МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2009
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2009
Proceedings of the Thirty Eighth Spring Conference of
the Union of Bulgarian Mathematicians
Borovetz, April 1–5, 2009

СИСТЕМАТИЗИРАНЕ НА СПЕЦИФИЧНИТЕ ЗА C++ ТРУДНОСТИ ПРИ УСВОЯВАНЕ НА ПОНЯТИЯТА КЛАС И ОБЕКТ В ОБУЧЕНИЕТО ПО ПРОГРАМИРАНЕ

Ивайло Дончев

Обучението по обектно-ориентирано програмиране (ООП) е съпътствано от множество трудности, независимо от използвания педагогически подход – с ранно или късно въвеждане на обектите. Сред научната общност няма единно мнение на какво се дължат тези трудности. В настоящата разработка се прави опит за систематизиране на характерните за курсове по програмиране на C++ трудности, свързани с правилното възприемане на фундаменталните за ООП понятия клас и обект. Предлагат се и конкретни средства за избягване или поне за намаляване на ефекта от тези трудности.

1. Въведение. Въпреки немалкият натрупан педагогически опит, усвояването на концепциите на ООП продължава да бъде трудна задача [1, 4, 13] и много изследвания са посветени на причините за това. Мненията са твърде разнопосочни. До преди няколко години най-цитираната причина беше преминаването от процедурен към обектно- ориентиран стил на програмиране, известна като "смяна на парадигмата" (*paradigm shift*) [10, 11]. В следствие на това много университети промениха своите учебни програми по програмиране, като възприеха подход на ранно въвеждане на обектите (*objects-first*) и ООП като уводен курс. Това, обаче, не винаги дава желаните резултати. Оказва се, че изучаването на ООП е трудно за начинаещи [12]. Основна прична за това е по-високата степен на абстракция в сравнение с процедурното програмиране [4, 6], поради което популярност придоби и "смесеният" подход: в един курс се комбинират елементи и от двете парадигми [2]. Така паралелно с уменията за откриване на необходимите за системата класове и обекти и правилните отношения между тях, се развиват и чисто процедурни програмистки умения, без които е невъзможна фазата на имплементиране. Въпреки това броят на трудностите, свързани с класовете и обектите, които срещат начинаещите програмисти, не намалява. Те варират в доста широки граници: проектиране на обектно-ориентирани системи [14, 5]; връзката между понятията клас, обект и променлива [7]; програмно решаване на проблеми [15, 4]; трудности със синтаксиса на езика [9]. В огромна част от случаите става дума за проблеми, възникнали в учебни курсове, ползващи езика Java.

Прави впечатление малкият брой публикации, свързани с проблемите при обучението по ООП на C++. Това едва ли се дължи на липсата на такива проблеми. В

българските университети при обучението по програмиране преобладава класическият подход *imperative-first* – с уводен курс по процедурно програмиране и късно въвеждане на класовете и обектите (такова е и историческото развитие на програмирането). Затова най-често се използва хибридният език C++, който позволява ефективно програмиране и в двата стила – процедурен и обектно-ориентиран. Студентите започват обучението си с процедурните аспекти на програмирането и се сблъскват с класовете и обектите след като са придобили известни умения за алгоритмично и абстрактно мислене – най-често в отделен, специализиран курс по ООП. Някои преподаватели включват и в уводния курс елементи на ООП, следвайки учебници като [3, 8], но отново преобладава езикът C++. Уводни курсове на Java са по-скоро изключение, но дори и те следват коментираните по-горе подходи и осигуряват добра процедурна и алгоритмична основа на студентите. Това е предпоставка за избягване на някои трудности (програмно решаване на проблеми, синтаксис), но поражда нови (най-често свързани с проектирането на системата в обектно-ориентиран стил).

В този доклад се разглеждат и систематизират трудностите, свързани с правилното възприемане и използване на основните за ООП понятия клас и обект и се предлагат средства за тяхното избягване или поне ограничаване. Тези трудности са открити при многогодишни наблюдения на лабораторни занятия, анализ на резултатите от контролни, тестове, самостоятелни и курсови работи на студенти, изучаващи дисциплините "Основи на програмирането", "Алгоритми и структури от данни", "ООП" и "Визуално програмиране" на C++ във ВТУ "Св.Св. Кирил и Методий".

