UDC: 004.4:004.23 INFO M: str. 46-53

IMPLEMENTACIJA DCI ARHITEKTURE PRIMENOM .NET PLATFORME
IMPLEMENTATION OF DCI ARCHITECTURE BY USING .NET PLATFORM

Branko Stevanović, Saša D. Lazarević

REZIME: DCI arhitektura je inovativan i interesantan princip koji promoviše mentalni model krajnjeg korisnika računarskog programa kao osnovni putokaz u kreiranju poslovnih aplikacija. Domenski objektu u DCI arhitekturi nemaju funkcionalnosti, one im se dodaju dinamički u vremenu izvršenja aplikacije. U ovom radu se istražuju mogućnosti dinamičke implementacije funkcionalnosti u vremenu izvršenja i predlaže AOP (Aspect Oriented Programing) kao moguće efikasno rešenje u realizaciji DCI arhitekture u Microsoft C# programskom jeziku.
KLJUČNE REČI: Konstrukcija softvera, Microsoft C#, DCI arhitektura, aspektno orijentisano programiranje, Dinamički programski jezici
ABSTRACT: DCI architecture is an innovative and interesting method that promotes the cognitive model of the program's end-user as the basic guideline for creating business applications. In DCI architecture, domain objects have no functionality - instead, it is added dynamically during the applications execution time. This work aims to explore the possibilities of dynamic implementation of functionality in execution time and suggests AOP (Aspect Oriented Programming) as a potentially efficient solution in realization of DCI architecture using Microsoft C# programming language.
KEY WORDS: Software construction, Microsoft C#, DCI architecture, Aspect oriented programming, Dynamic programming languages.

1. UVOD

Objektno orijentisano programiranje (OOP) je najčešća i najšire prihvaćena paradigma pisanja računarskih aplikacija danas. Razlog za to svakako leži u kvalitetu same ideje objektno orijentisanog programiranja, ali i u obrazovnim institucijama u kojima se izučava OOP paradigma, bogatstvu razvojnih okruženja, literature i širokoj zajednici programera i arhitekata koji stvaraju u nekom od objektno orijentisanih programskih jezika. Pisanje poslovnih aplikacija je posebno vezano za OOP okruženja i svedoci smo da je većina takvih aplikacija danas napisana u jednom od dva vodeća OOP programska jezika, C# (Microsoft) i Java (Oracle, nekada Sun Microsystems).

Kao i svaki proizvod i softverska aplikacija može da bude loša ili dobra. Uobičajen problem u korišćenju aplikacije jeste nerazumevanje toka rada programa od strane korisinka tog softvera. Uzrok tome najčešće jeste otklon od korisnikovog mentalnog i kognitivnog modela kao realne slike okruženja problema ka mentalnom i kognitivnom modelu softverskih inženjera i arhitekata koji problem vide na neki drugi, svoj način, obično dijametralno drugačije od onoga što vidi korisnik.

Originalno, ideja objektno orijentisanog programiranja je preslikavanje realnog problema u kod računarskog programa. Na taj način u programskom kodu postoji slika mentalnog modela problema a aplikacija postaje ekstenzija ljudskog uma. DCI (Data, Context, Interaction) arhitektura je zamišljena kao način da se pisanje aplikacija vrati ka korisnikovom mentalnom modelu. Rešavanje problema korisnika koje je prepušteno računaru, treba da bude replika načina na koji je taj problem rešavan bez računara. Tada korisnik sigurno razume šta softverski sistem radi i nema taj problem sa njegovom upotrebom.

Originalno predstavljena od strane autora Trigve Rinskaga (Trygve Reenskaug)[1] i Džejmsa Koplina (James O. Coplien), 2009. godine[1], DCI arhitektura razvoj poslovnih aplikacija vidi kao posao na dva fronta, opisu poslovnog sistema (ono što sistem jeste) i implementaciji poslovne logike sistema (ono što sistem radi). Ono što sistem jeste je često statični deo arhitekture, ono što se retko ili nikad ne menja i logično je da ga treba odvojiti od dela šta sistem radi, odnosno dela koji je podložan čestim promenama. Ono što sistem radi u DCI arhitekturi implementira se dinamički kroz uloge (Roles) i slučajeve korišćenja (Use cases). Problem sa tom paradigmom je očigledan u programskim jezicima sa statičkim proverama tipova, pre svega navedenim Java i C# jezicima u kojima se stvara većina poslovnih aplikacija. Ovde će biti ispitane i istražene mogućnosti u literaturi već opisanih načina implementacije DCI arhitekture u jeziku C# i predloženo rešenje koje nije zabeleženo u dostupnoj literaturi. Time se želi ukazati na mogućnost relativno jednostavnje implementacije DCI arhitekture kao interesantne paradigme u razvoju poslovnih aplikacija. U skladu sa tim ovaj rad treba da bude promocija DCI arhitekture sa mogućim implementacijama u realne poslovne probleme. Analizirani primeri su obrađeni u programskom jeziku C# koji je jedan od najviše korišćenih programskih jezika u rešavanju takvih problema.

2. DCI ARHITEKTURA

Svaki računarski program je u većoj ili manjioj meri uspešna slika nekog realnog ili virtualnog sistema. DCI arhi-

1 Trigve Rinskaga (Trygve Reenskaug) je takođe autor i idejni tvorac MVC (Model – View – Controler) arhitekture koja je obeležila poslednje decenije softverskog inženjerstva.

tektura je predstavljena 2009. godine kao pokušaj da se olakša problem opisa realnog, poslovnog okruženja računarskim programom. Mentalni model korisnika, njegovo viđenje problema koga treba rešiti računarom je ono što DCI arhitektura doživljava kao realno okruženje.

