Educational Technologies
Образователни технологии

ДЪЛБОКО КОПИЕ В C++ И JAVA

Христина Костадинова, Марияна Райкова
Нов български университет

Резюме. В статията се разглежда темата за програмната реализация на начините за копиране на обекти при разработване на софтуерни приложения. Акцентът е поставен върху проблема с т.нар. плитко копие (shallow copy) – ситуация, в която оригиналният обект и обектът копие използват общи ресурси, и страничните ефекти, които възникват вследствие на плитко копие. Описани са предимствата и недостатъците на различните подходи за реализиране на дълбоко копие в двата най-често използвани в обучението програмни езика: C++ и Java. Резултатите от прилагане на плитко копие и начините за реализация на дълбоко копие са представени чрез примери на двата езика.

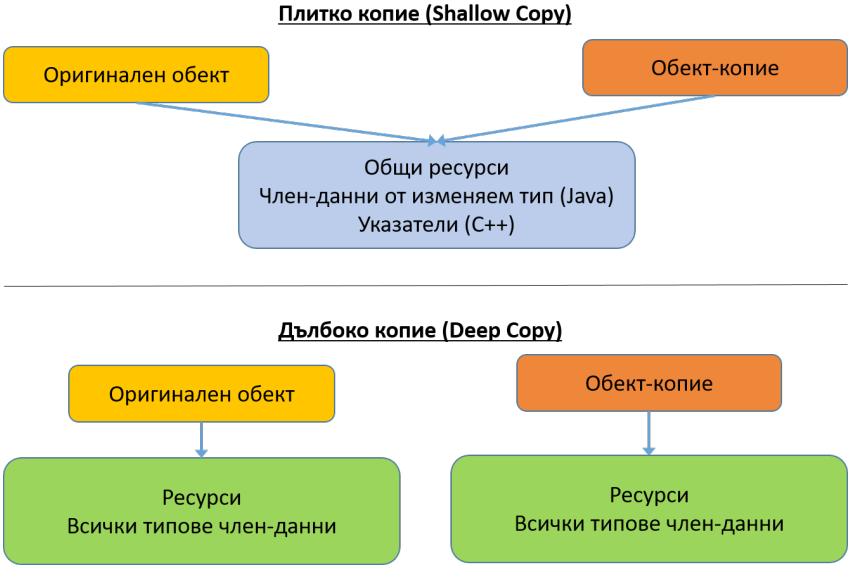
Ключови думи: плитко копие ; дълбоко копие; ООП; C++; Java

1. Какво е плитко и дълбоко копие?

При реализиране на софтуерни проекти, в които се използва обектно ориентирано програмиране (ООП), в много случаи е необходимо да се използват стойностите на член-данните на един обект за създаване на друг обект. Така създаваме т. нар. ***обект копие***, като използваме оригиналния обект. При такава ситуация трябва да се подхожда много внимателно, защото има риск от непредвидени резултати за стойностите на член-данните и на двата обекта.

Обектите могат да се копират чрез реализиране на: ***плитко копие*** (shallow copy) или ***дълбоко копие (***deep copy***)***.

Плитко копие се създава, когато обектът копие и оригиналният обект споделят едни и същи ресурси в паметта. Можем да получим плитко копие при използване на оператора за присвояване между обекти, подаване на обект като параметър на функция, присвояване в различни области на видимост и др. В такъв случай се стига до ситуация, в която промяната на обекта копие се отразява и на оригиналния обект. Това е в разрез с добрите принципи на програмиране и застрашава правилното изпълнение на програмата, често води до авариране на програмите по време на изпълнение и е много трудно за проследяване и коригиране. На фигура 1 е показана схемата на достъп до ресурсите на оригиналния обект и обекта копие при плитко и дълбоко копие.



Фигура 1. Плитко и дълбоко копие

Ще покажем проблемите, които възникват при използване на плитко копие, и различните подходи за реализиране на дълбоко копие. Ще обърнем внимание на предимствата и недостатъците на тези подходи и случаите, в които всеки от тях трябва да се използва. За тази цел ще поставим и решим следната задача.

