СТИЛЪТ НА ПРОГРАМАТА КАТО СРЕДСТВО ЗА ИЗБЯГВАНЕ НА ГРЕШКИ[1]

Теодоси Теодосиев

Шуменски университет "Епископ К.Преславски"
Факултет по математика и информатика
ул. "Университетска" 115, Шумен
t.teodosiev@fmi.shu-bg.net

Абстракт: *Представената работа разглежда въпроса за ролята на обучението в стил на програмиране за локализиране и избягване на грешки при програмиране. Това е особено важно за начинаещи програмисти. Коментирани са елементите на добрия стил и как да го постигнем. Разгледана е ролята на стила на програмиране в един от аспектите на влияние върху качеството на програмата, а именно минимизиране на опасността от грешки и подпомагане на откриването им. Показани са примери за това как стилът прави "очевадни" някои грешки и насочва вниманието ни към предпазване от допускане на обичайни грешки.*

Ключови думи: *програмиране, език за програмиране, обучение, стил на програмиране.*

Увод. Добрият стил на програмиране е субективен въпрос и е трудно да се определи. Все пак, има няколко елемента, общи за голям брой програмни парадигми[2]. Те са резултат от договорки между програмистите. Счита се, че в съответствие с правилата на добрия стил програмния код трябва да поддържа: *очевидна логика; естествени изрази, отсъствие на хитри трикове и необичайни конструкции (използването или избягване на определени конструкции като например GOTO); акуратно форматиране (използване на отстъпи и празни редове, употребата на интервали около оператори и ключови думи, подчертаването (отделянето) на ключови думи); използването и стил на коментари, използване на носещи смислова стойност имена на променливи и функции, стила и правописа на дефинирани от потребителя функции и процедури* [7]. Накратко казано: *простота, прегледност, коментиране и стандартизация*.

1. **Стил на програмиране.** Както е отбелязано в [1] главният тезис се състои в това, че стилът на програмиране – това е стил на мислене, проявяващ се в умението да се изобрази алгоритъма за решение на задача на конкретен език за програмиране. За съжаление гледането на стила на програмиране като на стил на мислене не е осъзнат докрай и от всички. Много често стилът на програмиране се свежда до технология на програмирането. В литературата се обръща се внимание главно на графичното оформяне на програмния код. Акцентът пада върху прегледността (отстъпите, интервалите, празните редове, дължината на кода) и стандартизацията на оформяне на кода (избор на имена, коментари и др.). Но очевидно, "лошо" разработена програма, записана и по правилата на структурното програмиране със спазване на изискванията за подреждане на кода, си остава "лоша" програма.

Ние сме свикнали с понятието „математическа култура" и добре разбираме какво се крие зад него. Точно по същия начин има смисъл да се говори за "програмистка култура", лежаща в основата на стила на програмиране.

Какво се нарича добър стил на програмиране, не е възможно да се дефинира строго и все пак: стилът на програмиране е резултат на дългогодишния опит на програмиста и е почти сигурен белег, по който може да се различи професионалния програмист от любителя и дилетанта.

Същността на добрия програмен стил е комуникацията. Добрият стил на програмиране е трудно да се научи, както добрия стил на английския (или всеки друг естествен) език. И в двата случая, документът няма никаква стойност, ако не изпрати своето послание на читателя. Всяка програма, която се използва, трябва да се поддържа от някого – както и някой трябва да бъде в състояние да разбере кода, като го прочете. Всяка програма, ще бъде по-лека за отстраняване на грешки, ако създателят внимателно обяснява какво се случва.

В рамките на програмния текст, програмистите имат три основни средства за комуникация на намеренията си: коментари (обяснение за участие в програмата); ясни имена на променливи, константи, изрази и подпрограми (чрез думите на самата програма) и празно пространство и привеждане в

[1] Работата е частично финансирана от фонд "НИ" на ШУ.

[2] Парадигма на програмиране — това е съвкупност от идеи и понятия, определящи стила на написване на програма. Парадигмата, най-напред се определя като базова програмна единица и самият принцип на достигане на модулност на програмата. В качество на такава единица се явява определението (декларативно, функционално програмиране), действие (императивно програмиране), правило (продукционно програмиране), диаграма на преходите (автоматно програмиране) и др. Важно е да се отбележи, че парадигмата на програмиране не се определя еднозначно от езика за програмиране – много съвременни езици за програмиране се явяват мултипарадигмени, т.е. допускат използване на различни парадигми.