2. Констатирани проблеми. Откритите трудности и проблеми могат да се класифицират в следните категории:

2.1. Трудности, свързани с изграждането на обектния модел на системата – откриване на необходимите класове и обекти, изграждане на релациите и взаимодействието между тях. В писмените работи на студенти често се срещат следните грешки:

– дефиниран е клас, който не се използва в програмата;

– използвани са обекти на потребителски клас, който не е дефиниран;

– в даден клас е описана член-променлива – обект на недефиниран клас;

– използва се наследяване за моделиране на "*has-a*" отношение, вместо агрегация;

– неоправдано се нарушава изискването за капсулиране на класа (поставят се членове-данни в `public` секцията или се реализират операции с приятелска функция, вместо с метод на класа);

Студентите трудно откриват необходимите класове, обекти и точните релации между тях. Решение на този проблем е по-ранното използване в курса на големи програмни проекти – с множество взаимодействащи си класове. При процедурното програмиране примерите са обикновено кратки – 2-3 процедури в проект, докато за добра примерна обектно- ориентирана програма са необходими поне 2-3 класа, всеки с по няколко метода. От тази гледна точка е оправдано още в уводния курс по програмиране да се разработват по-обемнии проекти – така преходът към ООП ще е по-лек.

Отчитайки ограничения брой часове за лабораторна работа, проектите по ООП могат да се дават в полуготов вид, като студентите трябва да добавят нова функци-

оналност на вече дефинирани класове и да създават изцяло нови класове и обекти, които си взаимодействат с вече функциониращите в програмата. Също така, за избягване на тези трудности са съществени ролята на самостоятелната работа и работата в екип. За създаване на добър програмен модел решаващи са знанията за обектна декомпозиция на проблема – предмет на обектно- ориентирания анализ и проектиране (ООАП). Затова най-удачният вариант е паралелното изучаване на двата курса – ООП и ООАП.

Когато преподавателят въвежда понятията клас и обект е допустимо първо да се ограничи разглеждането до един единствен клас с няколко негови обекта, но фактът, че обектите на различни класове си взаимодействат трябва да се демонстрира възможно по-рано, за да се избегне изграждането на погрешни представи за обектно-ориентирания модел.

2.2. Трудности при изграждането на цялостна картина на работата на програмата, която решава поставената задача. Важно изискване за усвояването на ООП е студентите да имат ясна представа и разбиране за процеса на изпълнение на програмата. Това от своя страна предполага разбиране на взаимодействието между обектите в нея.

Недостатък на повечето курсове по ООП е, че обикновено се съсредоточават върху създаването на класове и обекти, използвайки конкретен език за програмиране, без студентите да добият представа защо са необходими те. Липсва опитът от практическата разработка и представата за пълния цикъл на създаване, внедряване и поддръжка на софтуера, работата в екип. Ограниченият брой часове не позволяват създаването на голям реален проект в уводния курс по ООП, но в един практикум (като изборен или факултативен курс) с подобаващо място в него на самостоятелната работа, това е възможно.

За задължителния курс по ООП е достатъчно студентите да усетят динамичната природа на програмите като колекции от взаимодействащи си обекти, като всеки обект проявява поведение, определено от класа на който принадлежи. Това е трудно да се демонстрира с дидактически материали със статичен характер (например учебници). Затова използването в учебния процес на помощни средства за визуализация (често пренебрегвано от преподавателите) е от голяма полза за избягване на тези трудности.

2.3. Неразбиране на работата с паметта. Пропуски при дефинирането и използването на класове, съдържащи указатели. Проблемът се дължи на пропуски в усвояването на указателите още в уводния курс. Студентите често пропускат да заделят и освободят коректно памет чрез указателите. Срещат трудности и със синтаксиса, особено при конструирането на по-сложни изрази, в които участват указатели или адреси на обекти. Към това се добавят и особеностите на C++, свързани с конструирането и присвояването на обекти – компилаторът автоматично създава деструктор, копиращ конструктор и оператор за присвояване за всеки дефиниран клас, ако в класа няма дефинирани такива методи. Проблемът е в това, че копиращият конструктор и операторът за присвояване извършват т. нар. "плитко" копиране на обектите (*shallow copy*), което е некоректно за обекти, които съдържат указатели. Използването на такива обекти може да доведе до грешки по време на изпълнение на програмата. Анализът на учебната литература сочи, че не се отделя достатъчно внимание на този проблем. Студентите трябва да се научат да преде-

финират тези методи за всеки клас, който съдържа членове-данни, указващи други обекти така, че да извършват "дълбоко" копиране (*deep copy*) на обектите, а именно, да се заделя и освобождава коректно необходимата памет за сочените от указатели обекти. Добри примерни класове за тази цел са такива, съдържащи символни низове `char*` вместо `string`. За преодоляването на този проблем преподавателите трябва да насочат вниманието на студентите към писането на код и четенето на добър чужд код със съсредоточаване върху детайлите.