Задача. *Да се напише програма на C++ и Java, която реализира клас училище, характеризиращо се с: име, брой класни стаи и директор. Директорът на училището е клас, който се описва чрез името си.*

Ще използваме решението на поставената задача за демонстрация на ситуация, в която обект от клас „Училище“ служи за създаване на нов обект от клас „Училище“ със същите стойности на член-данните. Решенията са само демонстрационни и показват тези части от кода, които се отнасят до копирането на обектите.

2. Реализация на дълбоко копие в C++

Проблемите при плитко и дълбоко копие в езика C++ възникват, когато директно работим в паметта и боравим с нейното заделяне и освобождаване (т. нар. heap памет). Имаме нужда от реализиране на дълбоко копие, когато член-данна на класа е указател, или т.нар. динамична променлива, разположена в heap паметта. Най-често използваният пример за

онагледяване на нуждата от дълбоко копие е в случаите, когато член-данна на класа е динамичен масив (Meyers, 2017). Дълбоко копие е необходимо в следните ситуации:

1. създаване на обект чрез друг обект;
2. присвояване на един обект на друг обект;
3. подаване на обект по стойност като параметър на функция;
4. създаване (присвояване) на обект в различни области на видимост от оригиналния обект.

За решение на поставената задача създаваме клас „Училище“ с допълнителна член-данна: динамичен масив от учители, които преподават в училището.

```
class Teacher{
private:
    std::string name;
    std::string subject;
public:
    //контруктори, деструктор, get и set член-функции
};
class Principal{
private:
    std::string name;
public:
    //контруктори, деструктор, get и set член-функции
};
class School {
private:
    std::string name;
    int number Of Teachers;
    Principal principal;
    Teacher * teachers;
public:
    //контруктори, деструктор, get и set член-функции
…
};
//предефиниране на вмъкване в изходен поток на текстовото
съдържание на обекти от класовете
std::ostream&operator<<(std::ostream&, const Teacher&);
std::ostream&operator<<(std::ostream&, const Principal&);
std::ostream&operator<<(std::ostream&, const School&);
```

Листинг 1. Реализация на клас „Училище“ в C++

Ще разгледаме реализацията на дълбоко копие в C++, като опишем два основни подхода:

1. конструктор за копиране и предефиниране на оператор за присвояване;
2. клониране на обекти.