съответствие (организацията на думите в програмата). Всеки един от тези аспекти помага на комуникацията между създателя и читателя на програмата [5].

2. **Ролята на стила на програмата за избягване на грешки.** В изкуството стилът има роля само за класификацията на творбите и творците по определени характеристики. В програмирането освен тази функция, стилът носи и практически ползи: има препоръки, които правят четенето на програмата по-леко и ясно (без да влияят на верността и ефективността); **препоръки, които минимизират опасността от грешки, а понякога и са задължителни за избягване на грешки;** препоръки за ефективност (по време и ресурси). Както по-горе беше казано най-често коментирани в литературата са препоръките от първия вид. В предишна наша работа [4] са представени препоръки от третия вид. Тук ще се спрем на втория вид препоръки – как стилът на програмиране може да направи "очевадни" грешки, често допускани от "новаците" или поне да заостри вниманието ни върху "опасни участъци" от програмата.

Когато започвате работа като начинаещ програмист или пробвате да четете код на нов език за програмиране всичко ви изглежда еднакво непостижимо. Докато не разберете непосредствено езика за програмиране, вие не можете да видите очевидните синтактични грешки. В течение на първата фаза на изучаване вие започвате да разпознавате неща, които обикновено се наричат "стил на кодиране". Започвате да забелязвате код, който не съответства на строгите стандарти. Доколкото набирате все по-голям опит в работата в конкретна среда на разработка, се учите да виждате и други неща. Неща, които могат да са съвсем верни от гледна точка на използвания език за програмиране и абсолютно съответни на договорките за кодиране, но които ни карат да сме нащрек.

Подходящият избор на имена на променливи се разглежда като крайъгълен камък за добър стил. Добрият стил на програмиране изисква да се използват идентификатори с максимална изразителност, които имат висока мнемонична стойност, т.е. подсещат програмиста за ролята на именувания обект. Идентификаторите (т.е. имена на променливи, функции, типове, структури и т.н.), съобщават чрез името на обекта, функционалната му роля на читателя.

За пример, да погледнем следните фрагменти:

```
int a, b, c;
 if (a < 24  &&  b < 60 &&  c < 60 ) return 1; else return 0;
```

От избора на имената на променливите, значението на кода е трудно да се разбере. Въпреки това, ако имената на променливите са направени по-описателни:

```
 int  hours, minutes, seconds;
  if ( hours < 24 &&  minutes < 60 &&  seconds < 60) return  true; else return  false;
```

значението на кода е по-лесно да се различи, а именно: "Ако времето в 24-часов формат е валидно време, ще бъде върнато вярно, и невярно в противен случай". Неудачно избраното име на променлива прави кода по-труден за четене и разбиране.

Добре е, да избираме идентификатори по такъв начин, че грешка при писане на името, да не доведе до друг идентификатор, използван в нашата програма. По този начин, правописна грешка ще бъде уловена като грешка от компилатора (недекларирана идентификатор), вместо да доведе до трудна за намиране логическа грешка.

Като цяло, едносимволният идентификатор е лош избор, тъй като нарушава горните правила. Променливите *l, o* и *p* не са много информативни, и грешно преписаното *l* лесно може да се превърне в *1*, а грешно преписаното *o* лесно може да се превърне в 0 или *p*. Въпреки това, е традиционно ползването на едносимволни имена на променливи. В някои математически изрази, е уместно да се използват традиционните имена на променливи като *x* или *y*. Също така е подходящо да се използва *i, j* и *k* за параметри на цикъл *for* и индекси на масиви – но не за други цели!

Друг въпрос е изборът на кратки идентификатори, дългите имена могат да предизвикат сериозни затруднения при въвеждане на изходния код на програмата, независимо, че говорят повече. Този проблем го има още при преписването на текста от учебника или бялата дъска. Дългите имена водят до оператори на повече от един ред, което затруднява четенето. От друга страна значещите имена могат да заменят до голяма степен коментарите!?

Според някои автори не е особено трудно да се овладее една тясна английска лексика, която да се използва за именуване на обектите. По мое мнение за предпочитане е имената да са на български. По-лесно е за разбиране от новаците, но трябва да се изписват с латиница.

Важен елемент на програмния стил са коментарите. Коментарите трябва да пояснява само неща, които не са очевидни и да помагат на програмиста да се ориентира в алгоритъма и действието на програмата. Като цяло, програмната документация трябва да включва коментари, които пояснява какво прави всяка подпрограма, за какво е всяка променлива, както и обяснение за всеки сложен израз или част от код.

