас. В някои източници колизията на имената се разглежда като два независими

проблема – Name Clash Problem, отнасящ се до едноименните членове – данни и Selection Problem или Method Combination, засягащ едноименните методи.

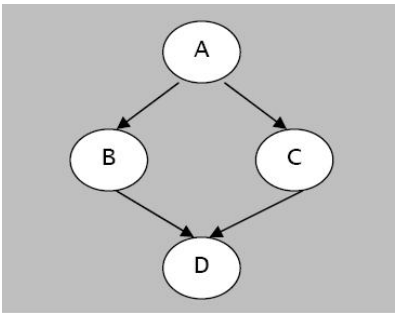
Пример за такава ситуация може да се даде с базови класове `Person` и `Car`, представящи, съответно, хора и автомобили (виж фиг. 4). И двата класа имат метод `display()`. Класът `CarPassport` наследява от `Car` и `Person` два метода с име `display()`. Проблемът е как да се разграничават те в `CarPassport`?



Фиг. 4. Колизия на имената при множествено наследяване

За справянето с този проблем програмистът може или да не допуска съвпадение на имената, или да използва заложения в конкретния език механизъм. Например, в C++ проблемът е решен чрез използване на квалифицирани имена – `Car::display()`, `Person::display()`.

Вторият важен проблем е **повторното наследяване на един и същ клас**. Това се получава когато един клас е предшественик на друг клас повече от веднъж. В почти всички учебници се дава примера с клас *D*, който има два базови – *B* и *C*, като всеки от тях е производен на *A* (фиг. 5). Тогава *D* ще наследи компонентите на *A* два пъти.



Фиг. 5. Повторно наследяване

Подходите за решаването на този проблем са три:

1. Компилаторът не допуска повторно наследяване. Така е решен проблемът в SmallTalk и Eiffel.

2. Допуска се повторно наследяване, но се изисква използването на пълните квалифицирани имена на компонентите. Това е едно от решенията, реализирани в C++.

3. Поддържа се само едно копие на данните. Това е второто решение, реализирано в C++. Повторно наследените класове се обявяват за **виртуални базови класове** в декларациите на своите наследници.

Когато един клас е виртуален, независимо от участието му в няколко списъка на базови класове, се създава само едно негово копие. В примера на фиг. 5, ако класът *A* се определи като виртуален базов за класовете *B* и *C*, класът *D* ще съдържа само един "поделен" базов клас *A*. Ще отбележим, че обявяването на *A* за виртуален не оказва никакво влияние на преките му наследници *B* и *C*, а само на *D*.

В някои източници, например [2] [9] [14], механизмът на виртуалните базови класове се нарича **виртуално наследяване** и би могло да се определи като, отделен вид наследяване:

> **Деф. 5.** Наследяване, при което една инстанция на базов клас се поделя между множество производни класове се нарича виртуално наследяване. [17].

Предимствата от използването му са безспорни, но има и ситуации, когато поддържането на две копия на базов клас е необходимо. Такъв пример откриваме в [13]. Имаме клас Scrollbar, базов за други два класа:

```
class Scrollbar
{
private:
    int x;
    int y;
public:
    void Scroll(units n);
//...
};
class HorizontalScrollbar : public Scrollbar {/*..*/};
class VerticalScrollbar : public Scrollbar {/*..*/}
```

Искаме прозорец с два скролбара – вертикален и хоризонтален. Това може да се постигне с множествено невиртуално наследяване:

```
class MultiScrollWindow: public VerticalScrollbar,
public HorizontalScrollbar {/*..*/};
MultiScrollWindow msw;
msw.HorizontalScrollbar::Scroll(5); // scroll left
msw.VerticalScrollbar::Scroll(12); //...and up
```

Така потребителят може да скролира в двете направления.

6. Изводи и препоръки, свързани с наследяването:

1. "*has-a*" отношенията се моделират по-добре с композиция, отколкото с наследяване.

