Зборник радова Факултета техничких наука,Нови Сад

UDK: 004.9
DOI: https://doi.org/10.24867/02BE39Ivetic

ФУНКЦИОНАЛНА ПАРАДИГМА У ПРОГРАМСКОМ ЈЕЗИКУ C#

FUNCTIONAL PARADIGM IN C# PROGRAMMING LANGUAGE

Марко Иветић, *Факултет техничких наука, Нови Сад*

Област – Рачунарство и аутоматика
Кратак садржај – *Тема рада је представљање концепата функционалног програмирања. У оквиру рада, поред описа самих концепата, приказани су и примери њихове употребе у програмском језику C#.*
Кључне речи: *Функционално програмирање, Објектно-оријентисано програмирање, Ламбда изрази, Чисте функције, Непроменљивост*
Abstract – *The subject of this paper is presenting concepts of functional programming. Beside the description of concepts, it also contains examples of their usage in C# programming language.*
Keywords: *Functional programming, Object-oriented progr., Lambda expression, Pure functions, Immutability*

1. УВОД

Програмирање је термин под којим се најчешће подразумева креирање рачунарских програма, што укључује детаљну теоријску разраду проблема, проналажења концептуалног решења и имплементацију коришћењем неког од програмских језика. Програмски језици [1] су формални језици који се могу користити за контролу понашања машина, нарочито рачунара. Програмски језици се користе да олакшају комуникацију са рачунаром приликом организовања и манипулације информација, али и прецизно изразе алгоритме.

Поред програмских језика развијено је и неколико програмских техника (парадигми) за писање програма.

Програмска парадигма [2] представља основни стил програмирања, служећи као начин прављења структуре и елемената рачунарских програма. Различити програмски језици подржавају различите парадигме програмирања. Већина језика подржава само једну парадигму, али има и језика који могу подржати и више парадигми.

Програмске парадигме се могу поделити у две групе, императивну и декларативну. Код императивне, у коју спадају процедурално и објектно-оријентисано програмирање, акценат је стављен на начин на који ће се задатак обавити. Код декларативне групе, у коју спадају функционално и логичко програмирање, акценат је на самом задатку и томе шта је потребно обавити, а не и како.

НАПОМЕНА:
Овај рад проистекао је из мастер рада чији ментор је био др Александар Купусинац.

2. УВОД У ФУНКЦИОНАЛНО ПРОГРАМИРАЊЕ

У рачунарству, функционално програмирање [3] је програмска парадигма која третира програм као израчунавање математичких функција и избегава стања и промењљиве податке. Посматра појединачне операције као израчунавање математичких функција. Супротно од стила императивног програмирања, код кога је акценат на промени стања, код функционалног програмирања акценат је стављен на примену функција. Спада у декларативне програмске парадигме јер акценат ставља на то шта посматрани код, односно апликација ради, а не како ради. Развојем математичког стила програмирања долази до удаљавања од конкретне рачунарске архитектуре која извршава програм. То за последицу има слабије перформансе функционалних језика у односу на императивне, те су они мање заступљени у развоју комерцијалних софтверских решења.

Предност функционалних језика је што су они ближи људском размишљању него рачунарима. У пракси, разлика између математичке функције и појма функција који се користи у императивном програмирању је у томе што императивне функције могу да имају бочне ефекте, јер могу да промене вредности већ извршених израчунавања. Због тога, њима недостаје референцијална транспарентност, другим речима, исти израз може да доведе до различитих резултата у различитим тренуцима, у зависности од стања програма који се извршава.

Са друге стране, у функционалном коду, излазна вредност функције зависи само од аргумената који се проследе функцији, па ће функције f два пута, са истом вредношћу аргумента x произвести исти резултат. Елеминисањем бочних ефеката, разумевање и предвиђање понашања програма може да постане много лакше, и ово је једна од кључних мотивација за развој функционалног програмирања. Елиминација бочних ефеката подразумева елиминацију измене вредности претходно израчунатих резултата (*mutable data*) и стања апликације.
Корени функционалног програмирања леже у ламбда рачуну [4], формалном систему развијеном током 1930-их (развио их је Алонзо Черч) ради проучавања дефиниције и примене функција и рекурзије. Ламбда рачун пружа теоријски оквир за израчунавање и описивање функција.
Иако се ради о математичкој апстракцији а не о програмском језику, он представља основу скоро свих модерних функционалних програмских језика.