Конструктор за копиране и предефиниране на оператор за присвояване в C++

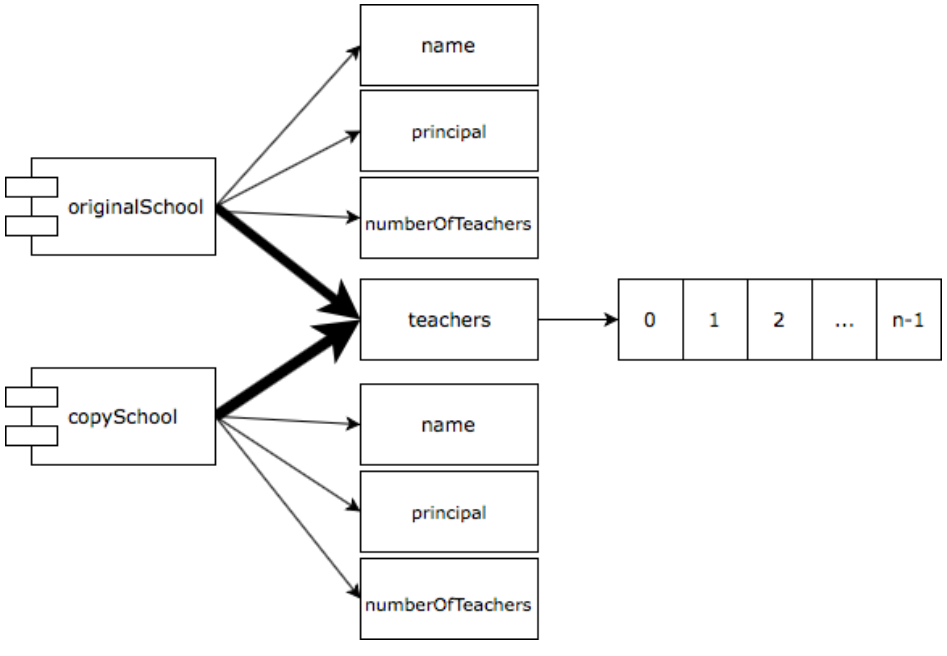
Един от най-разпространените подходи за осигуряване на дълбоко копие в C++ е използване на конструктор за копиране. В източниците за изучаване на езика е посочено, че за осигуряване на дълбоко копие е необходимо да се реализират конструктор за копиране, деструктор и да се предефинира операторът за присвояване (Todorova, 2011), (Meyers, 2017). В листинг 2 е демонстрирана реализацията на копиращ конструктор. Специфичното за езика е, че ако имаме обекти, разположени в heap паметта, е необходимо поелементно копиране на всички данни от тази памет за новия обект с цел избягване на ситуация споделени ресурси (фигура 2).

```
class School {
public:
    // конструктори get и set функции
…
    //копиращ конструктор
    School(const School & original){
    name = original.name;
    numberOfTeachers = original.numberOfTeachers;
    principal = original.principal;
    teachers = new Teacher[numberOfTeachers];
    for (intindex = 0; index<numberOfTeachers; index++) {
    teachers[index] = original.teachers[index];
    ...}
......}
......//деструктор
....~School(){
if(teachers!=nullptr) delete[] teachers;
teachers = nullptr;
....}
private:
....//член-данни на класа
};
int main()
{
Principal principal(„Иванов“);
Teacher * teachers = new Teacher[2];
teachers[0] = Teacher(„Димитрова“);
```

```
teachers[1] = Teacher(„Христов");
School originalSchool(„Георгиев", principal, 2, teachers);
....//автоматично извикване на копиращ конструктор, ако има
дефиниран такъв
....//или извършване на побитово плитко копие, ако не е
дефиниран копиращ конструктор
School copySchool = originalSchool;
//явно извикване на копиращ конструктор
School secondCopySchool(originalSchool);
if(teachers!=nullptr) delete[] teachers;
teachers = nullptr;
}
```

Листинг 2. Копиращ конструктор в C++

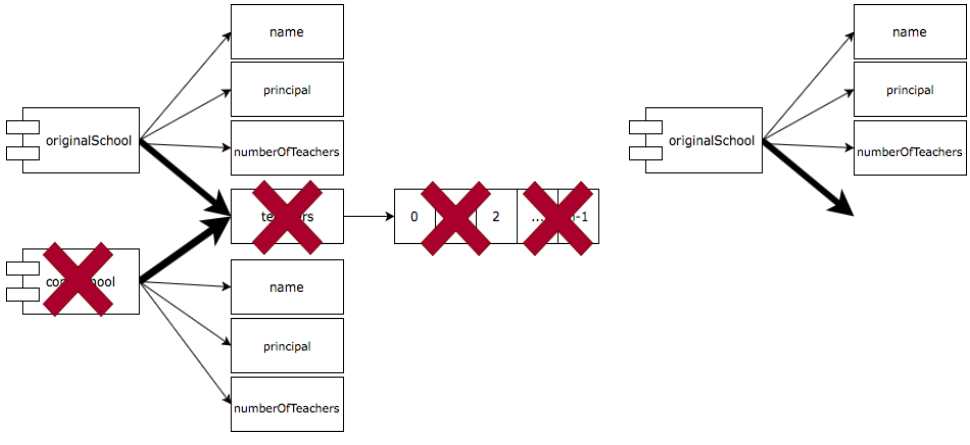
В класове, които имат член-данни външни ресурси (динамични обекти, масиви и др.), трябва да се извърши копиране на отделните елементи от оригиналния обект в тялото на копиращия конструктор. В листинг 2 е демонстрирано как се копират данни за масива с учители на даденото училище: необходимо е елемент по елемент от единия масив да бъдат копирани стойностите в масива на новия обект.



Фигура 2. Споделени ресурси между различни обекти в C++

Проблем със споделените ресурси настъпва при следните събития.

1. Промяна на данните за учители в някой от двата обекта би довела до промяна на учителите и в двете училища.
2. Излизане от област на видимост на един от обектите и извикване на неговия деструктор ще доведе до изтриване на учителите за двете училища (листинг 3).
3. Подаване на обект по стойност като параметър на функция (**`void`** `someFunction(School param)` в листинг 3) – след изпълнението на функцията нейната стекова рамка се освобождава и от стек паметта се „изтриват“ всички асоциирани с функцията данни заедно с копирания обект.