Korisnik računarske aplikacije u poslovnom sistemu skoro uvek obrađuje dokument. Dokument je fizička prezentacija podataka (Data). Zaposleni koji radi obradu dokumenata ima svoju ulogu (Role) u kontekstu poslovnog okruženja. Ono što definiše ulogu zaposlenog je posao koji on u toj ulozi obavlja. Ono što kvalifikuje zaposlenog da bude u ulozi je znanje da uradi posao koga zahteva ulogu. U velikoj većini poslovnih okruženja, takav posao je niz atomskih zadataka koji su povezani uslovom da je završetak jednog zadatka uslov za otpočinjanje drugog. Radi se, dakle, o algoritmu (postupku, proceduri). Kako svi algoritmi imaju ili treba da imaju ulaz i izlaz, radi se i o interakciji. Zaposleni koji zna sve potrebne algoritme na radnom mestu je kvalifikovan za ulogu tog radnog mesta. On može da nosi tu ulogu u poslovnom sistemu. Postoji velika verovatnoća da će se algoritmi vremenom menjati. Promenom algoritma, menjaju se i uloge. Zato uloge i algoritme ne bi trebalo posmatrati kao stabilan deo sistema. Oni se vremenom, izvesno, menjaju.

Ova tri nabrojana koncepta, model podataka, koncept uloga i interakcije, javljaju se kao potpuno nezavisni od konkretnog programskog jezika. Semantičko bogatstvo programskog jezika je da se ti koncepti mogu implementirati u njemu.

Logično je da korisnik najbolje poznaje funkcionisanje sistema. Posao softverskog arhitekte i inženjera je da kreiraju preslikavanje korisnikovog mentalnog modela okruženja i očekivane funkcionalnosti u kod programa. Idealno bi bilo ako bi to preslikavanje bilo jedan na jedan..

DCI arhitektura očekuje od softverskog arhitekte i softverskog inženjera sliku poslovnog sistema, onakvu kakva je u stvari. Da bi to bilo moguće softverski arhitekta i softverski inženjer moraju da razumeju razmatrani sistem na način na koji ga razumeju zaposleni u sistemu. Da bi ga razumeli moraju da uče od domenskih eksperata. Na taj način DCI arhitektura uključuje poznvavaoce domena i krajnje korisnike u razvoj softvera; Na taj način je DCI arhitektura i agilna (agile) u najboljem smislu.

2.1. Data – ono što sistem jeste

Data (podaci) spadaju u stabilni deo softvera. Podaci su materijal za gradnju domena, onoga što sistem jeste. U objektno orijentisanim jezicima to su klase. Klase koje opisuju domen se nazivaju često i domenske klase.

Domenske klase se sreću u mnogim softverskim arhitekturama pod drugim nazivima, najčešće kao Model u MVC, MVP i MVVM arhitekturama. Vrlo značajno je i iskustvo iz opisanih MV* (šta god) arhitektura da je domensku klasu, Model, potrebno očuvati jednostavnom, (u programerskom žargonu kaže se čak i glupom (dumb)). Ona treba da je opis entiteta, njihovih međusobnih veza i ništa više. Funkcionalnost je na nekom drugom mestu. DCI arhitektura, domenske klase, naziva Data objekti i deo su naziva arhitekture. Savremeni objektno orijentisani jezici bez ikakvog problema mogu da implementiraju koncept Data objekta na način kako je on promovisan u DCI arhitekturi.

2.2. Roles – funkcionalnosti sistema na način kako ih vidi korisnik sistema

Uloge su prirodan okvir za funkcionalnosti koje sistem opisuje. Posao DCI softverskog arhitekte je da projektuje takav sistem u kome je moguće naučiti Dumb objekat nekoj proceduri, algoritmu, dati mu ulogu. Takođe je posao DCI softverskog arhitekte da implementira razdvajanje onoga što sistem jeste od onoga što sistem radi na takav način da su domenski objekti uvek dumb – glupi, a da se uloge ostvaruju tako što se znanje domenskom objektu dinamički „upucava“ onda i samo onda kada je to potrebno. Tako imamo objekat sa funkcionalnošću koja je više osobina nego implementacija. Takođe imamo dobro odvojene celine softverske aplikacije, stabilni deo koji se uglavnom ne menja, ono što sistem jeste i deo koji je podložan frekventnim izmenama, ono što sistem radi. Na taj način konačno imamo i softverski sistem koji je slika kognitivnog modela korisnika aplikacije i koga onda korisnik razume i prihvata kao prirodnu ekstenziju svoga shvatanja sitema. Deo aplikacije u kome se znanje implementira u „glupi“ objekat je izuzetno zavisan od odabranog programskog jezika i okruženja u kome nastaje aplikacija. Ne postoji univerzalni princip kojim se ovakva funkcionalnost implementira. Opcije za implementaciju i jesu predmet istraživanja ovog rada.

2.3. Kontekst i interakcija

Kontekst je uvek okruženje. To je mesto u aplikaciji gde će se sresti uloge koje učestvuju u slučaju korišćenja. Potpuno isti koncept je i u realnom okruženju poslovne organizacije. Imamo obučene zaposlene i objekte njihovog rada u interakciji. To je ono što se želi opisati aplikacijom. To je i poslovni sistem kako ga vide korisnici, odnosno zaposleni u sistemu.