2. Въпреки важността на наследяването за ООП, при изграждането на модела на програмата трябва първо да се обмисля възможността за използване на композиция, поради нейната по-голяма гъвкавост. Наследяване трябва да се използва само, ако е очевидно необходимо.

3. `Private` наследяване трябва да се използва за моделиране на "*is implemented with*" отношения между класовете.

4. От полза е развиването на умения на студентите за комбинирано използване на композиция и наследяване чрез подходящи задачи, например [4, с. 281].

5. В курса по ООП множественото наследяване трябва да се използва за комбиниране на функционалността на няколко класа едновременно.

6. Поради високата си степен на сложност, особеностите на виртуалното и множествено наследяване да се разглеждат подробно във втората част на курса по ООП.

7. В повечето учебници проблемите, свързани с множественото наследяване се разглеждат с твърде абстрактни модели (класове `A, B, C, D,` или `Bas, Der1, Der2, Der1.2` и т.н.). По-удачно е да се използват примери от ежедневието на студентите, както това е направено в [9].

8. Трудностите при усвояването на наследяването могат да се избегнат чрез:

   а. използването на подходящи примери – Обяснява се най-добре с аналогии на наследяването в реалния свят, например, както децата наследяват характеристики и поведение от своите родители;

b. разглеждане на наследяването не само от езикова гледна точка, а и като инструмент на проектирането;
c. класифицирането и обсъждане на възможностите за използване на всички видове наследяване;
d. използването на UML нотация и помощни средства за онагледяване на класовите йерархии.

ЛИТЕРАТУРА

1. *Adamchik, V., Gunawardena, A. A.* Learning Objects Approach to Teaching Programming. Proceedings of the International Conference on Information Technology: Computers and Communications, IEEE Computer Society Washington, DC, USA, 2003, Page: 96

2. *Alexandrescu, A.* Modern C++ Design: Generic Programming and Design Patterns Applied, Addison Wesley, 2001, 352 pages

3. *Dodrill Gordon.* C++ Language Tutorial, Coronado Enterprises, 1997, http://mssls7.mssl.ucl.ac.uk/sw/help/cpp/cpplist.htm

4. *Eckel, B.* Thinking in Java, Prentice Hall, 4 edition, 2006, 1150 pages

5. *Eckerdal, A., Thuné, M.* Novice Java Programmers' Conceptions of "Object" and "Class", and Variation Theory, Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education, Caparica, Portugal, 2005, Pages: 89–93

6. *Grissom, Sc., Dulimarta, H.* An approach to teaching object oriented design in CS2, Journal of Computing Sciences in Colleges, Volume 20 , Issue 1 (October 2004), Pages: 106–113

7. *Grogono, P.* The Evolution of Programming Languages, Department of Computer Science, Concordia University, Montreal, Quebec, 1999, 118 pages

8. *Hekmat, Sh.* C++ Essentials, PragSoft, 2005, 311 pages

9. *Kalev Danny.* ANSI/ISO C++ Professional Programmer's Handbook, Que, 1999, 356 pages

10. *Kölling, M.* The design of an object-oriented environment and language for teaching, Dissertation for the degree of Ph.D. at the Basser Department of Computer Science, University of Sydney, 1999, 200 pages

11. *Kruse, R., Ryba, Al.* Data Structures and Program Design in C++, Prentice Hall, 2000, 734 pages

12. *Lafore, R.* C++ Interactive Course: Fast Mastery of C++, Waite Group Press, 1996, 944 pages

13. *Liang, D.* Introduction to Java Programming-Comprehensive Version, Prentice Hall, 6 edition, 2006, 1328 pages

14. *Liberty, J., Aklecha, V.* C++ Unleashed, Sams, 1998, 944 pages

15. *Liberty, J., Jones, B.* Sams Teach Yourself C++ in 21 Days (5th Edition), Sams, 2004, 936 pages