2.4. Тенденция цялата функционалност на програмата да се обединява в един клас. Това е пряко следствие от влиянието на императивния стил на програмиране. Студентите се стремят към централизирано управление на системата, а не към характерния за ООП разпределен контрол. Методите се използват като процедури, игнорира се използването на ключовите за ООП механизми наследяване и полиморфизъм. Когато процедурните конструкции и решаването на проблеми се въвеждат заедно, студентите се научават да мислят процедурно, независимо дали това е търсеният ефект от обучението. При по-късното изучаване на класовете и обектите, студентите нямат проблем с владеенето на синтаксиса на езика – как да дефинират класове и да създават техни обекти. Проблем възниква по-скоро от липсата на правилен логически модел за ефективното им използване. За справяне с този проблем помагат:

– изучаване на ООАП с акцент на декомпозирането на задачата в класове и обекти;

– избор на примерни задачи, в които полиморфизмът е естествена част от решението;

– акцентиране на операциите с обекти и комуникацията между обекти;

– класифициране на релациите между класовете и посочване на типични техни приложения.

2.5. Акцентиране върху данните, описани в класа, за сметка на аспектите, свързани с поведението на обектите. В учебните курсове често срещаме примерни класове, в които методите служат само за четене и модификация на данните. На практика такива класове приличат повече на структури, например:

```
class point {
   double x;
   double y;
public:
   point(double a, double b){x=a; y=b;}
   double get{_}x(){return x;}
   double get{_}y(){return y;}
   void set{_}x(double a){x=a;}
   void set{_}y(double a){y=a;}
};
```

Този проблем може да се избегне, ако примерните класове съдържат такива методи, че отговорът на съобщението да зависи от състоянието на обекта. За дефинирания по-горе клас би могло да се добави метод `quadrant()`, който определя в кой квадрант се намира точката:

```
int point::quadrant()}
{
   return (x>0&&y>0) ? 1 :
          ((x<0&&y>0)? 2 :
          ((x<0&&y<0)? 3 : ((x>0&&y<0) ? 4 : 0)));
}
```

и метод `move()`, който премества точката:

```
void point::move(double dx, double dy)
{
   x += dx; y += dy;
}
```

2.6. Трудности при използване на статичните компоненти на класовете. Механизмът на статичните компоненти (членове-данни и членове-функции на класовете) в C++ е реализация на идеята за метакласове (класове на класовете). Това усложнение на модела затруднява студентите – след като са възприели идеята, че всеки обект притежава свое копие на данните на класа, сега трябва да приемат, че има и членове-данни, които са общи за всички обекти. Освен това, тези данни могат да се използват и извън контекста на обектите.

Проблемът с възприемането на идеята за статични членове се среща по-рядко сред студенти, запознати в уводния курс по програмиране със статичните локални променливи във функциите. Независимо от предварителната подготовка и педагогическия подход, често срещана грешка е опитът за манипулиране на обект чрез статичен метод, като се пропуска фактът, че статичните методи не получават указател `this` като неявен параметър.

Преодоляването на тези трудности е възможно чрез демонстрация на подходящи примери за използване на статични методи и данни – такива, че да е оправдано в решението да се използват точно тези средства. Ето такъв пример:

```
class Employee {
     static int count; // статична член-променлива (брояч)
     char* name;       // име
     int year;         // година на раждане
     double salary;    // заплата
public:
     Employee(){name=NULL; year=0; salary=0; count++;}
     Employee(char*, int, double);
     ~Employee(){delete name; count--;}
     Employee(const Employee&);
     virtual void display();
};
Employee::Employee(char *a, int b, double c)
{
     int sz =strlen(a)+1;
     name = new char[sz];
```

```
    strcpy_s(name,sz,a);
    year = b;
    salary = c;
    count++;
}
// ..................................
int Employee::count = 0;
// ..................................
```

Тук член-променлива `count` служи за проследяване на броя на създадените обекти от класа `Employee`. Конструкторите на класа, включително копиращият, увеличават нейната стойност с `1`, а деструкторът я намалява с `1`.