Implementacija i konteksta i interakcije u aplikaciji uveliko zavisi od programskog okruženja, jezika koji se koristi. U dinamičkim programskim jezicima kao što su Python i Javascript, moguće je u samom kontekstu, a pre same interakcije „upucati“ znanje domenskom objektu. To je nemoguće u jezicima sa statičkom proverom tipova. Kod jezika sa statičkom proverom tipova učenje domenskog objekta mora se obaviti pre konteksta, odnosno objekat sa ulogom mora u kontekst doći kao „naučen“. Čak ni to nije moguće jednostavno uraditi na način koji bi bio univerzalan. Svaki jezik ima svoj način i često je više oblik „varanja“ ugrađenog sistema nego prirodan tok programske logike[5].

3. NAČINI IMPLEMENTACIJE DCI ARHITEKTURE U MICROSOFT C# PROGRAMSKOM JEZIKU

Način implementacije DCI arhitekture, barem onog dela koji opisuje šta sistem radi, mnogo zavisi od odabranog programskog jezika, čak, kako će u ovom radu biti prikazano i od verzije programskog jezika koji se koristi. Dinamička priroda interakcije u DCI arhitekturi implicira upotrebu dinamičkih jezika kao što su Python, Javascript i sl. ali u praksi generalno nije uobičajena upotreba tih jezika u razvoju poslovnih aplikacija. Najzastupljeniji jezici u razvoju poslovnih aplikacija su i dalje noviji objektno orijentisani jezici sa statičkom proverom tipova i to konkretno jezici Java (Oracle) i C# (Microsoft). Statička priroda tih jezika ih uveliko diskvalifikuje iz upotrebe u razvoju poslovnih aplikacija baziranih na DCI arhitekturi. Cilj ovog rada je da pokaže da postoje realne mogućnosti da se DCI arhitektura implementira u C# programskom jeziku.

Postojeća literatura predlaže više mogućih rešenja, od kojih su neka zasnovana na mogućnosti miksovanja programskih jezika u DotNet okruženju i korišćenje programskih jezika Scala i IronRuby u primarno C# okruženju [8]. Kako ta rešenja nisu čist C#, u ovom radu neće biti više reči o njima.

Od verzije 3.0 programskog jezika C# moguća je implementacija DCI arhitekture korišćenjem ekstenzija. Ekstenzija je tehnika kojom je moguće proširivati funkcionalnost novim metodama postojećih tipovima podataka bez potrebe za nasleđivanjem konkretnog tipa. [3] Samo rešenje moguće je videti na [4].

Drugi interesantni način je omogućen proširenjem jezika C# dinamičkim (dynamic) tipom podataka, dodatim u verziju 4.0 jezika. Dodatkom dynamic tipa, C# postaje ako ne dinamički onda svakako pseudo-dinamički programski jezik [6]. Implementacija DCI arhitekture korišćenjem novih dinamičkih svojstava jezika može se videti takođe na [7].

Kako se u primenjenim rešenjima sa ekstenzijama i sa dinamičkim tipovima podataka sve odvija u okviru samo C# jezika, ova dva metoda su uključena u istraživanje i kasnija uporedna merenja. Implementacija koja se koristi u istraživanju je replika rešenja predloženih u [8].

3.1. *Aspekti AOP kao način implementacije DCI arhitekture u C# jeziku*

Aspekti (Acpect Oriented Programming (AOP)) [9] je interesantan koncept u softverskom inženjerstvu i arhitekturi koji pokušava da reši problem sa često ponavljanim fragmentima koda. Na primer ako je potrebno upisivanje u log fajlove. Problem je što u objektu koji ima svoju specifičnu namenu postoji fragment koda koji rešava neki eksterni problem (pisanje u log fajl) i što se takav kod ponavlja iz objekta u objekt. Na taj način ruši se princip da objekat u OOP treba da radi samo ono što stvarno treba da radi. Ideja aspekta i AOP je da se kod koji obavlja funkcionalnost koja nije primarna za objekat piše eksterno i jednostavno „upuca“ u objekat, odnosno da mu se doda i taj aspekt, bez da se menja namena i dodaje nepotreban kod u osnovnu funkcionalnost objekta. Time je povećana modularnost, samim tim i olakšano održavanje aplikacije [9].

Problem je naravno u programskim jezicima sa statičkom proverom tipova, kod kojih je podrazumevano da je sve definisano pre kompajliranja i da nakon toga nema mogućnosti da se promene već definisani objekti. Autorima ovog teksta je bilo logično da ako postoji sistem koji u statički proveravanim jezicima, konkertno u C#, rešava problem aspekata, onda se taj princip verovatno može primeniti i u implementaciji DCI arhitekture u jeziku C# iako se ne radi o istoj nameni niti ideji.

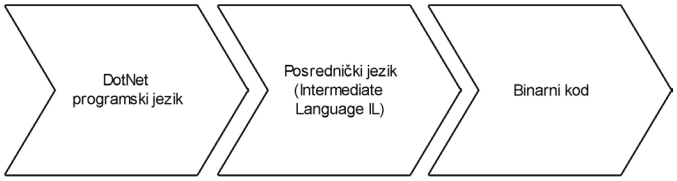
Problem sa aspektima postoji i u Java programskom jeziku i rešen je AspectJ jezikom koji je ekstenzija osnovnog jezika koji implementira aspekte. Slična rešenja postoje za C# programski jezik ali ali za razliku od AspectJ koji ima svoju primenu, nijedno rešenje koje bi bilo ekstenzija C# jezika nije zaživelo u DotNet zajednici.