16. *Lippman, St.* Essential C++, Addison Wesley, 2002, 416 pages

17. *Lippman, St., Lajoie, J.* C++ Primer (3rd edition), Addison Wesley, 1998, 1264 pages

18. *Meyers Scott.* Effective C++ Cd: 85 Specific Ways to Improve Your Programs and Designs, Addison Wesley Longman; CD-Rom edition (January 1999)

19. *Milewski Bartosz.* C++ In Action Industrial-Strength Programming Techniques, Addison Wesley, 2001, 485 pages

20. *Overland Brian.* C++ in Plain English, John Wiley & Sons; 3RD edition, 2000, 595 pages

21. *Singh, Gh.* Single Versus Multiple Inheritance in Object Oriented Programming, ACM SIGPLAN OOPS Messenger, Volume 6, Issue 1 (January 1995), ACM, New York, 1995, Pages: 30–39

22. *Stroustrup, Bj.* The C++ Programming Language, Addison-Wesley, 3 edition, 2000, 911 pages

23. *Stroustrup, Bj.* Why C++ is not just an object-oriented programming language, Addendum to the proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications, Austin, Texas, United States, 1995, Pages: 1–13

24. *Swan Tom.* Mastering Borland C++ 5, Sams; Bk&CD-Rom edition (May 1996), 1058 pages

25. *Taivalsaari, A.* On the Notion of Inheritance, ACM Computing Surveys, Vol. 28, № 3, September 1996, Pages: 438–479

26. *Walter, J., Kalev, D.* The Waite Group's C++ How-To, SAMS, 2001.

27. *Zhu, H., Zhou, M.* Methodology First and Language Second: A Way to Teach Object-Oriented Programming, Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Anaheim, CA, USA, 2003, Pages: 140–147

28. *Богданов, Д.* Обектно-ориентирано програмиране със C++. Техника, 1994, 239 с.

29. *Гама, Е., Хелм, Р.* Джонсън Р., Design Patterns (Шаблони за дизайн). СофтПрес, 2005, 464 с.

30. *Тодорова, М.* Програмиране на C++. Втора част. София, СИЕЛА, 2002, 482 с.

МЕХАНИЗМЪТ НА НАСЛЕДЯВАНЕ В УЧЕБНИЯ КУРС ПО ОБЕКТНО ОРИЕНТИРАНО ПРОГРАМИРАНЕ

ИВАЙЛО ДОНЧЕВ

Резюме

Наличието на механизъм за наследяване е критерият, по който даден език или техника се определят като обектно ориентирани. Ето защо неговото изучаване и придобиването на умения за правилното му прилагане са съществени за обучението по обектно ориентирано програмиране.

В тази статия се разглежда наследяването от две гледни точки: като инструмент за изграждане на отношения от вида „генеранизация / спецификация" и като средство за повторна използваемост на кода. Въвеждат се необходимите за учебния курс основни понятия; разглеждат се възможни класификации на видовете наследяване. От направения критичен анализ на учебната литература са откроени възможните опасности от прилагането на механизма на наследяване и са дадени конкретни препоръки за подобряване на учебния процес. Акцентирано е на особеностите и проблемите при използване на множественото наследяване.

THE INHERITANCE MECHANISM IN OBJECT ORIENTED PROGRAMMING COURSE

IVAYLO DONCHEV

Summary

The presence of inheritance mechanism is the criteria to prove a language or a technique as object-oriented. That's why learning inheritance and acquiring skills to apply it properly is crucial for object-oriented programming education to be successful.

In this article inheritance is examined from two viewpoints: as a tool for creating 'generalization-specification' relationships between classes, and as a code reusability technique.

The main concepts, necessary for the Object-oriented programming (OOP) course are introduced. Useful classifications of different types inheritance are given. From the critical analysis of the textbooks that was made, there are sharply outlined possible pitfalls when applying the inheritance mechanism and some recommendations that could improve learning OOP are given. The emphasis is of the special features and problems when using multiple inheritance.