Но има много добър стил и отвъд коментарите, както ще видим. Всеки път, когато декларираме променлива, е добре да се включва коментар. Всеки път, когато се започва писане на подпрограма, първо пишем коментар, за да я обясним. Това ни помага да си изясним за себе си това, което предстои да се направи. Ако срещнем трудности при писането на подпрограмата, това е сигурен знак, че сме направили лошо структуриране на програмата. Всеки път, когато напишем сложен израз, ако ни е било трудно да го съставим, трябва да очакваме, че ще бъде трудно да се разбере. Значи, нужен е коментар [5]. Следователно писането на коментари освен за читателя е много важно и за създателя на програмата. В резултат на обмислянето на коментара, той прави корекции и подобрения в програмата.

Силно интригуващ е въпросът, възможно ли е на порядък да се увеличат програмистките ни способности и какви методи (интелектуални, организационни или механични) трябва да се приложат за това към процеса на съставяне на програми. Работейки, върху него получаваме значително по-дълбоко разбиране за природата на трудността на задачите по програмиране, ставаме значително по-съзнателни в своя стил на програмиране, който много се подобрява, и откриваме, че много по-добре от преди контролираме, какво правим. Да не говорим, че това силно повлиява и преподавателската дейност.

Както казва в [3] Дейкстра *„... моята основна цел при обучението по програмиране е да науча студентите отначало да мислят, а не да бързат да се хвърлят в кодиране..."* . Видимо е, че начинаещите програмисти трябва достатъчно дълго да се обучават "с молив и лист" на елементи на логиката, правилно построяване на цикъл и т. н. преди да ги пуснем пред клавиатурата със задание да напишат работеща програма.

3. **Обучение в стил на програмиране.** Отчитайки известната истина, че пренаучаването е много трудно и бавно, то обучението в стил трябва да върви паралелно с усвояване на алгоритмите и езика за програмиране. Това прави много важен избора на първи език за програмиране. Изборът на първи език за програмиране и стила на програмиране имат помощна но не и маловажна роля в обучението по програмиране. С тези „патерици" можем да преодолеем препятствията, а ако те не са подходящи ни остава притеснението да паднем по пътя. В този контекст се повишават изискванията и към инструмента, който ще се използва.

Има особености на стила, които биха подпомогнали търсенето на грешки в програмата. Стилът може да ги прояви (да ги направи видими) за новака.

✓ Ляво сравнение. В езиците, които използват един символ *(=)* за присвояване и друг *(==)* за сравнение (например C/C + +, Java, PHP,Perl), когато присвояванията могат да бъдат направени в рамките на структурите за контрол, предимство е мястото на константи или изрази да е вляво на всяко сравнение. Изразите с ляво сравнение *(2==к)* в C/C++ биха предизвикали грешка при компилация при пропускане на едното = (обичаен пропуск при новаците), което няма да се случи при дясното (стандартно) сравнение.Тук компилаторът не открива грешка и тя става трудна за локализиране.

✓ Скоби след структури за контрол.

```
if (i != 0)
  run(i);
```

В този случай кодът е 100% правилен и съответства на договорките за кодиране, но факта, че тялото на оператора *if*, съдържащо само един ред, не е заградено във фигурни скоби може да ни застави да помислим подсъзнателно, ами ако някой добави още един оператор

```
if ( i != 0)
   repeat(i);
   run(i);
```

и забрави да постави фигурните скоби и така направи *run(i);* да се изпълнява безусловно? И така когато видим блок в кода, незатворен във фигурни скоби, бихме могли да почувстваме съвсем леко подозрение за нередност, което може да ни безпокои.

Препоръчително е да се иизбягват неясни езикови конструкции и уповаване на специфичните езикови предимства. Въобще, малко притеснителни са езиковите особености, които помагат нещо да се скрие. Често е по-добре да се използват излишни скоби, тъй като това не оставя съмнение за смисъла.

Има няколко правила за добро използване на променливи. Лесно е за следване:

✓ Не променяйте стойността на параметър на цикъл в рамките на този цикъл. С други думи, не бива да се прави следното:

```
for  i:=1 to n do
    begin i: =i+1 end;
```

На някои езици за програмиране компилаторът го приема за грешка или това предизвиква зацикляне (например в Паскал). В други това не предизвиква проблем, но все пак е добре да се избягва при началното обучение.

Това е едно проявление на общия принцип за употреба на променлива: да не се претоварва логическия смисъл на променливата. С помощта на *i* като параметър на цикъл, програмистът сигнализира на читателя нещо за *i.* Чрез свързване на променливата е въведено второ значение. Програмата е по-лесна за разбиране, ако се свърже и се използва друга променлива.