AOP u C# moguće je implementirati nekim od IOC (Inversion of Control) okvira kojih ima na tržištu od komercijalnih do besplatnih. U poslednjim inkarnaciijama razvojnog okruženja MS VisualStudio10 i MS VisualStudio12 ima ih i ugrađenih u okruženje (Unity framework i MEF – Microsoft Extension Framework). U svakoj od IOC implementacija koristi se uzor Proxy. Namena Proxy uzora je da preklopi ili sakrije neki drugi objekat i da svaki poziv ka objektu mora da pređe preko Proxy-ja. Klijent pri korišćenju uzora Proxy ne zna da postoji drugi objekat koga Proxy krije, njegovo znanje je ograničeno na objekat koji je Proxy [10]. Prava funkcionalnost je u stvari u objektu koji je sakriven.

U ovom radu više pažnje biće posvećeno drugom mogućem načinu implementacije aspekata u C# jeziku. Radi se o tehnici ILWeaving koju će i biti šire objašnjena.

3.1.1. IlWeaving tehnika

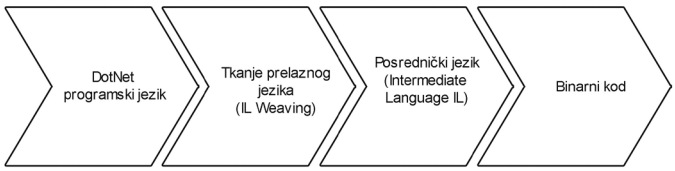
IL Weaving (Intermediate Language Weaving – upredanje u posrednički jezik) je nova, vrlo interesantna tehnologija prisutna na C# i DotNet sceni zadnjih par godina. Prilikom produkcije DotNet programa (na svim programskim dijalektima koji su u upotrebi u okruženju) izvršni tok izgleda kao na slici:



Slika 1. – *Tok prevođenja programa u DotNet okruženju*

Bilo bi idealno ako bi već statički proverene objekte od strane kompajlera mogli da izmenimo tako da im omogućimo novu funkcionalnost, svojstva, metode. Na taj način tipovi su statički provereni, a ipak dinamički izmenjivi. To je upravo ono što se želi i može uraditi IL Weaving tehnikama

IL Weaving ima tok prevođenja:



Slika 2. – *Tkanje posredničkog jezika (IL Weaving)*

U vremenu prevođenja (compile time) dešava se čarolija. Nakon statičke provere tipova, pre prevođenja u Intermediate Language, prekida se proces gradnje (Build) aplikacije. Na tom mestu se u kod objekta koji treba obogatiti aspektom „upucava" kod nove funkcionalnosti. Tako obogaćen objekat se šalje na dalji proces kompajliranja. Program onda vidi novonastali miks kao legitiman programski objekat. Korisnik ne primećuje ništa neobično u izvršenju. Cena se plaća kod prevođenja (kompajliranja) i to u trajanju procesa prevođenja, a u vremenu izvršenja to je najnormalniji izvršni fajl koji se izvršava kao da je i od početka pisan kao takav.

3.1.1.1. KAKO JE IL WEAVING MOGUĆ

Čist IL Weaving je moguć upotrebom ne toliko poznate Mono.Cecil biblioteke (DLL) koja je deo Mono projekta i koja radi upravno na način koji je prezentovan u prethodnom odeljku. Poseban problem sa Mono.Cecil bibliotekom je odsustvo dokumentacije što svakako komplikuje primenu i savladavanje jednog dosta komplikovanog koncepta. U narednom delu teksta biće prikazan jednostavan primer kojim se može prikazati kako izgleda proces programiranja sa Mono.Cecil bibliotekom. Dat je objekat PersonDumb:

```
public class PersonDumbObj
{
  public string Name {get;set;}
  public string Surname {get;set;}
  public PersonDumbObj() { }
  public void DoWhatYouNeedToDo() { }
}
```

Treba primetiti metod DoWhatYouNeedToDo koji je prazan, nema parametre i ne radi ništa.

Takođe dat je i objekat PersonFullObj:

```
public class PersonFullObj
{
  public string Name {get;set;}
  public string Surname {get;set;}
  public PersonDumbObj(){}

  public void DoWhatYouNeedToDo()
  {
    Console.WriteLine("Doing what i want");
  }
}
```

Izmene u objektu data su podebljanim slovima. Ovde treba obratiti pažnju na istu metodu kao i u prethodnom primeru, ovde je implementirana i jednostavna akciju koja treba da ispiše tekst u konzoli programskog okruženja. Dakle, postoje dva objekta, PersonDumbObj i PersonFullObj koji su potpuno isti, samo što u Dumb objektu metoda je prazna, dok u Full objektu metoda nešto radi. Ako analiziramo IntermediateLanguage kod oba objekat videćemo da IL kod metode DoWhatYouNeedToDo Dumb objekta izgleda:

```
1  .method public hidebysig
2      instance void DoWhatYouNeedToDo () cil managed
3  {
4      // Method begins at RVA 0x209b
5      // Code size 2 (0x2)
6      .maxstack 8
7
8      IL_0000: nop
9      IL_0001: ret
10 } // end of method PersonDumbObj::DoWhatYouNeedToDo
11
```