Фигура 3. Изтриване на споделени ресурси и странични ефекти в C++

За да се избегне копиране на обекти въобще при подаването им като параметри на функции, подаваме обектите не по стойност, а по адрес или като псевдоними. Така спестяваме ресурси (памет), тъй като за обекта в стековата рамка на функцията ще е необходимо само съхранение на адреса му. Вместо да декларираме функцията така: **`void`** `someFunction(School param);`, я декларираме **`void`** `someFunction(constSchool& param).`

```
void someFunction(School param){
cout << param;
}
int main()
{
Principal principal(„Иванов“);
```

```
Teacher * teachers = new Teacher[2];
teachers[0] = Teacher(„Димитрова");
teachers[1] = Teacher(„Христов");
    School originalSchool(„Георгиев", principal, 2, teachers);
     {
School copySchool = originalSchool;
cout << copySchool;
     //деструктуриране на обект copySchool и освобождаване
на паметта за неговия масив teachers
     }

      //при опит да се достъпи обекта, ако не сме имали
копиращ конструктор, програмата
     //ще аварира по време на изпълнение, тъй като целостта
на обекта е нарушена
      //при наличие на коректен копиращ конструктор, този
проблем не съществува
cout <<originalSchool;
some Function(originalSchool);
if(teachers!=nullptr) delete[] teachers;
teachers = nullptr;
}
```

Листинг 3. Напускане област на видимост след копиране на обект в C++

Предефиниране на оператор за присвояване в C++

Копиращият конструктор и операторът за присвояване се прилагат в различни ситуации. Ако при декларирането на обект използваме оператора равно, то ще се приложи по-горе разгледаният копиращ конструктор, а не присвояване. Например:

```
Stchool copySchool= originalSchool;
```

Операторът за присвояване се прилага, когато са декларирани два обекта и се извършва присвояване между тях. Например:

```
Stchool originalSchool;
Stchool copySchool;
copySchool= originalSchool;
```

Предефинираме оператора за присвояване, за да задължим компилатора да извършва дълбоко копиране на обекта. В C++ е позволено предефиниране на оператори, включително = и << (в листинг 1 е показано предефинирането на оператор <<).

```
School & operator =(const School & original){
if(this!=&original){
```

```
  name = original.name;
      numberOfTeachers = original.numberOfTeachers;
      principal = original.principal;
      if(teachers!=nullptr) delete[] teachers;
      teachers = new Teacher[numberOfTeachers];
      for (int index = 0; index<numberOfTeachers; index++) {
      teachers[index] = original.teachers[index];
          }
           }
  return *this;
        }
```

Листинг 4. Предефиниране на оператор = в C++

Клониране на обекти в C++

Клонирането на обекти в C++ се извършва при използване на наследяване, при което в дадена член-функция, която се предефинира (override) в наследяващия клас, се създава копие на обекта чрез указател към неговия базов тип. Клониращата член-функция се използва, когато не е предварително ясно от какъв тип ще бъде обектът, който ще се създава. Поради спецификата на езика C++ тук трябва изрично да се декларират член-функциите като виртуални, за да се реализира полиморфизъм. Клонирането на обектите се извършва от специално създадена член-функция, която се обръща към конструктора (Meyers, 2017). Разширяваме още веднъж примера за клас „Училище“, като създаваме един нов базов клас BaseSchool, в който са декларирани член-функция за клониране и виртуален деструктор (задължителен при реализиране на полиморфно поведение и йерархии от класове).

```
  class BaseSchool {
  public:
      virtual BaseSchool* clone() const = 0;
      virtual ~BaseSchool() = default;
  };
  class School : public BaseSchool {
  School* clone() const {
      return new School(*this);
     }
  }
  int main() {
  …
  School* school = new School(„Христо Ботев“, 2, principal,
teachers);
  School* schoolCopy = school->clone();
  …
  }
```

Листинг 5. Клониране на обекти в C++

Удачно е класът „Училище“ да съдържа реализация на член-функцията за клониране, което означава, че той извършва процеса на създаване на дълбокото копие на обекта.