Стилът може да се използва като средство за избягване на недостатъците на езика за програмиране. В [4] са показани примери за правилно подреждане на условията при изчисляване на конюнкция и дизюнкция с цел избягване на излизане от границите на масив или изчисляване на стойност на функция в недефинирана област, с което се предпазваме от грешки. Ето още два примера за често срещани и трудни за откриване от новаците грешки:

✓ Внимание при използване на изрази, включващи операция деление с цели операнди, защото резултатът е цяло число.

```
int n; double s=0;
cin>>n;
for (int i=1;i<=n; i++)
s=s+1/i;
cout<<s<<endl;
```

Резултатът независимо от *n* е *1*, което се дължи на целочисленото деление.

✓ *Избягване на смесване на типовете данни.*

```
int r, a, d, n;
 double p;
  cin>>a>>r;
  cin>>d;
  p=6*r*a / 2 - 3.14*d*d / 4;
  cout<<p<<endl;
```

При пресмятане на стойността на *p* израза *6*r*a / 2* е от тип *int*, което създава поне два проблема: при произведение, надхвърлящо границите на *int* се получава грешен резултат и после при делението, резултатът е също от тип *int*.

Преднамереното построяване на кода си така, че чувствителността ни към прецизността да бъде подразнена, прави голяма вероятността, нашият код да работи правилно.

Това е истинско изкуство: да създаваш "здрав код", буквално в движение изобретявайки договорки, които заставят грешките да изпъкват на екрана.

Обезпечението, че "лошия" код изглежда опасен – е прекрасен подход, но това не е задължително най-доброто от възможните решения за всички проблеми на безопасността. Това няма да покаже всички възможни грешки и недостатъци, защото ние не можем да прегледаме всеки ред от кода. Но със сигурност това е дяволски по-добре от нищо, и силно препоръчително да има договорки за кодиране, при съблюдаването на които "опасния" код изглежда неправилен. Получаваме изгода всеки път когато очите на програмиста пробягват по редовете на кода и той се убеждава, че специфичната грешка е проверена и предотвратена.

Смисълът от създаване на правила, при съблюдаването на които неправилния код изглежда неправилен се заключава в това, да поставим нужните неща колкото се може по-близо едно до друго на екрана. Когато преглеждаме код с цел откриване на грешки, трябва да знаем във всеки момент опасна или безопасна е променливата. Да се разположат зависимите редове от кода колкото е възможно по близо. Това увеличава шансовете, нашите очи да бъдат в състояние на обхванат всичко необходимо за разбирането. Не искаме тази информация да се намира в друг файл или на друга страница, където да я търсим. Трябва да видим всичко точно тук и в това е смисъла на правилата за именуване на променливите [6].

4. Заключение. Стилът на програмиране има отношение към съставянето на алгоритъма (постановката на мислене, добър алгоритъм), избора на инструмент (първи език за програмиране, правилно използване на езиковите конструкции), конструиране и оформяне на програмата (съобразяване с особеностите на компютърните пресмятания, използване на отстъпи и празни редове, избор на имена, коментари и др.). Всеки следващ елемент прибавен към предните подобрява качеството на програмата.

Подходът да напишем наистина надежден код се състои в това да се опитваме да използваме прости инструменти, които отчитат типичната човешка ненадеждност, а не сложни инструменти със скрити странични ефекти и неясни абстракции, които изискват безгрешност от програмиста.

Литература:

[1]. Боровин Г., и др., Ошибки-ловушки при програмировании на Фортране, „Наука", Москва, 1987

[2]. Боровски Б. и др., Справочник по изчислителна техника- програмиране и програмно осигуряване на ЦЕИМ, Техника, София,1990

[3]. Дейкстра Е., Почему программное обеспечение такое дорогое?

[4]. Теодосиев Т., Теодосиева Г., Математически проблеми в обучението по програмиране, Сборник доклади на Научно-приложна конференция „Математика, Информатика и Компютърни науки", В.Търново,2006, 458-462

[5]. Elements of programming style, Department of Computer Science Computer Program Documentation Standards: Version 1.3,
http://ei.cs.vt.edu/~cs2604/Standards/Standards.html

[6]. Spolsky Joel,Making Wrong Code Look Wrong, Joel on Software, 2005

[7]. http://en.wikipedia.org/wiki/Programming_style#Elements_of_good_style