Slika 3. – *Izgled IL koda prazne metode*

Dok ista metoda Full objekta u IL izgleda kao:

```
1  .method public hidebysig
2      instance void DoWhatYouNeedToDo () cil managed
3  {
4      // Method begins at RVA 0x209b
5      // Code size 2 (0x2)
6      .maxstack 8
7
8      IL_0000: nop
9      IL_0001: ldstr "Doing what I wont!"
10     IL_0006: call void [mscorlib]System.Console::WriteLine(string)
11     IL_000b: nop
12     IL_000c: ret
13 } // end of method PersonDumbObj::DoWhatYouNeedToDo
14
```

Slika 4. – *Izgled IL koda implementirane metode*

Odnosno razlika koja postoji u IL kodu je u zaokruženim redovima:

```
IL_0000: nop
IL_0001: ret
IL_0000: nop
IL_0001: ldstr "Doing what I wont!"
IL_0006: call void [mscorlib]System.Console::WriteLine(string)
IL_000b: nop
IL_000c: ret
```

Slika 5. – *Redovi IL u kojima postoj razlika*

Okvir pokazuje šta je dodato, odnosno šta je telo metode u IL kodu. Mono.Cecil biblioteka može upravo da uradi to, da u IL kod "upuca" novi kod, da sačuva na disku izmenjeno stanje i da od tog momenta progam ima sasvim drugo ponašanje, namenu, šta god programer odluči…

Kod koji u konkretnom primeru radi ovakvu izmenu u IL kodu dat je u snipetu:

```
MethodInfo consoleWriteLine =
        typeof(Console).GetMethod("WriteLine",new
Type[]{typeof(string)});

AssemblyDefinition definition = AssemblyFactory.
GetAssembly("PersonDumb.dll");
```

```
   TypeDefinition type = definition.MainModule.
Types["PersonDumb.PersonDumbObj"];

   MethodDefinition typeMethod = type.Methods.
OfType<MethodDefinition>().Where
            (m => m.Name == "DoWhatYouNeedToDo" &&
m.Parameters.Count == 0).Single();
   CilWorker cilWorker = typeMethod.Body.CilWorker;

   Instruction ldstr = cilWorker.Create(OpCodes.
Ldstr, "Doing what I wont!");
   cilWorker.InsertAfter(cilWorker.GetBody().
Instructions[0], ldstr);

   MethodReference writeLine = definition.MainModu-
le.Import(consoleWriteLine);
   Instruction callConsoleWriteLine = cilWorker.
Create(OpCodes.Call, writeLine);
   cilWorker.InsertAfter(ldstr, callConsoleWriteLi-
ne);
   Instruction nop = cilWorker.Create(OpCodes.Nop);
   cilWorker.InsertAfter(callConsoleWriteLine,
nop);

   AssemblyFactory.SaveAssembly(definition,"PersonD
umb.Changed.dll");
```

Pri kraju rutine vidi se red u kome se čuva izmenjeni DLL. IL kod izmenjenog DLL-a je potpuno isti kao i kod FullPersonObj objekta. Znači da je Mono.Cecil rutina uspešno "upucala" novo znanje u inače "dumb" DumbPersonObj objekat.

Verovatno je bitno primetiti i naglasiti postojanje metoda DoWthaYouNeedToDo u obe klase i u PersonDumbObj i u PersonFullObj, bez obzira što je u dumb objektu ona prazna. ILWeaving postupak mora znati mesto gde "upucava" izmenjenu ili novu rutinu. Ako ne postoji takvo mesto, sistem ne zna šta da presretne i zameni.

Ovim je pokazano na koji način se može pomoću biblioteke Mono.Cecil može menjati IL kod već kompajliranog objekta, dodavati rutine i učiti objekat ulogama koje su potrebne u slučajevima korišćenja.

3.1.1.2. KAKO U PRAKSI PRIMENITI IL WEAVING

Problem sa odsustvom dokumentacije Mono.Cecil biblioteke i samim tim dugog postupka navodi korisnike ka jednostavnijem rešenju. Na tržištu postoje dve primene Mono.Cecil biblioteke koje su namenjene AOP programiranju. Prva je PostSharp biblioteka, odličan, verovatno najpopularniji radni okvir (framework) kada je u pitanju AOP u profesionalnom razvoju poslovnih aplikacija. Drugi radni okvir je biblioteka LinFu (http://code.google.com/p/linfu/) autora Filipa Lorinoa (Philp Laureano). LinFu biblioteka je više generička nego PostSharp, odnosno pokriva više oblasti u savremenom programiranju poslovnih aplikacija. AOP je jedan od više problema koje rešava LinFu. AOP u LinFu bibliotekama je takođe zasnovan na primeni Mono.Cecil biblioteka [11] Ono što je autora ovog rada okrenulo ka LinFu bibliotekama je licenciranje – za razliku od PostSharp biblioteke koja je komercijalna, LinFu je besplatna biblioteka koja se može slobodno koristiit u komercijalnim projektima po uslovima GNU Lesser Public Licence.

Da bi sistem sa LinFu radio u aplikaciji potrebno je odraditi jedan ne tako uobičajen korak, potrebno je izmeniti MSBuild dokument u projektu na način da mu se pokaže gde se nalazi LinFu.Aop dll. Ovo je kritični deo primene LinFu. AOP. Šta se u stvari dobija ovom operacijom? Kada sistem krene u kompajliranje aplikacije, pre nego što generiše IL kod, biva zaustavljen, tada se dešava već opisana čarolija AOP, IL Weaving, sistem pronalazi dumb objekte i AOP funkcionalnosti koje im pripadaju, miksuje dumb objekte sa novim funkcionalnostima i šalje ih dalje na kompajliranje. Aplikacija u radu tako miksovane objekte vidi kao legitimne biblioteke i radi sa njima skoro kao „običnim" C# objektima. Na taj način uz cenu malo sporijeg kompajliranja imamo objekte koji su u vremenu izvršavanja po brzini jednaki ili skoro jednaki normalno pisanim objektima sa implementiranim funkcionalnostima u kodu. Proces izmene MSBuild dokumenta se radi samo jednom i samo na jednom mestu tako da nema uticaja na brzinu rada programera u razvoju. Kod koji se piše je jednostavan i sve govori u prilog ovom pristupu u razvoju DCI aplikacija.