3. Реализация на дълбоко копие в Java

Обикновено обектът-копие в Java се създава чрез използване на конструктора на класа и предаване на стойностите на член-данните на оригиналния обект. Това означава, че ако в нашия клас „Училище“ има член-данни, които са изменяеми (обект от клас „Директор“ (Principal)), те ще се използват едновременно от обекта копие и от оригиналния обект. Както се вижда от примера по-долу (листинг 6), при промяна на името на директора на обекта копие (school Copy) се променя и името на директора на оригиналния обект (school). Броят на класните стаи на оригиналния обект училище не се променя, защото е от примитивен тип int, а името на оригиналния обект училище не се променя, защото клас String е неизменяем. Ще имаме предвид, че при копирането на обекти в Java съществено значение има типът на всяка от член-данните на обекта, който се копира. Ако класът се състои от обекти от неизменяеми (immutable) класове, няма риск от ситуация с плитко копие. Ако класът се състои от обекти от изменяем тип (mutable), трябва да се реализират копия на всички обекти, от които се състои тази изменяема член-данна.

Особено внимание трябва да се обърне и на класове, които имат член-данни от тип „Колекция“ (Collection – съвкупност от елементи). Ако колекциите са изградени от изменяеми обекти, трябва да се осигури създаване на дълбоко копие на всички обекти и изграждащите ги обекти, както споменахме и в описанието на примера с клас „Училище“. Ако осигурим дълбоко копие само на колекцията, а не на всеки изменяем обект, който е член-данна на елемент от колекцията, ще се стигне до ситуация, в която колекцията на оригинала не се променя от обекта копие, но стойностите на член-данните на елементите в колекцията на оригинала се изменят от копието.

Ще продължим описанието на подходите за дълбоко копие с примера за клас „Училище“, който разгледахме по-горе.

Съществуват три основни подхода при имплементация на дълбоко копие в Java:

1. имплементиране на ***интерфейс Cloneable*** и предефиниране на метод clone();
2. създаване и използване на ***конструктор за копиране*** (Copy Constructor);
3. използване на ***сериализация*** (процес на запазване на състоянието на обекта) и ***десериализация*** (процес на възстановяване на състоянието на обекта) на данните.

```
public class Principal {
```

```
private String name;
// Конструктори, get и set методи
}
public class School {
private String name;
private int classrooms;
private Principal principal;
// Конструктори, get и set методи
}

public static void main (String[] args) {

Principal principal Ivanov = new Principal(„Иванов");
School school = new School(„Христо Ботев", 30, principal-
Ivanov);
System. out. println(„Оригиналният обект училище преди
промяната на копието: „);
System. out. println(school);
School school Copy = new School(school.getName(),
school.get Classrooms(), school.get Principal());

school Copy.setName(„Пенчо Славейков");
schoolCopy.setClassrooms(20);
schoolCopy.getPrincipal().setName(„Петров");
System.out.println(„Оригиналният обект училище след
промяната на копието: „);
System.out.println(school);
}
Резултат:
Оригиналният обект училище преди промяната на копието:
School{name='Христо Ботев', classrooms=30, principal=Prin-
cipal{name='Иванов'}}
Оригиналният обект училище след промяната на копието:
School {name='Христо Ботев', classrooms=30, princi-
pal=Principal{name='Петров'}}
```

Листинг 6. Плитко копие в Java

Интерфейс Cloneable и предефиниране на метод clone() в Java

Първият подход за реализиране на дълбоко копие, който разглеждаме, е с използване на ***метод*** **clone()** и имплементиране на ***интерфейс*** **Cloneable**. Необходимо е да декларираме, че класът „Училище“ ще може

да използва ***метод*** **clone()**, като използваме маркиращия ***интерфейс*** Cloneable. За тази цел клас „Училище“ имплементира интерфейс **Cloneable** и предефинира метод clone(), който е деклариран в базовия за всички класове в Java – клас Object. Всеки метод, който предефинира метод clone(), трябва да изхвърля изключение от тип CloneNotSupportedException и да реализира клонирането на всеки изменяем обект поотделно.