Ламбда рачун се користи у рачунарству због своје корисности у приказивању функционалног размишљања и итеративне редукције. Ламбда рачун садржи два упрошћавања која чине семантику једноставном. Поједностављује израчунавање уводећи неименоване функције и *curring.* Прво поједностављење - ламбда рачун третира функцију „анонимно“, без давања експлицитних имена. Друго поједностављење - ламбда рачун користи само функције једног улаза. Обична функција која захтева два улаза, може се написати као једна еквивалентна функција која прима један улаз, а да излаз враћа другу функцију, која заузврат прима један улаз.
У функционалном програмирању [2] се све заснива на идеји композиције једноставних функција, како би се добила комплексна структура програма, док се код објектног програмирања све заснива на композицији објеката (граф објеката).
LINQ [5] је базиран на композицији функција. *LINQ* (*Language Integrated Query*) је језик структурираних упита за претраживање, како локалних колекција објеката, тако и удаљених извора података, на такав начин да не нарушава безбедност података.
Он омогућава скраћено писање кода у раду са било каквим колекцијама и динамичким састављањем упита. Омогућава рад са различитим изворима података коришћењем истог начина кодирања.

2.1 Разлике функционалног у односу на објектно-оријентисано програмирање

Оно што је главна разлика ова два начина програмирања је сама чињеница да они представљају две различите парадигме у програмирању.

Функционално програмирање спада у декларативну (фокус на томе шта се ради) а објектно-оријентисано у императивну (фокус на то како се ради) програмску парадигму. Затим, код ООП-а редослед операција је изузетно битан. Када се измени стање неког објекта од стране различитих делова апликације, извршавање одређених функционалности може директно да зависи од стране њиховог редоследа.

Такође, са становишта једног дела апликације, неки други део апликације може објекат оставити у невалидном стању, што доводи до грешке, односно негативног *side* ефекта.

3. ОСНОВНИ КОНЦЕПТИ ФУНКЦИОНАЛНОГ ПРОГРАМИРАЊА

Функционално програмирање уводи неколико фундаменталних принципа, базирајући се на математичким основама ламбда калкулуса. У овом поглављу су представљени ти принципи, односно основни концепти функционалне програмске парадигме.

3.1 Функције вишег реда (Higher Order function)

У функционалним језицима, функција је основни градивни блок кода (за функцију се каже да је *first-class citizen*). Као што се у објектно-оријентисаним језицима креирају и користе објекти, тако се на исти начин у функционалним језицима креирају и користе функције. Једноставно речено, функција се може креирати, доделити некој променљивој, проследити другој функцији као аргумент или се вратити из друге функције као повратна вредност.

Функција која може да прихвати неку другу функцију као свој аргумент је функција вишег реда [6]. То је важан концепт који омогућава једноставну и природну имплементацију инверзије контроле, који је један од најважнијих принципа у развоју софтвера. Према концепту инверзије контроле, објекти не треба да креирају друге објекте од којих зависе, већ те објекте добијају екстерно.

3.2 Непроменљиви подаци (*Immutability*)

У императивном програмирању, варијабла може имати вредност, другачију у односу на прослеђену. Такође *for*, *while* и *do while* петље захтевају *mutable* варијабле чије промене доводе до услова за прекид и излазак из петље. У функционалном програмирању стање објекта се не мења. Уколико постоји потреба да се промени стање неког објекта, прави се копија оригиналног објекта и затим се та копија мења уместо промене самог оригиналног прослеђеног објекта.