Удачно е и привеждането на примери от библиотечните класове на C++, например често използваният `ios`.

2.7. Трудности по прилагане на динамичния полиморфизъм. Динамичният полиморфизъм се възприема от студентите като една от най-трудните концепции на ООП. Трудностите не са свързани с дефинирането на виртуални методи и абстрактни класове, а с използването им за реализиране на полиморфично поведение на обектите. За да придобият умения правилно да прилагат полиморфизма, студентите трябва да се "сблъскат" с него възможно най-рано – веднага след въвеждане на концепцията "наследяване" като нейно важно приложение. Тук отново се откроява полезната роля на ООАП за усвояването на тази концепция – студенти, изучавали елементи на ООАП срещат по-малки трудности с прилагането на полиморфизма.

За избягването на тези трудности спомага и изборът на близки до ежедневието на студентите, примерни задачи, в които полиморфизмът е естествена част от решението. Изцяло абстрактните модели, които срещаме в някои учебници с прекалено сложна и изкуствено създадена йерархия от класове с имена като B1, D12, D13, и т.н., не помагат на студентите да добият интуитивна представа за полиморфизма.

Забелязва се и връзка на този проблем с трудностите от т. 2.3. Динамичният полиморфизъм в C++ се реализира чрез указатели, които сочат елементи на различни класове. Евентуалните пропуски в усвояването на указателите тук рефлектират с още по-голяма сила.

2.8. Трудности при конструирането и разрушаването на обекти. От анализа на писмените работи и наблюденията по време на лабораторни занятия се открояват следните често срещани грешки:

– Неразбиране на механизма на предаване на параметри при изпълнение на конструктори, например:

```
point::point(int a, int b)
{a = x; b = y;} // вместо x = a; y = b;
```

Тези грешки водят началото си от уводния курс по програмиране, където са допуснати пропуски при изучаване на функциите. Друга типична грешка, дължаща се на същите причини, е следната:

```
point::point(int a, int b)
```

```
{
cout << "x= "; cin >> x; // вместо x = a;
cout << "y= "; cin >> y; // вместо y = b;
}
```

С оглед унифицирането на синтаксиса при дефиниране на конструктори, поподходящо е да се използва стила, характерен за конструкторите с параметри на производни класове, а именно:

```
point::point(int a, int b): x(a), y(b){}
```

– **Неразбиране кога се изпълняват конструкторите и деструкторите:** Тъй като извикването на конструкторите и деструкторите в повечето случаи става автоматично, за да проследят студентите тяхното изпълнение е удачно добавянето на информиращо съобщение, например:

```
class Person {
      char *name;
      int year;
      //.....................................
      ~Person(){delete name; cout << "Person destruction...\n";}
      //.....................................
};
```

– **Неотчитане на реда на изпълнение на конструктори и деструктори на производни и базови класове:** Този проблем е свързан с особеностите на механизма на наследяване и неговата реализация в C++. Студентите знаят теоретично правилата, но въпреки това допускат грешки в практическото им прилагане. Най-често срещани грешки са да се пропусне извикването на конструктор на базов клас от конструктора на негов производен, или да се смени реда на извикване на конструкторите на базовите класове при множествено наследяване, или да се пропусне извикването на конструктор на виртуален базов клас от конструктор на непряк негов наследник.

2.9. Трудности, свързани с терминологията. Тези проблеми се дължат на неоднозначния превод на термините на български език, използването на едни и същи ключови думи в коренно различен контекст или на различния смисъл, който отделните автори влагат в термините. От анализа на тестове, писмени работи и практически изпити се откроиха следните често срещани грешки, свързани с терминологията:

– **Смесване на понятията "предефиниране на функции", "предефиниране на методи" и "предефиниране на операции".** Макар тези понятия да имат общи особености, тяхното ясно разграничаване е задължително за усвояване на концепциите на ООП. Преподавателите трябва да акцентират не само на синтактичните особености на механизмите, но и да подчертаят разликите в семантиката и използването им.

Предефинирането на операции изисква специално внимание поради множеството ограничения, с които програмистът трябва да се съобразява. Важна разлика, на

която трябва да се акцентира е, че предефинирането на функции позволява едно и също име да се използва за функции с различен брой или тип на параметрите и върнатата стойност, докато предефинирането на операции позволява да се използват знаците за операции и с потребителски дефинирани типове – класове.

Друг важен аспект е разграничаването на предефинираните функции от предефинираните методи (член-функциите на класовете) в производни и базови класове, особено що се отнася до механизма на виртуалните методи и динамичното свързване.