В нашия пример клас „Директор“ имплементира интерфейс Cloneable и предефинира метод clone(), за да може в метод clone() на клас „Училище“ да се използва principal.clone(). Ако клас „Директор“ имаше член-данна от изменяем тип, трябваше да изпълним същите стъпки и за нея. Това е така, защото метод clone() всъщност реализира плитко копие.

В листинг 7 се виждат промените, които е необходимо да се направят в клас „Училище“ и клас „Директор“, за да се реализира дълбоко копие чрез клониране.

В обобщение на описаното по-горе, ако се използва подходът на клониране на обекти, трябва да се извършат три основни стъпки.

1. Имплементиране на маркиращ интерфейс Cloneable от всички изменяеми класове: класът, който ще се копира, и всички негови изменяеми член-данни.
2. Предефиниране на метод clone() с изрично клониране на всеки изменяем обект, който е член-данна на класа, в който е методът.
3. Деклариране или прихващане на изключението CloneNotSupporte-dException при декларация на всеки метод clone()

```
public class Principal imp lements Cloneable{
@Override
protected Object clone() throws CloneNotSupportedException {
return super.clone();
}
public class School implements Cloneable{...
@Override
public Object clone() throws CloneNotSupportedException {
School school = (School) super. clone();
school. principal= (Principal) this.principal.clone(); //
Cloning Principal, becauseitismutableobject
return school;
    }
}
public static void main(String[] args) {…
School school Copy = (School) school.clone();
}
```

Резултат:

```
Оригиналният обект училище преди промяната на копието:
School{name='Христо Ботев', classrooms=30, principal=Prin-
cipal{name='Иванов'}}
Оригиналният обект училище след промяната на копието:
School{name='Христо Ботев', classrooms=30, principal=Prin-
cipal{name='Иванов'}}
```

Листинг 7. Дълбоко копие в Java – интерфейс Cloneable иметод clone()

Конструктор за копиране в Java

Аналогично на копиращия конструктор в C++ и в Java той има един аргумент от типа на класа, в който е реализиран и връща нов обект от същия тип със стойности на член-данните като тези на аргумента. При неговата реализация трябва да създадем конструктор за копиране във всеки изменяем клас, от който има член-данна в класа, от който ще създаваме копие. В примера това са класовете „Училище“ и „Директор“. В листинг 8 се виждат имплементацията на конструкторите за копиране на двата класа и начинът за използването им при създаване на копието.

```
public class Principal {…
public Principal(Principal principal) {
  this.name = pricipal.name;
   }
}
public class School {…
public School(Schoolschool) {
this.name = school. name;
this. classrooms= school. classrooms;
this.principal= new Principal(school. principal);
   }
}
public static void main(String[] args) {…
School school Copy = new School(school);
}
Резултат:
Оригиналният обект училище преди промяната на копието:
School{name='Христо Ботев', classrooms=30, principal=Prin-
cipal{name='Иванов'}}
Оригиналният обект училище след промяната на копието:
School{name='Христо Ботев', classrooms=30, principal=Prin-
cipal{name='Иванов'}}
```

Листинг 8. Дълбоко копие в Java чрез копиращ конструктор

Кодът, който е описан в листинг 8, работи коректно, ако не се използва наследяване на обектите, които са член-данни на класа „Училище“.

В следващата реализация ще включим клас „Заместник-директор“ (Assistant Principal), който наследява клас „Директор“. „Заместник-директор“ освен име има още една булева член-данна, която служи, за да покаже дали заместник-директорът отговаря за учебната дейност.

```
public class AssistantPrincipal extends Principal {
private booleanresponsible For LearningActivities;
    // Copy constructor
public AssistantPrincipal(AssistantPrincipal principal,
booleanresponsibleForLearningActivities) {
super(principal);
this.responsible For Learning Activities= responsible-
ForLearningActivities;
    }
}
public static void main(String[] args) {
School school = new School(„Христо Ботев“, 30, new As-
sistantPrincipal(„Иванов“, true));
School schoolCopy = new School(school);
System.out.println(school);
System.out.println(schoolCopy);
}
Резултат:
School{name='Христо Ботев, classrooms=30, princi-
pal='Иванов' AssistantPrincipal{responsibleForLearningAc-
tivities=true}
School{name='Христо Ботев', classrooms=30, principal=Prin-
cipal{name='Иванов'}}
```