3.3 Чисте функције (*Pure functions*)

Чиста функција [7] је функција која нема бочне ефекте, која може да се позове велики број пута и да сваки пут врати исти резултат, који само и једино зависи од аргумената те функције, а не од глобалног стања. Користи вредности својих улазних параметара и не приступа нити мења вредности других променљивих. Самим тим се избегава нежељени *side* ефекти који могу проузроковати *bug*. Чињеница да функције у функционалном програмирању конзистентно враћају резултат за исте улазне вредности, оставља се слобода кориснику таквих функција да их позива као „црну кутију“.

3.4 Лењо израчунавање (*Lazy evaluation*)

Лењо израчунавање [8] је стратегија израчунавања која одлаже само израчунавање израза све док његова вредност није неопходна програму, и која такође избегава поновно израчунавање. Лењо израчунавање је врло често комбиновано са мемоизацијом. Након што је вредност функције израчуната за вредност једног или више улазних параметара, резултат се смешта у упоредну табелу (колекцију). Следећи пут када се та функција позове, претражује се колекција како би се видело да ли је резултат за те улазне параметре већ израчунат. Ако јесте, резултат из колекције се просто врати позиваоцу функције.

3.5 *Pattern matching*

Усклађивање структуре објекта са унапред одређеним обрасцима, тако да се може наставити са израчунавањем само ако је објекат у складу са одређеном структуром и вредностима.

3.6 Референцијална транспарентност (*Referential transparency*)

Постоји један важан аспект чистих функција који произилази из чињенице да оне зависе само од њихових аргумената, а то је референцијална транспарентност [9]. Референцијална транспарентност омогућава да се безбедно замени читав израз са својом одговарајућом вредношћу у свим каснијим референцама на њега, без промене понашања програма.

3.7 Рекурзија

Понављање делова програмског кода у смислу петљи не постоји у функционалним језицима, него се такви процеси решавају рекурзијом. Рекурзија може попримити многе облике у функционалним језицима и сматра се да је ефикаснија од механизама петљи у императивним језицима. То је и један од разлога што и већина императивних језика подржава рекурзију.

4. МЕТОДЕ И ПРИМЕРИ КОРИШЋЕЊА КОНЦЕПАТА ИЗ ФУНКЦИОНАЛНОГ ПРОГРАМИРАЊА У ООП-У НА ПРИМЕРУ C#

Сви претходно наведени концепти функционалног програмирања имају своје репрезенте у *C#*-у, што значи да се могу користити без оклевања. Међутим, *C#* је објектно-оријентисан језик и без обзира колико су концепти функционалног програмирања привлачни они нису природни за *C#* и морају се користити на адекватан начин.

У функционалном програмирању контрола тока извршавања нема *if-then-else*, нема петљи, него постоји *pattern matching*. Функција добије аргументе и ти аргументи се упоређују са очекиваним шаблонима. За сваки шаблон се зна шта треба да се уради. Направи се унија тих шаблона за које се зна, ако је она једнака домену функције, онда је та функција тотална, дефинисана за сваки аргумент. Било шта да позивалац функције проследи, функција ће моћи да произведе резултат. Слика 4.1 приказује функцију за израчунавање Фибоначијевих бројева, која је базирана на рекурзивним позивима.

```
using System;

namespace Demo
{
    0 references
    class Program
    {
        3 references
        static long Fibonacci(int n) =>
            n < 2
                ? n
                : Fibonacci(n - 1) + Fibonacci(n - 2);
        0 references
        static void Main(string[] args)
        {
            for (int i = 30; i < 43; i++)
            {
                Console.WriteLine($"{i}\t{Fibonacci(i)}");
            }

            Console.WriteLine("\nDone");
            Console.ReadLine();
        }
    }
}
```