– **Виртуалното наследяване се бърка с виртуални функции**. Тази грешка се дължи на това, че една и съща ключова дума – `virtual` се използва за спецификатор на два коренно различни механизма – на виртуалните базови класове и на виртуалните методи. Преподавателят трябва да подчертае, че при обявяването на един *базов клас* за виртуален спецификаторът `virtual` управлява механизма на наследяване, докато използването на същия спецификатор за обявяване на *метод* за виртуален управлява механизма на свързване (динамично или статично).

– **Неизяснени понятия "копиращ конструктор" и "конструктор за присвояване".** Дължат се на разлики в превода на термините на български език. Едни автори наричат "копиращ" този конструктор, който компилаторът създава автоматично за всеки клас (*implicit copy constructor*), а "за присвояване" – този, дефиниран от програмиста (*explicit copy constructor*). При други автори значението на термините е разменено. За да се избегнат неясnотите, предлагам да се използват термините "подразбиращ се копиращ конструктор" и "предефиниран копиращ конструктор".

2.10. Трудности при работа с тежките професионални програмни среди и липса на добри помощни средства. Средите за разработка на софтуер, които се използват за учебни цели и изискванията към тях се дискутират все по-често в научната литература. Преподавателите имат различен подход към този въпрос: един и същ програмен език се използва по различни начини, вариращи в широките граници от обикновен текстов редактор и компилатор от командния ред до професионални интегрирани среди за програмиране. Така се изграждат твърде различни програмистки умения.

Считам, че за усвояването на принципите на ООП голямо значение имат средствата, които средата за програмиране предоставя по отношение на възможностите за тестване, откриване на грешки и визуализация на класовете и обектите в програмата, техните връзки и комуникация. Само редактор и компилатор не са достатъчни. Другата крайност също не е за препоръчване: една тежка професионална среда, претрупана с възможности, които не се използват в учебния курс, може да се окаже стресираща за студентите. Разработените специално за обучение среди използват предимно езика Java и са неприложими в курсове по C++. Преподавателят по C++ има възможност да избере между олекотените безплатни версии, например Dev C++ и професионалните среди на Borland и Microsoft. Критериите при избора трябва да бъдат:

– леснота на използване;
– наличие на вградени помощни средства (съветници);
– наличие на средства за откриване на грешки и тестване;
– наличие на средства за визуализация;
– наличие на средства за проектиране;

– достъпност и популярност на средата.

Ако използваната среда не покрива всички нужди, естествено е използването на допълнителни помощни средства, например отделна среда за проектиране, дидактически средства, използващи анимация за онагледяване на взаимодействието между обектите в програмата и др.

2.11. Кодът, който студентите пишат, е с ниска степен на повторна използваемост и не е защитен от грешки по време на изпълнение на програмата. Ниската степен на повторна използваемост на кода, сращана в писмените работи на студентите, се дължи на недостатъчно разбиране на основните механизми за нейната реализация както в процедурното програмиране (подпрограми, функции и модули), така и в ООП (наследяване, агрегация, пространства на имената).

По отношение на защитеността на кода проблемът е по-скоро в това, че с цел изчистване на примерните програми от допълнителни елементи и акцентиране на конкретната изучавана конструкция, преподавателите често умишлено пропускат да включат в кода защита от грешки. Това важи както за процедурното програмиране, така и за ООП. Въпросът за защита от грешки и тяхната обработка се коментира чак в края на курса по ООП – при изучаване на обектно-ориентираната обработка на изключения.

Веднъж придобили навика да пропускат такъв код, студентите трудно се приучават да го използват, а неговото прилагане в ИТ индустрията е задължителен елемент. Ето защо е необходимо преподавателите възможно най-рано да въвеждат идеите за защита на програмите от грешки по време на изпълнението им. В курса по ООП също е възможно използване на стандартните начини за защита и обработка на грешки, усвоени в уводния курс, докато се въведе нелеката концепция на обработката на изключения. Заделянето на памет за динамичен масив от обекти например може да се прави така:

```
Employee *myarray;
if(!(myarray = new Employee[10000]))
{
     cout << "Error allocating memory..." << endl;
     return 0;
}
```

След като веднъж са свикнали да пишат защитен код, студентите лесно ще свикнат да използват и обектно-ориентирания механизъм за това:

```
try
{
      Employee* myarray = new Employee[10000];
}
catch (bad_alloc&)
{
      cout << "Error allocating memory... " << endl;
      //.............................
}
```