Листинг 9. Наследяване в Java и копиране на обекти чрез конструктор за копиране

В резултат на изпълнение на кода (листинг 9) се вижда, че обектът копие няма „Заместник-директор“, а само „Директор“. Конструкторът за копиране в клас „Училище“ не знае за съществуването на обект от клас „Заместник-директор“. Проблемът се решава, ако в клас „Директор“ и в неговите наследници се реализира метод за създаване на копие на обект „Директор“ и този метод се извика в конструктора за копиране на клас „Училище“, вместо да се използва директно копиращият конструктор на клас „Директор“ (Bloch, 2017). Проблемът с копиращия конструктор в ситуация на йерархии от класове се идентифицира лесно поради факта, че кодът е в разрез с принципа „Отворен/Затворен“ (Open/Closed Principle), който гласи, че всеки клас трябва да е затворен за модифициране и отворен за разширяване (Shvets, 2019) (Martin, 2018).

Извод (за C++ и Java): при йерархии от класове и полиморфизъм е по-удачно да се използва допълнителен метод за клониране (раздел **Клониране на обекти в C++**), който да извиква конструктора за копиране вместо директно да се извиква конструктор за копиране.

Използване на сериализация и десериализация на данните в Java

Третият подход за копиране на обекти в Java, който ще разгледаме, използва ***сериализация*** и ***десериализация*** на данните. Сериализация е процес на запазване на състоянието на обект (стойностите на член-данните му) в двоична форма. Десериализацията е обратният процес: възстановяване на обект, като се използват сериализираните данни. Процесът на запазване и възстановяване на данните се осъществява с използване на два допълнителни метода към клас „Училище“. Практически е по-удачно те да се реализират в отделен клас, но за целите на демонстрацията ще бъдат създадени в класа.

Необходимо е класът „Училище“ да бъде обозначен с маркиращия интерфейс `Serializable`, за да можем да запазим състоянието му. При наследяване трябва маркиращият интерфейс `Serializable` да бъде имплементиран на най-високо ниво в йерархията, за да може всички данни във всичките му наследници да се запазят. Всички класове, чиито обекти са член-данни на класа „Училище“, също трябва да имплементират `Serializable`.

Сериализацията на данните може да бъде осъществена и чрез използване на външни библиотеки, например Jackson и Apache Commons (SerializationUtils). Тук трябва да се отбележи, че процесът на сериализиране и десериализиране е бавен.

```
public class School implements Serializable {
public void serializeSchool(StringfilePath) {
try(File Output Streamfos = new FileOutputStream(filePath);
ObjectOutputStreamoutputStream = new ObjectOutputStream(-
fos);) {
outputStream.writeObject(this);
    } catch(IOExceptionex) {
System. err. println(ex);
    }
}
public School deserializeSchool(StringfilePath) {
School school = new School();
try(FileInputStreamfis = new FileInputStream(filePath);
ObjectInputStreaminputStream = new ObjectInputStream(fis);)
{
return (School) inputStream.readObject();
    } catch (ClassNotFoundExceptionex) {
```

```
System.err.println („Class not found: „ + ex);
    } catch( IOExceptionex) {
System.err.println(„IO error: „ + ex);
    }
return school;
}
public School deepCopyBySerialization(StringfilePath) {
serialize School(filePath);
return deserializeSchool(filePath);
}
public static void main(String[] args) {
StringfilePath = „school.ser“;
School school = new School(„Христо Ботев“, 30, newPrinci-
pal(„Иванов“));
School schoolCopy = new School(school.deepCopyBySerializa-
tion(filePath));
}
```

Листинг 10. Сериализация и десериализация на данни в Java

Предимства и недостатъци на подходите за реализиране на дълбоко копие в Java

Трите основни подхода за реализиране на дълбоко копие в Java, които са описани в предходните раздели, имат своите специфики и се използват в различни ситуации. Техните предимства и недостатъци са обобщени в таблица 1.