Слика 4.1 Једноставна функција за израчунавање Фибоначијевих бројева

Код са Слике 4.1 даје исправне резултате, међутим време извршавања овог кода, који се може сматрати једноставним, траје веома дуго. Због рекурзивних позива, ова функција дуго ради, иако је сама по себи доста тривијална. И управо овај пример показује како се понашају перформансе функција у пракси. Ова функција ће рекурзивно, за пример *Fibonacci* од 10 (*f(10)*), позвати *f(9) + f(8)*, затим да би израчунала *f(9)* мора да позове *f(8)* поново и *f(7)*, што намеће то да ће се онај крајњи *f(1)* позвати велики број пута. Једини битан елемент, везан за овај пример, је тај да се примети да ће тај *f(1)* који се позива велики број пута, сваки пут вратити исти резултат. У функционалном програмирању се таква функција зове „Чиста функција“.

У чистим функционалним језицима, који гарантују да су све функције чисте (*Pure*), постоји један принцип који се зове мемоизација. Мемоизација [10], у рачунарству, је техника оптимизације која се користи да би се убрзало извршавање рачунарских програма тако што се складиште резултати „скупих“ позива функција и враћају кеширани резултат када се поново јаве исти улази.

Ефекти коришћења мемоизације су најчешће позитивни и утрошак меморије је у прихватљивом обиму. На слици 4.2 приказан је пример функције за израчунавање Фибоначијевих бројева која користи кеширање тј. принцип мемоизације.

```
static IDictionary<int, long> dynamicCache = new Dictionary<int, long>();

3 references
static long DynamicFibonacci(int n)
{
    long value;
    if (!dynamicCache.TryGetValue(n, out value))
    {
        value = n < 2
            ? n
            : DynamicFibonacci(n - 1) + DynamicFibonacci(n - 2);
        dynamicCache[n] = value;
    }

    return value;
}
```

4.2 *Full memorization*

Динамичко програмирње [11] је метод за решавње сложеног проблема, тако што се он „разбије“ у колекцију једноставних и сваки од подпроблема се решава само једном и притом се решења чувају. У функцији *DynamicFibonacci*, свака појединачна ставка листе је израчуната на захтев, и не више од једном. Подаци се кеширају као парови кључ-вредност где је кључ примљени аргумент функције а вредност, повратна вредност те функције. Када тражени кључ у кешу не постоји, позива се целокупно тело функције. На крају функције, повратна вредност се сачува у кешу. На тај начин кеш је универзалан, независан од функције.

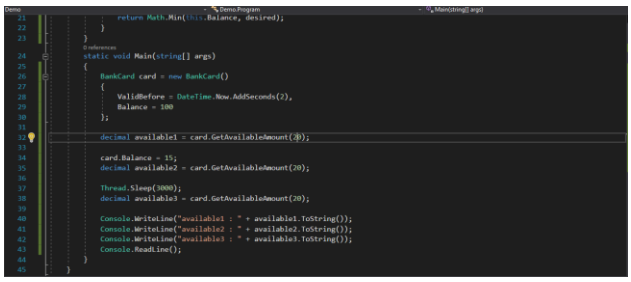
Следећи пример показује како стање објекта може утицати на повратну вредност једне функције. На слици 4.3 је приказан код који има *side* ефекат.

```
namespace Demo
{
    0 references
    class Program
    {
        2 references
        class BankCard
        {
            2 references
            public DateTime ValidBefore { get; set; }
            3 references
            public decimal Balance { get; set; }

            3 references
            public decimal GetAvailableAmount(decimal desired)
            {
                if (DateTime.Now.CompareTo(this.ValidBefore) >= 0)
                    return 0;
                return Math.Min(this.Balance, desired);
            }
        }
        0 references
        static void Main(string[] args)
        {
            BankCard card = new BankCard()
            {
```

Слика 4.3 *BankCard* класа

Следећа слика (4.4) приказује како се функција *GetAvailableAmount* може позивати са истим аргументом али сваки пут да врати другачији резултат.