3. Заключение. Факторите, които са причина за срещане на посочените трудности, не са еднородни. Някои са свързани с по-сложния синтаксис на C++ и необходимостта програмистът да се грижи за управлението на динамичната памет [3, 8], други се дължат на негативите от смяната на парадигмата [1, 4, 5]. За минимизиране на отрицателния ефект от тези трудности е необходимо подготовката на студентите за правилно възприемане на концепциите клас и обект да започне възможно най-рано – с развиване на способностите им за алгоритмично и абстрактно мислене, моделиране на реални ситуации в категориите на ООАП и добро усвояване на синтаксиса и семантиката на използвания език за програмиране още от първите теми в уводния курс по програмиране.

REFERENCES

[1] J. Börstler, H. Sharp. Learning and teaching object technology. *Editorial. Computer Science Education*, **13**, No.4 (2003), 243–247

[2] L. Cecchi, P. Crescenzi, G. Innocenti. C : C++ = JavaMM : Java. In: Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java (ACM International Conference Proceeding Series), vol. **42**, Kilkenny City, Ireland, 75–78.

[3] H. M. Deitel, P. J. Deitel. C++ How to Program'. Prentice Hall, Fifth Edition, 2005, 1536 p.

[4] A. Ebrahimi, C. Schweikert. Empirical study of novice programming with plans and objects. Working group reports on ITiCSE on Innovation and technology in computer science education, Bologna, Italy, 2006, 52–54.

[5] S. Garner, P. Haden, A. Robins. My Program is correct but it doesn't run: A preliminary investigation of novice programmers' problems. In: Proceedings of Australasian Computing Education Conference, 2005, 173–180.

[6] S. Hadjerrouit. A constructivist approach to object-oriented design and programming. In: Proceedings of the 4th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education. Cracow, Poland, 1999, 171–174.

[7] S. Holland, R. Griffiths M. Woodman. Avoiding object misconceptions. Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education, San Jose, California, 1997, 131–134.

[8] C. Horstmann. Computing Concepts with C++ Essentials". Wiley; Third Edition edition, 2002, 784 p.

[9] M. Hristova, A. Misra, M. Rutter, R. Mercuri. Identifying and Correcting Java Programming Errors for Introductory Computer Science Students. In: Proceedings of the 34th SIGCSE technical symposium on Computer science education, Reno, Navada, USA, 2003, 153–156.

[10] M. Kölling. The Problem of Teaching Object-Oriented Programming, Part 1: Languages. *Journal of Object-Oriented Programming*, **11**, No 8, (1999), 8–15.

[11] C. Liu, S. Goetze, B. Glynn. What Contributes to Successful Object-OrientedLearning? OOPSLA'92 Proceedings, 1992, 77–86.

[12] M. McCracken, V. Almstrum. A multi-national, multi-institutional study of assessment of programming skills of first-year CS Students. In: Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education. December 01, 2001, Canterbury, UK.

[13] A. Robins, J. Rountree. Learning and Teaching Programming: A Review and Discussion. *Journal of Computer Science Education*, **13**, (2003), 137–172,.

[14] B. Thomasson, M. Ratcliffe, L. Thomas. Identifying novice difficulties in object oriented design. ACM SIGCSE Bulletin, **38**, Issue 3 (2006), 28–32.
[15] F. Wei, S. Moritz, S. Parvez, G. Blank. A student model for object-oriented design and programming. Journal of Computing Sciences in Colleges, **20**, Issue 5 (2005), 260–273.

Ивайло Дончев
ВТУ "Св.Св. Кирил и Методий"
Педагогически факултет
Катедра "Информационни технологии"
Арх. Г. Козарев № 3
5000 Велико Търново
e-mail: i.donchev@abv.bg

SYSTEMATIZING THE C++ SPECIFIC DIFFICULTIES WHEN LEARNING THE CONCEPTS OF CLASS AND OBJECT IN PROGRAMMING TEACHING

Ivaylo Donchev

The teaching of Object-Oriented Programming (OOP) is accompanied by a number of difficulties whether the objects-early or objects-late pedagogical approach is employed. The scientific community has no unified opinion on the issue, what causes these difficulties. The present paper attempts to systematize the difficulties typical for C++ programming courses and related to the proper apprehension of the 'class' and 'object' concepts fundamental for OOP and proposes particular tools to avoid or at least to reduce the effects of them.