4. IMPLEMENTACIJA U REALNIM PRODUKCIONIM USLOVIMA

Implementacije korišćenjem ekstenzija i dynamic tipa podataka postoje objavljene i mogu se lako naći na [8]. U daljem tekstu biće prikazana implementacija AOP principa na realnom primeru upotrebom AOP paradigme.

4.1. Šta sistem radi – upotreba AOP principa

Implementacija upotrebom aspekata biti data na primeru koji je funkcionalno identičan primerima koji su prezentovani u literaturi pri primeni sa ekstenzijama i biblioteke dynamic u jeziku C#. Na taj način imamo "poštenu" situaciju koja se onda može meriti i porediti.

Postoje objekti računi koji su u interakciji transfera novca. Oba računa i izvor i destinacija su tipa SavingAccount koji je nasledio dumb apstraktnu klasu Account koja je potpuno ista u sva tri načina implementacije o kojoj se govori. Time se i poštuje preporuka da u DCI dumb klase treba da budu apstraktne kako bi ih obavezno nasledili objekti koji nose uloge. Saving Account u verziji koja implementira AOP način ima dodatu metodu DoWhatYouNeedToDo čija je uloga objašnjena u prethodnom tekstu.

Učenje klase SavingAccount odrađeno je u klasi koja se zove TransferMoneySourceTrait i koja izgleda kao u kodu:

```
   public class TransferMoneySourceTrait : IAround-
dInvoke
   {
        private SavingAccount source;
        private SavingAccount sink;
        private decimal amount;
        public TransferMoneySourceTrait( SavingAcco-
unt source,
                                SavingAccount sink,
                                decimal amount)
        {
          this.source = source;
          this.sink = sink;
          this.amount = amount;
        }

        public void AfterInvoke(IInvocationContext
context, object returnValue)
        {
          source.DecreaseBalance(amount);
          source.Log("Withdrawal: " + amount);
          sink.IncreaseBalance(amount);
          sink.Log("Receiving: " + amount);
        }

        public void BeforeInvoke(IInvocationContext
context)
        {
        }
     }
```

Klasa implementira interfejs IAroundInvoke iz LinFu frejmvorka sa rutinama BeforeInvoke i AfterInvoke. Kako se iz imena može zaključiti ova implementacija ima mogućnost da izvrši deo koda pre originalne metode (u posmatranom slučaju DoWhatYouNeedToDo metode), da odradi originalnu metodu i da konačno odradi metodu posle originalne. Navedeni primer koristi samo rutinu koja se izvršava posle originalne

U kontekstu je obavljeno učenje objekta Source, tipa SavingAccount novom znanju i to u metodi DoIt čije je kod dat:

```
   public void Doit()
   {
   var accountTest = Source;

        TransferMoneySourceTrait trait = new Transfe
rMoneySourceTrait(accountTest,
             Sink, Amount);
        IModifiableType acountTestModified = accoun-
tTest as IModifiableType;
        var provider = new SimpleAroundInvokeProvid
er(trait,
             c => c.TargetMethod.Name == "DoWhatYou-
NeedToDo");
        acountTestModified.IsInterceptionEnabled =
true;
        acountTestModified.AroundInvokeProvider =
provider;
        accountTest.DoWhatYouNeedToDo();
   }
```

Primetno je da veliki deo funkcionalnost DoIt nosi implementacija LinFu frejmworka. U navedenom kodu promenljiva provider je tipa SimpleAroundInvokeProvider koji obezbeđuje funkcionalnost upucavanja oko postojeće rutine. Parametri koje zahteva su klasa koja implementira interfejs IAroundInvoke konkretno TransferMoneySourceTrait koja je već prikazana i naziv rutine koja se presreće i to DoWhatYouNeedToDo rutine. Interfejs IModifiableType omogućava LinFu frejmvorku da izmeni skoro svaku klasu na nivou pojedinačne instance. Tako u gornjem primeru samo je instanca Source klase SavingAccount promenjena. Klasa Sink koja je istog tipa ostaje neizmenjena.

Arhitekta koji dizajnira aplikaciju lako može da izvuče kompletnu implementaciju LinFu frejmvorka u eksternu klasu kako bi omogućio izuzetno laganu implementaciju logike konteksta. Tako refaktorisana klasa može da izgleda kao u kodu:

```
   public class WeaveTraitUtility
   {
        public static void WeaveTrait(IAroundInvoke
useCaseTrait, SavingAccount account)
        {
              IModifiableType accountModifiable =
account as IModifiableType;
              var provider = new SimpleAroundInvok
eProvider(useCaseTrait,
              c => c.TargetMethod.Name == "DoWha-
tYouNeedToDo");
              accountModifiable.IsInterceptionEna-
bled = true;
              accountModifiable.AroundInvokeProvi-
der = provider;
        }
   }
```

U tom slučaju metoda DoIt u kontekstu bi izgledala kao u kodu:

```
   public void Doit()
   {
        var accountTest = Source;
        TransferMoneySourceTrait trait = new Tran
sferMoneySourceTrait(accountTest,

Sink, Amount);
        WeaveTraitUtility.WeaveTrait(trait, acco-
untTest);
        accountTest.DoWhatYouNeedToDo();
   }
```

što je već funkcionalnost koju može da primeni i junior programer u softverskom preduzeću.