Слика 4.4. Позивање методе *GetAvailableAmount*

На овом примеру се може видети како стање једног објекта може утицати на повратну вредност функције. Постоје три позива функције *GetAvailableAmount* и сва три позива враћају различити резултат. Први пут ће функција вратити 20, други пут 15, док ће трећи пут вратити 0, и то је типична ситуација у објектно-оријентисаном програмирању, која се избегава у функционалном. Гледано са стране функционалног програмирања, функција је чиста ако зависи само од улазних вредности и нема видљивог нежељеног ефекта. Функција *GetAvailableAmount* није чиста, зависи од променљивог стања објекта, који се може променити између позива.

Решење овог проблема би било да се уклони зависност од променљивих стања, или да се стања учине непроменљивим. Оба *property*-ја могу постати *read-only* и константни за цео „животни век“ објекта, а време које се проверава у методи, може бити прослеђено као аргумент.

На тај начин се метода чини чистом. Остатак кода, који инстанцира објекат класе *BankCard* и позива њену функција се неће компајлирати више. *Immutable state* (непроменљиво стање) је неопходно уколико је циљ да функција врати увек исти резултат за исте вредности аргумената.

Структура која се користи у *Immutable* колекцијама је *ImmutableList. ImmutableLis* је балансирано *AVL* стабло које гарантује да је дубина до *logaritam(n)*. Балансирање се врши динамички. Када постане „нагнуто“ на једну страну, онда се оно динамички ребалансира.

5. ЗАКЉУЧАК

Већини програмера објектно-оријентисана парадигма је једино позната. Разлог томе је мала примена функционалне парадигме у развоју модерних софтверских решења, пре свега због слабијих перформанси функционалних језика. Међутим, данас, уз јефтине рачунаркске компоненте и већу процесорску моћ рачунара, проблем перформанси функционалних језика је лако премостив.

Елементи функционалног програмирања су корисни, али исто тако не треба уклонити оно што је добро у објектно-оријентисаном програмирању. Нити је функционално програмирање „лек за све болести“, нити је то објектно-оријентисано.

Обе парадигме имају своје предности и мане, па примена концепата из функционалног програмирања на на местима где објектно заостаје, може представљати крајњи циљ који ће умногоме олакшати и улепшати код.

Коришћењем концепата из функционалног програмирања омогућава код са мање *bug*-ова, где се лакше и безбедније праве измене због *mutability* концепта и елиминације *side* ефеката. У оквиру рада, представљени су концепти функционалног програмирања и примери њихове употребе у објектно-оријентисаном језику као што је *C#*.

6. ЛИТЕРАТУРА

[1] https://en.wikipedia.org/wiki/Programming_language - Programming language

[2] https://en.wikipedia.org/wiki/Programming_paradigm - Programming paradigm

[3] https://en.wikipedia.org/wiki/Functional_programming - Functional programming

[4] https://en.wikipedia.org/wiki/Lambda_calculus - Lambda calculus

[5] https://msdn.microsoft.com/en-us/library/bb308959.aspx - LINQ

[6] https://en.wikipedia.org/wiki/Higher-order_function - Higher-order function

[7] https://en.wikipedia.org/wiki/Pure_function - Pure function

[8] https://en.wikipedia.org/wiki/Lazy_evaluation - Lazy evaluation

[9] https://en.wikipedia.org/wiki/Referential_transparency - Referential transparency

[10] https://en.wikipedia.org/wiki/Memoization - Memoization

[11] https://en.wikipedia.org/wiki/Dynamic_programming - Dynamic programming

Кратка биографија:

Марко Иветић, рођен је 23. септембра 1993. године у Новом Саду, Србија. Завршио је средњу економску школу „Светозар Милетић“ у Новом Саду, смер Финансијски техничар. Школске 2012/2013. године уписао је Факултет техничких наука у Новом Саду, одсек Рачунарство и аутоматика, усмерење Примењене рачунарске науке и информатика. Звање дипломирани инжењер електротехнике и рачунарства стекао је 2016. године. Исте године уписао је мастер академске студије на одсеку Рачунарство и аутоматика, усмерење Софтверско инжењерство. Положио је све испите прописане планом и програмом.