Таблица 1. Предимства и недостатъци на подходите за дълбоко копие в Java

Дълбоко копие в Java	Предимства	Недостатъци
Клониране: интерфейс Cloaneable и метод clone()	1. Не се налага да се копират примитивните данни. 2. Кодът за реализиране на метода за клониране е компактен. 3. Използва се при клониране на масиви.	1. Трябва да се имплементира интерфейс Cloneable. 2. Трябва да се предефинира метод clone() в цялата йерархия на наследяването. 3. Нямаме контрол върху конструирането на обекта, защото не се извиква конструктор. 4. Полета, дефинирани като final, не могат да се манипулират, защото те се променят само през конструктор. 5. Трябва да се прихване изключението CloneNotSupportedException.

Конструктор за копиране	1. Не се налага да се имплементира интерфейс. 2. Кодът не зависи от непознат механизъм за създаване на обекта. 3. Могат да се модифицират полета, декларирани final.	1. В ситуация на наследяване конструкторът за копиране на класа наследник не се извиква, т.е. копието не получава обект от класа наследник, а обект от базовия клас. Необходимо е да се реализира метод за клониране, който да извиква конструктора за копиране. Трябва отговорността за копирането на обекта да се делегира на негов собствен метод. 2. Когато нямаме достъп до класовете, в които трябва да се реализира конструкторът за копиране, не можем да го създадем.
Сериализиране на данните	1. Може да се използва при наследяване. 2. Може да се използват външни библиотеки, които реализират дълбоко копие, без да имаме достъп до класовете, чиито обекти ще копираме.	1. Трябва да се предефинира интерфейс Serializable. 2. Механизмът е бавен. Не се използва, ако се цели бързина при изпълнение. 3. Трябва да се внимава в ситуация на промяна на полетата на класа и десериализация на обекта.

Заключение – реализиране на дълбоко копие в Java и C++

В настоящата статия са представени различните начини за осигуряване на дълбоко копие в Java и C++. Основните методи, които се използват и в двете технологии, са: конструктор за копиране, клониране на обекти и използване на сериализация на данните. С един и същ пример са представени специфичните реализации на дълбокото копие в двата езика за програмиране и са описани основните предимства и недостатъци на трите метода. Решението за това кой от подходите трябва да се избере, зависи от редица фактори:

– дали имаме достъп до кода, който трябва да се използва за копиране на обектите;
– дали бързодействието е много важно за реализацията на проекта;
– използва ли се йерархия от класове и полиморфизъм.

Освен посочените по-горе фактори роля играе и специфичната реализация на подходите в двата езика, както и различните ситуации, при които се извършва и неявно копиране на обекти или тяхното деструктуриране. След анализиране на всички предпоставки се избира един от посочените начини за реализиране на дълбоко копие, като се стремим да спазваме принципите на изграждане на лесен за разширяване и сигурен код (Martin, 2018) (Shvets, 2019).

ЛИТЕРАТУРА

Тодорова, М. (2011). *Обектно ориентирано програмиране на базата на езика C++*. Ciela.

REFERENCES

Bloch, J. (2017). *Effective Java.* Pearson Education, Inc.

Martin, R. (2018). *Clean Architecture. A Craftman's Guide to Software Structure and Design.* Pearson Education, Inc.

Meyers, S. (2017). *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition).* Pearson Education, Inc.

Shvets, A. (2019). *Dive Into Design Patterns.*

DEEP COPY IN C++ AND JAVA

Abstract. The programme realization topic of the ways to copy cloning objects in object-oriented software projects is presented in the paper. The stress is on the subject of shallow and deep copy or the situations in which the copied objects have common resources with the original one. In many cases, programmers need to switch from one technology to another, which leads to the need of knowing the details of the implementation of a programme functionality in more than one programming languages. Different approaches for the implementation of the deep copy are demonstrated with examples in C++ and Java programming languages. Advantages and disadvantages of the presented methods are discussed.

Keywords: shallow copy; deep copy; C++; Java

✉ **Dr. Hristina Kostadinova**
✉ **Dr. Mariyana Raykova, Assist. Prof.**
Department of Computer Science
New Bulgarian University
21, Montevideo Blvd.
1618 Sofia, Bulgaria
E-mail: hkostadinova@nbu.bg
E-mail: mraykova@nbu.bg