4.2. Uporedna analiza

U prethodnom tekstu objašnjena je tehnika kojom se u C# programskom jeziku paradigma AOP implementira u DCI arhitekturu. Takođe navedeni su još dva od ranije poznata načina implementacije DCI u C#. Svaka od navedenih tehnika ima svoje slabosti i mane. Ovde je mesto da se prikažu i uporede kvaliteti navedenih metoda.

4.2.1. Implementacija korišćenjem ekstenzija u C# jeziku

Prednosti implementacije korišćenjem ekstenzija u C# jeziku su uglavnom u tome što je to najstarija poznata metoda i ima najviše primera koji su obrađeni u literaturi. Takođe ovo je metoda koja koristi samo ugrađene mogućnosti C# jezika. Problem sa ovom implementacijom je dosta komplikovana struktura koja nije intuitivna u dovoljnoj meri da bi je jednostavno primenjivao manje iskusan programer. Problem sa ekstenzijama takođe je i što su to statičke metode, tako da ne mogu da čuvaju stanja, mogu da implementiraju samo funkcionalnosti. Kako će se u daljem tekstu videti u pitanju je ipak implementacija koja pokazuje najveću brzinu u vremenu izvršenja.

4.2.2. Korišćenje dynamic tipa podataka u C# jeziku

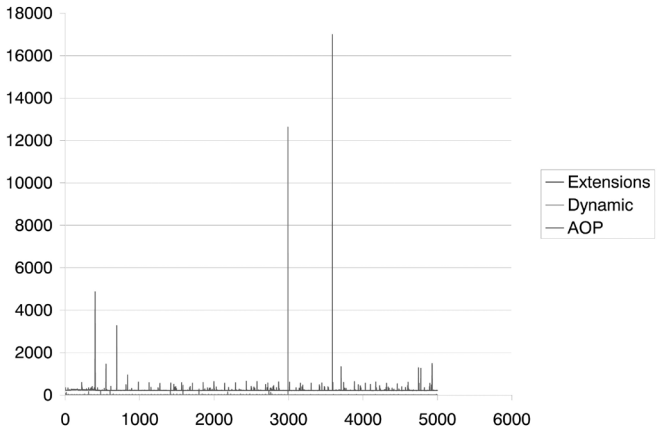
Dynamic tip podataka kao rešenje u implementaciji DCI arhitekture ima prednosti u relativno jednostavnoj strukturu kojom se primenjuje, takođe i u činjenici da je u potpunosti ugrađena u sam C# jezik. Problem može da bude u korišćenju lambda izraza koji mogu da budu zbunjujući za neiskusnog korisnika. Takođe problem je nedostatak intelisense pomoći kod implementacije dynamic tipova. Sistem ne zna šta programer hoće i sve provere ispravnosti koda u dynamic tipova su u vremenu izvršenja (runtime). U merenjima se ispostavilo da je ova metoda brza, skoro kao metoda sa ekstenzijama koja je šampion brzine u posmatranom slučaju.

4.2.3. AOP implementacija DCI arhitekture

AOP je i najviše objašnjena metoda u ovom tekstu. Pokazano je da konkretna implementacija arhitekturom može biti maksimalno pojednostavljena. Na tom nivou primenjiva je u slučaju kada je angažovan relativno neiskusan programer. Opet sama implementacija zahteva iskusnog programera koji može da se nosi sa tehnikom IL Weaving. Veliki problem takođe je nemogućnost postavljanja tačaka prekida (break point) u procesu traženja grešaka pri programiranju. Sistem jednostavno ne zna gde da prekine izvršenje nove klasu koja je nastala u procesu IL Weaving-a. Autori ovog teksta su u istraživanju imali još čudnih momenata, jedan od onih na koji i dalje nemaju objašnjenje je nemogućnost provere objekata na null vrednosti. Kompajlirana aplikacija na tom mestu izbacuje grešku bez ikakvog očiglednog razloga u vremenu izvršenja. U merenjima će biti pokazano i da je ovo najsporiji način implementacije. Svejedno, jednostavnost i elegancija kojom se AOP implementira autore teksta opredeljuje upravo za ovo rešenje kod implementacije DCI arhitekture.

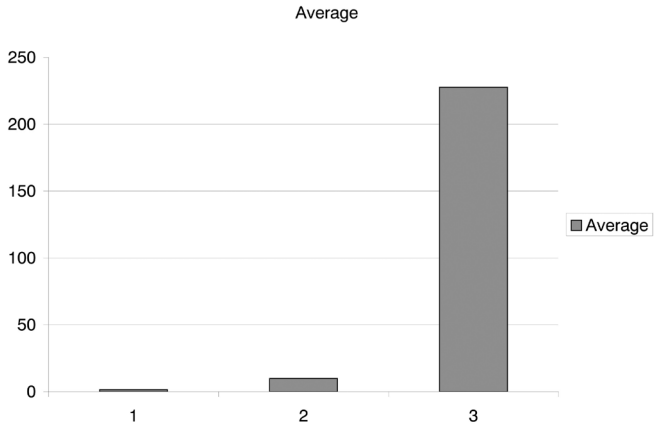
4.2.4. Merenja

Merenje je najbolji način da se dođe do konačnih sudova o kvalitetu pojedinačnih rešenja. U tom cilju napisana je aplikacija koja može da testira svako od rešenja određen broj puta i da prolazna vremena beleži u memoriju kako bi mogla biti grafički prikazana. Nakon 5000 krugova svaka od primenjenih varijanti dala je grafički prikazano sledeće rezultate:



Slika 6. – *grafik merenja trajanja metoda*

I u proseku gledano:



Slika 7. – *Prosečne vrednosti trajanja merenih metoda*

gde je očigledno da AOP rešenje ima najlošije rezultate. Ipak treba naglasiti da je merenje obavljeno u sistemskim jedinicama Tick koje su najmanje što računar pojedinačno može da zabeleži. U pokušaju da se izmere rezultati u milisekundama, dobijene su vrednosti od 0 milisekundi u svim prolazima za sve varijante koje su bile ispitivane.

Iako je AOP rešenje mnogostruko sporije od ostala dva rešenja, vrednost koju je ostvarilo i to rešenje ostaje ispod jedne milisekunde. Velika je verovatnoća da bi neki drugi proces u realnoj aplikaciji imao značajniji uticaj na sveukupnu brzinu nego sama implementacija AOP arhitekture.

5. ZAKLJUČAK

DCI arhitektura (barem prema dostupnim tekstovima i literaturi) nije doživela popularnost kakvu verovatno zaslužuje u zajednici profesionalnih programera koja se po pravilu angažuje na izradi poslovnih aplikacija. Razlog za to verovatno prvenstveno leži u tome što prema svojoj filozofiji DCI arhitektura nije namenjena jezicima koji su glavni tok, Java i C#. I kada se u obzir uzme da postoje implementacije u navedenim jezicima, predložena rešenja (upotrebom ekstenzija pogotovu) nisu jednostavna, uključuju varanja sistema i sigurno su problem kod eventualnog uvođenju manje iskusnih programera u produkciju koda. U uslovima kakvi vladaju na tržištu niko ne želi da gubi vreme na obučavanje mlađih i neiskusnijih kolega i biraju se rešenja koja se lakše implementiraju i ne zahtevaju veliku obuku.

Upotreba aspekta koja je detaljnije objašnjena u radu može da reši problem sa implementacijom, pokazano je da se princip može refaktorisati do nivoa na kome ga može implementirati i relativno neiskusan programer. Time bi arhitekturom aplikaciija bila podeljena na deo koji jeste složeniji ali je napisan samo jednom i izolovan je u posebnu klasu, dokument, dok je produkcija koda relativno jednostavna i pravolinijska.

Na taj način bi se mogao koristiti princip DCI arhitekture koji je generalno dobar i interesantan arhitektonski pristup razvoju poslovnih aplikacija, bez velikih gubitaka u obuci zaposlenih a uz sve prednosti koje donosi u organizaciji aplikacije i sveukupne filozofije pisanja koda prema meri krajnjih korisnika.

6. LITERATURA

[1] Reenskaug T., Coplien J.O., 2009, "The DCI Architecture: A New Vision of Object-Oriented Programming", http://www.artima.com/articles/dci_vision.html,

[2] Lazarević D.S., Petrović M., 2003 "The Correlation of Equality, Hashing and Inheritance", InfoM Volumen 8/2003, ISSN 1451-4397

[3] Extension Methods (C# Programming Guide), http://msdn.microsoft.com/en-us/library/vstudio/bb383977(v=vs.100).aspx

[4] Horsdal C.G, 2009, "DCI in C#", http://horsdal.blogspot.com/2009/05/dci-in-c.html

[5] Coplien J.O, Bjørnvig G., 2010, "Lean Software Architecture for Agile Software Development". Willey and Sons 2010 ISBN 978-0-470-68420-7

[6] Dynamic (C# Reference); http://msdn.microsoft.com/en-us/library/dd264741.aspx

[7] Lazarević D.S., Radenkovic S.; 2006. "Student Knowledge Evaluation System as a Support for Flexible Assessment", InfoM Volumen 17/2006; ISSN 1451-4397

[8] Horsdal C.G. 2010., "Options for DCI on .NET", http://horsdal.blogspot.com/2010/12/options-for-dci-on-net.html

[9] Aspect-oriented programming; http://en.wikipedia.org/wiki/Aspect-oriented_programming

[10] JohnsonR., Vlissides J., Helm R., Gamma E., 1994, "Design Patterns: Elements of Reusable Object-Oriented Software" Addison-Wesley.

[11] Laureano P., "Pervasive Method Interception and Replacement for Sealed Types in Any .NET Language" http://www.codeproject.com/Articles/23333/Introducing-LinFu-Part-VI-LinFu-AOP-Pervasive-Meth

Branko Stevanović MSc ,
Mail: branko@agenzzia.com
Visoka poslovno-tehnička škola strukovnih studija Užice,
Oblasti interesovanja: Softversko inženjerstvo, Multimedijalno računarstvo, Web dizajn

Saša D. Lazarević, Katedra za softversko inženjerstvo,
Fakultet organizacionih nauka,
Univerzitet u Beogradu
Kontakt: sasa.lazarevic@fon.bg.ac.rs
Oblasti interesovanja: konstrukcija softvera, testiranje softvera, kvalitet softvera, razvoj informacionih sistema, structure podataka i algoritmi, baze podataka, sistemi za upravljanje dokumentacijom, .NET platforma